

The documented source of Memoize, Advice and CollArgs

Memoize v1.4.1 Advice v1.1.1 Collargs v1.2.0
December 2, 2024 March 15, 2024 March 15, 2024

Sašo Živanović

✉ saso.zivanovic@guest.arnes.si
📁 spj.ff.uni-lj.si/zivanovic
🌐 github.com/sasozivanovic

This file contains the documented source code of package [Memoize](#) and, somewhat unconventionally, its two independently distributed auxiliary packages [Advice](#) and [CollArgs](#).

The source code of the \TeX parts of the package resides in `memoize.edtx`, `advice.edtx` and `collargs.edtx`. These files are written in [EasyDTX](#), a format of my own invention which is almost like the DTX format but eliminates the need for all those pesky `macrocode` environments: Any line introduced by a single comment counts as documentation, and to top it off, documentation lines may be indented. An `.edtx` file is converted to a `.dtx` by a little Perl script called `edtx2dtx`; there is also a rudimentary Emacs mode, implemented in `easydoctex-mode.el`, which takes care of fontification, indentation, and forward and inverse search.

The `.edtx` files contain the code for all three formats supported by the three packages — \LaTeX (guard `latex`), plain \TeX (guard `plain`) and \ConTeXt (guard `context`) — but upon reading the code, it will quickly become clear that Memoize was first developed for \LaTeX . In §1, we manually define whatever \LaTeX tools are “missing” in plain \TeX and \ConTeXt . Even worse, \ConTeXt code is often just the same as plain \TeX code, even in cases where I’m sure \ConTeXt offers the relevant tools. This nicely proves that I have no clue about \ConTeXt . If you are willing to \ConTeXt -ualize my code — please do so, your help is welcome!

The runtimes of Memoize (and also Advice) comprise of more than just the main runtime for each format. Memoize ships with two additional stub packages, `nomemoize` and `memoizable`, and a \TeX -based extraction script `memoize-extract-one`; Advice optionally offers a `TikZ` support defined in `advice-tikz.code.tex`. For the relation between guards and runtimes, consult the core of the `.ins` files below.

memoize.ins

```
\generate{%
  \file{memoize.sty}{\from{memoize.dtx}{mmz,latex}}%
  \file{memoize.tex}{\from{memoize.dtx}{mmz,plain}}%
  \file{t-memoize.tex}{\from{memoize.dtx}{mmz,context}}%
  \file{nomemoize.sty}{\from{memoize.dtx}{nommz,latex}}%
  \file{nomemoize.tex}{\from{memoize.dtx}{nommz,plain}}%
  \file{t-nomemoize.tex}{\from{memoize.dtx}{nommz,context}}%
  \file{memoizable.sty}{\from{memoize.dtx}{mmzable,latex}}%
  \file{memoizable.tex}{\from{memoize.dtx}{mmzable,plain}}%
  \file{t-memoizable.tex}{\from{memoize.dtx}{mmzable,context}}%
  \file{memoizable.code.tex}{\from{memoize.dtx}{mmzable,generic}}%
  \file{memoize-extract-one.tex}{\from{memoize.dtx}{extract-one}}%
  \file{memoize-biblatex.code.tex}{\from{memoize.dtx}{biblatex}}%
  \file{memoize-beamer.code.tex}{\from{memoize.dtx}{beamer}}%
```

advice.ins

```
\file{advice.sty}{\from{advice.dtx}{main,latex}}%
\file{advice.tex}{\from{advice.dtx}{main,plain}}%
\file{t-advice.tex}{\from{advice.dtx}{main,context}}%
\file{advice-tikz.code.tex}{\from{advice.dtx}{tikz}}%
```

collargs.ins

```
\file{collargs.sty}{\from{collargs.dtx}{latex}}%
\file{collargs.tex}{\from{collargs.dtx}{plain}}%
\file{t-collargs.tex}{\from{collargs.dtx}{context}}%
```

Memoize also contains two scripts, `memoize-extract` and `memoize-clean`. Both come in two functionally equivalent implementations: Perl (`.pl`) and a Python (`.py`). Their code is listed in §9.

Contents

1	First things first	3
2	The basic configuration	7
3	Memoization	11
3.1	Manual memoization	11
3.2	The memoization process	15
3.3	Context	24
3.4	C-memos	26
3.5	Cc-memos	28
3.6	The externs	31
4	Extraction	39
4.1	Extraction mode and method	39
4.2	The record files	41
4.2.1	The .mmz file	42
4.2.2	The shell scripts	43
4.2.3	The Makefile	44
4.3	T _E X-based extraction	45
4.3.1	memoize-extract-one.tex	47
5	Automemoization	49
5.1	L ^A T _E X-specific handlers	53
6	Support for various classes and packages	55
6.1	PGF	56
6.2	TikZ	56
6.3	Forest	57
6.4	Beamer	57
6.5	Biblatex	58
7	Initialization	62
8	Auxiliary packages	65
8.1	Extending commands and environments with Advice	65
8.1.1	Installation into a keypath	65
8.1.2	Submitting a command or environment	68
8.1.3	Executing a handled command	74
8.1.4	Environments	76
8.1.5	Error messages	78
8.1.6	Tracing	79
8.1.7	The TikZ collector	81
8.2	Argument collection with CollArgs	82
8.2.1	The keys	84
8.2.2	The central loop	89
8.2.3	Auxiliary macros	91
8.2.4	The handlers	96
8.2.5	The verbatim modes	114
8.2.6	Transition between the verbatim and the non-verbatim mode	119
9	The scripts	129
9.1	The Perl extraction script memoize-extract.pl	129
9.2	The Python extraction script memoize-extract.py	141
9.3	The Perl clean-up script memoize-clean.pl	150
9.4	The Python clean-up script memoize-clean.py	152

1 First things first

Identification of memoize, memoizable and nomemoize.

```
1 (*mmz)
2 <latex>\ProvidesPackage{memoize}[2024/12/02 v1.4.1 Fast and flexible externalization]
3 <context>%D \module[
4 <context>%D     file=t-memoize.tex,
5 <context>%D     version=1.4.1,
6 <context>%D     title=Memoize,
7 <context>%D     subtitle=Fast and flexible externalization,
8 <context>%D     author=Saso Zivanovic,
9 <context>%D     date=2024-12-02,
10 <context>%D     copyright=Saso Zivanovic,
11 <context>%D     license=LPPL,
12 <context>%D ]
13 <context>\writestatus{loading}{ConTeXt User Module / memoize}
14 <context>\unprotect
15 <context>\startmodule[memoize]
16 <plain>% Package memoize 2024/12/02 v1.4.1
17 </mmz>
18 (*mmzable)
19 <latex>\ProvidesPackage{memoizable}[2024/12/02 v1.4.1 A programmer's stub for Memoize]
20 <context>%D \module[
21 <context>%D     file=t-memoizable.tex,
22 <context>%D     version=1.4.1,
23 <context>%D     title=Memoizable,
24 <context>%D     subtitle=A programmer's stub for Memoize,
25 <context>%D     author=Saso Zivanovic,
26 <context>%D     date=2024-12-02,
27 <context>%D     copyright=Saso Zivanovic,
28 <context>%D     license=LPPL,
29 <context>%D ]
30 <context>\writestatus{loading}{ConTeXt User Module / memoizable}
31 <context>\unprotect
32 <context>\startmodule[memoizable]
33 <plain>% Package memoizable 2024/12/02 v1.4.1
34 </mmzable>
35 (*nommz)
36 <latex>\ProvidesPackage{nomemoize}[2024/12/02 v1.4.1 A no-op stub for Memoize]
37 <context>%D \module[
38 <context>%D     file=t-nomemoize.tex,
39 <context>%D     version=1.4.1,
40 <context>%D     title=Memoize,
41 <context>%D     subtitle=A no-op stub for Memoize,
42 <context>%D     author=Saso Zivanovic,
43 <context>%D     date=2024-12-02,
44 <context>%D     copyright=Saso Zivanovic,
45 <context>%D     license=LPPL,
46 <context>%D ]
47 <context>\writestatus{loading}{ConTeXt User Module / nomemoize}
48 <context>\unprotect
49 <context>\startmodule[nomemoize]
50 <mmz>% Package nomemoize 2024/12/02 v1.4.1
51 </nommz>
```

Required packages and L^AT_EXization of plain T_EX and ConT_EXt.

```
52 (*(mmz, mmzable, nommz) & (plain, context))
53 \input miniltx
54 </(mmz, mmzable, nommz) & (plain, context)>
```

Some stuff which is “missing” in miniltx, copied here from latex.ltx.

```

55 (*mmz & (plain, context))
56 \def\PackageWarning#1#2{%
57   \newlinechar`^^J\def\MessageBreak{^^J\space\space#1: }%
58   \message{#1: #2}}
59 </mmz & (plain, context)>

```

Same as the official definition, but without `\outer`. Needed for record file declarations.

```

60 (*mmz & plain)
61 \def\newtoks{\alloc@5\toks\toksdef@cclvi}
62 \def\newwrite{\alloc@7\write\chardef\sixt@n}
63 </mmz & plain>

```

I can't really write any code without `etoolbox` ...

```

64 (*mmz)
65 <latex>\RequirePackage{etoolbox}
66 <plain, context>\input etoolbox-generic

```

Setup the `memoize` namespace in `LuaTeX`.

```

67 \ifdefined\luatexversion
68   \directlua{memoize = {}}
69 \fi

```

`pdftexcmds.sty` eases access to some PDF primitives, but I cannot manage to load it in `ConTeXt`, even if it's supposed to be a generic package. So let's load `pdftexcmds.lua` and copy-paste what we need from `pdftexcmds.sty`.

```

70 <latex>\RequirePackage{pdftexcmds}
71 <plain>\input pdftexcmds.sty
72 (*context)
73 \directlua{%
74   require("pdftexcmds")
75   tex.enableprimitives('pdf@', {'draftmode'})
76 }
77 \long\def\pdf@mdfivesum#1{%
78   \directlua{%
79     oberdiek.pdftexcmds.mdfivesum("\luaescapestring{#1}", "byte")%
80   }%
81 }%
82 \def\pdf@system#1{%
83   \directlua{%
84     oberdiek.pdftexcmds.system("\luaescapestring{#1}")%
85   }%
86 }
87 \let\pdf@primitive\primitive

```

Lua function `oberdiek.pdftexcmds.filesize` requires the `kpse` library, which is not loaded in `ConTeXt`, see github.com/latex3/lua-uni-algos/issues/3, so we define our own `filesize` function.

```

88 \directlua{%
89   function memoize.filesize(filename)
90     local filehandle = io.open(filename, "r")

```

We can't easily use `~=`, as `~` is an active character, so the `else` workaround.

```

91     if filehandle == nil then
92       else
93         tex.write(filehandle:seek("end"))
94         io.close(filehandle)
95       end
96     end

```

```

97 }%
98 \def\pdf@filesize#1{%
99   \directlua{memoize.filesize("\luaescapestring{#1}")}%
100 }
101 </context>

```

Take care of some further differences between the engines.

```

102 \ifdef\pdftexversion{%
103 }{%
104   \def\pdfhorigin{1true in}%
105   \def\pdfvorigin{1true in}%
106   \ifdef\XeTeXversion{%
107     \let\quitvmode\leavevmode
108   }{%
109     \ifdef\luatexversion{%
110       \let\pdfpagewidth\pagewidth
111       \let\pdfpageheight\pageheight
112       \def\pdfmajorversion{\pdfvariable majorversion}%
113       \def\pdfminorversion{\pdfvariable minorversion}%
114     }{%
115       \PackageError{memoize}{Support for this TeX engine is not implemented}{}%
116     }%
117   }%
118 }
119 </mmz>

```

In ConT_EXt, `\unexpanded` means `\protected`, and the usual `\unexpanded` is available as `\normalunexpanded`. Option one: use dtx guards to produce the correct control sequence. I tried this option. I find it ugly, and I keep forgetting to guard. Option two: `\let` an internal control sequence, like `\mmz@unexpanded`, to the correct thing, and use that all the time. I never tried this, but I find it ugly, too, and I guess I would forget to use the new control sequence, anyway. Option three: use `\unexpanded` in the `.dtx`, and `sed` through the generated ConT_EXt files to replace all its occurrences by `\normalunexpanded`. Oh yeah!

Load `pgfkeys` in `nomemoize` and `memoizable`. Not necessary in `memoize`, as it is already loaded by `CollArgs`.

```

120 <*nommz, mmzable>
121 <latex>\RequirePackage{pgfkeys}
122 <plain>\input pgfkeys
123 <context>\input t-pgfkey
124 </nommz, mmzable>

```

Different formats of `memoizable` merely load `memoizable.code.tex`, which exists so that `memoizable` can be easily loaded by generic code, like a `tikz` library.

```

125 <mmzable&!generic>\input memoizable.code.tex

```

Shipout We will next load our own auxiliary package, `CollArgs`, but before we do that, we need to grab `\shipout` in plain T_EX. The problem is, `Memoize` needs to hack into the `shipout` routine, but it has best chances of working as intended if it redefines the *primitive* `\shipout`. However, `CollArgs` loads `pgfkeys`, which in turn (and perhaps with no for reason) loads `atbegshi`, which redefines `\shipout`. For details, see section 3.6. Below, we first check that the current meaning of `\shipout` is primitive, and then redefine it.

```

126 <*mmz>
127 <*plain>
128 \def\mmz@regular@shipout{%
129   \global\advance\mmzRegularPages1\relax
130   \mmz@primitive@shipout
131 }
132 \edef\mmz@temp{\string\shipout}%

```

```

133 \edef\mmz@tempa{\meaning\shipout}%
134 \ifx\mmz@temp\mmz@tempa
135 \let\mmz@primitive@shipout\shipout
136 \let\shipout\mmz@regular@shipout
137 \else
138 \PackageError{memoize}{Cannot grab \string\shipout, it is already redefined}{}%
139 \fi
140 </plain>

```

Our auxiliary package (^M§5.6.3, §8.2). We also need it in `nomemoize`, to collect manual environments.

```

141 <latex>\RequirePackage{advice}
142 <plain>\input advice
143 <context>\input t-advice
144 </mmz>

```

Loading order `memoize` and `nomemoize` are mutually exclusive, and `memoizable` must be loaded before either of them. `\mmz@loadstatus`: 1 = `memoize`, 2 = `memoizable`, 3 = `nomemoize`.

```

145 <*mmz, nommz>
146 \def\ifmmz@loadstatus#1{%
147 \ifnum#1=0\csname mmz@loadstatus\endcsname\relax
148 \expandafter\@firstoftwo
149 \else
150 \expandafter\@secondoftwo
151 \fi
152 }
153 </mmz, nommz>
154 <*mmz>
155 \ifmmz@loadstatus{3}{%
156 \PackageError{memoize}{Cannot load the package, as "nomemoize" is already
157 loaded. Memoization will NOT be in effect}{Packages "memoize" and
158 "nomemoize" are mutually exclusive, please load either one or the other.}%
159 <latex> \pgfkeys{/memoize/package options/.unknown/.code={}}
160 <latex> \ProcessPgfPackageOptions{/memoize/package options}
161 \endinput
162 }{}%
163 \ifmmz@loadstatus{2}{%
164 \PackageError{memoize}{Cannot load the package, as "memoizable" is already
165 loaded}{Package "memoizable" is loaded by packages which support
166 memoization. Memoize must be loaded before all such packages. The
167 compilation log can help you figure out which package loaded "memoizable";
168 please move
169 <latex> "\string\usepackage{memoize}"
170 <plain> "\string\input memoize"
171 <context> "\string\usemodule[memoize]"
172 before the
173 <latex> "\string\usepackage"
174 <plain> "\string\input"
175 <context> "\string\usemodule"
176 of that package.}%
177 <latex> \pgfkeys{/memoize/package options/.unknown/.code={}}
178 <latex> \ProcessPgfPackageOptions{/memoize/package options}
179 \endinput
180 }{}%
181 \ifmmz@loadstatus{1}{\endinput}{}%
182 \def\mmz@loadstatus{1}%
183 </mmz>
184 <*mmzable & generic>
185 \ifcsname mmz@loadstatus\endcsname\endinput\fi
186 \def\mmz@loadstatus{2}%
187 </mmzable & generic>

```

```

188 (*nommz)
189 \ifmmz@loadstatus{1}{%
190   \PackageError{nomemoize}{Cannot load the package, as "memoize" is already
191     loaded; memoization will remain in effect}{Packages "memoize" and
192     "nomemoize" are mutually exclusive, please load either one or the other.}%
193   \endinput }{}%
194 \ifmmz@loadstatus{2}{%
195   \PackageError{nomemoize}{Cannot load the package, as "memoizable" is already
196     loaded}{Package "memoizable" is loaded by packages which support
197     memoization. (No)Memoize must be loaded before all such packages. The
198     compilation log can help you figure out which package loaded
199     "memoizable"; please move
200 \latex}{"\string\usepackage{nomemoize}"
201 \plain}{"\string\input memoize"
202 \context}{"\string\usemodule[memoize]"
203   before the
204 \latex}{"\string\usepackage"
205 \plain}{"\string\input"
206 \context}{"\string\usemodule"
207   of that package.}%
208   \endinput
209 }{}%
210 \ifmmz@loadstatus{3}{\endinput}{}%
211 \def\mmz@loadstatus{3}%
212 </nommz>

```

```
213 (*mmz)
```

`\filetotoks` Read TeX file #2 into token register #1 (under the current category code regime); `\toksapp` is defined in CollArgs.

```

214 \def\filetotoks#1#2{%
215   \immediate\openin0{#2}%
216   #1={}%
217   \loop
218     \unless\ifeof0
219     \read0 to \totoks@temp

```

We need the `\expandafters` for our `\toksapp` macro.

```

220   \expandafter\toksapp\expandafter#1\expandafter{\totoks@temp}%
221   \repeat
222   \immediate\closein0
223 }

```

Other little things.

```

224 \newif\ifmmz@temp
225 \newtoks\mmz@temptoks
226 \newbox\mmz@box
227 \newwrite\mmz@out

```

2 The basic configuration

`\mmzset` The user primarily interacts with Memoize through the `pgfkeys`-based configuration macro `\mmzset`, which executes keys in path `/mmz`. In `nomemoize` and `memoizable`, it exists as a no-op.

```

228 \def\mmzset#1{\pgfkeys{/mmz}{#1}\ignorespaces}
229 </mmz>
230 (*nommz, mmzable & generic)
231 \def\mmzset#1{\ignorespaces}
232 </nommz, mmzable & generic>

```

`\nommzkeys` Any `/mmz` keys used outside of `\mmzset` must be declared by this macro for `nomemoize` package to work.

```
233 <mmz>\def\nommzkeys#1{  
234  <*nommz, mmzable & generic>  
235  \def\nommzkeys{\pgfqkeys{/mmz}}  
236  \pgfqkeys{/mmz}{.unknown/.code={\pgfkeysdef{\pgfkeyscurrentkey}{}}}  
237  </nommz, mmzable & generic>
```

enable These keys set \TeX -style conditional `\ifmemoize`, used as the central on/off switch for the functionality of the package — it is inspected in `\Memoize` and by run conditions of automemoization `\ifmemoize` handlers.

If used in the preamble, the effect of these keys is delayed until the beginning of the document. The delay is implemented through a special style, `begindocument`, which is executed at `begindocument` hook in \LaTeX ; in other formats, the user must invoke it manually (^M§5.1).

`Nomemoize` does not need the keys themselves, but it does need the underlying conditional — which will be always false.

```
238 <*mmz, nommz, mmzable & generic>  
239 \newif\ifmemoize  
240 </mmz, nommz, mmzable & generic>  
241 <*mmz>  
242 \mmzset{%  
243  enable/.style={begindocument/.append code=\memoizetrue},  
244  disable/.style={begindocument/.append code=\memoizefalse},  
245  begindocument/.append style={  
246    enable/.code=\memoizetrue,  
247    disable/.code=\memoizefalse,  
248  }},
```

`Memoize` is enabled at the beginning of the document, unless explicitly disabled by the user in the preamble.

```
249  enable,
```

options Execute the given value as a keylist of `Memoize` settings.

```
250  options/.style={#1},  
251 }
```

normal When `Memoize` is enabled, it can be in one of three modes (^M§2.4): **normal**, **readonly**, and **recompile**. The numeric constants are defined below. The mode is stored in `\mmz@mode`, and only **recompile** matters in `\Memoize` (and `\mmz@process@ccmemo`).¹

```
252 \def\mmz@mode@normal{0}  
253 \def\mmz@mode@readonly{1}  
254 \def\mmz@mode@recompile{2}  
255 \let\mmz@mode\mmz@mode@normal  
256 \mmzset{%  
257  normal/.code={\let\mmz@mode\mmz@mode@normal},  
258  readonly/.code={\let\mmz@mode\mmz@mode@readonly},  
259  recompile/.code={\let\mmz@mode\mmz@mode@recompile},  
260 }
```

prefix Key `prefix` determines the location of memo and extern files (`\mmz@prefix@dir`) and the first, fixed part of their basename (`\mmz@prefix@name`).

```
261 \mmzset{%  
262  prefix/.code={\mmz@parse@prefix{#1}},  
263 }
```

¹In fact, this code treats anything but 1 and 2 as normal.

`\mmz@split@prefix` This macro stores the detokenized expansion of #1 into `\mmz@prefix`, which it then splits into `\mmz@prefix@dir` and `\mmz@prefix@name` at the final /. The slash goes into `\mmz@prefix@dir`. If there is no slash, `\mmz@prefix@dir` is empty; in particular, it is empty under no memo dir.

```

264 \begingroup
265 \catcode`\/=12
266 \gdef\mmz@parse@prefix#1{%
267   \edef\mmz@prefix{\detokenize\expandafter{\expanded{#1}}}%
268   \def\mmz@prefix@dir{}%
269   \def\mmz@prefix@name{}%
270   \expandafter\mmz@parse@prefix@i\mmz@prefix/\mmz@eov
271 }
272 \gdef\mmz@parse@prefix@i#1/#2{%
273   \ifx\mmzeov#2%
274     \def\mmz@prefix@name{#1}%
275   \else
276     \appto\mmz@prefix@dir{#1/}%
277     \expandafter\mmz@parse@prefix@i\expandafter#2%
278   \fi
279 }
280 \endgroup

```

Key `prefix` concludes by performing two actions: it creates the given directory if `mkdir` is in effect, and notes the new prefix in record files (by eventually executing `record/prefix`, which typically puts a `\mmzPrefix` line in the `.mmz` file). In the preamble, only the final setting of `prefix` matters, so this key is only equipped with the action-triggering code at the beginning of the document.

```

281 \mmzset{%
282   begindocument/.append style={
283     prefix/.append code=\mmz@maybe@mkmemodir\mmz@record@prefix,
284   },

```

Consequently, the post-prefix-setting actions must be triggered manually at the beginning of the document. Below, we trigger directory creation; `record/prefix` will be called from `record/begin`, which is executed at the beginning of the document, so it shouldn't be mentioned here.

```

285   begindocument/.append code=\mmz@maybe@mkmemodir,
286 }

```

`mkdir` Should we create the memo/extern directory if it doesn't exist? And which command should we use to create it? Initially, we attempt to create this directory, and we attempt to do this via `memoize-extract.pl --mkdir`. The roundabout way of setting the initial value of `mkdir command` allows `extract=python` to change the initial value to `memoize-extract.py --mkdir` only in the case the user did not modify it.

```

287 \def\mmz@initial@mkdir@command{\mmzvalueof{perl extraction command} --mkdir}
288 \mmzset{

```

This conditional is perhaps a useless leftover from the early versions, but we let it be.

```

289   mkdir/.is if=mmz@mkdir,
290   mkdir command/.store in=\mmz@mkdir@command,
291   mkdir command/.expand once=\mmz@initial@mkdir@command,
292 }

```

The underlying conditional `\ifmmz@mkdir` is only ever used in `\mmz@maybe@mkmemodir` below, which is itself only executed at the end of `prefix` and in `begindocument`.

```

293 \newif\ifmmz@mkdir
294 \mmz@mkdirtrue

```

We only attempt to create the memo directory if `\ifmmz@mkdir` is in effect and if both `\mmz@mkdir@command` and `\mmz@prefix@dir` are specified (i.e. non-empty). In particular, no attempt to create it will be made when no memo dir is in effect.

```

295 \def\mmz@maybe@mkmemodir{%
296   \ifmmz@mkdir
297     \ifdefempty\mmz@mkdir@command{}{%
298       \ifdefempty\mmz@prefix@dir{}{%
299         \mmz@remove@quotes{\mmz@prefix@dir}\mmz@temp
300         \pdf@system{\mmz@mkdir@command\space"\mmz@temp"}%
301       }%
302     }%
303   \fi
304 }

```

memo dir Shortcuts for two handy settings of `prefix`. Key `no memo dir` will place the memos and externs `no memo dir` in the current directory, prefixed with `#1.`, where `#1` defaults to (unquoted) `\jobname`. The default `memo dir` places the memos and externs in a dedicated directory, `#1.memo.dir`; the filenames themselves have no prefix.

```

305 \mmzset{%
306   memo dir/.style={prefix={#1.memo.dir}},
307   memo dir/.default=\jobname,
308   no memo dir/.style={prefix={#1.}},
309   no memo dir/.default=\jobname,
310   memo dir,
311 }

```

`\mmz@remove@quotes` This macro removes fully expands `#1`, detokenizes the expansion and then removes all double quotes the string. The result is stored in the control sequence given in `#2`.

We use this macro when we are passing a filename constructed from `\jobname` to external programs.

```

312 \def\mmz@remove@quotes#1#2{%
313   \def\mmz@remove@quotes@end{\let#2\mmz@temp}%
314   \def\mmz@temp{}%
315   \expanded{%
316     \noexpand\mmz@remove@quotes@i
317     \detokenize\expandafter{\expanded{#1}}%
318     "\noexpand\mmz@eov
319   }%
320 }
321 \def\mmz@remove@quotes@i{%
322   \CollectArgumentsRaw
323   {\collargsReturnPlain
324     \collargsNoDelimiterstrue
325     \collargsAppendExpandablePostprocessor{\the\collargsArg}}%
326   }%
327   {u"u\mmz@eov}%
328   \mmz@remove@quotes@ii
329 }
330 \def\mmz@remove@quotes@ii#1#2{%
331   \appto\mmz@temp{#1}%
332   \ifx&#2&%
333     \mmz@remove@quotes@end
334     \expandafter\@gobble
335   \else
336     \expandafter\@firstofone
337   \fi
338   {\mmz@remove@quotes@i#2\mmz@eov}%
339 }

```

ignore spaces The underlying conditional will be inspected by automemoization handlers, to maybe put `\ignorespaces` after the invocation of the handler.

```
340 \newif\ifmmz@ignorespaces
341 \mmzset{
342   ignore spaces/.is if=mmz@ignorespaces,
343 }
```

verbatim These keys are tricky. For one, there's **verbatim**, which sets all characters' category codes to **verb** other, and there's **verb**, which leaves braces untouched (well, honestly, it redefines them). But **no verbatim** Memoize itself doesn't really care about this detail — it only uses the underlying conditional `\ifmmz@verbatim`. It is `CollArgs` which cares about the difference between the “long” and the “short” **verbatim**, so we need to tell it about it. That's why the **verbatim** options “append themselves” to `\mmzRawCollectorOptions`, which is later passed on to `\CollectArgumentsRaw` as a part of its optional argument.

```
344 \newif\ifmmz@verbatim
345 \def\mmzRawCollectorOptions{
346   \mmzset{
347     verbatim/.code={%
348       \def\mmzRawCollectorOptions{\collargsVerbatim}%
349       \mmz@verbatimtrue
350     },
351     verb/.code={%
352       \def\mmzRawCollectorOptions{\collargsVerb}%
353       \mmz@verbatimtrue
354     },
355     no verbatim/.code={%
356       \def\mmzRawCollectorOptions{\collargsNoVerbatim}%
357       \mmz@verbatimfalse
358     },
359 }
```

3 Memoization

3.1 Manual memoization

`\mmz` The core of this macro will be a simple invocation of `\Memoize`, but to get there, we have to collect the optional argument carefully, because we might have to collect the memoized code **verbatim**.

```
360 \protected\def\mmz{\futurelet\mmz@temp\mmz@i}
361 \def\mmz@i{%
```

Anyone who wants to call `\Memoize` must open a group, because `\Memoize` will close a group.

```
362   \begingroup
```

As the optional argument occurs after a control sequence (`\mmz`), any spaces were consumed and we can immediately test for the opening bracket.

```
363   \ifx\mmz@temp[%
364     \def\mmz@verbatim@fix{}%
365     \expandafter\mmz@ii
366   \else
```

If there was no optional argument, the opening brace (or the unlikely single token) of our mandatory argument is already tokenized. If we are requested to memoize in a **verbatim** mode, this non-verbatim tokenization was wrong, so we will use option `\collargsFixFromNoVerbatim` to ask `CollArgs` to fix the situation. (`\mmz@verbatim@fix` will only be used in the **verbatim** mode.)

```
367     \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
```

No optional argument, so we can skip `\mmz@ii`.

```
368 \expandafter\mmz@iii
369 \fi
370 }
371 \def\mmz@ii[#1]{%
```

Apply the options given in the optional argument.

```
372 \mmzset{#1}%
373 \mmz@iii
374 }
375 \def\mmz@iii{%
```

In the non-verbatim mode, we avoid collecting the single mandatory argument using `\CollectArguments`.

```
376 \ifmmz@verbatim
377 \expandafter\mmz@do@verbatim
378 \else
379 \expandafter\mmz@do
380 \fi
381 }
```

This macro grabs the mandatory argument of `\mmz` and calls `\Memoize`.

```
382 \long\def\mmz@do#1{%
383 \Memoize{#1}{#1}%
384 }%
```

The following macro uses `\CollectArgumentsRaw` of package `CollArgs` (§8.2) to grab the argument verbatim; the appropriate verbatim mode triggering raw option was put in `\mmzRawCollectorOptions` by key `verb(atim)`. The macro also `\mmz@verbatim@fix` contains the potential request for a category code fix (§8.2.6).

```
385 \def\mmz@do@verbatim#1{%
386 \expanded{%
387 \noexpand\CollectArgumentsRaw{%
388 \noexpand\collargsCaller{\noexpand\mmz}%
389 \expandonce\mmzRawCollectorOptions
390 \mmz@verbatim@fix
391 }%
392 }{+m}\mmz@do
393 }
```

`memoize` (*env.*) The definition of the manual memoization environment proceeds along the same lines as the definition of `\mmz`, except that we also have to implement space-trimming, and that we will collect the environment using `\CollectArguments` in both the verbatim and the non-verbatim and mode.

We define the \LaTeX , plain \TeX and \ConTeXt environments in parallel. The definition of the plain \TeX and \ConTeXt version is complicated by the fact that space-trimming is affected by the presence vs. absence of the optional argument (for purposes of space-trimming, it counts as present even if it is empty).

```
394 <*\latex>
```

We define the \LaTeX environment using `\newenvironment`, which kindly grabs any spaces in front of the optional argument, if it exists — and if doesn't, we want to trim spaces at the beginning of the environment body anyway.

```
395 \newenvironment{memoize}[1][\mmz@noarg]{%
```

We close the environment right away. We'll collect the environment body, complete with the end-tag, so we have to reintroduce the end-tag somewhere. Another place would be after the invocation of `\Memoize`, but that would put memoization into a double group and `\mmzAfterMemoization` would not work.

```
396 \end{memoize}%
```

We open the group which will be closed by `\Memoize`.

```
397 \begin{group}
```

As with `\mmz` above, if there was no optional argument, we have to ask Collargs for a fix. The difference is that, as we have collected the optional argument via `\newcommand`, we have to test for its presence in a roundabout way.

```
398 \def\mmz@temp{#1}%
399 \ifx\mmz@temp\mmz@noarg
400 \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
401 \else
402 \def\mmz@verbatim@fix{}%
403 \mmzset{#1}%
404 \fi
405 \mmz@env@iii
406 }{}
407 \def\mmz@noarg{\mmz@noarg}
408 </latex>
409 <plain>\def\memoize{%
410 <context>\def\startmemoize{%
411 <*plain, context>
412 \begin{group}
```

In plain `TeX` and `ConTeXt`, we don't have to worry about any spaces in front of the optional argument, as the environments are opened by a control sequence.

```
413 \futurelet\mmz@temp\mmz@env@i
414 }
415 \def\mmz@env@i{%
416 \ifx\mmz@temp[%]
417 \def\mmz@verbatim@fix{}%
418 \expandafter\mmz@env@ii
419 \else
420 \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
421 \expandafter\mmz@env@iii
422 \fi
423 }
424 \def\mmz@env@ii[#1]{%
425 \mmzset{#1}%
426 \mmz@env@iii
427 }
428 </plain, context>
429 \def\mmz@env@iii{%
430 \long\edef\mmz@do##1{%
```

`\unskip` will “trim” spaces at the end of the environment body.

```
431 \noexpand\Memoize{##1}{##1\unskip}%
432 }%
433 \expanded{%
434 \noexpand\CollectArgumentsRaw{%
```

`\CollectArgumentsRaw` will adapt the caller to the format automatically.

```
435 \noexpand\collargsCaller{memoize}%
```

verb(atim) is in here if it was requested.

```
436 \expandonce\mmzRawCollectorOptions
```

The category code fix, if needed.

```
437 \ifmmz@verbatim\mmz@verbatim@fix\fi
438 }%
```

Spaces at the beginning of the environment body are trimmed by setting the first argument to !t<space> and disappearing it with \collargsAppendExpandablePostprocessor{}; note that this removes any number of space tokens. \CollectArgumentsRaw automatically adapts the argument type b to the format.

```
439 }{&&{\collargsAppendExpandablePostprocessor{}}!t{ }+b{memoize}}{\mmz@do}%
440 }%
441 </mmz>
```

\nommz We throw away the optional argument if present, and replace the opening brace with begin-group plus \memoizefalse. This way, the “argument” of \nommz will be processed in a group (with Memoize disabled) and even the verbatim code will work because the “argument” will not have been tokenized.

As a user command, \nommz has to make it into package nomemoize as well, and we’ll \let \mmz equal it there; it is not needed in mmzable.

```
442 (*mmz, nommz)
443 \protected\def\nommz#1#{%
444 \afterassignment\nommz@i
445 \let\mmz@temp
446 }
447 \def\nommz@i{%
448 \bgroup
449 \memoizefalse
450 }
451 <nommz>\let\mmz\nommz
```

nomemoize (env.) We throw away the optional argument and take care of the spaces at the beginning and at the end of the body.

```
452 (*latex)
453 \newenvironment{nomemoize}[1][ ]{%
454 \memoizefalse
455 \ignorespaces
456 }{%
457 \unskip
458 }
459 </latex>
460 (*plain, context)
461 <plain>\def\nomemoize{%
462 <context>\def\startnomemoize{%
```

Start a group to delimit \memoizefalse.

```
463 \begingroup
464 \memoizefalse
465 \futurelet\mmz@temp\nommz@env@i
466 }
467 \def\nommz@env@i{%
468 \ifx\mmz@temp[%]
469 \expandafter\nommz@env@ii
```

No optional argument, no problems with spaces.

```
470 \fi
471 }
472 \def\nommz@env@ii[#1]{%
473 \ignorespaces
474 }
475 <plain>\def\endnomemoize{%
476 <context>\def\stopnomemoize{%
477 \endgroup
478 \unskip
479 }
480 </plain,context>
481 <*nommz>
482 <plain,latex>\let\memoize\nomemoize
483 <plain,latex>\let\endmemoize\endnomemoize
484 <context>\let\startmemoize\startnomemoize
485 <context>\let\stopmemoize\stopnomemoize
486 </nommz>
487 </mmz,nommz>
```

3.2 The memoization process

`\ifmemoizing` This conditional is set to true when we start memoization (but not when we start regular compilation or utilization); it should never be set anywhere else. It is checked by `\Memoize` to prevent nested memoizations, deployed in advice run conditions set by `run only if memoizing`, etc.

```
488 <*mmz,nommz,mmzable & generic>
489 \newif\ifmemoizing
```

`\ifinmemoize` This conditional is set to true when we start either memoization or regular compilation (but not when we start utilization); it should never be set anywhere else. It is deployed in the default advice run conditions, making sure that automemoized commands are not handled even when we're regularly compiling some code submitted to memoization.

```
490 \newif\ifinmemoize
```

`\mmz@maybe@scantokens` An auxiliary macro which rescans the given code using `\scantokens` if the verbatim mode is active. We also need it in `NoMemoize`, to properly grab verbatim manually memoized code.

```
491 </mmz,nommz,mmzable & generic>
492 <*mmz>
493 \def\mmz@maybe@scantokens{%
494 \ifmmz@verbatim
495 \expandafter\mmz@scantokens
496 \else
497 \expandafter\@firstofone
498 \fi
499 }
```

Without `\newlinechar=13`, `\scantokens` would see receive the entire argument as one long line — but it would not *see* the entire argument, but only up to the first newline character, effectively losing most of the tokens. (We need to manually save and restore `\newlinechar` because we don't want to execute the memoized code in yet another group.)

```
500 \long\def\mmz@scantokens#1{%
501 \expanded{%
502 \newlinechar=13
503 \unexpanded{\scantokens{#1\endinput}}%
504 \newlinechar=\the\newlinechar
505 }%
506 }
```

`\Memoize` Memoization is invoked by executing `\Memoize`. This macro is a decision hub. It test for the existence of the memos and externs associated with the memoized code, and takes the appropriate action (memoization: `\mmz@memoize`; regular compilation: `\mmz@compile`, utilization: `\mmz@process@cmemo` plus `\mmz@process@ccmemo` plus further complications) depending on the memoization mode (normal, readonly, recompile). Note that one should open a \TeX group prior to executing `\Memoize`, because `\Memoize` will close a group (^M§4.1).

`\Memoize` takes two arguments, which contain two potentially different versions of the code submitted to memoization: `#1` contains the code which $\langle code MD5 sum \rangle$ is computed off of, while `#2` contains the code which is actually executed during memoization and regular compilation. The arguments will contain the same code in the case of manual memoization, but they will differ in the case of automemoization, where the executable code will typically prefixed by `\AdviceOriginal`. As the two codes will be used not only by `\Memoize` but also by macros called from `\Memoize`, `\Memoize` stores them into dedicated toks registers, declared below.

```
507 \newtoks\mmz@mdfive@source
508 \newtoks\mmz@exec@source
```

Finally, the definition of the macro. In package `NoMemoize`, we should simply execute the code in the second argument. But in `Memoize`, we have work to do.

```
509 \let\Memoize\@secondoftwo
510 \long\def\Memoize#1#2{%
```

We store the first argument into token register `\mmz@mdfive@source` because we might have to include it in tracing info (when `trace` is in effect), or paste it into the c-memo (depending on `include source in cmemo`).

```
511 \mmz@mdfive@source{#1}%
```

We store the executable code in `\mmz@exec@source`. In the verbatim mode, the code will have to be rescanned. This is implemented by `\mmz@maybe@scantokens`, and we wrap the code into this macro right away, once and for all. Even more, we pre-expand `\mmz@maybe@scantokens` (three times), effectively applying the current `\ifmmz@verbatim` and eliminating the need to save and restore this conditional in `\mmz@compile`, which (regularly) compiles the code *after* closing the `\Memoize` group — after this pre-expansion, `\mmz@exec@source` will contain either `\mmz@scantokens{...}` or `\@firstofone{...}`.

```
512 \expandafter\expandafter\expandafter\expandafter
513 \expandafter\expandafter\expandafter
514 \mmz@exec@source
515 \expandafter\expandafter\expandafter\expandafter
516 \expandafter\expandafter\expandafter
517 {%
518 \mmz@maybe@scantokens{#2}%
519 }%
520 \mmz@trace@Memoize
```

In most branches below, we end up with regular compilation, so let this be the default action.

```
521 \let\mmz@action\mmz@compile
```

If `Memoize` is disabled, or if memoization is currently taking place, we will perform a regular compilation.

```
522 \ifmemoizing
523 \else
524 \ifmemoize
```

Compute $\langle code md5sum \rangle$ off of the salted source code, and globally store it into `\mmz@code@mdfivesum` — globally, because we need it in utilization to include externs, but the `\Memoize` group is closed (by `\mmzMemo`) while inputting the cc-memo.


```

525     \xdef\mmz@code@mdfivesum{\pdf@mdfivesum{%
526         \expanded{\the\mmzSalt}%
527         \the\mmz@mdfive@source
528     }}%
529     \mmz@trace@code@mdfive

```

Recompile mode forces memoization.

```

530     \ifnum\mmz@mode=\mmz@mode@recompile\relax
531         \ifnum\pdf@draftmode=0
532             \let\mmz@action\mmz@memoize
533         \fi
534     \else

```

In the normal and the readonly mode, we try to utilize the memos. The c-memo comes first. If the c-memo does not exist (or if something is wrong with it), `\mmz@process@cmemo` (defined in §3.4) will set `\ifmmz@abort` to true. It might also set `\ifmmzUnmemoizable` which means we should compile normally regardless of the mode.

```

535         \mmz@process@cmemo
536         \ifmmzUnmemoizable
537             \mmz@trace@cmemo@unmemoizable
538         \else
539             \ifmmz@abort

```

If there is no c-memo, or it is invalid, we memoize, unless the read-only mode is in effect.

```

540         \mmz@trace@process@cmemo@fail
541         \ifnum\mmz@mode=\mmz@mode@readonly\relax
542         \else
543             \ifnum\pdf@draftmode=0
544                 \let\mmz@action\mmz@memoize
545             \fi
546         \fi
547     \else
548         \mmz@trace@process@cmemo@ok

```

If the c-memo was fine, the formal action decided upon is to try utilizing the cc-memo. If it exists and everything is fine with it, `\mmz@process@ccmemo` (defined in section 3.5) will utilize it, i.e. the core of the cc-memo (the part following `\mmzMemo`) will be executed (typically including the single `\extern`). Otherwise, `\mmz@process@ccmemo` will trigger either memoization (in the normal mode) or regular compilation (in the readonly mode). This final decision is left to `\mmz@process@ccmemo` because if we made it here, the code would get complicated, as the cc-memo must be processed outside the `\Memoize` group and all the conditionals in this macro.

```

549         \let\mmz@action\mmz@process@ccmemo
550     \fi
551 \fi
552 \fi
553 \fi
554 \fi
555 \mmz@action
556 }

```

`\mmz@compile` This macro performs regular compilation — this is signalled to the memoized code and the memoization driver by setting `\ifinmemoize` to true for the duration of the compilation; `\ifmemoizing` is not touched. The group opened prior to the invocation of `\Memoize` is closed before executing the code in `\mmz@exec@source`, so that compiling the code has the same local effect as if was not submitted to memoization; it is closing this group early which complicates the restoration of `\ifinmemoize` at the end of compilation. Note that `\mmz@exec@source` is already set to properly deal with the current verbatim mode, so any further inspection of `\ifmmz@verbatim` is

unnecessary; the same goes for `\ifmmz@ignorespaces`, which was (or at least should be) taken care of by whoever called `\Memoize`.

```

557 \def\mmz@compile{%
558   \mmz@trace@compile
559   \expanded{%
560     \endgroup
561     \noexpand\inmemoizetrue
562     \the\mmz@exec@source
563     \ifinmemoize\noexpand\inmemoizetrue\else\noexpand\inmemoizefalse\fi
564   }%
565 }

```

`abortOnError` In Lua_T_EX, we can check whether an error occurred during memoization, and abort if it `\mmz@lua@atbeginmemoization` did. (We're going through `memoize.abort`, because `tex.print` does not seem to `\mmz@lua@atendmemoization` work during error handling.) We omit all this in Con_T_EXt, as it appears to stop on any error?

```

566 <!*context>
567 \ifdefined\luatexversion
568   \directlua{%
569     luatexbase.add_to_callback(
570       "show_error_message",
571       function()
572         memoize.abort = true
573         texio.write_nl(status.lasterrorstring)
574       end,
575       "Abort memoization on error"
576     )
577   }%
578   \def\mmz@lua@atbeginmemoization{%
579     \directlua{memoize.abort = false}%
580   }%
581   \def\mmz@lua@atendmemoization{%
582     \directlua{%
583       if memoize.abort then
584         tex.print("\noexpand\\mmzAbort")
585       end
586     }%
587   }%
588 \else
589 </!*context>
590   \let\mmz@lua@atbeginmemoization\relax
591   \let\mmz@lua@atendmemoization\relax
592 <!context> \fi

```

`\mmz@memoize` This macro performs memoization — this is signalled to the memoized code and the memoization driver by setting both `\ifinmemoize` and `\ifinmemoizing` to true.

```

593 \def\mmz@memoize{%
594   \mmz@trace@memoize
595   \memoizingtrue
596   \inmemoizetrue

```

Initialize the various macros and registers used in memoization (to be described below, or later). Note that most of these are global, as they might be adjusted arbitrarily deep within the memoized code.

```

597 \edef\memoizinggrouplevel{\the\currentgrouplevel}%
598 \global\mmz@abortfalse
599 \global\mmzUnmemoizablefalse
600 \global\mmz@seq 0
601 \global\setbox\mmz@tbe@box\vbox{}%

```

```

602 \global\mmz@ccmemo@resources{}%
603 \global\mmzCMemo{}%
604 \global\mmzCCMemo{}%
605 \global\mmzContextExtra{}%
606 \gdef\mmzAtEndMemoizationExtra{}%
607 \gdef\mmzAfterMemoizationExtra{}%
608 \mmz@lua@atbeginmemoization

```

Execute the pre-memoization hook, the memoized code (wrapped in the driver), and the post-memoization hook.

```

609 \mmzAtBeginMemoization
610 \mmzDriver{\the\mmz@exec@source}%
611 \mmzAtEndMemoization
612 \mmzAtEndMemoizationExtra
613 \mmz@lua@atendmemoization
614 \ifmmzUnmemoizable

```

To permanently prevent memoization, we have to write down the c-memo (containing `\mmzUnmemoizabletrue`). We don't need the extra context in this case.

```

615 \global\mmzContextExtra{}%
616 \gtoksapp\mmzCCMemo{\global\mmzUnmemoizabletrue}%
617 \mmz@write@cmemo
618 \mmz@trace@endmemoize@unmemoizable
619 \PackageInfo{memoize}{Marking this code as unmemoizable}%
620 \else
621 \ifmmz@abort

```

If memoization was aborted, we create an empty c-memo, to make sure that no leftover c-memo tricks Memoize into thinking that the code was successfully memoized.

```

622 \mmz@trace@endmemoize@aborted
623 \PackageInfo{memoize}{Memoization was aborted}%
624 \mmz@compute@context@mdfivesum
625 \mmz@write@cmemo
626 \else

```

If memoization was not aborted, we compute the *context md5sum*, open and write out the memos, and shipout the externs (as pages into the document).

```

627 \mmz@compute@context@mdfivesum
628 \mmz@write@cmemo
629 \mmz@write@ccmemo
630 \mmz@shipout@externs
631 \mmz@trace@endmemoize@ok
632 \fi
633 \fi

```

After closing the group, we execute the final, after-memoization hook (we pre-expand the regular macro; the extra macro was assigned to globally). In the after-memoization code, `\mmzIncludeExtern` points to a macro which can include the extern from `\mmz@tbe@box`, which makes it possible to typeset the extern by dropping the contents of `\mmzCCMemo` into this hook — but note that this will only work if `\ifmmzkeepexterns` was in effect at the end of memoization.

```

634 \expandafter\endgroup
635 \expandafter\let
636 \expandafter\mmzIncludeExtern\expandafter\mmz@include@extern@from@tbe@box
637 \mmzAfterMemoization
638 \mmzAfterMemoizationExtra
639 }

```

`\memoizinggrouplevel` This macro stores the group level at the beginning of memoization. It is deployed by `\IfMemoizing`, normally used by integrated drivers.

```
640 \def\memoizinggrouplevel{-1}%
```

`\mmzAbort` Memoized code may execute this macro to abort memoization.

```
641 \def\mmzAbort{\global\mmz@aborttrue}
```

`\ifmmz@abort` This conditional serves as a signal that something went wrong during memoization (where it is set to true by `\mmzAbort`), or c(c)-memo processing. The assignment to this conditional should always be global (because it may be set during memoization).

```
642 \newif\ifmmz@abort
```

`\mmzUnmemoizable` Memoized code may execute `\mmzUnmemoizable` to abort memoization and mark (in the c-memo) that memoization should never be attempted again. The c-memo is composed by `\mmz@memoize`.

```
643 \def\mmzUnmemoizable{\global\mmzUnmemoizabletrue}
```

`\ifmmzUnmemoizable` This conditional serves as a signal that the code should never be memoized. It can be set (a) during memoization (that's why it should be assigned globally), after which it is inspected by `\mmz@memoize`, and (b) from the c-memo, in which case it is inspected by `\Memoize`.

```
644 \newif\ifmmzUnmemoizable
```

`\mmzAtBeginMemoization` The memoization hooks and their keys. The hook macros may be set either before or during memoization. In the former case, one should modify the primary `\mmzAfterMemoization` macro (`\mmzAtBeginMemoization`, `\mmzAtEndMemoization`, `\mmzAfterMemoization`), `at begin memoization` and the assignment should be local. In the latter case, one should modify the `at end memoization` tra macro (`\mmzAtEndMemoizationExtra`, `\mmzAfterMemoizationExtra`; there is no `after memoization` `\mmzAtBeginMemoizationExtra`), and the assignment should be global. The keys automatically adapt to the situation, by appending either to the primary or the the extra macro; if `at begin memoization` is used during memoization, the given code is executed immediately. We will use this “extra” approach and the auto-adapting keys for other options, like `context`, as well.

```
645 \def\mmzAtBeginMemoization{}
646 \def\mmzAtEndMemoization{}
647 \def\mmzAfterMemoization{}
648 \mmzset{
649   at begin memoization/.code={%
650     \ifmemoizing
651       \expandafter\@firstofone
652     \else
653       \expandafter\appto\expandafter\mmzAtBeginMemoization
654     \fi
655     {#1}%
656   },
657   at end memoization/.code={%
658     \ifmemoizing
659       \expandafter\gappto\expandafter\mmzAtEndMemoizationExtra
660     \else
661       \expandafter\appto\expandafter\mmzAtEndMemoization
662     \fi
663     {#1}%
664   },
665   after memoization/.code={%
666     \ifmemoizing
667       \expandafter\gappto\expandafter\mmzAfterMemoizationExtra
668     \else
```

```

669     \expandafter\appto\expandafter\mmzAfterMemoization
670     \fi
671     {#1}%
672   },
673 }

```

driver This key sets the (formal) memoization driver. The function of the driver is to produce the memos and externs while executing the submitted code.

```

674 \mmzset{
675   driver/.store in=\mmzDriver,
676   driver=\mmzSingleExternDriver,
677 }

```

\ifmmzkeepexterns This conditional causes Memoize not to empty out `\mmz@tbe@box`, holding the externs collected during memoization, while shipping them out.

```

678 \newif\ifmmzkeepexterns

```

\mmzSingleExternDriver The default memoization driver externalizes the submitted code. It always produces exactly one extern, and including the extern will be the only effect of inputting the cc-memo (unless the memoized code contained some commands, like `\label`, which added extra instructions to the cc-memo.) The macro (i) adds `\quitvmode` to the cc-memo, if we're capturing into a horizontal box, and it puts it to the very front, so that it comes before any `\label` and `\index` replications, guaranteeing (hopefully) that they refer to the correct page; (ii) takes the code and typesets it in a box (`\mmz@box`); (iii) submits the box for externalization; (iv) adds the extern-inclusion code to the cc-memo, and (v) puts the box into the document (again prefixing it with `\quitvmode` if necessary). (The listing region markers help us present this code in the manual.)

```

679 \long\def\mmzSingleExternDriver#1{%
680   \xtoksapp\mmzCCMemo{\mmz@maybe@quitvmode}%
681   \setbox\mmz@box\mmz@capture{#1}%
682   \mmzExternalizeBox\mmz@box\mmz@temptoks
683   \xtoksapp\mmzCCMemo{\the\mmz@temptoks}%
684   \mmz@maybe@quitvmode\box\mmz@box
685 }

```

capture The default memoization driver uses `\mmz@capture` and `\mmz@maybe@quitvmode`, which are set by this key. `\mmz@maybe@quitvmode` will be expanded, but for \TeX , we have defined `\quitvmode` as a synonym for `\leavevmode`, which is a macro rather than a primitive, so we have to prevent its expansion in that case. It is easiest to just add `\noexpand`, regardless of the engine used.

```

686 \mmzset{
687   capture/.is choice,
688   capture/hbox/.code={%
689     \let\mmz@capture\hbox
690     \def\mmz@maybe@quitvmode{\noexpand\quitvmode}%
691   },
692   capture/vbox/.code={%
693     \let\mmz@capture\vbox
694     \def\mmz@maybe@quitvmode{}}%
695   },
696   capture=hbox,
697 }

```

The memoized code may be memoization-aware; in such a case, we say that the driver is *integrated* into the code. Code containing an integrated driver must take care to execute it only when memoizing, and not during a regular compilation. The following key and macro can help here, see [M§4.4.4](#) for details.

integrated driver This is an advice key, residing in `/mmz/auto`. Given $\langle suffix \rangle$ as the only argument, it declares conditional `\ifmemoizing $\langle suffix \rangle$` , and sets the driver for the automemoized command to a macro which sets this conditional to true. The declared conditional is *internal* and should not be used directly, but only via `\IfMemoizing` — because it will not be declared when package `NoMemoize` or only `Memoizable` is loaded.

```
698 \mmzset{
699   auto/integrated driver/.style={
700     after setup={\expandafter\newif\csname ifmmz@memoizing#1\endcsname},
701     driver/.expand once={%
702       \csname mmz@memoizing#1true\endcsname
```

Without this, we would introduce an extra group around the memoized code.

```
703     \@firstofone
704   }%
705 },
706 }
```

\IfMemoizing Without the optional argument, the condition is satisfied when the internal conditional `\ifmemoizing $\langle suffix \rangle$` , declared by `integrated driver`, is true. With the optional argument $\langle offset \rangle$, the current group level must additionally match the memoizing group level, modulo $\langle offset \rangle$ — this makes sure that the conditional comes out as false in a regular compilation embedded in a memoization.

```
707 \newcommand\IfMemoizing[2][\mmz@Ifmemoizing@nogrouplevel]{%>\fi
708 \csname ifmmz@memoizing#2\endcsname%>\if
```

One `\relax` is for the `\numexpr`, another for `\ifnum`. Complications arise when `#1` is the optional argument default (defined below). In that case, the content of `\mmz@Ifmemoizing@nogrouplevel` closes off the `\ifnum` conditional (with both the true and the false branch empty), and opens up a new one, `\iftrue`. Effectively, we're not testing for the group level match.

```
709 \ifnum\currentgrouplevel=\the\numexpr\memoizinggrouplevel+#1\relax\relax
710   \expandafter\expandafter\expandafter\@firstoftwo
711 \else
712   \expandafter\expandafter\expandafter\@secondoftwo
713 \fi
714 \else
715   \expandafter\@secondoftwo
716 \fi
717 }
718 \def\mmz@Ifmemoizing@nogrouplevel{0\relax\relax\fi\iftrue}
```

Tracing We populate the hooks which send the tracing info to the terminal.

```
719 \def\mmz@trace#1{\advice@typeout{[tracing memoize] #1}}
720 \def\mmz@trace@context{\mmz@trace{\space\space
721   Context: "\expandonce{\mmz@context@key}" --> \mmz@context@mdfivesum}}
722 \def\mmz@trace@Memoize@on{%
723   \mmz@trace{%
724     Entering \noexpand\Memoize (%
725     \ifmemoize enabled\else disabled\fi,
726     \ifnum\mmz@mode=\mmz@mode@recompile recompile\fi
727     \ifnum\mmz@mode=\mmz@mode@readonly readonly\fi
728     \ifnum\mmz@mode=\mmz@mode@normal normal\fi
729     \space mode) on line \the\inputlineno
730   }%
731   \mmz@trace{\space\space Code: \the\mmz@mdfive@source}%
732 }
733 \def\mmz@trace@code@mdfive@on{\mmz@trace{\space\space
734   Code md5sum: \mmz@code@mdfivesum}}
```

```

735 \def\mmz@trace@compile@on{\mmz@trace{\space\space Compiling}}
736 \def\mmz@trace@memoize@on{\mmz@trace{\space\space Memoizing}}
737 \def\mmz@trace@endmemoize@ok@on{\mmz@trace{\space\space
738   Memoization completed}}%
739 \def\mmz@trace@endmemoize@aborted@on{\mmz@trace{\space\space
740   Memoization was aborted}}
741 \def\mmz@trace@endmemoize@unmemoizable@on{\mmz@trace{\space\space
742   Marking this code as unmemoizable}}

```

No need for `\mmz@trace@endmemoize@fail`, as abortion results in a package warning anyway.

```

743 \def\mmz@trace@process@cmemo@on{\mmz@trace{\space\space
744   Attempting to utilize c-memo \mmz@cmemo@path}}
745 \def\mmz@trace@process@no@cmemo@on{\mmz@trace{\space\space
746   C-memo does not exist}}
747 \def\mmz@trace@process@cmemo@ok@on{\mmz@trace{\space\space
748   C-memo was processed successfully}\mmz@trace@context}
749 \def\mmz@trace@process@cmemo@fail@on{\mmz@trace{\space\space
750   C-memo input failed}}
751 \def\mmz@trace@cmemo@unmemoizable@on{\mmz@trace{\space\space
752   This code was marked as unmemoizable}}
753 \def\mmz@trace@process@ccmemo@on{\mmz@trace{\space\space
754   Attempting to utilize cc-memo \mmz@ccmemo@path\space
755   (\ifmmz@direct@ccmemo@input\else in\fi direct input)}}
756 \def\mmz@trace@resource@on#1{\mmz@trace{\space\space
757   Extern file does not exist: #1}}
758 \def\mmz@trace@process@ccmemo@ok@on{%
759   \mmz@trace{\space\space Utilization successful}}
760 \def\mmz@trace@process@no@ccmemo@on{%
761   \mmz@trace{\space\space CC-memo does not exist}}
762 \def\mmz@trace@process@ccmemo@fail@on{%
763   \mmz@trace{\space\space Cc-memo input failed}}

```

`tracing` The user interface for switching the tracing on and off; initially, it is off. Note that there is no `\mmzTracingOn` underlying conditional. The off version simply `\lets` all the tracing hooks to `\relax`, so that `\mmzTracingOff` the overhead of having the tracing functionality available is negligible.

```

764 \mmzset{%
765   trace/.is choice,
766   trace/.default=true,
767   trace/true/.code=\mmzTracingOn,
768   trace/false/.code=\mmzTracingOff,
769 }
770 \def\mmzTracingOn{%
771   \let\mmz@trace@memoize\mmz@trace@Memoize@on
772   \let\mmz@trace@code@mdfive\mmz@trace@code@mdfive@on
773   \let\mmz@trace@compile\mmz@trace@compile@on
774   \let\mmz@trace@memoize\mmz@trace@memoize@on
775   \let\mmz@trace@process@cmemo\mmz@trace@process@cmemo@on
776   \let\mmz@trace@endmemoize@ok\mmz@trace@endmemoize@ok@on
777   \let\mmz@trace@endmemoize@unmemoizable\mmz@trace@endmemoize@unmemoizable@on
778   \let\mmz@trace@endmemoize@aborted\mmz@trace@endmemoize@aborted@on
779   \let\mmz@trace@process@cmemo\mmz@trace@process@cmemo@on
780   \let\mmz@trace@process@cmemo@ok\mmz@trace@process@cmemo@ok@on
781   \let\mmz@trace@process@no@cmemo\mmz@trace@process@no@cmemo@on
782   \let\mmz@trace@process@cmemo@fail\mmz@trace@process@cmemo@fail@on
783   \let\mmz@trace@cmemo@unmemoizable\mmz@trace@cmemo@unmemoizable@on
784   \let\mmz@trace@process@ccmemo\mmz@trace@process@ccmemo@on
785   \let\mmz@trace@resource\mmz@trace@resource@on
786   \let\mmz@trace@process@ccmemo@ok\mmz@trace@process@ccmemo@ok@on
787   \let\mmz@trace@process@no@ccmemo\mmz@trace@process@no@ccmemo@on
788   \let\mmz@trace@process@ccmemo@fail\mmz@trace@process@ccmemo@fail@on
789 }

```

```

790 \def\mmzTracingOff{%
791   \let\mmz@trace@Memoize\relax
792   \let\mmz@trace@code@mdfive\relax
793   \let\mmz@trace@compile\relax
794   \let\mmz@trace@memoize\relax
795   \let\mmz@trace@process@cmemo\relax
796   \let\mmz@trace@endmemoize@ok\relax
797   \let\mmz@trace@endmemoize@unmemoizable\relax
798   \let\mmz@trace@endmemoize@aborted\relax
799   \let\mmz@trace@process@cmemo\relax
800   \let\mmz@trace@process@cmemo@ok\relax
801   \let\mmz@trace@process@no@cmemo\relax
802   \let\mmz@trace@process@cmemo@fail\relax
803   \let\mmz@trace@cmemo@unmemoizable\relax
804   \let\mmz@trace@process@ccmemo\relax
805   \let\mmz@trace@resource\@gobble
806   \let\mmz@trace@process@ccmemo@ok\relax
807   \let\mmz@trace@process@no@ccmemo\relax
808   \let\mmz@trace@process@ccmemo@fail\relax
809 }
810 \mmzTracingOff

```

3.3 Context

`\mmzContext` The context expression is stored in two token registers. Outside memoization, we will locally `\mmzContextExtra` assign to `\mmzContext`; during memoization, we will globally assign to `\mmzContextExtra`.

```

811 \newtoks\mmzContext
812 \newtoks\mmzContextExtra

```

`context` The user interface keys for context manipulation hide the complexity underlying the context `clear context` storage from the user.

```

813 \mmzset{%
814   context/.code={%
815     \ifmemoizing
816       \expandafter\gtoksapp\expandafter\mmzContextExtra
817     \else
818       \expandafter\toksapp\expandafter\mmzContext
819     \fi

```

We append a comma to the given context chunk, for disambiguation.

```

820   {#1,}%
821 },
822 clear context/.code={%
823   \ifmemoizing
824     \expandafter\global\expandafter\mmzContextExtra
825   \else
826     \expandafter\mmzContext
827   \fi
828   {}%
829 },
830 clear context/.value forbidden,

```

`meaning to context` Utilities to put the meaning of various stuff into context.

`csname meaning to context`

`key meaning to context`

`key value to context`

`/handlers/.meaning to context`

`/handlers/.value to context`

```

831 meaning to context/.code={\mmz@Cos\forcsvlist\mmz@mtoc{#1}},
832 csname meaning to context/.code={\mmz@Cos\mmz@mtoc@csname{#1}},
833 key meaning to context/.code={\mmz@Cos
834   \forcsvlist\mmz@mtoc\mmz@mtoc@keycmd{#1}},
835 key value to context/.code={\mmz@Cos\forcsvlist\mmz@mtoc@key{#1}},
836 /handlers/.meaning to context/.code={\mmz@Cos\expanded%

```



```

837     \noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath/.@cmd}}},
838 /handlers/.value to context/.code={\mmz@Cos
839     \expanded{\noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath}}},
840 }

```

```

\mmzSalt Salt functions in the same way as context outside memoization, but it con-
salt tributes to the source code hash, i.e. \mmzSalt is expanded when computing
clear salt \mmz@code@mdfivesum in \Memoize. It was realized that it is needed for full
meaning to salt Beamer support, see GitHub issue #27.
csname meaning to salt
key meaning to salt 841 \newtoks\mmzSalt
key value to salt 842 \mmzset{
/handlers/.meaning to salt 843 salt/.code=\expandafter\toksapp\expandafter\mmzSalt{#1},
/handlers/.value to salt 844 clear salt/.value forbidden,
845 clear salt/.code=\mmzSalt},
846 meaning to salt/.code={\mmz@coS\forcsvlist\mmz@mtoc{#1}},
847 csname meaning to salt/.code={\mmz@coS\mmz@mtoc@csname{#1}},
848 key meaning to salt/.code={\mmz@coS
849     \forcsvlist\mmz@mtoc\mmz@mtoc@keycmd{#1}},
850 key value to salt/.code={\mmz@coS
851     \forcsvlist\mmz@mtoc@key{#1}},
852 /handlers/.meaning to salt/.code={\mmz@coS\expanded{
853     \noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath/.@cmd}}},
854 /handlers/.value to salt/.code={\mmz@coS
855     \expanded{\noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath}}},
856 }

```

The following macros are shared between context and salt, so they should be preceded by either `\mmz@Cos` (for context) or `\mmz@coS` (for salt).

```

857 \def\mmz@Cos{\def\mmz@context@or@salt{context}}
858 \def\mmz@coS{\def\mmz@context@or@salt{salt}}
859 \def\mmz@mtoc#1{%
860     \collargs@cs@cases{#1}%
861     {\mmz@mtoc@cmd{#1}}%
862     {\mmz@mtoc@error@notcsorenv{#1}}%
863     {%
864         \mmz@mtoc@csname{%
865 <context>         start%
866                 #1}%
867         \mmz@mtoc@csname{%
868 <latex, plain>     end%
869 <context>         stop%
870                 #1}%
871     }%
872 }
873 \def\mmz@mtoc@cmd#1{%
874     \begingroup
875     \escapechar=-1
876     \expandafter\endgroup
877     \expandafter\mmz@mtoc@csname\expandafter{\string#1}%
878 }
879 \def\mmz@mtoc@csname#1{%
880     \pgfkeysvalueof{/mmz/\mmz@context@or@salt/.@cmd}%
881     \detokenize{#1}%
882     \ifcsname#1\endcsname
883     ={\expandafter\meaning\csname#1\endcsname}%
884     \fi
885     \pgfeov
886 }
887 \def\mmz@mtoc@key#1{\mmz@mtoc@csname{pgfk@#1}}
888 \def\mmz@mtoc@keycmd#1{\mmz@mtoc@csname{pgfk@#1/.@cmd}}
889 \def\mmz@mtoc@error@notcsorenv#1{%

```

```

890 \PackageError{memoize}{'\detokenize{#1}' passed to key 'meaning to \mmz@context@or@salt'
891   is neither a command nor an environment}{}%
892 }

```

3.4 C-memos

The path to a c-memo consists of the path prefix, the MD5 sum of the memoized code, and suffix .memo.

```

893 \def\mmz@cmemo@path{\mmz@prefix\mmz@code@mdfivesum.memo}

```

`\mmzCMemo` The additional, free-form content of the c-memo is collected in this token register.

```

894 \newtoks\mmzCMemo

```

`include source in cmemo` Should we include the memoized code in the c-memo? By default, yes.

```

\ifmmz@include@source

```

```

895 \mmzset{%
896   include source in cmemo/.is if=mmz@include@source,
897 }
898 \newif\ifmmz@include@source
899 \mmz@include@sourcetrue

```

`\mmz@write@cmemo` This macro creates the c-memo from the contents of `\mmzContextExtra` and `\mmzCMemo`.

```

900 \def\mmz@write@cmemo{%

```

Open the file for writing.

```

901   \immediate\openout\mmz@out{\mmz@cmemo@path}%

```

The memo starts with the `\mmzMemo` marker (a signal that the memo is valid).

```

902   \immediate\write\mmz@out{\noexpand\mmzMemo}%

```

We store the content of `\mmzContextExtra` by writing out a command that will (globally) assign its content back into this register.

```

903   \immediate\write\mmz@out{%
904     \global\mmzContextExtra{\the\mmzContextExtra}\collargs@percentchar
905   }%

```

Write out the free-form part of the c-memo.

```

906   \immediate\write\mmz@out{\the\mmzCMemo\collargs@percentchar}%

```

When `include source in cmemo` is in effect, add the memoized code, hiding it behind the `\mmzSource` marker.

```

907   \ifmmz@include@source
908     \immediate\write\mmz@out{\noexpand\mmzSource}%
909     \immediate\write\mmz@out{\the\mmz@mdfive@source}%
910   \fi

```

Close the file.

```

911   \immediate\closeout\mmz@out

```

Record that we wrote a new c-memo.

```

912   \pgfkeysalso{/mmz/record/new cmemo={\mmz@cmemo@path}}%
913 }

```

`\mmzSource` The c-memo memoized code marker. This macro is synonymous with `\endinput`, so the source following it is ignored when inputting the c-memo.

```
914 \let\mmzSource\endinput
```

`\mmz@process@cmemo` This macro inputs the c-memo, which will update the context code, which we can then compute the MD5 sum of.

```
915 \def\mmz@process@cmemo{%
916   \mmz@trace@process@cmemo
```

`\ifmmz@abort` serves as a signal that the c-memo exists and is of correct form.

```
917 \global\mmz@aborttrue
```

If c-memo sets `\ifmmzUnmemoizable`, we will compile regularly.

```
918 \global\mmzUnmemoizablefalse
919 \def\mmzMemo{\global\mmz@abortfalse}%
```

Just a safeguard ... c-memo assigns to `\mmzContextExtra` anyway.

```
920 \global\mmzContextExtra{}%
```

Input the c-memo, if it exists, and record that we have used it.

```
921 \IfFileExists{\mmz@cmemo@path}{%
922   \input{\mmz@cmemo@path}%
923   \pgfkeysalso{/mmz/record/used cmemo={\mmz@cmemo@path}}%
924 }-%
925   \mmz@trace@process@no@cmemo
926 }%
```

Compute the context MD5 sum.

```
927 \mmz@compute@context@mdfivesum
928 }
```

`\mmz@compute@context@mdfivesum` This macro computes the MD5 sum of the concatenation of `\mmzContext` and `\mmzContextExtra`, and writes out the tracing info when `trace context` is in effect. The argument is the tracing note.

```
929 \def\mmz@compute@context@mdfivesum{%
930   \xdef\mmz@context@key{\the\mmzContext\the\mmzContextExtra}%
```

A special provision for padding, which occurs in the context by default, and may contain otherwise undefined macros referring to the extern dimensions. We make sure that when we expand the context key, `\mmz@paddings` contains the stringified `\width` etc., while these macros (which may be employed by the end user in the context expression), are returned to their original definitions.

```
931 \begingroup
932 \begingroup
933 \def\width{\string\width}%
934 \def\height{\string\height}%
935 \def\depth{\string\depth}%
936 \edef\mmz@paddings{\mmz@paddings}%
937 \expandafter\endgroup
938 \expandafter\def\expandafter\mmz@paddings\expandafter{\mmz@paddings}%
```

We pre-expand the concatenated context, for tracing/inclusion in the cc-memo. In \LaTeX , we protect the expansion, as the context expression may contain whatever.

```
939 \<latex> \protected\xdef
940 \<!latex> \xdef
941   \mmz@context@key{\mmz@context@key}%
942 \endgroup
```

Compute the MD5 sum. We have to assign globally, because this macro is (also) called after inputting the c-memo, while the resulting MD5 sum is used to input the cc-memo, which happens outside the `\Memoize` group. `\mmz@context@mdfivesum`.

```
943 \xdef\mmz@context@mdfivesum{\pdf@mdfivesum{\expandonce\mmz@context@key}}%
944 }
```

3.5 Cc-memos

The `path` to a cc-memo consists of the path prefix, the hyphen-separated MD5 sums of the memoized code and the (evaluated) context, and suffix `.memo`.

```
945 \def\mmz@ccmemo@path{%
946 \mmz@prefix\mmz@code@mdfivesum-\mmz@context@mdfivesum.memo}
```

The structure of a cc-memo:

- the list of resources consisting of calls to `\mmzResource`;
- the core memo code (which includes the externs when executed), introduced by marker `\mmzMemo`; and,
- optionally, the context expansion, introduced by marker `\mmzThisContext`.

We begin the cc-memo with a list of extern files included by the core memo code so that we can check whether these files exist prior to executing the core memo code. Checking this on the fly, while executing the core memo code, would be too late, as that code is arbitrary (and also executed outside the `\Memoize` group).

`\mmzCCMemo` During memoization, the core content of the cc-memo is collected into this token register.

```
947 \newtoks\mmzCCMemo
```

`include context in ccmemo` Should we include the context expansion in the cc-memo? By default, no.

```
\ifmmz@include@context
```

```
948 \newif\ifmmz@include@context
949 \mmzset{%
950 include context in ccmemo/.is if=mmz@include@context,
951 }
```

`direct ccmemo input` When this conditional is false, the cc-memo is read indirectly, via a token register, `\ifmmz@direct@ccmemo@input` to facilitate inverse search.

```
952 \newif\ifmmz@direct@ccmemo@input
953 \mmzset{%
954 direct ccmemo input/.is if=mmz@direct@ccmemo@input,
955 }
```

`\mmz@write@ccmemo` This macro creates the cc-memo from the list of resources in `\mmz@ccmemo@resources` and the contents of `\mmzCCMemo`.

```
956 \def\mmz@write@ccmemo{%
```

Open the cc-memo file for writing. Note that the filename contains the context MD5 sum, which can only be computed after memoization, as the memoized code can update the context. This is one of the two reasons why we couldn't write the cc-memo directly into the file, but had to collect its contents into token register `\mmzCCMemo`.

```
957 \immediate\openout\mmz@out{\mmz@ccmemo@path}%
```

Token register `\mmz@ccmemo@resources` consists of calls to `\mmz@ccmemo@append@resource`, so the following code writes down the list of created externs into the cc-memo. Wanting to have this list at the top of the cc-memo is the other reason for the roundabout creation of the cc-memo — the resources become known only during memoization, as well.

```

958 \begingroup
959 \the\mmz@ccmemo@resources
960 \endgroup

```

Write down the content of `\mmzMemo`, but first introduce it by the `\mmzMemo` marker.

```

961 \immediate\write\mmz@out{\noexpand\mmzMemo}%
962 \immediate\write\mmz@out{\the\mmzCCMemo\collargs@percentchar}%

```

Write down the context tracing info when `include context in ccmemo` is in effect.

```

963 \ifmmz@include@context
964   \immediate\write\mmz@out{\noexpand\mmzThisContext}%
965   \immediate\write\mmz@out{\expandonce{\mmz@context@key}}}%
966 \fi

```

Insert the end-of-file marker and close the file.

```

967 \immediate\write\mmz@out{\noexpand\mmzEndMemo}%
968 \immediate\closeout\mmz@out

```

Record that we wrote a new cc-memo.

```

969 \pgfkeysalso{/mmz/record/new ccmemo={\mmz@ccmemo@path}}%
970 }

```

`\mmz@ccmemo@append@resource` Append the resource to the cc-memo (we are nice to external utilities and put each resource on its own line). #1 is the sequential number of the extern belonging to the memoized code; below, we assign it to `\mmz@seq`, which appears in `\mmz@extern@name`. Note that `\mmz@extern@name` only contains the extern filename — without the path, so that externs can be used by several projects, or copied around.

```

971 \def\mmz@ccmemo@append@resource#1{%
972   \mmz@seq=#1\relax
973   \immediate\write\mmz@out{%
974     \string\mmzResource{\mmz@extern@name}\collargs@percentchar}%
975 }

```

`\mmzResource` A list of these macros is located at the top of a cc-memo. The macro checks for the existence of the extern file, given as #1. If the extern does not exist, we redefine `\mmzMemo` to `\endinput`, so that the core content of the cc-memo is never executed; see also `\mmz@process@ccmemo` above.

```

976 \def\mmzResource#1{%

```

We check for existence using `\pdffilesize`, because an empty PDF, which might be produced by a failed \TeX -based extraction, should count as no file. The 0 behind `\ifnum` is there because `\pdffilesize` returns an empty string when the file does not exist.

```

977   \ifnum0\pdf@filesize{\mmz@prefix@dir#1}=0
978     \ifmmz@direct@ccmemo@input
979       \let\mmzMemo\endinput
980     \else

```

With indirect cc-memo input, we simulate end-of-input by grabbing everything up to the end-of-memo marker. In the indirect cc-memo input, a `\par` token shows up after `\mmzEndMemo`, I'm not sure why (`\everyeof={}` does not help).

```

981     \long\def\mmzMemo##1\mmzEndMemo\par{%
982     \fi
983     \mmz@trace@resource{#1}%
984     \fi

```

Map the extern sequential number to the path to the extern. We need this so that `prefix` can be changed via `\mmznext`. The problem is that when a cc-memo is utilized, `\mmzMemo` closes the Memoize group; the `prefix` is thereby forgotten. But at the point of execution of `\mmzResource`, the `prefix` setting is still known and can be baked into `\mmz@resource@<seq>`, which will be inspected by `\mmzIncludeExtern`.

```
985 \csxdef{mmz@resource@\the\mmz@seq}{\mmz@prefix@dir#1}%
986 \global\advance\mmz@seq1
987 }
```

`\mmz@process@ccmemo` This macro processes the cc-memo.

```
\mmzThisContext
\mmzEndMemo 988 \def\mmz@process@ccmemo{%
989 \mmz@trace@process@ccmemo
```

The following conditional signals whether cc-memo was successfully utilized. If the cc-memo file does not exist, `\ifmmz@abort` will remain true. If it exists, it is headed by the list of resources. If a resource check fails, `\mmzMemo` (which follows the list of resources) is redefined to `\endinput`, so `\ifmmz@abort` remains true. However, if all resource checks are successful, `\mmzMemo` marker is reached with the below definition in effect, so `\ifmmz@abort` becomes false. Note that this marker also closes the `\Memoize` group, so that the core cc-memo content is executed in the original group — and that this does not happen if anything goes wrong!

```
990 \global\mmz@aborttrue
```

Note that `\mmzMemo` may be redefined by `\mmzResource` upon an unavailable extern file.

```
991 \def\mmzMemo{%
992 \endgroup
993 \global\mmz@abortfalse
```

We `\let` the control sequence used for extern inclusion in the cc-memo to the macro which includes the extern from the extern file.

```
994 \let\mmzIncludeExtern\mmz@include@extern
995 }%
```

Define `\mmzEndMemo` wrt `\ifmmz@direct@ccmemo@input`, whose value will be lost soon because `\mmMemo` will close the group — that's also why this definition is global.

```
996 \xdef\mmzEndMemo{%
997 \ifmmz@direct@ccmemo@input
998 \noexpand\endinput
999 \else
```

In the indirect cc-memo input, a `\par` token shows up after `\mmzEndMemo`, I'm not sure why (`\everyeof={}` does not help).

```
1000 \unexpanded{%
1001 \def\mmz@temp\par}%
1002 \mmz@temp
1003 }%
1004 \fi
1005 }%
```

The cc-memo context marker, again wrt `\ifmmz@direct@ccmemo@input` and globally. With direct cc-memo input, this macro is synonymous with `\endinput`, so the (expanded) context following it is ignored when inputting the cc-memo. With indirect input, we simulate end-of-input by grabbing everything up to the end-of-memo marker (plus gobble the `\par` mentioned above).

```
1006 \xdef\mmzThisContext{%
1007 \ifmmz@direct@ccmemo@input
1008 \noexpand\endinput
1009 \else
```

```

1010     \unexpanded{%
1011         \long\def\mmz@temp##1\mmzEndMemo\par{}%
1012         \mmz@temp
1013     }%
1014     \fi
1015 }%

```

Reset `\mmz@seq` for `\mmzResource`.

```
1016 \global\mmz@seq=0
```

Input the cc-memo if it exists.

```

1017 \IfFileExists{\mmz@ccmemo@path}{%
1018     \ifmmz@direct@ccmemo@input
1019     \input{\mmz@ccmemo@path}%
1020     \else

```

Indirect cc-memo input reads the cc-memo into a token register and executes the contents of this register.

```

1021     \filetotoks\toks@{\mmz@ccmemo@path}%
1022     \the\toks@
1023     \fi

```

Record that we have used the cc-memo.

```

1024     \pgfkeysalso{/mmz/record/used ccmemo={\mmz@ccmemo@path}}%
1025 }{%
1026     \mmz@trace@process@no@ccmemo
1027 }%
1028 \ifmmz@abort

```

The cc-memo doesn't exist, or some of the resources don't. We need to memoize, but we'll do it only if `readonly` is not in effect, otherwise we'll perform a regular compilation. (Note that we are still in the group opened prior to executing `\Memoize`.)

```

1029     \mmz@trace@process@ccmemo@fail
1030     \ifnum\mmz@mode=\mmz@mode@readonly\relax
1031     \expandafter\expandafter\expandafter\mmz@compile
1032     \else
1033     \expandafter\expandafter\expandafter\mmz@memoize
1034     \fi
1035     \else
1036     \mmz@trace@process@ccmemo@ok
1037     \fi
1038 }

```

3.6 The externs

The [path](#) to an extern is like the path to a cc-memo, modulo suffix `.pdf`, of course. However, in case memoization of a chunk produces more than one extern, the filename of any non-first extern includes `\mmz@seq`, the sequential number of the extern as well (we start the numbering at 0). We will have need for several parts of the full path to an extern: the basename, the filename, the path without the suffix, and the full path.

```

1039 \newcount\mmz@seq
1040 \def\mmz@extern@basename{%
1041     \mmz@prefix@name\mmz@code@mdfivesum-\mmz@context@mdfivesum
1042     \ifnum\mmz@seq>0 -\the\mmz@seq\fi
1043 }
1044 \def\mmz@extern@name{\mmz@extern@basename.pdf}
1045 \def\mmz@extern@basepath{\mmz@prefix@dir\mmz@extern@basename}
1046 \def\mmz@extern@path{\mmz@extern@basepath.pdf}

```

`padding left` These options set the amount of space surrounding the bounding box of the externalized graphics in the resulting PDF, i.e. in the extern file. This allows the user to deal with TikZ overlays, `padding right` `\rlap` and `\llap`, etc.

```
padding top \rlap and \llap, etc.
padding bottom
1047 \mmzset{
1048 padding left/.store in=\mmz@padding@left,
1049 padding right/.store in=\mmz@padding@right,
1050 padding top/.store in=\mmz@padding@top,
1051 padding bottom/.store in=\mmz@padding@bottom,
```

`padding` A shortcut for setting all four paddings at once.

```
1052 padding/.style={
1053 padding left=#1, padding right=#1,
1054 padding top=#1, padding bottom=#1
1055 },
```

The default padding is what pdfTeX puts around the page anyway, 1 inch, but we'll use `1 in` rather than `1 true in`, which is the true default value of `\pdfhorigin` and `\pdfvorigin`, as we want the padding to adjust with magnification.

```
1056 padding=1in,
```

`padding to context` This key adds padding to the context. Note that we add the padding expression (`\mmz@paddings`, defined below, refers to all the individual padding macros), not the actual value (at the time of expansion). This is so because `\width`, `\height` and `\depth` are not defined outside extern shipout routines, and the context is evaluated elsewhere.

```
1057 padding to context/.style={
1058 context={padding=(\mmz@paddings)},
1059 },
```

Padding nearly always belongs into the context — the exception being memoized code which produces no externs (^M§4.4.2) — so we execute this key immediately.

```
1060 padding to context,
1061 }
1062 \def\mmz@paddings{%
1063 \mmz@padding@left,\mmz@padding@bottom,\mmz@padding@right,\mmz@padding@top
1064 }
```

`\mmzExternalizeBox` This macro is the public interface to externalization. In Memoize itself, it is called from the default memoization driver, `\mmzSingleExternDriver`, but it should be called by any driver that wishes to produce an extern, see ^M§4.4 for details. It takes two arguments:

- #1 The box that we want to externalize. It's content will remain intact. The box may be given either as a control sequence, declared via `\newbox`, or as box number (say, 0).
- #2 The token register which will receive the code that includes the extern into the document; it is the responsibility of the memoization driver to (globally) include the contents of the register in the cc-memo, i.e. in token register `\mmzCCMemo`. This argument may be either a control sequence, declared via `\newtoks`, or a `\toks` (*token register number*).

```
1065 \def\mmzExternalizeBox#1#2{%
1066 \begingroup
```

A courtesy to the user, so they can define padding in terms of the size of the externalized graphics.

```
1067 \def\width{\wd#1 }%
1068 \def\height{\ht#1 }%
1069 \def\depth{\dp#1 }%
```

Store the extern-inclusion code in a temporary macro, which will be smuggled out of the group.

```
1070 \xdef\mmz@global@temp{%
```


Executing `\mmzIncludeExtern` from the cc-memo will include the extern into the document.

```
1071 \noexpand\mmzIncludeExtern
```

`\mmzIncludeExtern` identifies the extern by its sequence number, `\mmz@seq`.

```
1072 {\the\mmz@seq}%
```

What kind of box? We `\noexpand` the answer just in case someone redefined them.

```
1073 \ifhbox#1\noexpand\hbox\else\noexpand\vbox\fi
```

The dimensions of the extern.

```
1074 {\the\wd#1}%
```

```
1075 {\the\ht#1}%
```

```
1076 {\the\dp#1}%
```

The padding values.

```
1077 {\the\dimexpr\mmz@padding@left}%
```

```
1078 {\the\dimexpr\mmz@padding@bottom}%
```

```
1079 {\the\dimexpr\mmz@padding@right}%
```

```
1080 {\the\dimexpr\mmz@padding@top}%
```

```
1081 }%
```

Prepend the new extern box into the global extern box where we collect all the externs of this memo. Note that we `\copy` the extern box, retaining its content — we will also want to place the extern box in its regular place in the document.

```
1082 \global\setbox\mmz@tbe@box\vbox{\copy#1\unvbox\mmz@tbe@box}%
```

Add the extern to the list of resources, which will be included at the top of the cc-memo, to check whether the extern files exists at the time the cc-memo is utilized. In the cc-memo, the list will contain full extern filenames, which are currently unknown, but no matter; right now, providing the extern sequence number suffices, the full extern filename will be produced at the end of memoization, once the context MD5 sum is known.

```
1083 \xtoksapp\mmz@ccmemo@resources{%
```

```
1084 \noexpand\mmz@ccmemo@append@resource{\the\mmz@seq}%
```

```
1085 }%
```

Increment the counter containing the sequence number of the extern within this memo.

```
1086 \global\advance\mmz@seq1
```

Assign the extern-including code into the token register given in #2. This register may be given either as a control sequence or as `\toks<token register number>`, and this is why we have temporarily stored the code (into `\mmz@global@temp`) globally: a local storage with `\expandafter\endgroup\expandafter` here would fail with the receiving token register given as `\toks<token register number>`.

```
1087 \endgroup
```

```
1088 #2\expandafter{\mmz@global@temp}%
```

```
1089 }
```

`\mmz@ccmemo@resources` This token register, populated by `\mmz@externalize@box` and used by `\mmz@write@ccmemo`, holds the list of externs produced by memoization of the current chunk.

```
1090 \newtoks\mmz@ccmemo@resources
```

`\mmz@tbe@box` `\mmz@externalize@box` does not directly dump the extern into the document (as a special page). Rather, the externs are collected into `\mmz@tbe@box`, whose contents are dumped into the document at the end of memoization of the current chunk. In this way, we guarantee that aborted memoization does not pollute the document.

```
1091 \newbox\mmz@tbe@box
```

`\mmz@shipout@externs` This macro is executed at the end of memoization, when the externs are waiting for us in `\mmz@tbe@box` and need to be dumped into the document. It loops through the contents of `\mmz@tbe@box`,² putting each extern into `\mmz@box` and calling `\mmz@shipout@extern`. Note that the latter macro is executed within the group opened by `\vbox` below.

```
1092 \def\mmz@shipout@externs{%
1093   \global\mmz@seq 0
1094   \setbox\mmz@box\vbox{%
```

Set the macros below to the dimensions of the extern box, so that the user can refer to them in the padding specification (which is in turn used in the page setup in `\mmz@shipout@extern`).

```
1095   \def\width{\wd\mmz@box}%
1096   \def\height{\ht\mmz@box}%
1097   \def\depth{\dp\mmz@box}%
1098   \vskip1pt
1099   \ifmmzkeepexterns\expandafter\unvcopy\else\expandafter\unvbox\fi\mmz@tbe@box
1100   \@whiles\ifdim\opt=\lastskip\fi{%
1101     \setbox\mmz@box\lastbox
1102     \mmz@shipout@extern
1103   }%
1104 }%
1105 }
```

`\mmz@shipout@extern` This macro ships out a single extern, which resides in `\mmz@box`, and records the creation of the new extern.

```
1106 \def\mmz@shipout@extern{%
```

Calculate the expected width and height. We have to do this now, before we potentially adjust the box size and paddings for magnification.

```
1107 \edef\expectedwidth{\the\dimexpr
1108   (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)}%
1109 \edef\expectedheight{\the\dimexpr
1110   (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box + (\mmz@padding@bottom)}%
```

Apply the inverse magnification, if `\mag` is not at the default value. We'll do this in a group, which will last until shipout.

```
1111 \begingroup
1112 \ifnum\mag=1000
1113 \else
1114   \mmz@shipout@mag
1115 \fi
```

Setup the geometry of the extern page. In plain \TeX and \LaTeX , setting `\pdfpagewidth` and `\pdfpageheight` seems to do the trick of setting the extern page dimensions. In \ConTeXt , however, the resulting extern page ends up with the PDF `/CropBox` specification of the current regular page, which is then used (ignoring our `mediabox` requirement) when we're including the extern into the document by `\mmzIncludeExtern`. Typically, this results in a page-sized extern. I'm not sure how to deal with this correctly. In the workaround below, we use Lua function `backends.codeinjections.setupcanvas` to set up page dimensions: we first

²The looping code is based on TeX.SE answer tex.stackexchange.com/a/25142/16819 by Bruno Le Floch.

remember the current page dimensions (`\edef\mmz@temp`), then set up the extern page dimensions (`\expanded{...}`), and finally, after shipping out the extern page, revert to the current page dimensions by executing `\mmz@temp` at the very end of this macro.

```

1116 <*plain, latex>
1117 \pdfpagewidth\dimexpr
1118 (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)\relax
1119 \pdfpageheight\dimexpr
1120 (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box+ (\mmz@padding@bottom)\relax
1121 </plain, latex>
1122 <*context>
1123 \edef\mmz@temp{%
1124   \noexpand\directlua{
1125     backends.codeinjections.setupcanvas({
1126       paperwidth=\the\numexpr\pagewidth,
1127       paperheight=\the\numexpr\pageheight
1128     })
1129   }%
1130 }%
1131 \expanded{%
1132   \noexpand\directlua{
1133     backends.codeinjections.setupcanvas({
1134       paperwidth=\the\numexpr\dimexpr
1135         \mmz@padding@left + \wd\mmz@box + \mmz@padding@right\relax,
1136       paperheight=\the\numexpr\dimexpr
1137         \mmz@padding@top + \ht\mmz@box + \dp\mmz@box+ \mmz@padding@bottom\relax
1138     })
1139   }%
1140 }%
1141 </context>
1142 \mmz@shipout@unrotate

```

We complete the page setup by setting the content offset.

```

1143 \hoffset\dimexpr\mmz@padding@left - \pdfhorigin\relax
1144 \voffset\dimexpr\mmz@padding@top - \pdfvorigin\relax

```

We shipout the extern page using the `\shipout` primitive, so that the extern page is not modified, or even registered, by the shipout code of the format or some package. I can't imagine those shipout routines ever needing to know about the extern page. In fact, most often knowing about it would be undesirable. For example, `LATEX` and `ConTEXt` count the “real” pages, but usually to know whether they are shipping out an odd or an even page, or to make the total number of pages available to subsequent compilations. Taking the extern pages into account would disrupt these mechanisms.

Another thing: delayed `\writes`. We have to make sure that any `LATEX`-style protected stuff in those is not expanded. We don't bother introducing a special group, as we'll close the `\mag` group right after the shipout anyway.

```

1145 <latex> \let\protect\noexpand
1146 \pdf@primitive\shipout\box\mmz@box
1147 <context> \mmz@temp
1148 \endgroup

```

Advance the counter of shipped-out externs. We do this before preparing the recording information below, because the extern extraction tools expect the extern page numbering to start with 1.

```

1149 \global\advance\mmzExternPages1

```

Prepare the macros which may be used in `record/<type>/new` extern code.

```

1150 \edef\externbasepath{\mmz@extern@basepath}%

```

Adding up the counters below should result in the real page number of the extern. Macro `\mmzRegularPages` holds the number of pages which were shipped out so far using the regular shipout routine of the format; `\mmzExternPages` holds the number of shipped-out extern pages; and `\mmzExtraPages` holds, or at least should hold, the number of pages shipped out using any other means.

```
1151 \edef\pagenumber{%
1152   \the\numexpr\mmzRegularPages
```

In L^AT_EX, the `\mmzRegularPages` holds to number of pages already shipped out. In ConT_EXt, the counter is already increased while processing the page, so we need to subtract 1.

```
1153 <context>   -1%
1154           +\mmzExternPages+\mmzExtraPages
1155           }%
```

Record the creation of the new extern. We do this after shipping out the extern page, so that the recording mechanism can serve as an after-shipout hook, for the unlikely situation that some package really needs to do something when our shipout happens. Note that we absolutely refuse to provide a before-shipout hook, because we can't allow anyone messing with our extern, and that using this after-shipout "hook" is unnecessary for counting extern shipouts, as we already provide this information in the public counter `\mmzExternPages`.

```
1156 \mmzset{record/new extern/.expanded=\mmz@extern@path}%
```

Advance the sequential number of the extern, in the context of the current memoized code chunk. This extern numbering starts at 0, so we only do this after we wrote the cc-memo and called `record/new extern`.

```
1157 \global\advance\mmz@seq1
1158 }
```

`\mmz@shipout@mag` This macro applies the inverse magnification, so that the extern ends up with its natural size on the extern page.

```
1159 \def\mmz@shipout@mag{%
```

We scale the extern box using the PDF primitives: `q` and `Q` save and restore the current graphics state; `cm` applies the given coordinate transformation matrix. ($a\ b\ c\ d\ e\ f\ cm$ transforms (x, y) into $(ax + cy + e, bx + dy + f)$.)

```
1160 \setbox\mmz@box\hbox{%
1161   \pdfliteral{q \mmz@inverse@mag\space 0 0 \mmz@inverse@mag\space 0 0 cm}%
1162   \copy\mmz@box\relax
1163   \pdfliteral{Q}%
1164   }%
```

We first have to scale the paddings, as they might refer to the `\width` etc. of the extern.

```
1165 \dimen0=\dimexpr\mmz@padding@left\relax
1166 \edef\mmz@padding@left{\the\dimexpr\mmz@inverse@mag\dimen0}%
1167 \dimen0=\dimexpr\mmz@padding@bottom\relax
1168 \edef\mmz@padding@bottom{\the\dimexpr\mmz@inverse@mag\dimen0}%
1169 \dimen0=\dimexpr\mmz@padding@right\relax
1170 \edef\mmz@padding@right{\the\dimexpr\mmz@inverse@mag\dimen0}%
1171 \dimen0=\dimexpr\mmz@padding@top\relax
1172 \edef\mmz@padding@top{\the\dimexpr\mmz@inverse@mag\dimen0}%
```

Scale the extern box.

```
1173 \wd\mmz@box=\mmz@inverse@mag\wd\mmz@box\relax
1174 \ht\mmz@box=\mmz@inverse@mag\ht\mmz@box\relax
1175 \dp\mmz@box=\mmz@inverse@mag\dp\mmz@box\relax
1176 }
```

`\mmz@inverse@mag` The inverse magnification factor, i.e. the number we have to multiply the extern dimensions with so that they will end up in their natural size. We compute it, once and for all, at the beginning of the document. To do that, we borrow the little macro `\Pgf@geT` from `pgfutil-common` (but rename it).

```

1177 {\catcode`\p=12\catcode`\t=12\gdef\mmz@Pgf@geT#1pt{#1}}
1178 \mmzset{begindocument/.append code={%
1179     \edef\mmz@inverse@mag{\expandafter\mmz@Pgf@geT\the\dimexpr 1000pt/\mag}%
1180 }}

```

`\mmz@shipout@unrotate` Remove any rotation attribute given to the PDF page. I don't know enough about PDF page attributes to be sure this works in general. I have tested the code on simple documents with \LaTeX and plain \TeX in \TeX live 2023 and 2024. It seems that the “unrotation” is unnecessary with \LaTeX 3's PDF management module (activated by `\DocumentMetadata{}` before the document class declaration).

```

1181 \ifdef\XeTeXversion{%
1182     \def\mmz@shipout@unrotate{%
1183     }{%
1184     \ifdef\luatexversion{%
1185         \def\mmz@shipout@unrotate{%
1186 <latex>         \IfPDFManagementActiveTF{ }\pdfvariable pageattr {/Rotate 0}}%
1187 <plain, context>         \pdfvariable pageattr {/Rotate 0}}%
1188         }%
1189     }{%
1190     \def\mmz@shipout@unrotate{%
1191 <latex>         \IfPDFManagementActiveTF{ }\pdfpageattr{/Rotate 0}}%
1192 <plain, context>         \pdfpageattr{/Rotate 0}}%
1193         }%
1194     }%
1195 }

```

`\mmzRegularPages` This counter holds the number of pages shipped out by the format's shipout routine. \LaTeX and \ConTeXt keep track of this in dedicated counters, so we simply use those. In plain \TeX , we have to hack the `\shipout` macro to install our own counter. In fact, we already did this while loading the required packages, in order to avoid it being redefined by `atbegshi` first. All that is left to do here is to declare the counter.

```

1196 <latex> \let\mmzRegularPages\ReadOnlyShipoutCounter
1197 <context> \let\mmzRegularPages\realpageno
1198 <plain> \newcount\mmzRegularPages

```

`\mmzExternPages` This counter holds the number of extern pages shipped out so far.

```

1199 \newcount\mmzExternPages

```

The total number of new externs is announced at the end of the compilation, so that \TeX editors, `latexmk` and such can propose recompilation.

```

1200 \mmzset{
1201     enddocument/afterlastpage/.append code={%
1202         \ifnum\mmzExternPages>0
1203             \PackageWarning{memoize}{The compilation produced \the\mmzExternPages\space
1204                 new extern\ifnum\mmzExternPages>1 s\fi}%
1205         \fi
1206     },
1207 }

```

`\mmzExtraPages` This counter will probably remain at zero forever. It should be advanced by any package which (like `Memoize`) ships out pages bypassing the regular shipout routine of the format.

```

1208 \newcount\mmzExtraPages

```

`\mmz@include@extern` This macro, called from cc-memos as `\mmzIncludeExtern`, inserts an extern file into the document. We first look up the path to the extern based on the extern sequential number; the dictionary was set up by `\mmzResource` statements in the cc-memo.

```

1209 \def\mmz@include@extern#1{%
1210   \expandafter\expandafter\expandafter\mmz@include@extern@i
1211   \expandafter\expandafter\expandafter{%
1212     \csname mmz@resource@#1\endcsname}%
1213 }

```

`#1` is the path to the extern filename, `#2` is either `\hbox` or `\vbox`, `#3`, `#4` and `#5` are the (expected) width, height and the depth of the externalized box; `#6–#9` are the paddings (left, bottom, right, and top).

```

1214 \def\mmz@include@extern@i#1#2#3#4#5#6#7#8#9{%

```

Use the primitive PDF graphics inclusion commands to include the extern file. Set the correct depth or the resulting box, and shift it as specified by the padding.

```

1215   \setbox\mmz@box=#2{%
1216     \setbox0=\hbox{%
1217       \lower\dimexpr #5+#7\relax\hbox{%
1218         \hskip -#6\relax
1219         \setbox0=\hbox{%
1220           \mmz@insertpdfpage{#1}{1}%
1221         }%
1222       \unhbox0
1223     }%
1224   }%
1225   \wd0 \dimexpr\wd0-#8\relax
1226   \ht0 \dimexpr\ht0-#9\relax
1227   \dp0 #5\relax
1228   \box0
1229 }%

```

Check whether the size of the included extern is as expected. There is no need to check `\dp`, we have just set it. (`\mmz@if@roughly@equal` is defined in section 4.3.)

```

1230   \mmz@tempfalse
1231   \mmz@if@roughly@equal{\mmz@tolerance}{#3}{\wd\mmz@box}{%
1232     \mmz@if@roughly@equal{\mmz@tolerance}{#4}{\ht\mmz@box}{%
1233       \mmz@temptrue
1234     }{}{}%
1235   \ifmmz@temp
1236   \else
1237     \mmz@use@memo@warning{\mmz@extern@path}{#3}{#4}{#5}%
1238   \fi

```

Use the extern box, with the precise size as remembered at memoization.

```

1239   \wd\mmz@box=#3\relax
1240   \ht\mmz@box=#4\relax
1241   \box\mmz@box

```

Record that we have used this extern.

```

1242   \pgfkeysalso{/mmz/record/used extern={\mmz@extern@path}}%
1243 }

```

```

1244 \def\mmz@use@memo@warning#1#2#3#4{%
1245   \PackageWarning{memoize}{Unexpected size of extern "#1";
1246     expected #2\space x \the\dimexpr #3+#4\relax,
1247     got \the\wd\mmz@box\space x \the\dimexpr\the\ht\mmz@box+\the\dp\mmz@box\relax}%
1248 }

```

`\mmz@insertpdfpage` This macro inserts a page from the PDF into the document. We define it according to which engine is being used. Note that ConT_EXt always uses LuaT_EX.

```

1249 <latex,plain>\ifdef\luatexversion{%
1250   \def\mmz@insertpdfpage#1#2{% #1 = filename, #2 = page number
1251     \saveimageresource page #2 mediabox {#1}%
1252     \useimageresource\lastsavedimageresourceindex
1253   }%
1254   <*/latex,plain>
1255 }{%
1256   \ifdef\XeTeXversion{%
1257     \def\mmz@insertpdfpage#1#2{%
1258       \XeTeXpdffile #1 page #2 media
1259     }%
1260   }{% pdfLaTeX
1261     \def\mmz@insertpdfpage#1#2{%
1262       \pdfximage page #2 mediabox {#1}%
1263       \pdfrefximage\pdflastximage
1264     }%
1265   }%
1266 }
1267 </latex,plain>

```

`\mmz@include@extern@from@tbe@box` Include the extern number #1 residing in `\mmz@tbe@box` into the document. It may be called as `\mmzIncludeExtern` from after memoization hook if `\ifmmzkeepexterns` was set to true during memoization. The macro takes the same arguments as `\mmzIncludeExtern` but disregards all but the first one, the extern sequential number. Using this macro, a complex memoization driver can process the cc-memo right after memoization, by issuing `\global\mmzkeepexternstrue\xtoksapp\mmzAfterMemoizationExtra{\the\mmzCCMemo}`.

```

1268 \def\mmz@include@extern@from@tbe@box#1#2#3#4#5#6#7#8#9{%
1269   \setbox0\vbox{%
1270     \@tempcnta#1\relax
1271     \vskip1pt
1272     \unvcopy\mmz@tbe@box
1273     \@whilenum\@tempcnta>0\do{%
1274       \setbox0\lastbox
1275       \advance\@tempcnta-1\relax
1276     }%
1277     \global\setbox1\lastbox
1278     \@whilesw\ifdim0pt=\lastskip\fi{%
1279       \setbox0\lastbox
1280     }%
1281     \box\mmz@box
1282   }%
1283   \box1
1284 }

```

4 Extraction

4.1 Extraction mode and method

extract This key selects the extraction mode and method. It normally occurs in the package options list, less commonly in the preamble, and never in the document body.

```

1285 \def\mmzvalueof#1{\pgfkeysvalueof{/mmz/#1}}
1286 \mmzset{
1287   extract/.estore in=\mmz@extraction@method,
1288   extract/.value required,
1289   begindocument/.append style={extract/.code=\mmz@preamble@only@error},

```

`extract/perl` Any other value will select internal extraction with the given method. Memoize ships with two extraction scripts, a Perl script and a Python script, which are selected by `extract=perl` (the default) and `extract=python`, respectively. We run the scripts in verbose mode (without `-q`), and keep the `.mmz` file as is (without `-k`), i.e. we're not commenting out the `\mmzNewExtern` lines, because we're about to overwrite it anyway. We inform the script about the format of the document (`-F`).

```

1290 extract/perl/.code={%
1291   \mmz@clear@extraction@log
1292   \pdf@system{%
1293     \mmzvalueof{perl extraction command}\space
1294     \mmzvalueof{perl extraction options}%
1295   }%
1296   \mmz@check@extraction@log{perl}%
1297 },
1298 perl extraction command/.initial=memoize-extract.pl,
1299 perl extraction options/.initial={\space
1300 <latex> -F latex
1301 <plain> -F plain
1302 <context> -F context
1303   \jobname\space
1304 },
1305 extract=perl,
1306 extract/python/.code={%
1307   \mmz@clear@extraction@log
1308   \pdf@system{%
1309     \mmzvalueof{python extraction command}\space
1310     \mmzvalueof{python extraction options}%
1311   }%
1312   \mmz@check@extraction@log{python}%

```

Change the initial value of `mkdir` command to `memoize-extract.py --mkdir`, but only in the case the user did not modify it.

```

1313   \ifx\mmz@mkdir@command\mmz@initial@mkdir@command
1314     \def\mmz@mkdir@command{\mmzvalueof{python extraction command} --mkdir}%
1315   \fi
1316 },
1317 python extraction command/.initial=memoize-extract.py,
1318 python extraction options/.initial={\space
1319 <latex> -F latex
1320 <plain> -F plain
1321 <context> -F context
1322   \jobname\space
1323 },
1324 }
1325 \def\mmz@preamble@only@error{%
1326   \PackageError{memoize}{%
1327     Ignoring the invocation of "\pgfkeyscurrentkey".
1328     This key may only be executed in the preamble}{}%
1329 }

```

The extraction log — As we cannot access the exit status of a system command in $\text{T}_{\text{E}}\text{X}$, we communicate with the system command via the “extraction log file,” produced by both $\text{T}_{\text{E}}\text{X}$ -based extraction and the Perl and Python extraction script. This file signals whether the embedded extraction was successful — if it is, the file ends if `\endinput` — and also contains any warnings and errors thrown by the script. As the log is really a $\text{T}_{\text{E}}\text{X}$ file, the idea is to simply input it after extracting each extern (for $\text{T}_{\text{E}}\text{X}$ -based extraction) or after the extraction of all externs (for the external scripts).

```

1330 \def\mmz@clear@extraction@log{%
1331   \begingroup

```



```

1332 \immediate\openout0{\jobname.mmz.log}%
1333 \immediate\closeout0
1334 \endgroup
1335 }

#1 is the extraction method.

1336 \def\mmz@check@extraction@log#1{%
1337 \begingroup \def\extractionmethod{#1}%
1338 \mmz@tempfalse \let\mmz@orig@endinput\endinput
1339 \def\endinput{\mmz@temptrue\mmz@orig@endinput}%
1340 \@input{\jobname.mmz.log}%
1341 \ifmmz@temp \else \mmz@extraction@error \fi \endgroup }
1342 \def\mmz@extraction@error{%
1343 \PackageError{memoize}{Extraction of externs from document
1344 "\jobname.pdf" using method "\extractionmethod" was
1345 unsuccessful}{The extraction script "\mmzvalueof{\extractionmethod\space
1346 extraction command}" wasn't executed or didn't finish execution
1347 properly.}}

```

4.2 The record files

record This key activates a record *<type>*: the hooks defined by that record *<type>* will henceforth be executed at the appropriate places.

A *<hook>* of a particular *<type>* resides in pgfkeys path `/mmz/record/<type>/<hook>`, and is invoked via `/mmz/record/<hook>`. Record type activation thus appends a call of the former to the latter. It does so using handler `.try`, so that unneeded hooks may be left undefined.

```

1348 \mmzset{
1349 record/.style={%
1350 record/begin/.append style={
1351 /mmz/record/#1/begin/.try,

```

The `begin` hook also executes the `prefix` hook, so that `\mmzPrefix` surely occurs at the top of the `.mmz` file. Listing each prefix type separately in this hook ensures that `prefix` of a certain type is executed after that type's `begin`.

```

1352 /mmz/record/#1/prefix/.try/.expanded=\mmz@prefix,
1353 },
1354 record/prefix/.append style={/mmz/record/#1/prefix/.try={##1}},
1355 record/new extern/.append style={/mmz/record/#1/new extern/.try={##1}},
1356 record/used extern/.append style={/mmz/record/#1/used extern/.try={##1}},
1357 record/new cmemo/.append style={/mmz/record/#1/new cmemo/.try={##1}},
1358 record/new ccmemo/.append style={/mmz/record/#1/new ccmemo/.try={##1}},
1359 record/used cmemo/.append style={/mmz/record/#1/used cmemo/.try={##1}},
1360 record/used ccmemo/.append style={/mmz/record/#1/used ccmemo/.try={##1}},
1361 record/end/.append style={/mmz/record/#1/end/.try},
1362 },
1363 }

```

no record This key deactivates all record types. Below, we use it to initialize the relevant keys; in the user code, it may be used to deactivate the preactivated `mmz` record type.

```

1364 \mmzset{
1365 no record/.style={%

```

The `begin` hook clears itself after invocation, to prevent double execution. Consequently, `record/begin` may be executed by the user in the preamble, without any ill effects.

```

1366 record/begin/.style={record/begin/.style={}},

```

The `prefix` key invokes itself again when the group closes. This way, we can correctly track the path prefix changes in the `.mmz` even if `path` is executed in a group.

```

1367 record/prefix/.code={\aftergroup\mmz@record@prefix},
1368 record/new extern/.code={},
1369 record/used extern/.code={},
1370 record/new cmemo/.code={},
1371 record/new ccmemo/.code={},
1372 record/used cmemo/.code={},
1373 record/used ccmemo/.code={},

```

The `end` hook clears itself after invocation, to prevent double execution. Consequently, `record/end` may be executed by the user before the end of the document, without any ill effects.

```

1374 record/end/.style={record/end/.code={}},
1375 }
1376 }

```

We define this macro because `\aftergroup`, used in `record/prefix`, only accepts a token.

```

1377 \def\mmz@record@prefix{%
1378 \mmzset{/mmz/record/prefix/.expanded=\mmz@prefix}%
1379 }

```

Initialize the hook keys, preactivate `mmz` record type, and execute hooks `begin` and `end` at the edges of the document.

```

1380 \mmzset{
1381 no record,
1382 record=mmz,
1383 begindocument/.append style={record/begin},
1384 enddocument/afterlastpage/.append style={record/end},
1385 }

```

4.2.1 The `.mmz` file

Think of the `.mmz` record file as a \TeX -readable log file, which lets the extraction procedure know what happened in the previous compilation. The file is in \TeX format, so that we can trigger internal \TeX -based extraction by simply inputting it. The commands it contains are intentionally as simple as possible (just a macro plus braced arguments), to facilitate parsing by the external scripts.

`record/mmz/...` These hooks simply put the calls of the corresponding macros into the file. All but hooks but `begin` and `end` receive the full path to the relevant file as the only argument (ok, `prefix` receives the full path prefix, as set by key `path`).

```

1386 \mmzset{
1387 record/mmz/begin/.code={%
1388 \newwrite\mmz@mmzout

```

The record file has a fixed name (the jobname plus the `.mmz` suffix) and location (the current directory, i.e. the directory where \TeX is executed from; usually, this will be the directory containing the \TeX source).

```

1389 \immediate\openout\mmz@mmzout{\jobname.mmz}%
1390 },

```

The `\mmzPrefix` is used by the clean-up script, which will remove all files with the given path prefix but (unless called with `--all`) those mentioned in the `.mmz`. Now this script could in principle figure out what to remove by inspecting the paths to utilized/created memos/externs in the `.mmz` file, but this method could lead to problems in case of an incomplete (perhaps empty) `.mmz` file created by a failed compilation. Recording the path prefix in the `.mmz` radically

increases the chances of a successful clean-up, which is doubly important, because a clean-up is sometimes precisely what we need to do to recover after a failed compilation.

```

1391 record/mMZ/prefix/.code={%
1392   \immediate\write\mmz@mmzout{\noexpand\mmzPrefix{#1}}%
1393 },
1394 record/mMZ/new extern/.code={%

```

While this key receives a single formal argument, Memoize also prepares macros `\externbasepath` (`#1` without the `.pdf` suffix), `\pagenumber` (of the extern page in the document PDF), and `\expectedwidth` and `\expectedheight` (of the extern page).

```

1395   \immediate\write\mmz@mmzout{%
1396     \noexpand\mmzNewExtern{#1}{\pagenumber}{\expectedwidth}{\expectedheight}%
1397   }%

```

Support latexmk:

```

1398 <latex> \typeout{No file #1}%
1399 },
1400 record/mMZ/new cmemo/.code={%
1401   \immediate\write\mmz@mmzout{\noexpand\mmzNewCMemo{#1}}%
1402 },
1403 record/mMZ/new ccmemo/.code={%
1404   \immediate\write\mmz@mmzout{\noexpand\mmzNewCCMemo{#1}}%
1405 },
1406 record/mMZ/used extern/.code={%
1407   \immediate\write\mmz@mmzout{\noexpand\mmzUsedExtern{#1}}%
1408 },
1409 record/mMZ/used cmemo/.code={%
1410   \immediate\write\mmz@mmzout{\noexpand\mmzUsedCMemo{#1}}%
1411 },
1412 record/mMZ/used ccmemo/.code={%
1413   \immediate\write\mmz@mmzout{\noexpand\mmzUsedCCMemo{#1}}%
1414 },
1415 record/mMZ/end/.code={%

```

Add the `\endinput` marker to signal that the file is complete.

```

1416   \immediate\write\mmz@mmzout{\noexpand\endinput}%
1417   \immediate\closeout\mmz@mmzout
1418 },

```

4.2.2 The shell scripts

We define two shell script record types: `sh` for Linux, and `bat` for Windows.

`sh` These keys set the shell script filenames.

`bat`

```

1419 sh/.store in=\mmz@shname,
1420 sh=memoize-extract.\jobname.sh,
1421 bat/.store in=\mmz@batname,
1422 bat=memoize-extract.\jobname.bat,

```

`record/sh/...` Define the Linux shell script record type.

```

1423 record/sh/begin/.code={%
1424   \newwrite\mmz@shout
1425   \immediate\openout\mmz@shout{\mmz@shname}%
1426 },
1427 record/sh/new extern/.code={%
1428   \begingroup

```

Macro `\mmz@tex@extraction@systemcall` is customizable through `tex extraction command`, `tex extraction options` and `tex extraction script`.

```

1429 \immediate\write\mmz@shout{\mmz@tex@extraction@systemcall}%
1430 \endgroup
1431 },
1432 record/sh/end/.code={%
1433 \immediate\closeout\mmz@shout
1434 },

```

`record/bat/...` Rinse and repeat for Windows.

```

1435 record/bat/begin/.code={%
1436 \newwrite\mmz@batout
1437 \immediate\openout\mmz@batout{\mmz@batname}%
1438 },
1439 record/bat/new extern/.code={%
1440 \begingroup
1441 \immediate\write\mmz@batout{\mmz@tex@extraction@systemcall}%
1442 \endgroup
1443 },
1444 record/bat/end/.code={%
1445 \immediate\closeout\mmz@batout
1446 },

```

4.2.3 The Makefile

The implementation of the Makefile record type is the most complex so far, as we need to keep track of the targets.

`makefile` This key sets the makefile filename.

```

1447 makefile/.store in=\mmz@makefilename,
1448 makefile=memoize-extract.\jobname.makefile,
1449 }

```

We need to define a macro which expands to the tab character of catcode “other”, to use as the recipe prefix.

```

1450 \begingroup
1451 \catcode\^^I=12
1452 \gdef\mmz@makefile@recipe@prefix{^^I}%
1453 \endgroup

```

`record/makefile/...` Define the Makefile record type.

```

1454 \mmzset{
1455 record/makefile/begin/.code={%

```

We initialize the record type by opening the file and setting makefile variables `.DEFAULT_GOAL` and `.PHONY`.

```

1456 \newwrite\mmz@makefileout
1457 \newtoks\mmz@makefile@externs
1458 \immediate\openout\mmz@makefileout{\mmz@makefilename}%
1459 \immediate\write\mmz@makefileout{.DEFAULT_GOAL = externs}%
1460 \immediate\write\mmz@makefileout{.PHONY: externs}%
1461 },

```

The crucial part, writing out the extraction rule. The target comes first, then the recipe, which is whatever the user has set by `tex extraction command`, `tex extraction options` and `tex extraction script`.

```

1462 record/makefile/new extern/.code={%

```

The target extern file:

```
1463 \immediate\write\mmz@makefileout{#1:}%  
1464 \begingroup
```

The recipe is whatever the user set by `tex extraction command`, `tex extraction options` and `tex extraction script`.

```
1465 \immediate\write\mmz@makefileout{%  
1466 \mmz@makefile@recipe@prefix\mmz@tex@extraction@systemcall}%  
1467 \endgroup
```

Append the extern file to list of targets.

```
1468 \xtoksapp\mmz@makefile@externs{#1\space}%  
1469 },  
1470 record/makefile/end/.code={%
```

Before closing the file, we list the extern files as the prerequisites of our phony default target, `externs`.

```
1471 \immediate\write\mmz@makefileout{externs: \the\mmz@makefile@externs}%  
1472 \immediate\closeout\mmz@makefileout  
1473 },  
1474 }
```

4.3 T_EX-based extraction

`extract/tex` We trigger the T_EX-based extraction by inputting the `.mmz` record file.

```
1475 \mmzset{  
1476 extract/tex/.code={%  
1477 \begingroup  
1478 \@input{\jobname.mmz}%  
1479 \endgroup  
1480 },  
1481 }
```

`\mmzUsedCMemo` We can ignore everything but `\mmzNewExterns`. All these macros receive a single argument.

```
\mmzUsedCCMemo 1482 \def\mmzUsedCMemo#1{}  
\mmzUsedExtern 1483 \def\mmzUsedCCMemo#1{}  
  \mmzNewCMemo 1484 \def\mmzUsedExtern#1{}  
\mmzNewCCMemo 1485 \def\mmzNewCMemo#1{}  
  \mmzPrefix 1486 \def\mmzNewCCMemo#1{}  
  1487 \def\mmzPrefix#1{}
```

`\mmzNewExtern` Command `\mmzNewExtern` takes four arguments. It instructs us to extract page `#2` of document `\jobname.pdf` to file `#1`. During the extraction, we will check whether the size of the extern matches the given expected width (`#3`) and total height (`#4`).

We perform the extraction by an embedded T_EX call. The system command that gets executed is stored in `\mmz@tex@extraction@systemcall`, which is set by `tex extraction command` and friends; by default, we execute `pdftex`.

```
1488 \def\mmzNewExtern#1{%
```

The T_EX executable expects the basename as the argument, so we strip away the `.pdf` suffix.

```
1489 \mmz@new@extern@i#1\mmz@temp  
1490 }  
1491 \def\mmz@new@extern@i#1.pdf\mmz@temp#2#3#4{%  
1492 \begingroup
```

Define the macros used in `\mmz@tex@extraction@systemcall`.

```
1493 \def\externbasepath{#1}%
1494 \def\pagenumber{#2}%
1495 \def\expectedwidth{#3}%
1496 \def\expectedheight{#4}%
```

Empty out the extraction log.

```
1497 \mmz@clear@extraction@log
```

Extract.

```
1498 \pdf@system{\mmz@tex@extraction@systemcall}%
```

Was the extraction successful? We temporarily redefine the extraction error message macro (suited for the external extraction scripts, which extract all externs in one go) to report the exact problematic extern page.

```
1499 \let\mmz@extraction@error\mmz@pageextraction@error
1500 \mmz@check@extraction@log{tex}%
1501 \endgroup
1502 }
```

```
1503 \def\mmz@pageextraction@error{%
1504 \PackageError{memoize}{Extraction of extern page \pagenumber\space from
1505 document "jobname.pdf" using method "\extractionmethod" was
1506 unsuccessful.}{Check the log file to see if the extraction script was
1507 executed at all, and if it finished successfully. You might also want to
1508 inspect "\externbasepath.log", the log file of the embedded TeX compilation
1509 which ran the extraction script}}
```

tex extraction command Using these keys, we set the system call which will be invoked for each extern page. The **tex extraction options** value of this key is expanded when executing the system command. The user may deploy **tex extraction script** the following macros in the value of these keys:

- `\externbasepath`: the extern PDF that should be produced, minus the `.pdf` suffix;
- `\pagenumber`: the page number to be extracted;
- `\expectedwidth`: the expected width of the extracted page;
- `\expectedheight`: the expected total height of the extracted page;

```
1510 \def\mmz@tex@extraction@systemcall{%
1511 \mmzvalueof{tex extraction command}\space
1512 \mmzvalueof{tex extraction options}\space
1513 "\mmzvalueof{tex extraction script}"%
1514 }
```

The default system call for \TeX -based extern extraction. As this method, despite being \TeX -based, shares no code with the document, we're free to implement it with any engine and format we want. For reasons of speed, we clearly go for the plain `pdf \TeX` .³ We perform the extraction by a little \TeX script, `memoize-extract-one`, inputted at the end of the value given to `tex extraction script`.

```
1515 \mmzset{
1516 tex extraction command/.initial=pdf $\text{\TeX}$ ,
1517 tex extraction options/.initial={%
1518 -halt-on-error
1519 -interaction=batchmode
1520 -jobname "\externbasepath"
1521 },
1522 tex extraction script/.initial={%
```

³I implemented the first version of \TeX -based extraction using \LaTeX and package `graphicx`, and it was (running with `pdf \TeX` engine) almost four times slower than the current plain \TeX implementation.

```

1523 \def\noexpand\fromdocument{\jobname.pdf}%
1524 \def\noexpand\pagenumber{\pagenumber}%
1525 \def\noexpand\expectedwidth{\expectedwidth}%
1526 \def\noexpand\expectedheight{\expectedheight}%
1527 \def\noexpand\logfile{\jobname.mmz.log}%
1528 \unexpanded{%
1529 \def\warningtemplate{%
1530 <latex> \noexpand\PackageWarning{memoize}{\warningtext}%
1531 <plain> \warning{memoize: \warningtext}%
1532 <context> \warning{memoize: \warningtext}%
1533 }}%
1534 \ifdef\XeTeXversion{}{%
1535 \def\noexpand\mmzpdfmajorversion{\the\pdfmajorversion}%
1536 \def\noexpand\mmzpdfminorversion{\the\pdfminorversion}%
1537 }%
1538 \noexpand\input memoize-extract-one
1539 },
1540 }
1541 </mmz>

```

4.3.1 memoize-extract-one.tex

The rest of the code of this section resides in file `memoize-extract-one.tex`. It is used to extract a single extern page from the document; it also checks whether the extern page dimensions are as expected, and passes a warning to the main job if that is not the case. For the reason of speed, the extraction script is in plain \TeX format. For the same reason, it is compiled by pdf \TeX engine by default, but we nevertheless take care that it will work with other (supported) engines as well.

```

1542 <{*extract-one}>
1543 \catcode`\@11\relax
1544 \def\@firstoftwo#1#2{#1}
1545 \def\@secondoftwo#1#2{#2}

```

Set the PDF version (maybe) passed to the script via `\mmzpdfmajorversion` and `\mmzpdfminorversion`.

```

1546 \ifdefined\XeTeXversion
1547 \else
1548 \ifdefined\luatexversion
1549 \def\pdfmajorversion{\pdfvariable majorversion}%
1550 \def\pdfminorversion{\pdfvariable minorversion}%
1551 \fi
1552 \ifdefined\mmzpdfmajorversion
1553 \pdfmajorversion\mmzpdfmajorversion\relax
1554 \fi
1555 \ifdefined\mmzpdfminorversion
1556 \pdfminorversion\mmzpdfminorversion\relax
1557 \fi
1558 \fi

```

Allocate a new output stream, always — `\newwrite` is `\outer` and thus cannot appear in a conditional.

```

1559 \newwrite\extractionlog

```

Are we requested to produce a log file?

```

1560 \ifdefined\logfile
1561 \immediate\openout\extractionlog{\logfile}%

```

Define a macro which both outputs the warning message and writes it to the extraction log.

```

1562 \def\doublewarning#1{%
1563   \message{#1}%
1564   \def\warningtext{#1}%

```

This script will be called from different formats, so it is up to the main job to tell us, by defining macro `\warningtemplate`, how to throw a warning in the log file.

```

1565   \immediate\write\extractionlog{%
1566     \ifdefined\warningtemplate\warningtemplate\else\warningtext\fi
1567   }%
1568 }%
1569 \else
1570   \let\doublewarning\message
1571 \fi
1572 \newif\ifforce
1573 \ifdefined\force
1574   \csname force\force\endcsname
1575 \fi

```

`\mmz@if@roughly@equal` This macro checks whether the given dimensions (#2 and #3) are equal within the tolerance given by #1. We use the macro both in the extraction script and in the main package. (We don't use `\ifpdfabsdim`, because it is unavailable in X_YT_EX.)

```

1576 </extract-one>
1577 < *mmz, extract-one >
1578 \def\mmz@tolerance{0.01pt}
1579 \def\mmz@if@roughly@equal#1#2#3{%
1580   \dimen0=\dimexpr#2-#3\relax
1581   \ifdim\dimen0<0pt
1582     \dimen0=-\dimen0\relax
1583   \fi
1584   \ifdim\dimen0>#1\relax
1585     \expandafter\@secondoftwo
1586   \else

```

The exact tolerated difference is, well, tolerated. This is a must to support `tolerance=0pt`.

```

1587   \expandafter\@firstoftwo
1588   \fi
1589 }%
1590 </mmz, extract-one >
1591 < *extract-one >

```

Grab the extern page from the document and put it in a box.

```

1592 \ifdefined\XeTeXversion
1593   \setbox0=\hbox{\XeTeXpdffile \fromdocument\space page \pagenumber media}%
1594 \else
1595   \ifdefined\luatexversion
1596     \saveimageresource page \pagenumber mediabox {\fromdocument}%
1597     \setbox0=\hbox{\useimageresource\lastsavedimageresourceindex}%
1598   \else
1599     \pdfximage page \pagenumber mediabox {\fromdocument}%
1600     \setbox0=\hbox{\pdfrefximage\pdflastximage}%
1601   \fi
1602 \fi

```

Check whether the extern page is of the expected size.

```

1603 \newif\ifbaddimensions
1604 \ifdefined\expectedwidth
1605   \ifdefined\expectedheight
1606     \mmz@if@roughly@equal{\mmz@tolerance}{\wd0}{\expectedwidth}{%
1607       \mmz@if@roughly@equal{\mmz@tolerance}{\ht0}{\expectedheight}%

```



```

1608     {}%
1609     {\baddimensionstrue}%
1610   }{\baddimensionstrue}%
1611   \fi
1612 \fi

```

We'll setup the page geometry of the extern file and shipout the extern — if all is well, or we're forced to do it.

```

1613 \ifdefined\luatexversion
1614   \let\pdfpagewidth\pagewidth
1615   \let\pdfpageheight\pageheight
1616   \def\pdfhorigin{\pdfvariable horigin}%
1617   \def\pdfvorigin{\pdfvariable vorigin}%
1618 \fi
1619 \def\do@shipout{%
1620   \pdfpagewidth=\wd0
1621   \pdfpageheight=\ht0
1622   \ifdefined\XeTeXversion
1623     \hoffset -1 true in
1624     \voffset -1 true in
1625   \else
1626     \pdfhorigin=0pt
1627     \pdfvorigin=0pt
1628   \fi
1629   \shipout\box0
1630 }
1631 \ifbaddimensions
1632   \doublewarning{I refuse to extract page \pagenumber\space from
1633     "\fromdocument", because its size (\the\wd0 \space x \the\ht0) is not
1634     what I expected (\expectedwidth\space x \expectedheight)}%
1635   \ifforce\do@shipout\fi
1636 \else
1637   \do@shipout
1638 \fi

```

If logging is in effect and the extern dimensions were not what we expected, write a warning into the log.

```

1639 \ifdefined\logfile
1640   \immediate\write\extractionlog{\noexpand\endinput}%
1641   \immediate\closeout\extractionlog
1642 \fi
1643 \bye
1644 </extract-one>

```

5 Automemoization

Install the advising framework implemented by our auxiliary package Advice, which automemoization depends on. This will define keys `auto`, `activate` etc. in our keypath.

```

1645 < *mmz >
1646 \mmzset{
1647   .install advice={setup key=auto, activation=deferred},

```

We switch to the immediate activation at the end of the preamble.

```

1648   begindocument/before/.append style={activation=immediate},
1649 }

```

manual Unless the user switched on `manual`, we perform the deferred (de)activations at the beginning of the document (and then clear the style, so that any further deferred activations will start with a

clean slate). In L^AT_EX, we will use the latest possible hook, `begindocument/end`, as we want to hack into commands defined by other packages. (The T_EX conditional needs to be defined before using it in `.append` code below.

```

1650 \newif\ifmmz@manual
1651 \mmzset{
1652   manual/.is if=mmz@manual,
1653   begindocument/end/.append code={%
1654     \ifmmz@manual
1655     \else
1656       \pgfkeysalso{activate deferred,activate deferred/.code={}}%
1657     \fi
1658   },

```

Announce Memoize’s run conditions and handlers.

```

1659   auto/.cd,
1660   run if memoization is possible/.style={
1661     run conditions=\mmz@auto@rc@if@memoization@possible
1662   },
1663   run if memoizing/.style={run conditions=\mmz@auto@rc@if@memoizing},
1664   apply options/.style={
1665     bailout handler=\mmz@auto@bailout,
1666     outer handler=\mmz@auto@outer,
1667   },
1668   memoize/.style={
1669     run if memoization is possible,
1670     apply options,
1671     inner handler=\mmz@auto@memoize
1672   },
1673   <math>
1674   noop/.style={run if memoization is possible, noop \AdviceType},
1675   noop command/.style={apply options, inner handler=\mmz@auto@noop},
1676   noop environment/.style={
1677     outer handler=\mmz@auto@noop@env, bailout handler=\mmz@auto@bailout},
1678   </math>
1679   <plain,context> noop/.style={inner handler=\mmz@auto@noop},
1680   nomemoize/.style={noop, options=disable},
1681   replicate/.style={run if memoizing, inner handler=\mmz@auto@replicate},
1682   to context/.style={run if memoizing, outer handler=\mmz@auto@tocontext},
1683 }

```

Abortion We cheat and let the `run conditions` do the work — it is cheaper to just always abort than to invoke the outer handler. (As we don’t set `\AdviceRuntrue`, the run conditions will never be satisfied.)

```

1684 \mmzset{
1685   auto/abort/.style={run conditions=\mmzAbort},
1686 }

```

And the same for unmemoizable:

```

1687 \mmzset{
1688   auto/unmemoizable/.style={run conditions=\mmzUnmemoizable},
1689 }

```

For one, we abort upon `\pdfsavepos` (called `\savepos` in LuaT_EX). Second, unless in LuaT_EX, we submit `\errmessage`, which allows us to detect at least some errors — in LuaT_EX, we have a more bullet-proof system of detecting errors, see `\mmz@memoize` in §3.2.

```

1690 \ifdef\luatexversion{%
1691   \mmzset{auto=\savepos{abort}}
1692 }{%

```

```

1693 \mmzset{
1694   auto=\pdfsavepos{abort},
1695   auto=\errmessage{abort},
1696 }
1697 }

```

run if memoization is possible These run conditions are used by `memoize` and `noop`: Memoize should be `\mmz@auto@rc@if@memoization@possible` enabled, but we should not be already within Memoize, i.e. memoizing or normally compiling some code submitted to memoization.

```

1698 \def\mmz@auto@rc@if@memoization@possible{%
1699   \ifmemoize
1700     \ifinmemoize
1701     \else
1702     \AdviceRuntrue
1703     \fi
1704 \fi
1705 }

```

run if memoizing These run conditions are used by `\label` and `\ref`: they should be handled only during `\mmz@auto@rc@if@memoizing` memoization (which implies that Memoize is enabled).

```

1706 \def\mmz@auto@rc@if@memoizing{%
1707   \ifmemoizing\AdviceRuntrue\fi
1708 }

```

\mmznxt The next-options, set by this macro, will be applied to the next, and only next instance of automemoization. We set the next-options globally, so that only the linear order of the invocation matters. Note that `\mmznxt`, being a user command, must also be defined in package `nomemoize`.

```

1709 </mmz>
1710 <nommz> \def\mmznxt#1{\ignorespaces}
1711 <*mmz>
1712 \def\mmznxt#1{\gdef\mmz@next{#1}\ignorespaces}
1713 \mmznxt{}%

```

apply options The outer and the bailout handler defined here work as a team. The outer handler's job is to `\mmz@auto@outer` apply the auto- and the next-options; therefore, the bailout handler must consume the next-
`\mmz@auto@bailout` options as well. To keep the option application local, the outer handler opens a group, which is expected to be closed by the inner handler. This key is used by `memoize` and `noop` command.

```

1714 \def\mmz@auto@outer{%
1715   \begingroup
1716   \mmzAutoInit
1717   \AdviceCollector
1718 }
1719 \def\mmz@auto@bailout{%
1720   \mmznxt{}%
1721 }

```

\mmzAutoInit Apply first the auto-options, and then the next-options (and clear the latter). Finally, if we have any extra collector options (set by the `verbatim` keys), append them to Advice's (raw) collector options.

```

1722 \def\mmzAutoInit{%
1723   \ifdefempty\AdviceOptions{\expandafter\mmzset\expandafter{\AdviceOptions}}%
1724   \ifdefempty\mmz@next{\expandafter\mmzset\expandafter{\mmz@next}\mmznxt{}}%
1725   \eappto\AdviceRawCollectorOptions{\expandonce\mmzRawCollectorOptions}%
1726 }

```

`memoize` This key installs the inner handler for memoization. If you compare this handler to the definition `\mmz@auto@memoize` of `\mmz` in section 3.1, you will see that the only thing left to do here is to start memoization with `\Memoize`, everything else is already done by the advising framework, as customized by `Memoize`.

The first argument to `\Memoize` is the memoization key (which the code `md5sum` is computed off of); it consists of the handled code (the contents of `\AdviceReplaced`) and its arguments, which were collected into `##1`. The second argument is the code which the memoization driver will execute. `\AdviceOriginal`, if invoked right away, would execute the original command; but as this macro is only guaranteed to refer to this command within the advice handlers, we expand it before calling `\Memoize`. that command.

Note that we don't have to define different handlers for commands and environments, and for different \TeX formats. When memoizing command `\foo`, `\AdviceReplaced` contains `\foo`. When memoizing environment `foo`, `\AdviceReplaced` contains `\begin{foo}`, `\foo` or `\startfoo`, depending on the format, while the closing tag (`\end{foo}`, `\endfoo` or `\stopfoo`) occurs at the end of the collected arguments, because `apply options` appended `\collargsEndTagtrue` to `raw collector options`.

This macro has no formal parameters, because the collected arguments will be grabbed by `\mmz@marshal`, which we have to go through because executing `\Memoize` closes the memoization group and we lose the current value of `\ifmmz@ignorespaces`. (We also can't use `\aftergroup`, because closing the group is not the final thing `\Memoize` does.)

```
1727 \long\def\mmz@auto@memoize#1{%
1728   \expanded{%
1729     \noexpand\Memoize
1730     {\expandonce\AdviceReplaced\unexpanded{#1}}%
1731     {\expandonce\AdviceOriginal\unexpanded{#1}}%
1732     \ifmmz@ignorespaces\ignorespaces\fi
1733   }%
1734 }
```

`noop` The no-operation handler can be used to apply certain options for the span of the execution `\mmz@auto@noop` of the handled command or environment. This is exploited by `auto/nomemoize`, which sets `\mmz@auto@noop@env` `disable` as an auto-option.

The handler for commands and non- \LaTeX environments is implemented as an inner handler. On its own, it does nothing except honor `verbatim` and `ignore spaces` (only takes care of `verbatim` and `ignore spaces` (in the same way as the memoization handler above), but it is intended to be used alongside the default outer handler, which applies the `auto-` and the `next-`options. As that handler opens a group (and this handler closes it), we have effectively delimited the effect of those options to this invocation of the handled command or environment.

```
1735 \long\def\mmz@auto@noop#1{%
1736   \expandafter\mmz@maybe@scantokens\expandafter{\AdviceOriginal#1}%
1737   \expandafter\endgroup
1738   \ifmmz@ignorespaces\ignorespaces\fi
1739 }
```

In \LaTeX , and only there, commands and environments need separate treatment. As \LaTeX environments introduce a group of their own, we can simply hook our initialization into the beginning of the environment (as a one-time hook). Consequently, we don't need to collect the environment body, so this can be an outer handler.

```
1740 <*\latex>
1741 \def\mmz@auto@noop@env{%
1742   \AddToHookNext{env/\AdviceName/begin}{%
1743     \mmzAutoInit
1744     \ifmmz@ignorespaces\ignorespacesafterend\fi
1745   }%
1746   \AdviceOriginal
1747 }
1748 </\latex>
```

`replicate` This inner handler writes a copy of the handled command or environment's invocation into `\mmz@auto@replicate` the cc-memo (and then executes it). As it is used alongside `run if memoizing`, the replicated command in the cc-memo will always execute the original command. The system works even if replication is off when the cc-memo is input; in that case, the control sequence in the cc-memo directly executes the original command.

This handler takes an option, `expanded` — if given, the collected arguments will be expanded (under protection) before being written into the cc-memo.

```

1749 \def\mmz@auto@replicate#1{%
1750   \begingroup
1751   \let\mmz@auto@replicate@expansion\unexpanded
1752   \expandafter\pgfqkeys\expanded{{/mmz/auto/replicate}{\AdviceOptions}}%
1753 <latex> \let\protect\noexpand
1754   \expanded{%
1755     \endgroup
1756     \noexpand\gtoksapp\noexpand\mmzCCMemo{%
1757       \expandonce\AdviceReplaced\mmz@auto@replicate@expansion{#1}}%
1758     \expandonce\AdviceOriginal\unexpanded{#1}%
1759   }%
1760 }
1761 \pgfqkeys{/mmz/auto/replicate}{
1762   expanded/.code={\let\mmz@auto@replicate@expansion\@firstofone},
1763 }

```

`to context` This outer handler appends the original definition of the handled command to the `\mmz@auto@tocontext` text. The `\expandafter` are there to expand `\AdviceName` once before fully expanding `\AdviceGetOriginalCname`.

```

1764 \def\mmz@auto@tocontext{%
1765   \expanded{%
1766     \noexpand\pgfkeysvalueof{/mmz/context/.cmd}%
1767     original "\AdviceNamespace" csname "\AdviceCname"={%
1768       \noexpand\expanded{%
1769         \noexpand\noexpand\noexpand\meaning
1770         \noexpand\AdviceCnameGetOriginal{\AdviceNamespace}{\AdviceCname}}%
1771     }%
1772   }%
1773 }%
1774 \pgfeov
1775 \AdviceOriginal
1776 }

```

5.1 L^AT_EX-specific handlers

We handle cross-referencing (both the `\label` and the `\ref` side) and indexing. Note that the latter is a straightforward instance of replication.

```

1777 <!*latex>
1778 \mmzset{
1779   auto/.cd,
1780   ref/.style={outer handler=\mmz@auto@ref\mmzNoRef, run if memoizing},
1781   force ref/.style={outer handler=\mmz@auto@ref\mmzForceNoRef, run if memoizing},
1782 }
1783 \mmzset{
1784   auto=\ref{ref},
1785   auto=\pageref{ref},
1786   auto=\label{run if memoizing, outer handler=\mmz@auto@label},
1787   auto=\index{replicate, args=m, expanded},
1788 }

```

`ref` These keys install an outer handler which appends a cross-reference to the context. `force ref` does this even if the reference key is undefined, while `ref` aborts memoization in such a case — `\mmz@auto@ref`

the idea is that it makes no sense to memoize when we expect the context to change in the next compilation anyway.

Any command taking a mandatory braced reference key argument potentially preceded by optional arguments of (almost) any kind may be submitted to these keys. This follows from the parameter list of `\mmz@auto@ref@i`, where #2 grabs everything up to the first opening brace. The downside of the flexibility regarding the optional arguments is that unbraced single-token reference keys will cause an error, but as such usages of `\ref` and friends should be virtually inexistent, we let the bug stay.

#1 should be either `\mmzNoRef` or `\mmzForceNoRef`. #2 will receive any optional arguments of `\ref` (or `\pageref`, or whatever), and #3 in `\mmz@auto@ref@i` is the cross-reference key.

```
1789 \def\mmz@auto@ref#1#2#{\mmz@auto@ref@i#1{#2}}
1790 \def\mmz@auto@ref@i#1#2#3{%
1791   #1{#3}%
1792   \AdviceOriginal#2{#3}%
1793 }
```

`\mmzForceNoRef` These macros do the real job in the outer handlers for cross-referencing, but it might be useful to have them publicly available. `\mmzForceNoRef` appends the reference key to the context. `\mmzNoRef` only does that if the reference is defined, otherwise it aborts the memoization.

```
1794 \def\mmzForceNoRef#1{%
1795   \mmz@Cos
1796   \expandafter\mmz@mtoc@csname\expandafter{\expanded{r@#1}}%
1797   \ignorespaces
1798 }
1799 \def\mmzNoRef#1{%
1800   \ifcsundef{r@#1}{\mmzAbort}{\mmzForceNoRef{#1}}%
1801   \ignorespaces
1802 }
```

`refrange` Let's rinse and repeat for reference ranges. The code is virtually the same as above, but we grab two reference key arguments (#3 and #4) in the final macro.
`force refrange`
`\mmz@auto@refrange`

```
1803 \mmzset{
1804   auto/.cd,
1805   refrange/.style={outer handler=\mmz@auto@refrange\mmzNoRef,
1806     bailout handler=\relax, run if memoizing},
1807   force refrange/.style={outer handler=\mmz@auto@refrange\mmzForceNoRef,
1808     bailout handler=\relax, run if memoizing},
1809 }
1810 \def\mmz@auto@refrange#1#2#{\mmz@auto@refrange@i#1{#2}}
1811 \def\mmz@auto@refrange@i#1#2#3#4{%
1812   #1{#3}%
1813   #1{#4}%
1814   \AdviceOriginal#2{#3}{#4}%
1815 }
```

`multiref` And one final time, for “multi-references”, such as `cleveref`'s `\cref`, which can take a comma-separated list of reference keys in the sole argument. Again, only the final macro is any different, `\mmz@auto@multiref` this time distributing #1 (`\mmzNoRef` or `\mmzForceNoRef`) over #3 by `\forcsvlist`.

```
1816 \mmzset{
1817   auto/.cd,
1818   multiref/.style={outer handler=\mmz@auto@multiref\mmzNoRef,
1819     bailout handler=\relax, run if memoizing},
1820   force multiref/.style={outer handler=\mmz@auto@multiref\mmzForceNoRef,
1821     bailout handler=\relax, run if memoizing},
1822 }
1823 \def\mmz@auto@multiref#1#2#{\mmz@auto@multiref@i#1{#2}}
1824 \def\mmz@auto@multiref@i#1#2#3{%
```

```

1825 \forcsvlist{#1}{#3}%
1826 \AdviceOriginal#2{#3}%
1827 }

```

`\mmz@auto@label` The outer handler for `\label` must be defined specifically for this command. The generic replicating handler is not enough here, as we need to replicate both the invocation of `\label` and the definition of `\@currentlabel`.

```

1828 \def\mmz@auto@label#1{%
1829 \xtoksapp\mmzCCMemo{%
1830 \noexpand\mmzLabel{#1}{\expandonce\@currentlabel}%
1831 }%
1832 \AdviceOriginal{#1}%
1833 }

```

`\mmzLabel` This is the macro that `\label`'s handler writes into the cc-memo. The first argument is the reference key; the second argument is the value of `\@currentlabel` at the time of invocation `\label` during memoization, which this macro temporarily restores.

```

1834 \def\mmzLabel#1#2{%
1835 \begingroup
1836 \def\@currentlabel{#2}%
1837 \label{#1}%
1838 \endgroup
1839 }
1840 </latex>

```

6 Support for various classes and packages

As the support for foreign classes and packages is only loaded in `begindocument/before`, any keys defined there are undefined in the preamble, and can only be used in the document body. We don't want to burden the author with this detail, so we try executing any unknown keys once again at `begindocument/end`.

```

1841 \def\mmz@unknown{}
1842 \mmzset{
1843 .unknown/.code={%
1844 \eappto\mmz@unknown{,\pgfkeyscurrentkey={#1}}}%
1845 },
1846 begindocument/end/.append code={%
1847 \pgfkeyslet{/mmz/.unknown/.@cmd}\undefined
1848 \expandafter\pgfkeysalso\expandafter{\mmz@unknown}%
1849 },
1850 }

1851 <*latex>
1852 \AddToHook{shipout/before}[memoize]{\global\advance\mmzExtraPages-1\relax}
1853 \AddToHook{shipout/after}[memoize]{\global\advance\mmzExtraPages1\relax}
1854 \mmzset{auto=\DiscardShipoutBox{
1855 outer handler=\global\advance\mmzExtraPages1\relax\AdviceOriginal}}
1856 </latex>

```

An auxiliary macro for loading the support code `memoize-*.code.tex` from hook `begindocument/before` where `@` already has catcode 'other'. The `\input` statement should be enclosed in `\mmz@makeatletter` and `\mmz@restoreatcatcode`.

```

1857 \def\mmz@makeatletter{%
1858 \edef\mmz@restoreatcatcode{\catcode`\noexpand\@the\catcode`\@relax}%
1859 \catcode`\@=11
1860 }

```

Utility macro for clarity below. #1 is the name of the package which should be loaded (used with L^AT_EX) and #2 is the name of the command which should be defined (used with plain T_EX and ConT_EXt) for #3 to be executed at the beginning of the document.

```

1861 \def\mmz@if@package@loaded#1#2#3{%
1862   \mmzset{%
1863     begindocument/before/.append code={%
1864 <latex>   \ifpackageloaded{#1}{%
1865 <plain, context>   \ifdefined#2%
1866               #3%
1867 <plain, context>   \fi
1868 <latex>   }{}%
1869   }%
1870 }%
1871 }

```

6.1 PGF

```

1872 \mmz@if@package@loaded{pgf}{%
1873 <plain>   \pgfpicture
1874 <context> \startpgfpicture
1875 }{%

```

We have very little code here, so we don't bother introducing `memoize-pgf.code.tex`.

```

1876 \def\mmzPgfAtBeginMemoization{%
1877   \edef\mmz@pgfpictureid{%
1878     \the\pgf@picture@serial@count
1879   }%
1880 }%
1881 \def\mmzPgfAtEndMemoization{%
1882   \edef\mmz@temp{%
1883     \the\numexpr\pgf@picture@serial@count-\mmz@pgfpictureid\relax
1884   }%
1885   \ifx\mmz@temp=0
1886     \else
1887       \xtoksapp\mmzCCMemo{\noexpand\mmzStepPgfPictureId{\mmz@temp}}%
1888     \fi
1889   }%
1890 \def\mmzStepPgfPictureId##1{%
1891   \global\advance\pgf@picture@serial@count##1\relax
1892 }%
1893 \mmzset{%
1894   at begin memoization=\mmzPgfAtBeginMemoization,
1895   at end memoization=\mmzPgfAtEndMemoization,
1896 } }%

```

6.2 TikZ

In this section, we activate TikZ support (the collector is defined by Advice). All the action happens at the end of the preamble, so that we can detect whether TikZ was loaded (regardless of whether Memoize was loaded before TikZ, or vice versa), but still input the definitions.

```

1898 \mmz@if@package@loaded{tikz}{\tikz}{%
1899   \input advice-tikz.code.tex

```

We define and activate the automemoization handlers for the TikZ command and environment.

```

1900 \mmzset{%
1901   auto={tikzpicture}{memoize},
1902   auto=\tikz{memoize, collector=\AdviceCollectTikZArguments},
1903 }%
1904 }

```


6.3 Forest

Forest will soon feature extensive memoization support, but for now, let's just enable the basic, single extern externalization. Command `\Forest` is defined using `xparse`, so `args` is unnecessary.

```
1905 <*\latex>
1906 \mmz@if@package@loaded{forest}{\Forest}{%
1907   \mmzset{
1908     auto={forest}{memoize},
1909     auto=\Forest{memoize},
1910   }%
1911 }
1912 </\latex>
```

6.4 Beamer

The Beamer code is explained in [M§4.2.4](#).

```
1913 <*\latex>
1914 \AddToHook{begindocument/before}{%
1915   \@ifclassloaded{beamer}{%
1916     \mmz@makeatletter
1917     \input{memoize-beamer.code.tex}%
1918     \mmz@restoreatcatcode
1919   }{}%
1920   \@ifpackageloaded{beamerarticle}{%
1921     \mmz@makeatletter
1922     \input{memoize-beamer.code.tex}%
1923     \mmz@restoreatcatcode
1924   }{}%
1925 }
1926 </\latex>
1927 </mmz>
1928 <*\beamer>
1929 \mmzset{
1930   per overlay/.code={},
1931   beamer mode to prefix/.style={
1932     prefix=\mmz@prefix@dir\mmz@prefix@name\beamer@currentmode_mode.
1933   },
1934 }%
1935 \@ifclassloaded{beamer}{%
1936   \mmzset{
1937     per overlay/.style={
1938       /mmz/context={%
1939         overlay=\csname beamer@overlaynumber\endcsname,
1940         pauses=\ifmemoizing
1941           \mmzBeamerPauses
1942         \else
1943           \expandafter\the\csname c@beamerpauses\endcsname
1944         \fi
1945       },
1946       /mmz/at begin memoization={%
1947         \xdef\mmzBeamerPauses{\the\c@beamerpauses}%
1948         \xtoksapp\mmzCMemo{%
1949           \noexpand\mmzSetBeamerOverlays{\mmzBeamerPauses}{\beamer@overlaynumber}}%
1950         \gtoksapp\mmzCCMemo{%
1951           \only<all:\mmzBeamerOverlays>{}}%
1952       },
1953       /mmz/at end memoization={%
1954         \xtoksapp\mmzCCMemo{%
1955           \noexpand\setcounter{beamerpauses}{\the\c@beamerpauses}}%
1956       },
1957       /mmz/per overlay/.code={},
1958     },
```

```

1959 }
1960 \def\mmzSetBeamerOverlays#1#2{%
1961   \ifnum\c@beamerpauses=#1\relax
1962     \gdef\mmzBeamerOverlays{#2}%
1963     \ifnum\beamer@overlaynumber<#2\relax \mmz@temptrue \else \mmz@tempfalse \fi
1964   \else
1965     \mmz@temptrue
1966   \fi
1967   \ifmmz@temp
1968     \appto\mmzAtBeginMemoization{%
1969       \gtoksapp\mmzCMemo{\mmzSetBeamerOverlays{#1}{#2}}}%
1970   \fi
1971 }%
1972 }%
1973 </beamer>
1974 < *mmz >

```

6.5 Biblatex

```

1975 < *latex >
1976 \mmzset{
1977   biblatex/.code={%
1978     \mmz@if@package@loaded{biblatex}{}{%
1979       \mmz@makeatletter
1980       \input memoize-biblatex.code.tex
1981       \mmz@restreatcatcode
1982       \mmzset{#1}%
1983     }%
1984   },
1985 }
1986 </latex>
1987 </mmz>
1988 < *biblatex >
1989 \mmzset{%

```

Advise macro `\entry` occurring in `.bbl` files to collect the entry, verbatim. `args: m = citation key, &&{...}u = the entry, verbatim, braced` — so `\blx@bbl@entry` will receive two mandatory arguments.

```

1990   auto=\blx@bbl@entry{
1991     inner handler=\mmz@biblatex@entry,
1992     args={%
1993       m%
1994       &&{\collargsVerb
1995         \collargsAppendExpandablePostprocessor{\the\collargsArg}}%
1996       }u{\endentry}%
1997   },

```

No braces around the collected arguments, as each is already braced on its own.

```

1998   raw collector options=\collargsReturnPlain,
1999 },

```

`cite` Define handlers for citation commands.

```

volcite
cites
volcites
2000   auto/cite/.style={
2001     run conditions=\mmz@biblatex@cite@rc,
2002     outer handler=\mmz@biblatex@cite@outer,
2003     args=l*m,
2004     raw collector options=\mmz@biblatex@def@star\collargsReturnNo,
2005     inner handler=\mmz@biblatex@cite@inner,
2006   },

```

We need a dedicated `volcite` even though `\volcite` executes `\cite` because otherwise, we would end up with `\cite{volume}{key}` in the cc-memo when `biblatex ccmemo cite=replicate`.

```

2007 auto/volcite/.style={
2008   run if memoizing,
2009   outer handler=\mmz@biblatex@cite@outer,
2010   args=lml*m,
2011   raw collector options=\mmz@biblatex@def@star\collargsReturnNo,
2012   inner handler=\mmz@biblatex@cite@inner,
2013 },
2014 auto/cites/.style={
2015   run conditions=\mmz@biblatex@cites@rc,
2016   outer handler=\mmz@biblatex@cites@outer,
2017   args=l*m,
2018   raw collector options=
2019     \mmz@biblatex@def@star\collargsClearArgsfalse\collargsReturnNo,
2020   inner handler=\mmz@biblatex@cites@inner,
2021 },
2022 auto/volcites/.style={
2023   run if memoizing,
2024   outer handler=\mmz@biblatex@cites@outer,
2025   args=lml*m,
2026   raw collector options=
2027     \mmz@biblatex@def@star\collargsClearArgsfalse\collargsReturnNo,
2028   inner handler=\mmz@biblatex@cites@inner,
2029 },

```

`biblatex ccmemo cite` What to put into the cc-memo, `\nocite` or the handled citation command?

```

2030 biblatex ccmemo cite/.is choice,
2031 biblatex ccmemo cite/nocite/.code={%
2032   \let\mmz@biblatex@do@ccmemo\mmz@biblatex@do@nocite
2033 },
2034 biblatex ccmemo cite/replicate/.code={%
2035   \let\mmz@biblatex@do@ccmemo\mmz@biblatex@do@replicate
2036 },
2037 }%

```

`\mmz@biblatex@entry` This macro stores the MD5 sum of the `\entry` when reading the `.bbl` file.

```

2038 \def\mmz@biblatex@entry#1#2{%
2039   \edef\mmz@temp{\pdf@mdfivesum{#2}}%
2040   \global\cslet{mmz@bbl@#1}\mmz@temp
2041   \mmz@scantokens{\AdviceOriginal{#1}#2}%
2042 }

```

`\mmz@biblatex@cite@rc` Run if memoizing but not within a `\volcite` command. Applied to `\cite(s)`.

```

\mmz@biblatex@cites@rc
2043 \def\mmz@biblatex@cite@rc{%
2044   \ifmemoizing

```

We cannot use the official `\ifvolcite`, or even the `blx@volcite` toggle it depends on, because these are defined/set within the next-citation hook, which is yet to be executed. So we depend on the internal detail that `\volcite` and friends redefine `\blx@citeargs` to `\blx@volciteargs`.

```

2045   \ifx\blx@citeargs\blx@volciteargs
2046   \else
2047     \AdviceRuntrue
2048   \fi
2049 \fi
2050 }
2051 \def\mmz@biblatex@cites@rc{%
2052   \ifmemoizing

```

The internal detail with `\volcites`: it defines a hook.

```

2053   \ifdef\blx@hook@mcite@before{}{\AdviceRuntrue}%
2054   \fi
2055 }

```

`\mmz@biblatex@cite@outer` Initialize the macro receiving the citation key(s), and execute the collector.

```
2056 \def\mmz@biblatex@cite@outer{%
2057   \gdef\mmz@biblatex@keys{}%
2058   \AdviceCollector
2059 }
```

`\mmz@biblatex@mark@citation@key` We *append* to `\mmz@biblatex@keys` to automatically collect all citation keys of a `\cites` command; note that we use this system for `\cite` as well.

```
2060 \def\mmz@biblatex@def@star{%
2061   \collargsAlias{*}{&&\mmz@biblatex@mark@citation@key}}%
2062 }
2063 \def\mmz@biblatex@mark@citation@key{%
2064   \collargsAppendPreprocessor{\xappto\mmz@biblatex@keys{\the\collargsArg}}%
2065 }
```

`\mmz@biblatex@cite@inner` This macro puts the cites reference keys into the context, and adds `\nocite`, or the handled citation command, to the cc-memo.

```
2066 \def\mmz@biblatex@cite@inner{%
2067   \mmz@biblatex@do@context
2068   \mmz@biblatex@do@ccmemo
2069   \expandafter\AdviceOriginal\the\collargsArgs
2070 }
2071 \def\mmz@biblatex@do@context{%
2072   \expandafter\forcsvlist
2073     \expandafter\mmz@biblatex@do@context@one
2074     \expandafter{\mmz@biblatex@keys}%
2075 }
2076 \def\mmz@biblatex@do@context@one#1{%
2077   \mmz@Cos
2078   \mmz@mtoc@csname{mmz@bbl@#1}%
2079   \ifcsdef{mmz@bbl@#1}{\mmzAbort}%
2080 }
2081 \def\mmz@biblatex@do@nocite{%
2082   \xtoksapp\mmzCCMemo{%
2083     \noexpand\nocite{\mmz@biblatex@keys}%
2084   }%
2085 }
2086 \def\mmz@biblatex@do@replicate{%
2087   \xtoksapp\mmzCCMemo{%
2088     {%
2089       \nullfont
```

It is ok to use `\AdviceName` here, as the cc-memo is never input during memoization.

```
2090     \expandonce\AdviceName\the\collargsArgs
2091   }%
2092 }%
2093 }
2094 \let\mmz@biblatex@do@ccmemo\mmz@biblatex@do@nocite
```

`\mmz@biblatex@cites@outer` Same as for `\cite`, but we iterate the collector as long as the arguments continue.

```
2095 \def\mmz@biblatex@cites@outer{%
2096   \global\collargsArgs{}%
2097   \gdef\mmz@biblatex@keys{}%
2098   \AdviceCollector
2099 }
2100 \def\mmz@biblatex@cites@inner{%
2101   \futurelet\mmz@temp\mmz@biblatex@cites@inner@again
2102 }
```

If the following token is an opening brace or bracket, the multicite arguments continue.

```
2103 \def\mmz@biblatex@cites@inner@again{%
2104   \mmz@tempfalse
2105   \ifx\mmz@temp\bgroup
2106     \mmz@temptrue
2107   \else
2108     \ifx\mmz@temp[%]
2109       \mmz@temptrue
2110     \fi
2111   \fi
2112   \ifmmz@temp
2113     \expandafter\AdviceCollector
2114   \else
2115     \expandafter\mmz@biblatex@cites@inner@finish
2116   \fi
2117 }
2118 \def\mmz@biblatex@cites@inner@finish{%
2119   \mmz@biblatex@do@context
2120   \mmz@biblatex@do@ccmemo
2121   \expandafter\AdviceOriginal\the\collargsArgs
2122 }
```

[Advise](#) the citation commands.

```
2123 \mmzset{
2124   auto=\cite{cite},
2125   auto=\Cite{cite},
2126   auto=\parencite{cite},
2127   auto=\Parencite{cite},
2128   auto=\footcite{cite},
2129   auto=\footcitetext{cite},
2130   auto=\textcite{cite},
2131   auto=\Textcite{cite},
2132   auto=\smartcite{cite},
2133   auto=\Smartcite{cite},
2134   auto=\supercite{cite},
2135   auto=\cites{cites},
2136   auto=\Cites{cites},
2137   auto=\parencites{cites},
2138   auto=\Parencites{cites},
2139   auto=\footcites{cites},
2140   auto=\footcitetexts{cites},
2141   auto=\smartcites{cites},
2142   auto=\Smartcites{cites},
2143   auto=\textcites{cites},
2144   auto=\Textcites{cites},
2145   auto=\supercites{cites},
2146   auto=\autocite{cite},
2147   auto=\Autocite{cite},
2148   auto=\autocites{cites},
2149   auto=\Autocites{cites},
2150   auto=\citeauthor{cite},
2151   auto=\Citeauthor{cite},
2152   auto=\citetitle{cite},
2153   auto=\citeyear{cite},
2154   auto=\citedate{cite},
2155   auto=\citeurl{cite},
2156   auto=\nocite{cite},
2157   auto=\fullcite{cite},
2158   auto=\footfullcite{cite},
2159   auto=\volcite{volcite},
2160   auto=\Volcite{volcite},
2161   auto=\volcites{volcites},
```

```

2162 auto=\Volcites{volcites},
2163 auto=\pvolcite{volcite},
2164 auto=\Pvolcite{volcite},
2165 auto=\pvolcites{volcites},
2166 auto=\Pvolcites{volcites},
2167 auto=\fvolcite{volcite},
2168 auto=\Fvolcite{volcite},
2169 auto=\fvolcites{volcites},
2170 auto=\Fvolcites{volcites},
2171 auto=\ftvolcite{volcite},
2172 auto=\ftvolcites{volcites},
2173 auto=\Ftvolcite{volcite},
2174 auto=\Ftvolcites{volcites},
2175 auto=\svolcite{volcite},
2176 auto=\Svolcite{volcite},
2177 auto=\svolcites{volcites},
2178 auto=\Svolcites{volcites},
2179 auto=\tvolcite{volcite},
2180 auto=\Tvolcite{volcite},
2181 auto=\tvolcites{volcites},
2182 auto=\Tvolcites{volcites},
2183 auto=\avolcite{volcite},
2184 auto=\Avolcite{volcite},
2185 auto=\avolcites{volcites},
2186 auto=\Avolcites{volcites},
2187 auto=\notecite{cite},
2188 auto=\Notecite{cite},
2189 auto=\pnotecite{cite},
2190 auto=\Pnotecite{cite},
2191 auto=\fnotecite{cite},

```

Similar to `volcite`, these commands must be handled specifically in order to function correctly with `biblatex ccmemo cite=replicate`.

```

2192 auto=\citename{cite, args=1*mlm},
2193 auto=\citelist{cite, args=1*mlm},
2194 auto=\citefield{cite, args=1*mlm},
2195 }
2196 </biblatex>
2197 <*mmz>

```

7 Initialization

`begindocument/before` These styles contain the initialization and the finalization code. They were populated throughout the source. Hook `begindocument/before` contains the package support code, which must be loaded while still in the preamble. Hook `begindocument` contains the initialization code whose execution doesn't require any particular timing, as long as it happens at the beginning of the document. Hook `begindocument/end` is where the commands are activated; this must crucially happen as late as possible, so that we successfully override foreign commands (like `hyperref`'s definitions). In `LATEX`, we can automatically execute these hooks at appropriate places:

```

2198 <*latex>
2199 \AddToHook{begindocument/before}{\mmzset{begindocument/before}}
2200 \AddToHook{begindocument}{\mmzset{begindocument}}
2201 \AddToHook{begindocument/end}{\mmzset{begindocument/end}}
2202 \AddToHook{enddocument/afterlastpage}{\mmzset{enddocument/afterlastpage}}
2203 </latex>

```

In plain `TEX`, the user must execute these hooks manually; but at least we can group them together and given them nice names. Provisionally, manual execution is required in `ConTEXt` as

well, as I'm not sure where to execute them — please help!

```
2204 <{*plain, context}>
2205 \mmzset{
2206   begin document/.style={begindocument/before, begindocument, begindocument/end},
2207   end document/.style={enddocument/afterlastpage},
2208 }
2209 <{/plain, context}>
```

We clear the hooks after executing the last of them.

```
2210 \mmzset{
2211   begindocument/end/.append style={
2212     begindocument/before/.code={},
2213     begindocument/.code={},
2214     begindocument/end/.code={},
2215   }
2216 }
```

Formats other than plain T_EX need a way to prevent extraction during package-loading.

```
2217 <{!plain}>\mmzset{extract/no/.code={}}
```

[memoize.cfg](#) Load the configuration file. Note that `nomemoize` must input this file as well, because any special memoization-related macros defined by the user should be available; for example, my `memoize.cfg` defines `\ifregion` (see ^M§2.6).

```
2218 <{/mmz}>
2219 <{mmz, nommz}>\InputIfFileExists{memoize.cfg}{-}{-}
2220 <{*mmz}>
```

For formats other than plain T_EX, we also save the current (initial or `memoize.cfg`-set) value of `extract`, so that we can restore it when package options include `extract=no`. Then, `extract` can be called without an argument in the preamble, triggering extraction using this method; this is useful e.g. if Memoize is compiled into a format.

```
2221 <{!plain}>\let\mmz@initial@extraction@method\mmz@extraction@method
```

[Process](#) the package options (except in plain T_EX).

```
2222 <{*latex}>
2223 \DeclareUnknownKeyHandler[mmz]{%
2224   \expanded{\noexpand\pgfqkeys{/mmz}{#1\IfBlankF{#2}{-}{#2}}}}
2225 \ProcessKeyOptions[mmz]
2226 <{/latex}>
2227 <{context}>\expandafter\mmzset\expandafter{\currentmoduleparameters}
```

In L^AT_EX, `nomemoize` has to process package options as well, otherwise L^AT_EX will complain.

```
2228 <{/mmz}>
2229 <{*latex & nommz}>
2230 \DeclareUnknownKeyHandler[mmz]{-}
2231 \ProcessKeyOptions[mmz]
2232 <{/latex & nommz}>
```

[Extern extraction](#) We redefine `extract` to immediately trigger extraction. This is crucial in plain T_EX, where extraction must be invoked after loading the package, but also potentially useful in other formats when package options include `extract=no`.

```
2233 <{*mmz}>
2234 \mmzset{
2235   extract/.is choice,
2236   extract/.default=\mmz@extraction@method,
```

But only once:

```
2237 extract/.append style={
2238   extract/.code={\PackageError{memoize}{Key "extract" was invoked twice.}{In
2239     principle, externs should be extracted only once.  If you really want
2240     to extract again, execute "extract/<method>".}},
2241 },
```

In formats other than plain T_EX, we remember the current `extract` code and then trigger the extraction.

```
2242 (!plain) /utils/exec={\pgfkeysgetvalue{/mmz/extract/.@cmd}\mmz@temp@extract},
2243 (!plain) extract=\mmz@extraction@method,
2244 }
```

Option `extract=no` (which only exists in formats other than plain T_EX) should allow for an explicit invocation of `extract` in the preamble.

```
2245 (*!plain)
2246 \def\mmz@temp{no}
2247 \ifx\mmz@extraction@method\mmz@temp
2248   \pgfkeyslet{/mmz/extract/.@cmd}\mmz@temp@extract
2249   \let\mmz@extraction@method\mmz@initial@extraction@method
2250 \fi
2251 \let\mmz@temp@extract\relax
2252 </!plain>
```

Memoize was not really born for the draft mode, as it cannot produce new externs there. But we don't want to disable the package, as utilization and pure memoization are still perfectly valid in this mode, so let's just warn the user.

```
2253 \ifnum\pdf@draftmode=1
2254   \PackageWarning{memoize}{No externalization will be performed in the draft mode}%
2255 \fi
2256 </mmz>
```

Several further things which need to be defined as dummies in `nomemoize/memoizable`.

```
2257 (*nommz, mmzable & generic)
2258 \pgfkeys{%
2259   /handlers/.meaning to context/.code={},
2260   /handlers/.value to context/.code={},
2261 }
2262 \let\mmzAbort\relax
2263 \let\mmzUnmemoizable\relax
2264 \newcommand\IfMemoizing[2][\@secondoftwo]
2265 \let\mmzNoRef\@gobble
2266 \let\mmzForceNoRef\@gobble
2267 \newtoks\mmzContext
2268 \newtoks\mmzContextExtra
2269 \newtoks\mmzCMemo
2270 \newtoks\mmzCCMemo
2271 \newcount\mmzExternPages
2272 \newcount\mmzExtraPages
2273 \let\mmzTracingOn\relax
2274 \let\mmzTracingOff\relax
2275 </nommz, mmzable & generic>
```

The end of `memoize`, `nomemoize` and `memoizable`.

```
2276 (*mmz, nommz, mmzable)
2277 (plain)\resetatcatcode
2278 (context)\stopmodule
2279 (context)\protect
2280 </mmz, nommz, mmzable>
```


That's all, folks!

8 Auxiliary packages

8.1 Extending commands and environments with Advice

```
2281 (*main)
2282 <latex>\ProvidesPackage{advice}[2024/03/15 v1.1.1 Extend commands and environments]
2283 <context>%D \module[
2284 <context>%D     file=t-advice.tex,
2285 <context>%D     version=1.1.1,
2286 <context>%D     title=Advice,
2287 <context>%D     subtitle=Extend commands and environments,
2288 <context>%D     author=Saso Zivanovic,
2289 <context>%D     date=2024-03-15,
2290 <context>%D     copyright=Saso Zivanovic,
2291 <context>%D     license=LPPL,
2292 <context>%D ]
2293 <context>\writestatus{loading}{ConTeXt User Module / advice}
2294 <context>\unprotect
2295 <context>\startmodule[advice]
```

Required packages

```
2296 <plain, context>\input miniltx
2297 <latex>\RequirePackage{collargs}
2298 <plain>\input collargs
2299 <context>\input t-collargs
```

In \LaTeX , we also require `xparse`. Even though `\NewDocumentCommand` and friends are integrated into the \LaTeX kernel, `\GetDocumentCommandArgSpec` is only available through `xparse`.

```
2300 <latex>\RequirePackage{xparse}
```

8.1.1 Installation into a keypath

`.install advice` This handler installs the advising mechanism into the handled path, which we shall henceforth also call the (advice) namespace.

```
2301 \pgfkeys{
2302   /handlers/.install advice/.code={%
2303     \edef\auto@install@namespace{\pgfkeyscurrentpath}%
2304     \def\advice@install@setupkey{advice}%
2305     \def\advice@install@activation{immediate}%
2306     \pgfkeys{/advice/install}{#1}%
2307     \expanded{\noexpand\advice@install
2308       {\auto@install@namespace}%
2309       {\advice@install@setupkey}%
2310       {\advice@install@activation}%
2311     }%
2312   },
```

setup key These keys can be used in the argument of `.install advice` to configure the installation. By **activation** default, the setup key is `advice` and activation is `immediate`.

```
2313   /advice/install/.cd,
2314   setup key/.store in=\advice@install@setupkey,
2315   activation/.is choice,
2316   activation/.append code=\def\advice@install@activation{#1},
2317   activation/immediate/.code={},
2318   activation/deferred/.code={},
2319 }
```

#1 is the installation keypath (in Memoize, /mmz); #2 is the setup key name (in Memoize, auto, and this is why we document it as such); #3 is the initial activation regime.

```
2320 \def\advice@install#1#2#3{%
```

Switch to the installation keypath.

```
2321 \pgfqkeys{#1}{%
```

auto These keys submit a command or environment to advising. The namespace is hard-coded into **auto csname** these keys via #1; their arguments are the command/environment (cs)name, and setup keys **auto key** belonging to path *(installation keypath)*/\meta{setup key name}.

```
2322 #2/.code 2 args={%
```

auto key' Call the internal setup macro, wrapping the received keylist into a **pgfkeys** invocation.

```
2323 \AdviceSetup{#1}{#2}{##1}{\pgfqkeys{#1/#2}{##2}}%
```

Activate if not already activated (this can happen when updating the configuration). Note we don't call **\advice@activate** directly, but use the public keys; in this way, activation is automatically deferred if so requested. (We don't use **\pgfkeysalso** to allow **auto** being called from any path.)

```
2324 \pgfqkeys{#1}{try activate, activate={##1}}%
```

```
2325 },
```

A variant without activation.

```
2326 #2'/.code 2 args={%
2327 \AdviceSetup{#1}{#2}{##1}{\pgfqkeys{#1/#2}{##2}}%
2328 },
2329 #2 csname/.style 2 args={
2330 #2/.expand once=\expandafter{\csname ##1\endcsname}{##2},
2331 },
2332 #2 csname'/.style 2 args={
2333 #2'/.expand once=\expandafter{\csname ##1\endcsname}{##2},
2334 },
2335 #2 key/.style 2 args={
2336 #2/.expand once=%
2337 \expandafter{\csname pgfk@##1/.@cmd\endcsname}%
2338 {collector=\advice@pgfkeys@collector,##2},
2339 },
2340 #2 key'/.style 2 args={
2341 #2'/.expand once=%
2342 \expandafter{\csname pgfk@##1/.@cmd\endcsname}%
2343 {collector=\advice@pgfkeys@collector,##2},
2344 },
```

activation This key, residing in the installation keypath, forwards the request to the /advice path **activation** subkeys, which define **activate** and **friends** in the installation keypath. Initially, the activation regime is whatever the user has requested using the **.install advice** argument (here #3).

```
2345 activation/.style={/advice/activation/##1={#1}},
```

```
2346 activation=#3,
```

activate deferred The deferred activations are collected in this style, see section `refsec:code:advice:activation` for details.

```
2347 activate deferred/.code={},
```

activate csname For simplicity of implementation, the `csname` versions of `activate` and `deactivate` accept a single `csname`. This way, they can be defined right away, as they don't change with the type of activation (immediate vs. deferred).

```
2348 activate csname/.style={activate/.expand once={\csname##1\endcsname}},
2349 deactivate csname/.style={deactivate/.expand once={\csname##1\endcsname}},
```

activate key (De)activation of `pgfkeys` keys. Accepts a list of key names, requires full key names.

```
deactivate key
2350 activate key/.style={activate@key={#1/activate}{##1}},
2351 deactivate key/.style={activate@key={#1/deactivate}{##1}},
2352 activate@key/.code n args=2{%
2353   \def\advice@temp{ }%
2354   \def\advice@do####1{%
2355     \appto\advice@temp{\expandonce{\csname pgfk@####1/.@cmd\endcsname}}}%
2356   \forcsvlist\advice@do{##2}%
2357   \pgfkeysalso{##1/.expand once=\advice@temp}%
2358 },
```

The rest of the keys defined below reside in the `auto` subfolder of the installation keypath.

```
2359 #2/.cd,
```

run conditions These keys are used to setup the handling of the command or environment. The

outer handler storage macros (`\AdviceRunConditions` etc.) have public names as they also play

bailout handler a crucial role in the handler definitions, see section 8.1.3.

```
collector
  args 2360 run conditions/.store in=\AdviceRunConditions,
collector options 2361 bailout handler/.store in=\AdviceBailoutHandler,
2362 outer handler/.store in=\AdviceOuterHandler,
clear collector options 2363 collector/.store in=\AdviceCollector,
raw collector options 2364 collector options/.code={\appto\AdviceCollectorOptions{##1}},
clear raw collector options 2365 clear collector options/.code={\def\AdviceCollectorOptions{ }},
inner handler 2366 raw collector options/.code={\appto\AdviceRawCollectorOptions{##1}},
options 2367 clear raw collector options/.code={\def\AdviceRawCollectorOptions{ }},
clear options 2368 args/.store in=\AdviceArgs,
2369 inner handler/.store in=\AdviceInnerHandler,
2370 options/.code={\appto\AdviceOptions{##1}},
2371 clear options/.code={\def\AdviceOptions{ }},
```

A user-friendly way to set options: any unknown key is an option.

```
2372 .unknown/.code={%
2373   \appto{\AdviceOptions}{,\pgfkeyscurrentname={\unexpanded{##1}}}%
2374 },
```

The default values of the keys, which equal the initial values for commands, as assigned by `\advice@setup@init@command`.

```
2375 run conditions/.default=\AdviceRuntrue,
2376 bailout handler/.default=\relax,
2377 outer handler/.default=\AdviceCollector,
2378 collector/.default=\advice@CollectArgumentsRaw,
2379 collector options/.value required,
2380 raw collector options/.value required,
2381 args/.default=\advice@noargs,
2382 inner handler/.default=\advice@error@noinnerhandler,
2383 options/.value required,
```

reset This key resets the advice settings to their initial values, which depend on whether we're handling a command or environment.

```
2384 reset/.code={\csname\advice@setup@init@\AdviceType\endcsname},
```

`after setup` The code given here will be executed once we exit the setup group. `integrated driver` of Memoize uses it to declare a conditional.

```
2385   after setup/.code={\appto\AdviceAfterSetup{##1}},
```

In L^AT_EX, we finish the installation by submitting `\begin`; the submission is funky, because the run conditions handler actually hacks the standard handling procedure. Note that if `\begin` is not activated, environments will not be handled, and that the automatic activation might be deferred.

```
2386 <latex>   #1/#2=\begin{run conditions=\advice@begin@rc},
2387   }%
2388 }
```

8.1.2 Submitting a command or environment

`\AdviceSetup` Macro `\advice@setup` is called by key `auto` to submit a command or environment to advising. `\AdviceName` It receives four arguments: `#1` is the installation keypath / storage namespace; `#2` is the name of the setup key; `#3` is the submitted command or environment; `#4` is the setup code (which is only grabbed by `\advice@setup@i`).

Executing this macro defines macros `\AdviceName`, holding the control sequence of the submitted command or the environment name, and `\AdviceType`, holding `command` or `environment`; they are used to set up some initial values, and may be used by user-defined keys in the `auto` path, as well (see `/mmz/auto/noop` for an example). The macro then performs internal initialization, and finally calls the second part, `\advice@setup@i`, with the command's *storage* name as the first argument.

This macro also serves as the programmer's interface to `auto`, the idea being that an advanced user may write code `#4` which defined the settings macros (`\AdviceOuterHandler` etc.) without deploying `pgfkeys`. (Also note that activation at the end only occurs through the `auto` interface.)

```
2389 \def\AdviceSetup#1#2#3{%
```

Open a group, so that we allow for embedded `auto` invocations.

```
2390   \begingroup
2391   \def\AdviceName{#3}%
2392   \advice@def@AdviceCname
```

Command, complain, or environment?

```
2393   \collargs@cs@cases{#3}{%
2394     \def\AdviceType{command}%
2395     \advice@setup@init@command
2396     \advice@setup@i{#3}{#1}{#3}%
2397   }{%
2398     \advice@error@advice@notcs{#1/#2}{#3}%
2399   }{%
2400     \def\AdviceType{environment}%
2401     \advice@setup@init@environment
2402 <latex>   \advice@setup@i{#3}%
2403 <plain>   \expandafter\advice@setup@i\expandafter{\csname #3\endcsname}%
2404 <context> \expandafter\advice@setup@i\expandafter{\csname start#3\endcsname}%
2405     {#1}{#3}%
2406   }%
2407 }
```

The arguments of `\advice@setup@i` are a bit different than for `\advice@setup`, because we have inserted the storage name as `#1` above, and we lost the setup key name `#2`. Here, `#2` is the installation keypath / storage namespace, `#3` is the submitted command or environment; and `#4` is the setup code.

What is the difference between the storage name (`#1`) and the command/environment name (`#3`, and also the contents of `\AdviceName`), and why do we need both? For commands, there is

actually no difference; for example, when submitting command `\foo`, we end up with `#1=#3=\foo`. And there is also no difference for \LaTeX environments; when submitting environment `foo`, we get `#1=#3=foo`. But in plain \TeX , `#1=\foo` and `#3=foo`, and in \ConTeXt , `#1=\startfoo` and `#3=foo` — which should explain the guards and `\expandafters` above.

And why both `#1` and `#3`? When a handled command is executed, it loads its configuration from a macro determined by the storage namespace and the (`\stringified`) storage name, e.g. `/mmz` and `\foo`. In plain \TeX and \ConTeXt , each environment is started by a dedicated command, `\foo` or `\startfoo`, so these control sequences (`\stringified`) must act as storage names. (Not so in \LaTeX , where an environment configuration is loaded by `\begin`'s handler, which can easily work with storage name `foo`. Even more, having `\foo` as an environment storage name would conflict with the storage name for the (environment-internal) command `\foo` — yes, we can submit either `foo` or `\foo`, or both, to advising.)

```
2408 \def\advice@setup@i#1#2#3#4{%
```

Load the current configuration of the handled command or environment — if it exists.

```
2409 \advice@setup@init@i{#2}{#1}%
2410 \advice@setup@init@I{#2}{#1}%
2411 \def\AdviceAfterSetup{}%
```

Apply the setup code/keys.

```
2412 #4%
```

Save the resulting configuration. This closes the group, because the config is saved outside it.

```
2413 \advice@setup@save{#2}{#1}%
2414 }
```

Initialize the configuration of a command or environment. Note that the default values of the keys equal the initial values for commands. Nothing would go wrong if these were not the same, but it's nice that the end-user can easily revert to the initial values.

```
2415 \def\advice@setup@init@common{%
2416 \def\AdviceRunConditions{\AdviceRuntrue}%
2417 \def\AdviceBailoutHandler{\relax}%
2418 \def\AdviceOuterHandler{\AdviceCollector}%
2419 \def\AdviceCollector{\advice@CollectArgumentsRaw}%
2420 \def\AdviceCollectorOptions{}%
2421 \def\AdviceInnerHandler{\advice@error@noinnerhandler}%
2422 \def\AdviceOptions{}%
2423 }
2424 \def\advice@setup@init@command{%
2425 \advice@setup@init@common
2426 \def\AdviceRawCollectorOptions{}%
2427 \def\AdviceArgs{\advice@noargs}%
2428 }
2429 \def\advice@setup@init@environment{%
2430 \advice@setup@init@common
2431 \edef\AdviceRawCollectorOptions{%
2432 \noexpand\collargsEnvironment{\AdviceName}%
```

When grabbing an environment body, the end-tag will be included. This makes it possible to have the same inner handler for commands and environments.

```
2433 \noexpand\collargsEndTagtrue
2434 }%
2435 \def\AdviceArgs{+b}%
2436 }
```

We need to initialize `\AdviceOuterHandler` etc. so that `\advice@setup@store` will work.

```
2437 \advice@setup@init@command
```

The configuration storage The remaining macros in this subsection deal with the configuration storage space, which is set up in a way to facilitate fast loading during the execution of handled commands and environments.

The configuration of a command or environment is stored in two parts: the first stage settings comprise the run conditions, the bailout handler and the outer handler; the second stage settings contain the rest. When a handled command is invoked, only the first stage settings are immediately loaded, for speed; the second stage settings are only loaded if the run conditions are satisfied.

`\advice@init@i` The two-stage settings are stored in control sequences `\advice@i<namespace>//<storage name>` and `\advice@I<namespace>//<storage name>`, respectively, and accessed using macros `\advice@init@i` and `\advice@init@I`.

Each setting storage macro contains a sequence of items, where each item is either of form `\def\AdviceSetting{<value>}`. This allows us store multiple settings in a single macro (rather than define each control-sequence-valued setting separately, which would use more string memory), and also has the consequence that we don't require the handlers to be defined when submitting a command (whether that's good or bad could be debated: as things stand, any typos in handler declarations will only yield an error once the handled command is executed).

```
2438 \def\advice@init@i#1#2{\csname advice@i#1//\string#2\endcsname}
2439 \def\advice@init@I#1#2{\csname advice@I#1//\string#2\endcsname}
```

We make a copy of these for setup; the originals might be swapped for tracing purposes.

```
2440 \let\advice@setup@init@i\advice@init@i
2441 \let\advice@setup@init@I\advice@init@I
```

`\advice@setup@save` To save the configuration at the end of the setup, we construct the storage macros out of `\AdviceRunConditions` and friends. Stage-one contains only `\AdviceRunConditions` and `\AdviceBailoutHandler`, so that `\advice@handle` can bail out as quickly as possible if the run conditions are not met.

```
2442 \def\advice@setup@save#1#2{%
2443   \expanded{%
```

Close the group before saving. Note that `\expanded` has already expanded the settings macros.

```
2444   \endgroup
2445   \noexpand\csdef{advice@i#1//\string#2}{%
2446     \def\noexpand\AdviceRunConditions{\expandonce\AdviceRunConditions}%
2447     \def\noexpand\AdviceBailoutHandler{\expandonce\AdviceBailoutHandler}%
2448   }%
2449   \noexpand\csdef{advice@I#1//\string#2}{%
2450     \def\noexpand\AdviceOuterHandler{\expandonce\AdviceOuterHandler}%
2451     \def\noexpand\AdviceCollector{\expandonce\AdviceCollector}%
2452     \def\noexpand\AdviceRawCollectorOptions{%
2453       \expandonce\AdviceRawCollectorOptions}%
2454     \def\noexpand\AdviceCollectorOptions{\expandonce\AdviceCollectorOptions}%
2455     \def\noexpand\AdviceArgs{\expandonce\AdviceArgs}%
2456     \def\noexpand\AdviceInnerHandler{\expandonce\AdviceInnerHandler}%
2457     \def\noexpand\AdviceOptions{\expandonce\AdviceOptions}%
2458   }%
2459   \expandonce{\AdviceAfterSetup}%
2460 }%
2461 }
```

activation/immediate These two subkeys of `/advice/activation` install the immediate and the deferred activation code into the installation keypath. They are invoked by key `<installation keypath>/activation=<type>`.

Under the deferred activation regime, the commands are not (de)activated right away. Rather, the (de)activation calls are collected in style `activate deferred`, which should be

executed by the installation keypath owner, if and when they so desire. (Be sure to switch to `activation=immediate` before executing `activate deferred`, otherwise the activation will only be deferred once again.)

```

2462 \pgfkeys{
2463   /advice/activation/deferred/.style={
2464     #1/activate/.style={%
2465       activate deferred/.append style={#1/activate={##1}}},
2466     #1/deactivate/.style={%
2467       activate deferred/.append style={#1/deactivate={##1}}},
2468     #1/force activate/.style={%
2469       activate deferred/.append style={#1/force activate={##1}}},
2470     #1/try activate/.style={%
2471       activate deferred/.append style={#1/try activate={##1}}},
2472   },

```

activate The “real,” immediate `activate` and `deactivate` take a comma-separated list of commands or environments and (de)activate them. If `try activate` is in effect, no error is thrown upon failure.

deactivate If `force activate` is in effect, activation proceeds even if we already had the original definition; it does not apply to deactivation. These conditionals are set to false after every invocation of key (de)activate, so that they only apply to the immediately following (de)activate. (`#1` below is the `<namespace>`; `##1` is the list of commands to be (de)activated.)

```

2473   /advice/activation/immediate/.style={
2474     #1/activate/.code={%
2475       \forcsvlist{\advice@activate{#1}}{##1}%
2476       \advice@activate@forcefalse
2477       \advice@activate@tryfalse
2478     },
2479     #1/deactivate/.code={%
2480       \forcsvlist{\advice@deactivate{#1}}{##1}%
2481       \advice@activate@forcefalse
2482       \advice@activate@tryfalse
2483     },
2484     #1/force activate/.is if=advice@activate@force,
2485     #1/try activate/.is if=advice@activate@try,
2486   },
2487 }
2488 \newif\ifadvice@activate@force
2489 \newif\ifadvice@activate@try

```

\advice@original@csname Activation replaces the original meaning of the handled command with our definition. We **\advice@original@cs** store the original definition into control sequence `\advice@o<namespace>//<storage name>` **\AdviceGetOriginal** (with a `\stringified <storage name>`). Internally, during (de)activation and handling, we access it using `\advice@original@csname` and `\advice@original@cs`. Publicly it should always be accessed by `\AdviceGetOriginal`, which returns the argument control sequence if that control sequence is not handled.

Using the internal command outside the handling context, we could fall victim to scenario such as the following. When we memoize something containing a `\label`, the produced cc-memo contains code eventually executing the original `\label`. If we called the original `\label` via the internal macro there, and the user `deactivated \label` on a subsequent compilation, the cc-memo would not call `\label` anymore, but `\relax`, resulting in a silent error. Using `\AdviceGetOriginal`, the original `\label` will be executed even when not activated.

However, not all is bright with `\AdviceGetOriginal`. Given an activated control sequence (`#2`), a typo in the namespace argument (`#1`) will lead to an infinite loop upon the execution of `\AdviceGetOriginal`. In the manual, we recommend defining a namespace-specific macro to avoid such typos.

```

2490 \def\advice@original@csname#1#2{advice@o#1//\string#2}
2491 \def\advice@original@cs#1#2{\csname advice@o#1//\string#2\endcsname}

```

```

2492 \def\AdviceGetOriginal#1#2{%
2493   \ifcsname advice@o#1//\string#2\endcsname
2494     \expandonce{\csname advice@o#1//\string#2\expandafter\endcsname\expandafter}%
2495   \else
2496     \unexpanded\expandafter{\expandafter#2\expandafter}%
2497   \fi
2498 }

```

\AdviceCsnameGetOriginal A version of `\AdviceGetOriginal` which accepts a control sequence name as the second argument.

```

2499 \begingroup
2500 \catcode`\/=0
2501 \catcode`\=12
2502 /gdef/advice@backslash@other{\}%
2503 /endgroup
2504 \def\AdviceCsnameGetOriginal#1#2{%
2505   \ifcsname advice@o#1//\advice@backslash@other#2\endcsname
2506     \expandonce{\csname advice@o#1//\advice@backslash@other#2\expandafter\endcsname
2507       \expandafter}%
2508   \else
2509     \expandonce{\csname#2\expandafter\endcsname\expandafter}%
2510   \fi
2511 }

```

\advice@activate These macros execute either the command, or the environment (de)activator.

```

\advice@deactivate
2512 \def\advice@activate#1#2{%
2513   \collargs@cs@cases{#2}%
2514   {\advice@activate@cmd{#1}{#2}}%
2515   {\advice@error@activate@notcsorenv{#1}}%
2516   {\advice@activate@env{#1}{#2}}%
2517 }
2518 \def\advice@deactivate#1#2{%
2519   \collargs@cs@cases{#2}%
2520   {\advice@deactivate@cmd{#1}{#2}}%
2521   {\advice@error@activate@notcsorenv{de}{#1}}%
2522   {\advice@deactivate@env{#1}{#2}}%
2523 }

```

\advice@activate@cmd We are very careful when we're activating a command, because activating means rewriting its original definition. Configuration by `auto` did not touch the original command; activation will. So, the leitmotif of this macro: safety first. (`#1` is the namespace, and `#2` is the command to be activated.)

```

2524 \def\advice@activate@cmd#1#2{%
    Is the command defined?
2525   \ifdef{#2}{%
    Yes, the command is defined. Let's see if it's safe to activate it. We'll do this by checking whether
    we have its original definition in our storage. If we do, this means that we have already activated
    the command. Activating it twice would lead to the loss of the original definition (because the
    second activation would store our own redefinition as the original definition) and consequently
    an infinite loop (because once — well, if — the handler tries to invoke the original command, it
    will execute itself all over).
2526     \ifcsdef{\advice@original@csname{#1}{#2}}{%

```

Yes, we have the original definition, so the safety check failed, and we shouldn't activate again. Unless ... how does its current definition look like?

```

2527     \advice@if@our@definition{#1}{#2}{%

```


Well, the current definition of the command matches what we would put there ourselves. The command is definitely activated, and we refuse to activate again, as that would destroy the original definition.

```
2528     \advice@activate@error@activated{#1}{#2}{Command}{already}%
2529     }{%
```

We don't recognize the current definition as our own code (despite the fact that we have surely activated the command before, given the result of the first safety check). It appears that someone else was playing fast and loose with the same command, and redefined it after our activation. (In fact, if that someone else was another instance of Advice, from another namespace, forcing the activation will result in the loss of the original definition and the infinite loop.) So it *should* be safe to activate it (again) ... but we won't do it unless the user specifically requested this using `force activate`. Note that without `force activate`, we would be stuck in this branch, as we could neither activate (again) nor deactivate the command.

```
2530     \ifadvice@activate@force
2531     \advice@activate@cmd@do{#1}{#2}%
2532     \else
2533     \advice@activate@error@activated{#1}{#2}{Command}{already}%
2534     \fi
2535     }%
2536     }{%
```

No, we don't have the command's original definition, so it was not yet activated, and we may activate it.

```
2537     \advice@activate@cmd@do{#1}{#2}%
2538     }%
2539     }{%
2540     \advice@activate@error@undefined{#1}{#2}{Command}{}%
2541     }%
2542 }
```

`\advice@deactivate@cmd` The deactivation of a command follows the same template as activation, but with a different logic, and of course a different effect. In order to deactivate a command, both safety checks discussed above must be satisfied: we must have the command's original definition, *and* our redefinition must still reside in the command's control sequence — the latter condition prevents overwriting someone else's redefinition with the original command. As both conditions must be unavoidably fulfilled, `force activate` has no effect in deactivation (but `try activate` has).

```
2543 \def\advice@deactivate@cmd#1#2{%
2544   \ifdef{#2}{%
2545     \ifcsdef{\advice@original@csname{#1}{#2}}{%
2546       \advice@if@our@definition{#1}{#2}{%
2547         \advice@deactivate@cmd@do{#1}{#2}%
2548       }{%
2549         \advice@deactivate@error@changed{#1}{#2}%
2550       }%
2551     }{%
2552       \advice@activate@error@activated{#1}{#2}{Command}{not yet}%
2553     }%
2554   }{%
2555     \advice@activate@error@undefined{#1}{#2}{Command}{de}%
2556   }%
2557 }
```

`\advice@if@our@definition` This macro checks whether control sequence #2 was already activated (in namespace #1) in the sense that its current definition contains the code our activation would put there: `\advice@handle{#1}{#2}` (protected).

```

2558 \def\advice@if@our@definition#1#2{%
2559   \protected\def\advice@temp{\advice@handle{#1}{#2}}%
2560   \ifx#2\advice@temp
2561     \expandafter\@firstoftwo
2562   \else
2563     \expandafter\@secondoftwo
2564   \fi
2565 }

```

`\advice@activate@cmd@do` This macro saves the original command, and redefines its control sequence. Our redefinition must be `\protected` — even if the original command wasn’t fragile, our replacement certainly is. (Note that as we require ε -TeX anyway, we don’t have to pay attention to L^AT_EX’s robust commands by redefining their “inner” command. Protecting our replacement suffices.)

```

2566 \def\advice@activate@cmd@do#1#2{%
2567   \cslet{\advice@original@csname{#1}{#2}}#2%
2568   \protected\def#2{\advice@handle{#1}{#2}}%
2569   \PackageInfo{advice (#1)}{Activated command "\string#2"}%
2570 }

```

`\advice@deactivate@cmd@do` This macro restores the original command, and removes its definition from our storage — this also serves as a signal that the command is not activated anymore.

```

2571 \def\advice@deactivate@cmd@do#1#2{%
2572   \letcs#2{\advice@original@csname{#1}{#2}}%
2573   \csundef{\advice@original@csname{#1}{#2}}%
2574   \PackageInfo{advice (#1)}{Deactivated command "\string#2"}%
2575 }

```

8.1.3 Executing a handled command

`\advice@handle` An invocation of this macro is what replaces the original command and runs the whole shebang. The system is designed to bail out as quickly as necessary if the run conditions are not met (plus L^AT_EX’s `\begin` will receive a very special treatment for this reason).

We first check the run conditions, and bail out if they are not satisfied. Note that only the stage-one config is loaded at this point. It sets up the following macros (while they are public, neither the end user nor the installation keypath owner should ever have to use them):

- `\AdviceRunConditions` executes `\AdviceRuntrue` if the command should be handled; set by `run conditions`.
- `\AdviceBailoutHandler` will be executed if the command will not be handled, after all; set by `bailout handler`.

```

2576 \def\advice@handle#1#2{%
2577   \advice@init@i{#1}{#2}%
2578   \AdviceRunfalse
2579   \AdviceRunConditions
2580   \advice@handle@rc{#1}{#2}%
2581 }

```

`\advice@handle@rc` We continue the handling in a new macro, because this is the point where the handler for `\begin` will hack into the regular flow of events.

```

2582 \def\advice@handle@rc#1#2{%
2583   \ifAdviceRun
2584     \expandafter\advice@handle@outer
2585   \else

```

Bailout is simple: we first execute the handler, and then the original command.

```

2586     \AdviceBailoutHandler
2587     \expandafter\advice@original@cs

```

```

2588 \fi
2589 {#1}{#2}%
2590 }

```

`\advice@handle@outer` To actually handle the command, we first setup some macros:

- `\AdviceNamespace` holds the installation keypath / storage name space.
- `\AdviceName` holds the control sequence of the handled command, or the environment name.
- `\AdviceReplaced` holds the “substituted” code. For commands, this is the same as `\AdviceName`. For environment `foo`, it equals `\begin{foo}` in L^AT_EX, `\foo` in plain T_EX and `\startfoo` in ConT_EXt.
- `\AdviceOriginal` executes the original definition of the handled command or environment.

```

2591 \def\advice@handle@outer#1#2{%
2592   \def\AdviceNamespace{#1}%
2593   \def\AdviceName{#2}%
2594   \advice@def@AdviceCname
2595   \let\AdviceReplaced\AdviceName
2596   \def\AdviceOriginal{\AdviceGetOriginal{#1}{#2}}%

```

We then load the stage-two settings. This defines the following macros:

- `\AdviceOuterHandler` will effectively replace the command, if it will be handled; set by `outer handler`.
- `\AdviceCollector` collects the arguments of the handled command, perhaps consulting `\AdviceArgs` to learn about its argument structure.
- `\AdviceRawCollectorOptions` contains the options which will be passed to the argument collector, in the “raw” format.
- `\AdviceCollectorOptions` contains the additional, user-specified options which will be passed to the argument collector.
- `\AdviceArgs` contains the `xparse`-style argument specification of the command, or equals `\advice@noargs` to signal that command was defined using `xparse` and that the argument specification should be retrieved automatically.
- `\AdviceInnerHandler` is called by the argument collector once it finishes its work. It receives all the collected arguments as a single (braced) argument.
- `\AdviceOptions` holds options which may be used by the outer or the inner handler; Advice does not need or touch them.

```

2597 \advice@init@I{#1}{#2}%

```

All prepared, we execute the outer handler.

```

2598 \AdviceOuterHandler
2599 }
2600 \def\advice@def@AdviceCname{%
2601   \begingroup
2602   \escapechar=-1
2603   \expandafter\expandafter\expandafter\endgroup
2604   \expandafter\expandafter\expandafter\def
2605   \expandafter\expandafter\expandafter\AdviceCname
2606   \expandafter\expandafter\expandafter{\expandafter\string\AdviceName}%
2607 }

```

`\ifAdviceRun` This conditional is set by the run conditions macro to signal whether we should run the outer (true) or the bailout (false) handler.

```

2608 \newif\ifAdviceRun

```

`\advice@CollectArgumentsRaw` This is the default collector, which will collect the argument using `CollArgs`’ command `\CollectArgumentsRaw`. It will provide that command with:

- the collector options, given in the raw format:
 - the caller (`\collargsCaller`),

- the raw options (`\AdviceRawCollectorOptions`), and
- the user options (`\AdviceRawCollectorOptions`, wrapped in `\collargsSet`;
- the argument specification `\AdviceArgs` of the handled command; and
- the inner handler `\AdviceInnerHandler` to execute after collecting the arguments; the inner handler receives the collected arguments as a single braced argument.

If the argument specification is not defined (either the user did not set it, or has reset it by writing `args` without a value), it is assumed that the handled command was defined by `xparse` and `\AdviceArgs` will be retrieved by `\GetDocumentCommandArgSpec`.

```

2609 \def\advice@CollectArgumentsRaw{%
2610   \AdviceIfArgs{}{%
2611     \expandafter\GetDocumentCommandArgSpec\expandafter{\AdviceName}%
2612     \let\AdviceArgs\ArgumentSpecification
2613   }%
2614   \expanded{%
2615     \noexpand\CollectArgumentsRaw{%
2616       \noexpand\collargsCaller{\expandonce\AdviceName}%
2617       \expandonce\AdviceRawCollectorOptions
2618       \ifdefempty\AdviceCollectorOptions{}{%
2619         \noexpand\collargsSet{\expandonce\AdviceCollectorOptions}%
2620       }%
2621     }%
2622     {\expandonce\AdviceArgs}%
2623     {\expandonce\AdviceInnerHandler}%
2624   }%
2625 }

```

`\AdviceIfArgs` If the value of `args` is “real”, i.e. an `xparse` argument specification, execute the first argument. If `args` was set to the special value `\advice@noargs`, signaling a command defined by `\NewDocumentCommand` or friends, execute the second argument. (Ok, in reality anything other than `\advice@noargs` counts as real “real”.)

```

2626 \def\advice@noargs@text{\advice@noargs}
2627 \def\AdviceIfArgs{%
2628   \ifx\AdviceArgs\advice@noargs@text
2629     \expandafter\@secondoftwo
2630   \else
2631     \expandafter\@firstoftwo
2632   \fi
2633 }

```

`\advice@pgfkeys@collector` A `pgfkeys` collector is very simple: the sole argument of the any key macro, regardless of the argument structure of the key, is everything up to `\pgfeov`.

```

2634 \def\advice@pgfkeys@collector#1\pgfeov{%
2635   \AdviceInnerHandler{#1}%
2636 }

```

8.1.4 Environments

`\advice@activate@env` Things are simple in `TEX` and `ConTEXt`, as their environments are really commands. So `\advice@deactivate@env` rather than activating environment name `#2`, we (de)activate command `\#2` or `\start#2`, depending on the format.

```

2637   <plain, context>
2638 \def\advice@activate@env#1#2{%
2639   \expanded{%
2640     \noexpand\advice@activate@cmd{#1}{\expandonce{\csname
2641   <context>       start%
2642     #2\endcsname}}%
2643   }%
2644 }

```

```

2645 \def\advice@deactivate@env#1#2{%
2646   \expanded{%
2647     \noexpand\advice@deactivate@cmd{#1}{\expandonce{\csname
2648 (context)       start%
2649                 #2\endcsname}}}%
2650   }%
2651 }
2652 </plain, context>

```

We activate commands by redefining them; that's the only way to do it. But we won't activate a L^AT_EX environment `foo` by redefining command `\foo`, where the user's definition for the start of the environment actually resides, as such a redefinition would be executed too late, deep within the group opened by `\begin`, following many internal operations and public hooks. We handle L^AT_EX environments by defining an outer handler for `\begin` (consequently, L^AT_EX environment support can be (de)activated by the user by saying `(de)activate=\begin`), and activating an environment will be nothing but setting a mark, by defining a dummy control sequence `\advice@original@csname{#1}{#2}`, which that handler will inspect. Note that `force activate` has no effect here.

```

2653 (*latex)
2654 \def\advice@activate@env#1#2{%
2655   \ifcsdef{\advice@original@csname{#1}{#2}}{%
2656     \advice@activate@error@activated{#1}{#2}{Environment}{already}%
2657   }{%
2658     \csdef{\advice@original@csname{#1}{#2}}{%
2659       \PackageInfo{advice (#1)}{Activated environment "#2"}%
2660     }%
2661   }
2662 \def\advice@deactivate@env#1#2{%
2663   \ifcsdef{\advice@original@csname{#1}{#2}}{%
2664     \csundef{\advice@original@csname{#1}{#2}}{%
2665   }{%
2666     \advice@activate@error@activated{#1}{#2}{Environment}{not yet}%
2667     \PackageInfo{advice (#1)}{Deactivated environment "#2"}%
2668   }%
2669 }

```

`\advice@begin@rc` This is the handler for `\begin`. It is very special, for speed. It is meant to be declared as the run conditions component, and it hacks into the normal flow of handling. It knows that after executing the run conditions macro, `\advice@handle` eventually (the tracing info may interrupt here as `#1`) continues by `\advice@handle@rc{\namespace}{\handled control sequence}`, so it grabs all these (`#2` is the `\namespace` and `#3` is the `\handled control sequence`), i.e. `\begin`) plus the environment name (`#4`).

```

2670 \def\advice@begin@rc#1\advice@handle@rc#2#3#4{%

```

We check whether environment `#4` is activated (in namespace `#2`) by inspecting whether activation dummy is defined. If it is not, we execute the original `\begin` (`\advice@original@cs{#2}{#3}`), followed by the environment name (`#4`). Note that we *don't* execute the environment's bailout handler here: we haven't checked its run conditions yet, as the environment is simply not activated.

```

2671   \ifcsname\advice@original@csname{#2}{#4}\endcsname
2672     \expandafter\advice@begin@env@rc
2673   \else
2674     \expandafter\advice@original@cs
2675   \fi
2676   {#2}{#3}{#4}%
2677 }

```

`\advice@begin@env@rc` Starting from this point, we essentially replicate the workings of `\advice@handle`, adapted to L^AT_EX environments.

```
2678 \def\advice@begin@env@rc#1#2#3{%
```

We first load the stage-one configuration for environment #3 in namespace #1.

```
2679 \advice@init@i{#1}{#3}%
```

This defined `\AdviceRunConditions` for the environment. We can now check its run conditions. If they are not satisfied, we bail out by executing the environment's bailout handler followed by the original `\begin` (`\advice@original@cs{#1}{#2}`) plus the environment name (#3).

```
2680 \AdviceRunConditions
2681 \ifAdviceRun
2682 \expandafter\advice@begin@env@outer
2683 \else
2684 \AdviceBailoutHandler
2685 \expandafter\advice@original@cs
2686 \fi
2687 {#1}{#2}{#3}%
2688 }
```

`\advice@begin@env@outer` We define the macros expected by the outer handler, see `\advice@handle@outer`, load the second-stage configuration, and execute the environment's outer handler.

```
2689 \def\advice@begin@env@outer#1#2#3{%
2690 \def\AdviceNamespace{#1}%
2691 \def\AdviceName{#3}%
2692 \let\AdviceCsname\advice@undefined
2693 \def\AdviceReplaced{#2{#3}}%
2694 \def\AdviceOriginal{\AdviceGetOriginal{#1}{#2}{#3}}%
2695 \advice@init@I{#1}{#3}%
2696 \AdviceOuterHandler
2697 }
2698 {//latex}
```

8.1.5 Error messages

Define error messages for the entire package. Note that `\advice@(de)activate@error@...` implement try activate.

```
2699 \def\advice@activate@error@activated#1#2#3#4{%
2700 \ifadvice@activate@try
2701 \else
2702 \PackageError{advice (#1)}{#3 "\string#2" is #4 activated}{}%
2703 \fi
2704 }
2705 \def\advice@activate@error@undefined#1#2#3#4{%
2706 \ifadvice@activate@try
2707 \else
2708 \PackageError{advice (#1)}{#3 "\string#2" you are trying to #4activate is not defined}{}%
2709 \fi
2710 }
2711 }
2712 \def\advice@deactivate@error@changed#1#2{%
2713 \ifadvice@activate@try
2714 \else
2715 \PackageError{advice (#1)}{The definition of "\string#2" has changed since we
2716 have activated it. Has somebody overridden our command?}{If you have tried
2717 to deactivate so that you could immediately reactivate, you may want to try
2718 "force activate".}%
2719 \fi
```

```

2720 }
2721 \def\advice@error@advice@notcs#1#2{%
2722   \PackageError{advice}{The first argument of key "#1" should be either a single
2723     control sequence or an environment name, not "#2"}{}%
2724 }
2725 \def\advice@error@activate@notcsorenv#1#2{%
2726   \PackageError{advice}{Each item in the value of key "#1activate" should be
2727     either a control sequence or an environment name, not "#2".}{}%
2728 }
2729 \def\advice@error@storecs@notcs#1#2{%
2730   \PackageError{advice}{The value of key "#1" should be a single control sequence,
2731     not "\string#2"}{}%
2732 }
2733 \def\advice@error@noinnerhandler#1{%
2734   \PackageError{advice (\AdviceNamespace)}{The inner handler for
2735     "\expandafter\string\AdviceName" is not defined}{}%
2736 }

```

8.1.6 Tracing

We implement tracing by adding the tracing information to the handlers after we load them. So it is the handlers themselves which, if and when they are executed, will print out that this is happening.

`\AdviceTracingOn` Enable and disable tracing.

```

\AdviceTracingOff
2737 \def\AdviceTracingOn{%
2738   \let\advice@init@i\advice@trace@init@i
2739   \let\advice@init@I\advice@trace@init@I
2740 }
2741 \def\AdviceTracingOff{%
2742   \let\advice@init@i\advice@setup@init@i
2743   \let\advice@init@I\advice@setup@init@I
2744 }

```

`\advice@typeout` The tracing output routine; the typeout macro depends on the format. In \LaTeX , we use stream `\advice@trace` `\@unused`, which is guaranteed to be unopened, so that the output will go to the terminal and the log. ConTeXt, we don't muck about with write streams but simply use Lua function `texio.write_nl`. In plain \TeX , we use either Lua or the stream, depending on the engine; we use a high stream number 128 although the good old 16 would probably work just as well.

```

2745 <plain> \ifdefined\luatexversion
2746 <!latex> \long\def\advice@typeout#1{\directlua{texio.write_nl("\luaescapestring{#1}")}}
2747 <plain> \else
2748 <latex> \def\advice@typeout{\immediate\write\@unused}
2749 <plain> \def\advice@typeout{\immediate\write128}
2750 <plain> \fi
2751 \def\advice@trace#1{\advice@typeout{[tracing advice] #1}}

```

`\advice@trace@init@i` Install the tracing code.

```

\advice@trace@init@I
2752 \def\advice@trace@init@i#1#2{%
2753   \advice@trace{Advising \detokenize\expandafter{\string#2} (\detokenize{#1})}%
2754   \advice@trace{\space\space Original command meaning:
2755     \expandafter\expandafter\expandafter\meaning\advice@original@cs{#1}{#2}}%
2756   \advice@setup@init@i{#1}{#2}%
2757   \edef\AdviceRunConditions{%

```

We first execute the original run conditions, so that we can show the result.

```

2758   \expandonce\AdviceRunConditions
2759   \noexpand\advice@trace{\space\space
2760     Executing run conditions:

```

```

2761     \detokenize\expandafter{\AdviceRunConditions}
2762     -->
2763     \noexpand\ifAdviceRun true\noexpand\else false\noexpand\fi
2764   }%
2765 }%
2766 \edef\AdviceBailoutHandler{%
2767   \noexpand\advice@trace{\space\space
2768     Executing bailout handler:
2769     \detokenize\expandafter{\AdviceBailoutHandler}}%
2770   \expandonce\AdviceBailoutHandler
2771 }%
2772 }
2773 \def\advice@trace@init@I#1#2{%
2774   \advice@setup@init@I{#1}{#2}%
2775   \edef\AdviceOuterHandler{%
2776     \noexpand\advice@trace{\space\space
2777       Executing outer handler:
2778       \detokenize\expandafter{\AdviceOuterHandler}}%
2779     \expandonce\AdviceOuterHandler
2780 }%
2781   \edef\AdviceCollector{%
2782     \noexpand\advice@trace{\space\space
2783       Executing collector:
2784       \detokenize\expandafter{\AdviceCollector}}%
2785     \noexpand\advice@trace{\space\space\space\space
2786       Argument specification:
2787       \detokenize\expandafter{\AdviceArgs}}%
2788     \noexpand\advice@trace{\space\space\space\space
2789       Options:
2790       \detokenize\expandafter{\AdviceCollectorOptions}}%
2791     \noexpand\advice@trace{\space\space\space\space
2792       Raw options:
2793       \detokenize\expandafter{\AdviceRawCollectorOptions}}%

```

Collargs' return complicates tracing of the received argument. We put the code for remembering its value among the raw collector options. The default is 0; it is needed when we're using a collector other than `\CollectArguments`, the assumption being that external collectors will always return the collected arguments braced.

```

2794   \unexpanded{%
2795     \gdef\advice@collargs@return{0}%
2796     \appto\AdviceRawCollectorOptions{\advice@remember@collargs@return}%
2797   }%
2798   \expandonce\AdviceCollector
2799 }%
2800 \edef\advice@inner@handler@trace@do{%
2801   \noexpand\advice@trace{\space\space
2802     Executing inner handler:
2803     \detokenize\expandafter{\AdviceInnerHandler}}%

```

When this macro is executed, the received arguments are waiting for us in `\toks0`.

```

2804   \noexpand\advice@trace{\space\space\space\space
2805     Received arguments\noexpand\advice@inner@handler@trace@printcollargsreturn:
2806     \noexpand\detokenize\noexpand\expandafter{\unexpanded{\the\toks0}}}%
2807   \noexpand\advice@trace{\space\space\space\space
2808     Options:
2809     \detokenize\expandafter{\AdviceOptions}}%
2810   \expandonce{\AdviceInnerHandler}%
2811 }%
2812 \def\AdviceInnerHandler{\advice@inner@handler@trace}%
2813 }
2814 \def\advice@remember@collargs@return{%

```



```

2815 \global\let\advice@collargs@return\collargsReturn
2816 }

```

This is the entry point into the tracing inner handler. It will either get the received arguments as a braced argument (when Collargs' return=0), or from \collargsArgs otherwise. We don't simply always inspect \collargsArgs because foreign argument collectors will not use this token register; the assumption is that they will always return the collected arguments braced.

```

2817 \def\advice@inner@handler@trace{%
2818   \ifnum\advice@collargs@return=0
2819     \expandafter\advice@inner@handler@trace@i
2820   \else
2821     \expandafter\advice@inner@handler@trace@ii
2822   \fi
2823 }
2824 \def\advice@inner@handler@trace@i#1{%
2825   \toks0={#1}%
2826   \advice@inner@handler@trace@do{#1}%
2827 }
2828 \def\advice@inner@handler@trace@ii{%
2829   \expandafter\toks\expandafter0\expandafter{\the\collargsArgs}%
2830   \advice@inner@handler@trace@do
2831 }
2832 \def\advice@inner@handler@trace@printcollargsreturn{%
2833   \ifnum\advice@collargs@return=0
2834   \else
2835     \space(collargs return=%
2836     \ifcase\advice@collargs@return braced\or plain\or no\fi
2837   )%
2838   \fi
2839 }

```

```

2840 <plain>\resetatcatcode
2841 <context>\stopmodule
2842 <context>\protect
2843 </main>

```

8.1.7 The TikZ collector

In this section, we implement the argument collector for command \tikz, which has idiosyncratic syntax, see §12.2.2 of the TikZ & PGF manual:

- \tikz<animation spec>[<options>]{<picture code>}
- \tikz<animation spec>[<options>]<picture command>;

where <animation spec> = (:<key>={<value>})*.

The TikZ code resides in a special file. It is meant to be \input at any time, so we need to temporarily assign @ category code 11.

```

2844 <*tikz>
2845 \edef\advice@resetatcatcode{\catcode`\noexpand\@the\catcode`\@relax}%
2846 \catcode`\@=11
2847 \def\AdviceCollectTikZArguments{%

```

We initialize the token register which will hold the collected arguments, and start the collection. Nothing of note happens until ...

```

2848   \toks0={}%
2849   \advice@tikz@anim
2850 }
2851 \def\advice@tikz@anim{%
2852   \pgfutil@ifnextchar[{\advice@tikz@opt}{%
2853     \pgfutil@ifnextchar:{\advice@tikz@anim@a}{%
2854       \advice@tikz@code}}%

```

```

2855 }
2856 \def\advice@tikz@anim@a#1=#2{%
2857   \toksapp0{#1={#2}}}%
2858   \advice@tikz@anim
2859 }
2860 \def\advice@tikz@opt[#1]{%
2861   \toksapp0{[#1]}%
2862   \advice@tikz@code
2863 }
2864 \def\advice@tikz@code{%
2865   \pgfutil@ifnextchar\bgroup\advice@tikz@braced\advice@tikz@single
2866 }
2867 \long\def\advice@tikz@braced#1{\toksapp0{#1}\advice@tikz@done}
2868 \def\advice@tikz@single#1;{\toksapp0{#1;}\advice@tikz@done}

```

... we finish collecting the arguments, when we execute the inner handler, with the (braced) collected arguments is its sole argument.

```

2869 \def\advice@tikz@done{%
2870   \expandafter\AdviceInnerHandler\expandafter{\the\toks0}%
2871 }
2872 \adviceresetatcatcode
2873 </tikz>

```

Local Variables: TeX-engine: luatex TeX-master: "doc/memoize-code.tex" TeX-auto-save: nil End:

8.2 Argument collection with CollArgs

Package CollArgs provides commands `\CollectArguments` and `\CollectArgumentsRaw`, which (what a surprise!) collect the arguments conforming to the given (slightly extended) `xparse` argument specification. The package was developed to help out with automemoization (see section 5). It started out as a few lines of code, but had grown once I realized I want automemoization to work for verbatim environments as well — the environment-collecting code is based on Bruno Le Floch’s package `cprotect` — and had then grown some more once I decided to support the `xparse` argument specification in full detail, and to make the verbatim mode flexible enough to deal with a variety of situations.

The implementation of this package does not depend on `xparse`. Perhaps this is a mistake, especially as the `xparse` code is now included in the base \LaTeX , but the idea was to have a light-weight package (not sure this is the case anymore, given all the bells and whistles), to have its functionality available in plain \TeX and \ConTeXt as well (same as Memoize), and, perhaps most importantly, to have the ability to collect the arguments verbatim.

Identification

```

2874 <latex>\ProvidesPackage{collargs}[2024/03/15 v1.2.0 Collect arguments of any command]
2875 <context>%D \module[
2876 <context>%D     file=t-collargs.tex,
2877 <context>%D     version=1.2.0,
2878 <context>%D     title=CollArgs,
2879 <context>%D     subtitle=Collect arguments of any command,
2880 <context>%D     author=Saso Zivanovic,
2881 <context>%D     date=2024-03-15,
2882 <context>%D     copyright=Saso Zivanovic,
2883 <context>%D     license=LPPL,
2884 <context>%D ]
2885 <context>\writestatus{loading}{ConTeXt User Module / collargs}
2886 <context>\unprotect
2887 <context>\startmodule[collargs]

```

Required packages

```

2888 <latex>\RequirePackage{pgfkeys}

```

```

2889 <plain>\input pgfkeys
2890 <context>\input t-pgfkey
2891 <latex>\RequirePackage{etoolbox}
2892 <plain, context>\input etoolbox-generic
2893 <plain>\edef\resetatcatcode{\catcode`\noexpand\@the\catcode`\@relax}
2894 <plain>\catcode`\@11\relax

```

`\toksapp` Macros for appending to a token register. We don't have to define them in LuaTeX, where they exist as primitives. Same as these primitives, our macros accept either a register number or a `\etoksapp` `\toksdeffed` control sequence as the (unbraced) #1; #2 is the text to append.

```

<xtoksapp>
2895 \ifdefined\luatexversion
2896 \else
2897 \def\toksapp{\toks@cs@or@num\@toksapp}
2898 \def\gtoksapp{\toks@cs@or@num\@gtoksapp}
2899 \def\etoksapp{\toks@cs@or@num\@etoksapp}
2900 \def\xtoksapp{\toks@cs@or@num\@xtoksapp}
2901 \def\toks@cs@or@num#1#2#3{%

```

Test whether #2 (the original #1) is a number or a control sequence.

```
2902 \ifnum-2>-1#2
```

It is a number. `\toks@cs@or@num@num` will gobble `\toks@cs@or@num@cs` below.

```
2903 \expandafter\toks@cs@or@num@num
```

The register control sequence in #2 is skipped over in the false branch.

```
2904 \fi
2905 \toks@cs@or@num@cs{#1}{#2}%
2906 }
```

#1 is one of `\@toksapp` and friends. The second macro prefixes the register number by `\toks`.

```
2907 \def\toks@cs@or@num@cs#1#2{#1{#2}}
2908 \def\toks@cs@or@num@num\toks@cs@or@num@cs#1#2{#1{\toks#2 }}

```

Having either `\tokscs` or `\toks<number>` in #1, we can finally do the real job.

```
2909 \long\def\@toksapp#1#2{#1\expandafter{\the#1#2}}%
2910 \long\def\@etoksapp#1#2{#1\expandafter{\expanded{\the#1#2}}}%
2911 \long\def\@gtoksapp#1#2{\global#1\expandafter{\the#1#2}}%
2912 \long\def\@xtoksapp#1#2{\global#1\expandafter{\expanded{\the#1#2}}}%
2913 \fi

```

`\CollectArguments` `\CollectArguments` takes three arguments: the optional #1 is the option list, processed `\CollectArgumentsRaw` by `pgfkeys` (given the grouping structure, these options will apply to all arguments); the mandatory #2 is the `xparse`-style argument specification; the mandatory #3 is the “next” command (or a sequence of commands). The argument list is expected to start immediately after the final argument; `\CollectArguments` parses it, effectively figuring out its extent, and then passes the entire argument list to the “next” command (as a single argument).

`\CollectArgumentsRaw` differs only in how it takes and processes the options. For one, these should be given as a mandatory argument. Furthermore, they do not take the form of a keylist, but should deploy the “programmer’s interface.” #1 should thus be a sequence of invocations of the macro counterparts of the keys defined in section 8.2.1, which can be recognized as starting with `\collargs` followed by a capital letter, e.g. `\collargsCaller`. Note that `\collargsSet` may also be used in #1. (The “optional,” i.e. bracketed, argument of `\CollectArgumentsRaw` is in fact mandatory.)

```

2914 \protected\def\CollectArguments{%
2915 \pgf@keys@utilifnextchar[\CollectArguments@i{\CollectArgumentsRaw{}}]%
2916 }
2917 \def\CollectArguments@i[#1]{\CollectArgumentsRaw{\collargsSet{#1}}
2918 \protected\def\CollectArgumentsRaw#1#2#3{%

```

This group will be closed by `\collargs@.` once we grinded through the argument specification.

```
2919 \beginingroup
```

Initialize category code fixing; see section 8.2.6 for details. We have to do this before applying the settings, so that `\collargsFixFromNoVerbatim` et al can take effect.

```
2920 \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
2921 \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
2922 \global\collargs@double@fixfalse
```

Apply the settings.

```
2923 \collargs@verbatim@wrap{#1}%
```

Initialize the space-grabber.

```
2924 \collargs@init@grabspaces
```

Remember the code to execute after collection.

```
2925 \def\collargs@next{#3}%
```

Initialize the token register holding the collected arguments.

```
2926 \ifcollargsClearArgs
2927   \global\collargsArgs{}%
2928 \fi
```

Execute the central loop macro, which expects the argument specification #2 to be delimited from the following argument tokens by a dot.

```
2929 \collargs@#2.%
2930 }
```

`\collargsSet` This macro processes the given keys in the `/collargs` keypath. When it is used to process options given by the end user (the optional argument to `\CollectArguments`, and the options given within the argument specification, using the new modifier `&`), its invocation should be wrapped in `\collargs@verbatim@wrap` to correctly deal with the changes of the verbatim mode.

```
2931 \def\collargsSet#1{\pgfqkeys{/collargs}{#1}}
```

8.2.1 The keys

`\collargs@cs@cases` If the first argument of this auxiliary macro is a single control sequence, then the second argument is executed. If the first argument starts with a control sequence but this control sequence does not form the entire argument, the third argument is executed. Otherwise, the fourth argument is executed.

This macro is defined in package `CollArgs` because we use it in key `caller` below, but it is really useful in package `Auto`, where having it we don't have to bother the end-user with a separate keys for commands and environments, but automatically detect whether the argument of `auto` and `(de)activate` is a command or an environment.

```
2932 \def\collargs@cs@cases#1{\collargs@cs@cases@i#1\collargs@cs@cases@end}
2933 \let\collargs@cs@cases@end\relax
2934 \def\collargs@cs@cases@i{\futurelet\collargs@temp\collargs@cs@cases@ii}
2935 \def\collargs@cs@cases@ii#1#2\collargs@cs@cases@end{%
2936   \ifcat\noexpand\collargs@temp\relax
2937     \ifx\relax#2\relax
2938       \expandafter\expandafter\expandafter\@firstofthree
2939     \else
2940       \expandafter\expandafter\expandafter\@secondofthree
2941     \fi
```

```

2942 \else
2943 \expandafter\@thirdofthree
2944 \fi
2945 }
2946 \def\@firstofthree#1#2#3{#1}
2947 \def\@secondofthree#1#2#3{#2}
2948 \def\@thirdofthree#1#2#3{#3}

```

`caller` Every macro which grabs a part of the argument list will be accessed through the “caller” control sequence, so that \TeX 's reports of any errors in the argument structure can contain a command name familiar to the author.⁴ For example, if the argument list “originally” belonged to command `\foo` with argument structure `r()`, but no parentheses follow in the input, we want \TeX to complain that `Use of \foo doesn't match its definition`. This can be achieved by setting `caller=\foo`; the default is `caller=\CollectArguments`, which is still better than seeing an error involving some random internal control sequence. It is also ok to set an environment name as the caller, see below.

The key and macro defined below store the caller control sequence into `\collargs@caller`, e.g. when we say `caller=\foo`, we effectively execute `\def\collargs@caller{\foo}`.

```

2949 \collargsSet{
2950 caller/.code={\collargsCaller{#1}},
2951 }
2952 \def\collargsCaller#1{%
2953 \collargs@cs@cases{#1}{%
2954 \let\collargs@temp\collargs@caller@cs
2955 }{%
2956 \let\collargs@temp\collargs@caller@csandmore
2957 }{%
2958 \let\collargs@temp\collargs@caller@env
2959 }%
2960 \collargs@temp{#1}%
2961 }
2962 \def\collargs@caller@cs#1{%

```

If `#1` is a single control sequence, just use that as the caller.

```

2963 \def\collargs@caller{#1}%
2964 }
2965 \def\collargs@caller@csandmore#1{%

```

If `#1` starts with a control sequence, we don't complain, but convert the entire `#1` into a control sequence.

```

2966 \begingroup
2967 \escapechar -1
2968 \expandafter\endgroup
2969 \expandafter\def\expandafter\collargs@caller\expandafter{%
2970 \csname\string#1\endcsname
2971 }%
2972 }
2973 \def\collargs@caller@env#1{%

```

If `#1` does not start with a control sequence, we assume that is an environment name, so we prepend `start` in `ConTeXt`, and dress it up into `\begin{#1}` in \LaTeX .

```

2974 \expandafter\def\expandafter\collargs@caller\expandafter{%
2975 \csname
2976 <context> start%
2977 <latex> begin{%
2978 #1%
2979 <latex> }%

```

⁴The idea is borrowed from package `environ`, which is in turn based on code from `amsmath`.

```

2980     \endcsname
2981   }%
2982 }
2983 \collargsCaller\CollectArguments

```

`\ifcollargs@verbatim` The first of these conditional signals that we're collecting the arguments in one of the `\ifcollargs@verbatimbraces` verbatim modes; the second one signals the `verb` mode in particular.

```

2984 \newif\ifcollargs@verbatim
2985 \newif\ifcollargs@verbatimbraces

```

`verbatim` These keys set the verbatim mode macro which will be executed by `\collargsSet` after `verb` processing all keys. The verbatim mode macros `\collargsVerbatim`, `\collargsVerb` `no verbatim` and `\collargsNoVerbatim` are somewhat complex; we postpone their definition until section 8.2.5. Their main effect is to set conditionals `\ifcollargs@verbatim` and `\ifcollargs@verbatimbraces`, which are to be inspected by the argument type handlers — and to make the requested category code changes, of course.

Here, note that the verbatim-selection code is not executed while the keylist is being processed. Rather, the verbatim keys simply set the macro which will be executed *after* the keylist is processed, and this is why processing of a keylist given by the user must be always wrapped in `\collargs@verbatim@wrap`.

```

2986 \collargsSet{
2987   verbatim/.code={\let\collargs@apply@verbatim\collargsVerbatim},
2988   verb/.code={\let\collargs@apply@verbatim\collargsVerb},
2989   no verbatim/.code={\let\collargs@apply@verbatim\collargsNoVerbatim},
2990 }
2991 \def\collargs@verbatim@wrap#1{%
2992   \let\collargs@apply@verbatim\relax
2993   #1%
2994   \collargs@apply@verbatim
2995 }

```

`fix from verbatim` These keys and macros should be used to request a category code fix, when the offending `fix from verb` tokenization took place prior to invoking `\CollectArguments`; see section 8.2.6 for `fix from no verbatim` details. While I assume that only `\collargsFixFromNoVerbatim` will ever be used `\collargsFixFromVerbatim` (and it is used by `\mmz`), we provide macros for all three transitions, for completeness.

```

\collargsFixFromVerb
\collargsFixFromNoVerbatim
2996 \collargsSet{
2997   fix from verbatim/.code={\collargsFixFromVerbatim},
2998   fix from verb/.code={\collargsFixFromVerb},
2999   fix from no verbatim/.code={\collargsFixFromNoVerbatim},
3000 }

```

```

3001 \def\collargsFixFromNoVerbatim{%
3002   \global\collargs@fix@requestedtrue
3003   \global\let\ifcollargs@last@verbatim\iffalse
3004 }
3005 \def\collargsFixFromVerbatim{%
3006   \global\collargs@fix@requestedtrue
3007   \global\let\ifcollargs@last@verbatim\iftrue
3008   \global\let\ifcollargs@last@verbatimbraces\iftrue
3009 }
3010 \def\collargsFixFromVerb{%
3011   \global\collargs@fix@requestedtrue
3012   \global\let\ifcollargs@last@verbatim\iftrue
3013   \global\let\ifcollargs@last@verbatimbraces\iffalse
3014 }

```

`braces` Set the characters which are used as the grouping characters in the full verbatim mode. The user is only required to do this when multiple character pairs serve as the grouping characters. The underlying macro, `\collargsBraces`, will be defined in section 8.2.5.

```

3015 \collargsSet{
3016   braces/.code={\collargsBraces{#1}}%
3017 }

```

environment Set the environment name.

```

\collargsEnvironment
3018 \collargsSet{
3019   environment/.estore in=\collargs@b@envname
3020 }
3021 \def\collargsEnvironment#1{\edef\collargs@b@envname{#1}}
3022 \collargsEnvironment{}

```

begin tag When `begin tag/end tag` is in effect, the `begin/end-tag` will be will be prepended/ap-
end tag pended to the environment body. `tags` is a shortcut for setting `begin tag` and `end tag`
tags simultaneously.

```

\ifcollargsBeginTag
\ifcollargsEndTag
\ifcollargsAddTags
3023 \collargsSet{
3024   begin tag/.is if=collargsBeginTag,
3025   end tag/.is if=collargsEndTag,
3026   tags/.style={begin tag=#1, end tag=#1},
3027   tags/.default=true,
3028 }
3029 \newif\ifcollargsBeginTag
3030 \newif\ifcollargsEndTag

```

ignore nesting When this key is in effect, we will ignore any `\begin{<name>}`s and simply grab
\ifcollargsIgnoreNesting everything up to the first `\end{<name>}` (again, the markers are automatically adapted
to the format).

```

3031 \collargsSet{
3032   ignore nesting/.is if=collargsIgnoreNesting,
3033 }
3034 \newif\ifcollargsIgnoreNesting

```

ignore other tags This key is only relevant in the non-verbatim and partial verbatim modes in L^AT_EX.

\ifcollargsIgnoreOtherTags When it is in effect, CollArgs checks the environment name following each `\begin`
and `\end`, ignoring the tags with an environment name other than `\collargs@b@envname`.

```

3035 \collargsSet{
3036   ignore other tags/.is if=collargsIgnoreOtherTags,
3037 }
3038 \newif\ifcollargsIgnoreOtherTags

```

(append/prepend) (pre/post)processor These keys and macros populate the list of preprocessors,
\collargs(Append/Prepend) (Pre/Post)processor `\collargs@preprocess@arg`, and the list of post-processors,
`\collargs@postprocess@arg`, executed in `\collargs@appendarg`.

```

3039 \collargsSet{
3040   append preprocessor/.code={\collargsAppendPreprocessor{#1}},
3041   prepend preprocessor/.code={\collargsPrependPreprocessor{#1}},
3042   append postprocessor/.code={\collargsAppendPostprocessor{#1}},
3043   prepend postprocessor/.code={\collargsPrependPostprocessor{#1}},
3044 }
3045 \def\collargsAppendPreprocessor#1{\appto\collargs@preprocess@arg{#1}}
3046 \def\collargsPrependPreprocessor#1{\preto\collargs@preprocess@arg{#1}}
3047 \def\collargsAppendPostprocessor#1{\appto\collargs@postprocess@arg{#1}}
3048 \def\collargsPrependPostprocessor#1{\preto\collargs@postprocess@arg{#1}}

```

clear (pre/post)processors These keys and macros clear the pre- and post-processor lists, which are
\collargsClear(Pre/Post)processors initially empty as well.

```

3049 \def\collargs@preprocess@arg{}

```

```

3050 \def\collargs@postprocess@arg{}
3051 \collargsSet{
3052   clear preprocessors/.code={\collargsClearPreprocessors},
3053   clear postprocessors/.code={\collargsClearPostprocessors},
3054 }
3055 \def\collargsClearPreprocessors{\def\collargs@preprocess@arg{}}%
3056 \def\collargsClearPostprocessors{\def\collargs@postprocess@arg{}}%

```

(append/prepend) expandable (pre/post)processor These keys and macros simplify the definition of expandable processors. Note that expandable processors are added to the same list as non-expandable processors.

```

3057 \collargsSet{
3058   append expandable preprocessor/.code={\collargsAppendExpandablePreprocessor{#1}},
3059   prepend expandable preprocessor/.code={\collargsPrependExpandablePreprocessor{#1}},
3060   append expandable postprocessor/.code={\collargsAppendExpandablePostprocessor{#1}},
3061   prepend expandable postprocessor/.code={\collargsPrependExpandablePostprocessor{#1}},
3062 }
3063 \def\collargsAppendExpandablePreprocessor#1{%
3064   \appto\collargs@preprocess@arg{%
3065     \collargsArg\expandafter{\expanded{#1}}}%
3066   }%
3067 }
3068 \def\collargsPrependExpandablePreprocessor#1{%
3069   \preto\collargs@preprocess@arg{%
3070     \collargsArg\expandafter{\expanded{#1}}}%
3071   }%
3072 }
3073 \def\collargsAppendExpandablePostprocessor#1{%
3074   \appto\collargs@postprocess@arg{%
3075     \collargsArg\expandafter{\expanded{#1}}}%
3076   }%
3077 }
3078 \def\collargsPrependExpandablePostprocessor#1{%
3079   \preto\collargs@postprocess@arg{%
3080     \collargsArg\expandafter{\expanded{#1}}}%
3081   }%
3082 }

```

no delimiters When this conditional is in effect, the delimiter wrappers set by `\collargs@wrap` are `\ifcollargsNoDelimiters` ignored by `\collargs@appendarg`.

```

3083 \collargsSet{%
3084   no delimiters/.is if=collargsNoDelimiters,
3085 }
3086 \newif\ifcollargsNoDelimiters

```

clear args When this conditional is set to false, the global token register `\collargsArgs` receiving `\ifcollargsClearArgs` the collected arguments is not cleared prior to argument collection.

```

3087 \collargsSet{%
3088   clear args/.is if=collargsClearArgs,
3089 }
3090 \newif\ifcollargsClearArgs
3091 \collargsClearArgstrue

```

return Exiting `\CollectArguments`, should the next-command be followed by the braced collected `\collargsReturn` arguments, collected arguments as they are, or nothing?

```

3092 \collargsSet{%
3093   return/.is choice,
3094   return/braced/.code=\collargsReturnBraced,

```



```

3095 return/plain/.code=\collargsReturnPlain,
3096 return/no/.code=\collargsReturnNo,
3097 }
3098 \def\collargsReturnBraced{\def\collargsReturn{0}}
3099 \def\collargsReturnPlain{\def\collargsReturn{1}}
3100 \def\collargsReturnNo{\def\collargsReturn{2}}
3101 \collargsReturnBraced

```

alias

```

\collargsAlias
3102 \collargsSet{%
3103 alias/.code 2 args=\collargsAlias{#1}{#2}%
3104 }
3105 \def\collargsAlias#1#2{%
3106 \csdef{collargs@#1}{\collargs@@#2}%
3107 }

```

8.2.2 The central loop

The central loop is where we grab the next *<token>* from the argument specification and execute the corresponding argument type or modifier handler, `\collargs@<token>`. The central loop consumes the argument type *<token>*; the handler will see the remainder of the argument specification (which starts with the arguments to the argument type, if any, e.g. by `()` of `d()`), followed by a dot, and then the tokens list from which the arguments are to be collected. It is the responsibility of handler to preserve the rest of the argument specification and reexecute the central loop once it is finished.

`\collargs@` Each argument is processed in a group to allow for local settings. This group is closed by `\collargs@appendarg`.

```

3108 \def\collargs@{%
3109 \begingroup
3110 \collargs@@@
3111 }

```

`\collargs@@@` This macro is where modifier handlers reenter the central loop — we don't want modifiers to open a group, because their settings should remain in effect until the next argument. Furthermore, modifiers do not trigger category code fixes.

```

3112 \def\collargs@@@#1{%
3113 \collargs@in@{#1}{&+!>.%}
3114 \ifcollargs@in@
3115 \expandafter\collargs@@@iii
3116 \else
3117 \expandafter\collargs@@@i
3118 \fi
3119 #1%
3120 }
3121 \def\collargs@@@i#1.{%

```

Fix the category code of the next argument token, if necessary, and then proceed with the main loop.

```

3122 \collargs@fix{\collargs@@@i#1.%}
3123 }

```

Reset the fix request and set the last verbatim conditionals to the current state.

```

3124 \def\collargs@@@ii{%
3125 \global\collargs@fix@requestedfalse
3126 \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
3127 \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
3128 \collargs@@@iii
3129 }

```

Call the modifier or argument type handler denoted by the first token of the remainder of the argument specification.

```
3130 \def\collargs@@@iii#1{%
3131   \ifcsname collargs@#1\endcsname
3132     \csname collargs@#1\expandafter\endcsname
3133   \else
```

We throw an error if the token refers to no argument type or modifier.

```
3134     \collargs@error@badtype{#1}%
3135   \fi
3136 }
```

Throwing an error stops the processing of the argument specification, and closes the group opened in `\collargs@i`.

```
3137 \def\collargs@error@badtype#1#2.{%
3138   \PackageError{collargs}{Unknown xparse argument type or modifier "#1"
3139     for "\expandafter\string\collargs@caller\space"}{}}%
3140   \endgroup
3141 }
```

`\collargs@&` We extend the `xparse` syntax with modifier `&`, which applies the given options to the following (and only the following) argument. If `&` is followed by another `&`, the options are expected to occur in the raw format, like the options given to `\CollectArgumentsRaw`. Otherwise, the options should take the form of a keylist, which will be processed by `\collargsSet`. In any case, the options should be given within the argument specification, immediately following the (single or double) `&`.

```
3142 \csdef{collargs@&}{%
3143   \futurelet\collargs@temp\collargs@amp@i
3144 }
3145 \def\collargs@amp@i{%
```

In `ConTeXt`, `&` has character code “other” in the text.

```
3146 (!context) \ifx\collargs@temp&%
3147 (context)  \expandafter\ifx\detokenize{&}\collargs@temp
3148             \expandafter\collargs@amp@raw
3149           \else
3150             \expandafter\collargs@amp@set
3151           \fi
3152 }
3153 \def\collargs@amp@raw#1#2{%
3154   \collargs@verbatim@wrap{#2}%
3155   \collargs@@@
3156 }
3157 \def\collargs@amp@set#1{%
3158   \collargs@verbatim@wrap{\collargsSet{#1}}%
3159   \collargs@@@
3160 }
```

`\collargs@+` This modifier makes the next argument long, i.e. accept paragraph tokens.

```
3161 \csdef{collargs@+}{%
3162   \collargs@longtrue
3163   \collargs@@@
3164 }
3165 \newif\ifcollargs@long
```

`\collargs@>` We can simply ignore the processor modifier. (This, `xparse`'s processor, should not be confused with `CollArgs`'s processors, which are set using keys `append` `preprocessor` etc.)

```
3166 \csdef{collargs@>}#1{\collargs@@@}
```

`\collargs@!` Should we accept spaces before an optional argument following a mandatory argument (`xparse` manual, §1.1)? By default, yes. This modifier is only applicable to types `d` and `t`, and derived types, but, unlike `xparse`, we don't bother to enforce this; when used with other types, `!` simply has no effect.

```
3167 \csdef{collargs@!}{%
3168   \collargs@grabspacesfalse
3169   \collargs@@@
3170 }
```

`\collargsArgs` This token register is where we store the collected argument tokens. All assignments to this register are global, because it needs to survive the groups opened for individual arguments.

```
3171 \newtoks\collargsArgs
```

`\collargsArg` An auxiliary, but publicly available token register, used for processing the argument, and by some argument type handlers.

```
3172 \newtoks\collargsArg
```

`\collargs@.` This fake argument type is used to signal the end of the argument list. Note that this really counts as an extension of the `xparse` argument specification.

```
3173 \csdef{collargs@.}{%
```

Close the group opened in `\collargs@.`

```
3174   \endgroup
```

Close the main `\CollectArguments` group, fix the category code of the next token if necessary, and execute the next-code, followed by the collected arguments in braces. Any over-grabbed spaces are reinserted into the input stream, non-verbatim.

```
3175   \expanded{%
3176     \endgroup
3177     \noexpand\collargs@fix{%
3178       \expandonce\collargs@next
3179       \ifcase\collargsReturn\space
3180         {\the\collargsArgs}%
3181       \or
3182         \the\collargsArgs
3183       \fi
3184       \collargs@spaces
3185     }%
3186   }%
3187 }
```

8.2.3 Auxiliary macros

`\collargs@appendarg` This macro is used by the argument type handlers to append the collected argument to the storage (`\collargsArgs`).

```
3188 \long\def\collargs@appendarg#1{%
```

Temporarily store the collected argument into a token register. The processors will manipulate the contents of this register.

```
3189   \collargsArg={#1}%
```

This will clear the double-fix conditional, and potentially request a normal, single fix. We can do this here because this macro is only called when something is actually collected. For details, see section 8.2.6.

```
3190 \ifcollargs@double@fix
3191 \collargs@cancel@double@fix
3192 \fi
```

Process the argument with user-definable preprocessors, the wrapper defined by the argument type, and user-definable postprocessors.

```
3193 \collargs@preprocess@arg
3194 \ifcollargsNoDelimiters
3195 \else
3196 \collargs@process@arg
3197 \fi
3198 \collargs@postprocess@arg
```

Append the processed argument, preceded by any grabbed spaces (in the correct mode), to the storage.

```
3199 \xtoksapp\collargsArgs{\collargs@grabbed@spaces\the\collargsArg}%
```

Initialize the space-grabber.

```
3200 \collargs@init@grabspaces
```

Once the argument was appended to the list, we can close its group, opened by `\collargs@`.

```
3201 \endgroup
3202 }
```

`\collargs@wrap` This macro is used by argument type handlers to declare their delimiter wrap, like square brackets around the optional argument of type o. It uses `\collargs@addwrap`, defined in section 8.2.1, but adds to `\collargs@process@arg`, which holds the delimiter wrapper defined by the argument type handler. Note that this macro *appends* a wrapper, so multiple wrappers are allowed — this is used by type e handler.

```
3203 \def\collargs@wrap#1{%
3204 \appto\collargs@process@arg{%
3205 \long\def\collargs@temp##1{#1}%
3206 \expandafter\expandafter\expandafter\collargsArg
3207 \expandafter\expandafter\expandafter{%
3208 \expandafter\collargs@temp\expandafter{\the\collargsArg}%
3209 }%
3210 }%
3211 }
3212 \def\collargs@process@arg{}
```

`\collargs@defcollector` These macros streamline the usage of the “caller” control sequence. They are like a `\collargs@defusecollector` `\def`, but should not be given the control sequence to define, as they will automatically define the control sequence residing in `\collargs@caller`; the usage is thus `\collargs@defcollector<parameters>{<definition>}`. For example, if `\collargs@caller` holds `\foo`, `\collargs@defcollector#1{(#1)}` is equivalent to `\def\foo#1{(#1)}`. Macro `\collargs@defcollector` will only define the caller control sequence to be the collector, while `\collargs@defusecollector` will also immediately execute it.

```
3213 \def\collargs@defcollector#1{%
3214 \ifcollargs@long\long\fi
3215 \expandafter\def\collargs@caller#1%
3216 }
3217 \def\collargs@defusecollector#1{%
```

```

3218 \afterassignment\collargs@caller
3219 \ifcollargs@long\long\fi
3220 \expandafter\def\collargs@caller#1%
3221 }
3222 \def\collargs@letusecollector#1{%
3223 \expandafter\let\collargs@caller#1%
3224 \collargs@caller
3225 }
3226 \newif\ifcollargs@grabspaces
3227 \collargs@grabspacestrue

```

`\collargs@init@grabspaces` The space-grabber macro `\collargs@grabspaces` should be initialized by executing this macro. If `\collargs@grabspaces` is called twice without an intermediate initialization, it will assume it is in the same position in the input stream and simply bail out.

```

3228 \def\collargs@init@grabspaces{%
3229 \gdef\collargs@gs@state{0}%
3230 \gdef\collargs@spaces{}%
3231 \gdef\collargs@otherspaces{}%
3232 }

```

`\collargs@grabspaces` This auxiliary macro grabs any following spaces, and then executes the next-code given as the sole argument. The spaces will be stored into two macros, `\collargs@spaces` and `\collargs@otherspaces`, which store the spaces in the non-verbatim and the verbatim form. With the double storage, we can grab the spaces in the verbatim mode and use them non-verbatim, or vice versa. The macro takes a single argument, the code to execute after maybe grabbing the spaces.

```

3233 \def\collargs@grabspaces#1{%
3234 \edef\collargs@gs@next{\unexpanded{#1}}%
3235 \ifnum\collargs@gs@state=0
3236 \gdef\collargs@gs@state{1}%
3237 \expandafter\collargs@gs@i
3238 \else
3239 \expandafter\collargs@gs@next
3240 \fi
3241 }
3242 \def\collargs@gs@i{%
3243 \futurelet\collargs@temp\collargs@gs@g
3244 }

```

We check for grouping characters even in the verbatim mode, because we might be in the partial verbatim.

```

3245 \def\collargs@gs@g{%
3246 \ifcat\noexpand\collargs@temp\bgroup
3247 \expandafter\collargs@gs@next
3248 \else
3249 \ifcat\noexpand\collargs@temp\egroup
3250 \expandafter\expandafter\expandafter\collargs@gs@next
3251 \else
3252 \expandafter\expandafter\expandafter\collargs@gs@ii
3253 \fi
3254 \fi
3255 }
3256 \def\collargs@gs@ii{%
3257 \ifcollargs@verbatim
3258 \expandafter\collargs@gs@iii
3259 \else
3260 \expandafter\collargs@gs@iii
3261 \fi
3262 }

```

This works because the character code of a space token is always 32.

```
3263 \def\collargs@gs@iii{%
3264   \expandafter\ifx\space\collargs@temp
3265     \expandafter\collargs@gs@iv
3266   \else
3267     \expandafter\collargs@gs@next
3268   \fi
3269 }
3270 \expandafter\def\expandafter\collargs@gs@iv\space{%
3271   \gappto\collargs@spaces{ }%
3272   \xappto\collargs@otherspaces{\collargs@otherspace}%
3273   \collargs@gs@i
3274 }
```

We need the space of category 12 above.

```
3275 \begingroup\catcode\ =12\relax\gdef\collargs@otherspace{ }\endgroup
3276 \def\collargs@gs@iii#1{%
```

Macro `\collargs@cc` recalls the “outside” category code of character `#1`; see section [8.2.5](#).

```
3277   \ifnum\collargs@cc{#1}=10
```

We have a space.

```
3278     \expandafter\collargs@gs@iv
3279   \else
3280     \ifnum\collargs@cc{#1}=5
```

We have a newline.

```
3281     \expandafter\expandafter\expandafter\collargs@gs@v
3282   \else
3283     \expandafter\expandafter\expandafter\collargs@gs@next
3284   \fi
3285 \fi
3286 #1%
3287 }
3288 \def\collargs@gs@iv#1{%
3289   \gappto\collargs@otherspaces{#1}%
```

No matter how many verbatim spaces we collect, they equal a single non-verbatim space.

```
3290   \gdef\collargs@spaces{ }%
3291   \collargs@gs@i
3292 }
3293 \def\collargs@gs@v{%
```

Only add the first newline.

```
3294   \ifnum\collargs@gs@state=2
3295     \expandafter\collargs@gs@next
3296   \else
3297     \expandafter\collargs@gs@vi
3298   \fi
3299 }
3300 \def\collargs@gs@vi#1{%
3301   \gdef\collargs@gs@state{2}%
3302   \gappto\collargs@otherspaces{#1}%
3303   \gdef\collargs@spaces{ }%
3304   \collargs@gs@i
3305 }
```

`\collargs@maybegrabspace`s This macro grabs any following spaces, but it will do so only when conditional `\ifcollargs@grabspace`s, which can be *unset* by modifier `!`, is in effect. The macro is used by handlers for types `d` and `t`.

```

3306 \def\collargs@maybegrabspace{%
3307   \ifcollargs@grabspace
3308     \expandafter\collargs@grabspace
3309   \else
3310     \expandafter\@firstofone
3311   \fi
3312 }

```

`\collargs@grabbed@space`s This macro expands to either the verbatim or the non-verbatim variant of the grabbed spaces, depending on the verbatim mode in effect at the time of expansion.

```

3313 \def\collargs@grabbed@space{%
3314   \ifcollargs@verbatim
3315     \collargs@otherspace
3316   \else
3317     \collargs@space
3318   \fi
3319 }

```

`\collargs@reinsert@space`s Inserts the grabbed spaces back into the input stream, but with the category code appropriate for the verbatim mode then in effect. After the insertion, the space-grabber is initialized and the given next-code is executed in front of the inserted spaces.

```

3320 \def\collargs@reinsert@space#1{%
3321   \expanded{%
3322     \unexpanded{%
3323       \collargs@init@grabspace
3324       #1%
3325     }%
3326     \collargs@grabbed@space
3327   }%
3328 }

```

`\collargs@ifnextcat` An adaptation of `\pgf@keys@utilifnextchar` which checks whether the *category* code of the next non-space character matches the category code of `#1`.

```

3329 \long\def\collargs@ifnextcat#1#2#3{%
3330   \let\pgf@keys@utilreserved@d=#1%
3331   \def\pgf@keys@utilreserved@a{#2}%
3332   \def\pgf@keys@utilreserved@b{#3}%
3333   \futurelet\pgf@keys@utillet@token\collargs@ifncat}
3334 \def\collargs@ifncat{%
3335   \ifx\pgf@keys@utillet@token\pgf@keys@utilstoken
3336     \let\pgf@keys@utilreserved@c\collargsxifnch
3337   \else
3338     \ifcat\noexpand\pgf@keys@utillet@token\pgf@keys@utilreserved@d
3339       \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@a
3340     \else
3341       \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@b
3342     \fi
3343   \fi
3344   \pgf@keys@utilreserved@c}
3345 {%
3346   \def\:\{\collargs@xifncat}
3347   \expandafter\gdef\:\{\futurelet\pgf@keys@utillet@token\collargs@ifncat}
3348 }

```

`\collargs@forrange` This macro executes macro `\collargs@do` for every integer from #1 and #2, both inclusive. `\collargs@do` should take a single parameter, the current number.

```

3349 \def\collargs@forrange#1#2{%
3350   \expanded{%
3351     \noexpand\collargs@forrange@i{\number#1}{\number#2}%
3352   }%
3353 }
3354 \def\collargs@forrange@i#1#2{%
3355   \ifnum#1>#2 %
3356     \expandafter\@gobble
3357   \else
3358     \expandafter\@firstofone
3359   \fi
3360   {%
3361     \collargs@do{#1}%
3362     \expandafter\collargs@forrange@i\expandafter{\number\numexpr#1+1\relax}{#2}%
3363   }%
3364 }

```

`\collargs@forranges` This macro executes macro `\collargs@do` for every integer falling into the ranges specified in #1. The ranges should be given as a comma-separated list of from-to items, e.g. 1-5,10-11.

```

3365 \def\collargs@forranges{\forcsvlist\collarg@forrange@i}
3366 \def\collarg@forrange@i#1{\collarg@forrange@ii#1-}
3367 \def\collarg@forrange@ii#1-#2-{\collargs@forrange{#1}{#2}}

```

`\collargs@percentchar` This macro holds the percent character of category 12.

```

3368 \begingroup
3369 \catcode`\%=12
3370 \gdef\collargs@percentchar{%
3371 \endgroup

```

8.2.4 The handlers

`\collargs@l` We will first define the handler for the very funky argument type 1, which corresponds to T_EX's `\def\foo#1#{...}`, which grabs (into #1) everything up to the first opening brace — not because this type is important or even recommended to use, but because the definition of the handler is very simple, at least for the non-verbatim case.

```
3372 \def\collargs@l#1. {%
```

Any pre-grabbed spaces in fact belong into the argument.

```

3373   \collargs@reinsert@spaces{\collargs@l@i#1.}%
3374 }
3375 \def\collargs@l@i{%

```

We request a correction of the category code of the delimiting brace if the verbatim mode changes for the next argument; for details, see section 8.2.6.

```
3376   \global\collargs@fix@requestedtrue
```

Most handlers will branch into the verbatim and the non-verbatim part using conditional `\ifcollargs@verbatim`. This handler is a bit special, because it needs to distinguish verbatim and non-verbatim *braces*, and braces are verbatim only in the full verbatim mode, i.e. when `\ifcollargs@verbatimbraces` is true.

```

3377   \ifcollargs@verbatimbraces
3378     \expandafter\collargs@l@verb
3379   \else
3380     \expandafter\collargs@l@ii
3381   \fi
3382 }

```


We grab the rest of the argument specification (`#1`), to be reinserted into the token stream when we reexecute the central loop.

```
3383 \def\collargs@l@ii#1.{%
```

In the non-verbatim mode, we merely have to define and execute the collector macro. The parameter text `##1##` (note the doubled hashes), which will put everything up to the first opening brace into the first argument, looks funky, but that's all.

```
3384 \collargs@defusecollector##1##{%
```

We append the collected argument, `#1`, to `\collargsArgs`, the token register holding the collected argument tokens.

```
3385 \collargs@appendarg{##1}%
```

Back to the central loop, with the rest of the argument specification reinserted.

```
3386 \collargs@#1.%
```

```
3387 }%
```

```
3388 }
```

```
3389 \def\collargs@l@verb#1.{%
```

In the verbatim branch, we need to grab everything up to the first opening brace of category code 12, so we want to define the collector with parameter text `##1{`, with the opening brace of category 12. We have stored this token in macro `\collargs@other@bgroup`, which we now need to expand.

```
3390 \expandafter\collargs@defusecollector
```

```
3391 \expandafter##\expandafter1\collargs@other@bgroup{%
```

Appending the argument works the same as in the non-verbatim case.

```
3392 \collargs@appendarg{##1}%
```

Reexecuting the central loop macro is a bit more involved, as we need to reinsert the verbatim opening brace (contrary to the regular brace above, the verbatim brace is consumed by the collector macro) back into the token stream, behind the reinserted argument specification.

```
3393 \expanded{%
```

```
3394 \noexpand\collargs@\unexpanded{#1.}%
```

```
3395 \collargs@other@bgroup
```

```
3396 }%
```

```
3397 }%
```

```
3398 }
```

`\collargs@u` Another weird type — `u<tokens>` reads everything up to the given `<tokens>`, i.e. this is \TeX 's `\def\foo#1<tokens>{...}` — but again, simple enough to allow us to showcase solutions to two recurring problems.

We start by branching into the verbatim mode (full or partial) or the non-verbatim mode.

```
3399 \def\collargs@u{%
```

```
3400 \ifcollargs@verbatim
```

```
3401 \expandafter\collargs@u@verb
```

```
3402 \else
```

```
3403 \expandafter\collargs@u@i
```

```
3404 \fi
```

```
3405 }
```

To deal with the verbatim mode, we only need to convert the above `<tokens>` (i.e. the argument of `u` in the argument specification) to category 12, i.e. we have to `\detokenize` them. Then, we may proceed as in the non-verbatim branch, `\collargs@u@ii`.

```
3406 \def\collargs@u@verb#1{%
```

The `\string` here is a temporary solution to a problem with spaces. Our verbatim mode has them of category “other”, but `\detokenize` produces a space of category “space” behind control words.

```
3407 \expandafter\collargs@u@i\expandafter{\detokenize\expandafter{\string#1}}%
3408 }
```

We then reinsert any pre-grabbed spaces into the stream, but we take care not to destroy the braces around our delimiter in the argument specification.

```
3409 \def\collargs@u@i#1#2.{%
3410 \collargs@reinsert@spaces{\collargs@u@ii{#1}#2.}%
3411 }
3412 \def\collargs@u@ii#1#2.{%
```

`#1` contains the delimiter tokens, so `##1` below will receive everything in the token stream up to these. But we have a problem: if we defined the collector as for the non-verbatim `l`, and the delimiter happened to be preceded by a single brace group, we would lose the braces. For example, if the delimiter was `-` and we received `{foo}-`, we would collect `foo-`. We solve this problem by inserting `\collargs@empty` (with an empty definition) into the input stream (at the end of this macro) — this way, the delimiter can never be preceded by a single brace group — and then expanding it away before appending to storage (within the argument of `\collargs@defusecollector`).

```
3413 \collargs@defusecollector##1#1{%
```

Define the wrapper which will add the delimiter tokens (`#1`) after the collected argument. The wrapper will be applied during argument processing in `\collargs@appendarg` (sandwiched between used-definable pre- and post-processors).

```
3414 \collargs@wrap{####1#1}%
```

Expand the first token in `##1`, which we know to be `\collargs@empty`, with empty expansion.

```
3415 \expandafter\collargs@appendarg\expandafter{##1}%
3416 \collargs@#2.%
3417 }%
```

Insert `\collargs@empty` into the input stream, in front of the “real” argument tokens.

```
3418 \collargs@empty
3419 }
3420 \def\collargs@empty{}
```

`\collargs@r` Finally, a real argument type: required delimited argument.

```
3421 \def\collargs@r{%
3422 \ifcollargs@verbatim
3423 \expandafter\collargs@r@verb
3424 \else
3425 \expandafter\collargs@r@i
3426 \fi
3427 }
3428 \def\collargs@r@verb#1#2{%
3429 \expandafter\collargs@r@i\detokenize{#1#2}%
3430 }
3431 \def\collargs@r@i#1#2#3.{%
```

We will need to use the `\collargs@empty` trick from type `u`, but with an additional twist: we need to insert it *after* the opening delimiter `#1`. To do this, we consume the opening delimiter by the “outer” collector below — we need to use the collector so that we get a nice error message when the opening delimiter is not present — and have this collector define the “inner” collector in the spirit of type `u`.

The outer collector has no parameters, it just requires the presence of the opening delimiter.

```
3432 \collargs@defcollector#1{%
```

The inner collector will grab everything up to the closing delimiter.

```
3433 \collargs@defusecollector####1#2{%
```

Append the collected argument `####1` to the list, wrapping it into the delimiters (`#1` and `#2`), but not before expanding its first token, which we know to be `\collargs@empty`.

```
3434 \collargs@wrap{#1#####1#2}%
3435 \expandafter\collargs@appendarg\expandafter{####1}%
3436 \collargs@#3.%
3437 }%
3438 \collargs@empty
3439 }%
```

Another complication: our delimited argument may be preceded by spaces. To replicate the argument tokens faithfully, we need to collect them before trying to grab the argument itself.

```
3440 \collargs@grabspaces\collargs@caller
3441 }
```

`\collargs@R` Discard the default and execute `r`.

```
3442 \def\collargs@R#1#2#3{\collargs@r#1#2}
```

`\collargs@d` Optional delimited argument. Very similar to `r`.

```
3443 \def\collargs@d{%
3444 \ifcollargs@verbatim
3445 \expandafter\collargs@d@verb
3446 \else
3447 \expandafter\collargs@d@i
3448 \fi
3449 }
3450 \def\collargs@d@verb#1#2{%
3451 \expandafter\collargs@d@i\detokenize{#1#2}%
3452 }
3453 \def\collargs@d@i#1#2#3.{%
```

This macro will be executed when the optional argument is not present. It simply closes the argument's group and reexecutes the central loop.

```
3454 \def\collargs@d@noopt{%
3455 \global\collargs@fix@requestedtrue
3456 \endgroup
3457 \collargs@#3.%
3458 }%
```

The collector(s) are exactly as for `r`.

```
3459 \collargs@defcollector#1{%
3460 \collargs@defusecollector####1#2{%
3461 \collargs@wrap{#1#####1#2}%
3462 \expandafter\collargs@appendarg\expandafter{####1}%
3463 \collargs@#3.%
3464 }%
3465 \collargs@empty
3466 }%
```

This macro will check, in conjunction with `\futurelet` below, whether the optional argument is present or not.

```

3467 \def\collargs@d@ii{%
3468   \ifx#1\collargs@temp
3469     \expandafter\collargs@caller
3470   \else
3471     \expandafter\collargs@d@noopt
3472   \fi
3473 }%
```

Whether spaces are allowed in front of this type of argument depends on the presence of modifier `!`.

```

3474 \collargs@maybegrabspace{\futurelet\collargs@temp\collargs@d@ii}%
3475 }
```

`\collargs@D` Discard the default and execute `d`.

```

3476 \def\collargs@D#1#2#3{\collargs@d#1#2}
```

`\collargs@o` `o` is just `d` with delimiters `[` and `]`.

```

3477 \def\collargs@o{\collargs@d[]}
```

`\collargs@O` `O` is just `d` with delimiters `[` and `]` and the discarded default.

```

3478 \def\collargs@O#1{\collargs@d[]}
```

`\collargs@t` An optional token. Similar to `d`.

```

3479 \def\collargs@t{%
3480   \ifcollargs@verbatim
3481     \expandafter\collargs@t@verb
3482   \else
3483     \expandafter\collargs@t@i
3484   \fi
3485 }
3486 \def\collargs@t@space{ }
3487 \def\collargs@t@verb#1{%
3488   \let\collargs@t@space\collargs@otherspace
3489   \expandafter\collargs@t@i\expandafter{\detokenize{#1}}%
3490 }
3491 \def\collargs@t@i#1{%
3492   \expandafter\ifx\space#1%
3493     \expandafter\collargs@t@s
3494   \else
3495     \expandafter\collargs@t@I\expandafter#1%
3496   \fi
3497 }
3498 \def\collargs@t@s#1.{%
3499   \collargs@grabspace{%
3500     \ifcollargs@grabspace
3501       \collargs@appendarg{}}%
3502   \else
3503     \expanded{%
3504       \noexpand\collargs@init@grabspace
3505       \noexpand\collargs@appendarg{\collargs@grabbed@spaces}}%
3506   }%
3507   \fi
3508   \collargs@#1.%
3509 }%
3510 }
```

```

3511 \def\collargs@t@I#1#2.{%
3512   \def\collargs@t@noopt{%
3513     \global\collargs@fix@requestedtrue
3514     \endgroup
3515     \collargs@#2.%
3516   }%
3517 \def\collargs@t@opt##1{%
3518   \collargs@appendarg{#1}%
3519   \collargs@#2.%
3520 }%
3521 \def\collargs@t@ii{%
3522   \ifx#1\collargs@temp
3523     \expandafter\collargs@t@opt
3524   \else
3525     \expandafter\collargs@t@noopt
3526   \fi
3527 }%
3528 \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@t@ii}%
3529 }
3530 \def\collargs@t@opt@space{%
3531   \expanded{\noexpand\collargs@t@opt{\space}\expandafter}\romannumeral-0%
3532 }%

```

`\collargs@s` The optional star is just a special case of `t`.

```

3533 \def\collargs@s{\collargs@t*}

```

`\collargs@m` Mandatory argument. Interestingly, here's where things get complicated, because we have to take care of several \TeX quirks.

```

3534 \def\collargs@m{%
3535   \ifcollargs@verbatim
3536     \expandafter\collargs@m@verb
3537   \else
3538     \expandafter\collargs@m@i
3539   \fi
3540 }

```

The non-verbatim mode. First, collect any spaces in front of the argument.

```

3541 \def\collargs@m@i#1.{%
3542   \collargs@grabspaces{\collargs@m@checkforgroup#1.}%
3543 }

```

Is the argument in braces or not?

```

3544 \def\collargs@m@checkforgroup#1.{%
3545   \edef\collargs@action{\unexpanded{\collargs@m@checkforgroup@i#1.}}%
3546   \futurelet\collargs@token\collargs@action
3547 }
3548 \def\collargs@m@checkforgroup@i{%
3549   \ifcat\noexpand\collargs@token\bgroup
3550     \expandafter\collargs@m@group
3551   \else
3552     \expandafter\collargs@m@token
3553   \fi
3554 }

```

The argument is given in braces, so we put them back around it (`\collargs@wrap`) when appending to the storage.

```

3555 \def\collargs@m@group#1.{%
3556   \collargs@defusecollector##1{%

```

```

3557 \collargs@wrap{####1}%
3558 \collargs@appendarg{##1}%
3559 \collargs@#1.%
3560 }%
3561 }

```

The argument is a single token, we append it to the storage as is.

```

3562 \def\collargs@m@token#1.{%
3563 \collargs@defusecollector##1{%
3564 \collargs@appendarg{##1}%
3565 \collargs@#1.%
3566 }%
3567 }

```

The verbatim mode. Again, we first collect any spaces in front of the argument.

```

3568 \def\collargs@m@verb#1.{%
3569 \collargs@grabspace{\collargs@m@verb@checkforgroup#1.}%
3570 }

```

We want to check whether we're dealing with a braced argument. We're in the verbatim mode, but are braces verbatim as well? In other words, are we in `verbatim` or `verb` mode? In the latter case, braces are regular, so we redirect to the regular mode.

```

3571 \def\collargs@m@verb@checkforgroup{%
3572 \ifcollargs@verbatimbraces
3573 \expandafter\collargs@m@verb@checkforgroup@i
3574 \else
3575 \expandafter\collargs@m@checkforgroup
3576 \fi
3577 }

```

Is the argument in verbatim braces?

```

3578 \def\collargs@m@verb@checkforgroup@i#1.{%
3579 \def\collargs@m@verb@checkforgroup@ii{\collargs@m@verb@checkforgroup@iii#1.}%
3580 \futurelet\collargs@temp\collargs@m@verb@checkforgroup@ii
3581 }
3582 \def\collargs@m@verb@checkforgroup@iii#1.{%
3583 \expandafter\ifx\collargs@other@bgroup\collargs@temp

```

Yes, the argument is in (verbatim) braces.

```

3584 \expandafter\collargs@m@verb@group
3585 \else

```

We need to manually check whether the following token is a (verbatim) closing brace, and throw an error if it is.

```

3586 \expandafter\ifx\collargs@other@egroup\collargs@temp
3587 \expandafter\expandafter\expandafter\collargs@m@verb@egrouperror
3588 \else

```

The argument is a single token.

```

3589 \expandafter\expandafter\expandafter\collargs@m@v@token
3590 \fi
3591 \fi
3592 #1.%
3593 }
3594 \def\collargs@m@verb@egrouperror#1.{%
3595 \PackageError{collargs}{%
3596 Argument of \expandafter\string\collargs@caller\space has an extra
3597 \iffalse{\else\string}}}%
3598 }

```

A single-token verbatim argument.

```
3599 \def\collargs@m@v@token#1.#2{%
```

Is it a control sequence? (Macro `\collargs@cc` recalls the “outside” category code of character `#1`; see section 8.2.5.)

```
3600 \ifnum\collargs@cc{#2}=0
3601   \expandafter\collargs@m@v@token@cs
3602 \else
3603   \expandafter\collargs@m@token
3604 \fi
3605 #1.#2%
3606 }
```

Is it a one-character control sequence?

```
3607 \def\collargs@m@v@token@cs#1.#2#3{%
3608   \ifnum\collargs@cc{#3}=11
3609     \expandafter\collargs@m@v@token@cs@letter
3610   \else
3611     \expandafter\collargs@m@v@token@cs@nonletter
3612   \fi
3613   #1.#2#3%
3614 }
```

Store `\<token>`.

```
3615 \def\collargs@m@v@token@cs@nonletter#1.#2#3{%
3616   \collargs@appendarg{#2#3}%
3617   \collargs@#1.%
3618 }
```

Store `\` to a temporary register, we’ll parse the control sequence name now.

```
3619 \def\collargs@m@v@token@cs@letter#1.#2{%
3620   \collargsArg{#2}%
3621   \def\collargs@tempa{#1}%
3622   \collargs@m@v@token@cs@letter@i
3623 }
```

Append a letter to the control sequence.

```
3624 \def\collargs@m@v@token@cs@letter@i#1{%
3625   \ifnum\collargs@cc{#1}=11
3626     \toksapp\collargsArg{#1}%
3627     \expandafter\collargs@m@v@token@cs@letter@i
3628   \else
```

Finish, returning the non-letter to the input stream.

```
3629     \expandafter\collargs@m@v@token@cs@letter@ii\expandafter#1%
3630   \fi
3631 }
```

Store the verbatim control sequence.

```
3632 \def\collargs@m@v@token@cs@letter@ii{%
3633   \expanded{%
3634     \unexpanded{%
3635       \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3636     }%
3637     \noexpand\collargs@\expandonce\collargs@tempa.%
3638   }%
3639 }
```

The verbatim mandatory argument is delimited by verbatim braces. We have to use the heavy machinery adapted from `cprotect`.

```

3640 \def\collargs@m@verb@group#1.#2{%
3641   \let\collargs@begintag\collargs@other@bgroup
3642   \let\collargs@endtag\collargs@other@egroup
3643   \def\collargs@tagarg{}%
3644   \def\collargs@commandatend{\collargs@m@verb@group@i#1.}%
3645   \collargs@readContent
3646 }

```

This macro appends the result given by the heavy machinery, waiting for us in macro `\collargsArg`, to `\collargsArgs`, but not before dressing it up (via `\collargs@wrap`) in a pair of verbatim braces.

```

3647 \def\collargs@m@verb@group@i{%
3648   \edef\collargs@temp{%
3649     \collargs@other@bgroup\unexpanded{##1}\collargs@other@egroup}%
3650   \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3651   \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3652   \collargs@
3653 }

```

`\collargs@g` An optional group: same as `m`, but we simply bail out if we don't find the group character.

```

3654 \def\collargs@g{%
3655   \def\collargs@m@token{%
3656     \global\collargs@fix@requestedtrue
3657     \endgroup
3658     \collargs@
3659   }%
3660   \let\collargs@m@v@token\collargs@m@token
3661   \collargs@m
3662 }

```

`\collargs@G` Discard the default and execute `g`.

```

3663 \def\collargs@G#1{\collargs@g}

```

`\collargs@v` Verbatim argument. The code is executed in the group, deploying `\collargsVerbatim`. The grouping characters are always set to braces, to mimic `xparse` perfectly.

```

3664 \def\collargs@v#1.{%
3665   \begingroup
3666   \collargsBraces{}}%
3667   \collargsVerbatim
3668   \collargs@grabspaces{\collargs@v@i#1.}%
3669 }
3670 \def\collargs@v@i#1.#2{%
3671   \expandafter\ifx\collargs@other@bgroup#2%

```

If the first token we see is an opening brace, use the `cprotect` adaptation to grab the group.

```

3672   \let\collargs@begintag\collargs@other@bgroup
3673   \let\collargs@endtag\collargs@other@egroup
3674   \def\collargs@tagarg{}%
3675   \def\collargs@commandatend{%
3676     \edef\collargs@temp{%
3677       \collargs@other@bgroup\unexpanded{####1}\collargs@other@egroup}%
3678     \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3679     \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3680     \endgroup
3681     \collargs@#1.%

```



```

3682 }%
3683 \expandafter\collargs@readContent
3684 \else

```

Otherwise, the verbatim argument is delimited by two identical characters (#2).

```

3685 \collargs@defcollector##1#2{%
3686 \collargs@wrap{#2###1#2}%
3687 \collargs@appendarg{##1}%
3688 \endgroup
3689 \collargs@#1.%
3690 }%
3691 \expandafter\collargs@caller
3692 \fi
3693 }

```

`\collargs@b` Environments. Here's where all hell breaks loose. We survive by adapting some code from Bruno Le Floch's `cprotect`. We first define the environment-related keys, then provide the handler code, and finish with the adaptation of `cprotect`'s environment-grabbing code.

The argument type `b` token may be followed by a braced environment name (in the argument specification).

```

3694 \def\collargs@b{%
3695 \collargs@ifnextcat\bgroup\collargs@bg\collargs@bi
3696 }
3697 \def\collargs@bg#1{%
3698 \edef\collargs@b@envname{#1}%
3699 \collargs@bi
3700 }
3701 \def\collargs@bi#1.{%

```

Convert the environment name to verbatim if necessary.

```

3702 \ifcollargs@verbatim
3703 \edef\collargs@b@envname{\detokenize\expandafter{\collargs@b@envname}}%
3704 \fi

```

This is a format-specific macro which sets up `\collargs@begintag` and `\collargs@endtag`.

```

3705 \collargs@bi@defCPTbeginend
3706 \edef\collargs@tagarg{%
3707 \ifcollargs@verbatimbraces
3708 \else
3709 \ifcollargsIgnoreOtherTags
3710 \collargs@b@envname
3711 \fi
3712 \fi
3713 }%

```

Run this after collecting the body.

```

3714 \def\collargs@commandatend{%

```

In \LaTeX , we might, depending on the verbatim mode, need to check whether the environment name is correct.

```

3715 <latex> \collargs@bii

```

In plain \TeX and \ConTeXt , we can skip directly to `\collargs@biii`.

```

3716 <plain,context> \collargs@biii
3717 #1.%
3718 }%

```

Collect the environment body, but first, put any grabbed spaces back into the input stream.

```
3719 \collargs@reinsert@spaces\collargs@readContent
3720 }
3721 <*/latex>
```

In L^AT_EX in the regular and the partial verbatim mode, we search for `\begin/\end` — as we cannot search for braces — either as control sequences in the regular mode, or as strings in the partial verbatim mode. (After search, we will have to check whether the argument of `\begin/\end` matches our environment name.) In the full verbatim mode, we can search for the entire string `\begin/\end{<name>}`.

```
3722 \def\collargs@bi@defCPTbeginend{%
3723 \edef\collargs@begintag{%
3724 \ifcollargs@verbatim
3725 \expandafter\string
3726 \else
3727 \expandafter\noexpand
3728 \fi
3729 \begin
3730 \ifcollargs@verbatimbraces
3731 \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
3732 \fi
3733 }%
3734 \edef\collargs@endtag{%
3735 \ifcollargs@verbatim
3736 \expandafter\string
3737 \else
3738 \expandafter\noexpand
3739 \fi
3740 \end
3741 \ifcollargs@verbatimbraces
3742 \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
3743 \fi
3744 }%
3745 }
3746 </latex>
3747 <*plain, context>
```

We can search for the entire `\<name>/\end<name>` (in T_EX) or `\start<name>/\stop<name>` (in ConT_EXt), either as a control sequence (in the regular mode), or as a string (in the verbatim modes).

```
3748 \def\collargs@bi@defCPTbeginend{%
3749 \edef\collargs@begintag{%
3750 \ifcollargs@verbatim
3751 \expandafter\expandafter\expandafter\string
3752 \else
3753 \expandafter\expandafter\expandafter\noexpand
3754 \fi
3755 \csname
3756 <context> start%
3757 \collargs@b@envname
3758 \endcsname
3759 }%
3760 \edef\collargs@endtag{%
3761 \ifcollargs@verbatim
3762 \expandafter\expandafter\expandafter\string
3763 \else
3764 \expandafter\expandafter\expandafter\noexpand
3765 \fi
3766 \csname
3767 <plain> end%
```

```

3768 <context>      stop%
3769              \collargs@b@envname
3770              \endcsname
3771            }%
3772          }
3773 </plain, context>
3774 <*latex>

```

Check whether we're in front of the (braced) environment name (in L^AT_EX), and consume it.

```

3775 \def\collargs@bii{%
3776   \ifcollargs@verbatimbraces
3777     \expandafter\collargs@biii
3778   \else
3779     \ifcollargsIgnoreOtherTags

```

We shouldn't check the name in this case, because it was already checked, and consumed.

```

3780     \expandafter\expandafter\expandafter\collargs@biii
3781   \else
3782     \expandafter\expandafter\expandafter\collargs@b@checkend
3783   \fi
3784 \fi
3785 }
3786 \def\collargs@b@checkend#1.{%
3787   \collargs@grabspaces{\collargs@b@checkend@i#1.}%
3788 }
3789 \def\collargs@b@checkend@i#1.#2{%
3790   \def\collargs@temp{#2}%
3791   \ifx\collargs@temp\collargs@b@envname
3792   \else
3793     \collargs@b@checkend@error
3794   \fi
3795   \collargs@biii#1.%
3796 }
3797 \def\collargs@b@checkend@error{%
3798   \PackageError{collargs}{Environment "\collargs@b@envname" ended as
3799     "\collargs@temp"}{}%
3800 }
3801 </latex>

```

This macro stores the collected body.

```

3802 \def\collargs@biii{%

```

Define the wrapper macro (`\collargs@temp`).

```

3803   \collargs@b@def@wrapper

```

Execute `\collargs@appendarg` to append the body to the list. Expand the wrapper in `\collargs@temp` first and the body in `\collargsArg` next.

```

3804   \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%

```

Reexecute the central loop.

```

3805   \collargs@
3806 }
3807 \def\collargs@b@def@wrapper{%
3808 <latex>   \edef\collargs@temp{\collargs@b@envname}}%
3809   \edef\collargs@temp{%

```

Was the begin-tag requested?

```

3810   \ifcollargsBeginTag

```

`\collargs@begintag` is already adapted to the format and the verbatim mode.

```
3811 \expandonce\collargs@begintag
```

Add the braced environment name in \LaTeX in the regular and partial verbatim mode.

```
3812 {*latex}
3813 \ifcollargs@verbatimbraces\else\collargs@temp\fi
3814 {/latex}
3815 \fi
```

This is the body.

```
3816 #####1%
```

Rinse and repeat for the end-tag.

```
3817 \ifcollargsEndTag
3818 \expandonce\collargs@endtag
3819 {*latex}
3820 \ifcollargs@verbatimbraces\else\collargs@temp\fi
3821 {/latex}
3822 \fi
3823 }%
3824 \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3825 }
```

`\collargs@readContent` This macro, which is an adaptation of `cprotect`'s environment-grabbing code, collects some delimited text, leaving the result in `\collargsArg`. Before calling it, one must define the following macros: `\collargs@begintag` and `\collargs@endtag` are the content delimiters; `\collargs@tagarg`, if non-empty, is the token or grouped text which must follow a delimiter to be taken into account; `\collargs@commandatend` is the command that will be executed once the content is collected.

```
3826 \def\collargs@readContent{%
```

Define macro which will search for the first begin-tag.

```
3827 \ifcollargs@long\long\fi
3828 \collargs@CPT@def\collargs@gobbleOneB\collargs@begintag{%
```

Assign the collected tokens into a register. The first token in `##1` will be `\collargs@empty`, so we expand to get rid of it.

```
3829 \toks0\expandafter{##1}%
```

`cprotect` simply grabs the token following the `\collargs@begintag` with a parameter. We can't do this, because we need the code to work in the non-verbatim mode, as well, and we might stumble upon a brace there. So we take a peek.

```
3830 \futurelet\collargs@temp\collargs@gobbleOneB@i
3831 }%
```

Define macro which will search for the first end-tag. We make it long if so required (by +).

```
3832 \ifcollargs@long\long\fi
3833 \collargs@CPT@def\collargs@gobbleUntilE\collargs@endtag{%
```

Expand `\collargs@empty` at the start of `##1`.

```
3834 \expandafter\toksapp\expandafter0\expandafter{##1}%
3835 \collargs@gobbleUntilE@i
3836 }%
```

Initialize.

```
3837 \collargs@begins=0\relax
3838 \collargsArg{}%
3839 \toks0{}%
```

We will call `\collargs@gobbleUntilE` via the caller control sequence.

```
3840 \collargs@letusecollector\collargs@gobbleUntilE
```

We insert `\collargs@empty` to avoid the potential debracing problem.

```
3841 \collargs@empty
3842 }
```

How many begin-tags do we have opened?

```
3843 \newcount\collargs@begins
```

An auxiliary macro which `\defs #1` so that it will grab everything up until `#2`. Additional parameters may be present before the definition.

```
3844 \def\collargs@CPT@def#1#2{%
3845 \expandafter\def\expandafter#1%
3846 \expandafter##\expandafter1#2%
3847 }
```

A quark quard.

```
3848 \def\collargs@qend{\collargs@qend}
```

This macro will collect the “environment”, leaving the result in `\collargsArg`. It expects `\collargs@begintag`, `\collargs@endtag` and `\collargs@commandatend` to be set.

```
3849 \def\collargs@gobbleOneB@i{%
3850 \def\collargs@begins@increment{1}%
3851 \ifx\collargs@qend\collargs@temp
```

We have reached the fake begin-tag. Note that we found the end-tag.

```
3852 \def\collargs@begins@increment{-1}%
```

Gobble the quark guard.

```
3853 \expandafter\collargs@gobbleOneB@v
3854 \else
```

Append the real begin-tag to the temporary tokens.

```
3855 \etoksapp0{\expandonce\collargs@begintag}%
3856 \expandafter\collargs@gobbleOneB@ii
3857 \fi
3858 }%
```

Do we have to check the tag argument (i.e. the environment name after `\begin`)?

```
3859 \def\collargs@gobbleOneB@ii{%
3860 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
3861 \expandafter\collargs@gobbleOneB@vi
3862 \else
```

Yup, so let's (carefully) collect the tag argument.

```
3863   \expandafter\collargs@gobbleOneB@iii
3864   \fi
3865 }
3866 \def\collargs@gobbleOneB@iii{%
3867   \collargs@grabspaces{%
3868     \collargs@letusecollector\collargs@gobbleOneB@iv
3869   }%
3870 }
3871 \def\collargs@gobbleOneB@iv#1{%
3872   \def\collargs@temp{#1}%
3873   \ifx\collargs@temp\collargs@tagarg
```

This is the tag argument we've been waiting for!

```
3874   \else
```

Nope, this `\begin` belongs to someone else.

```
3875   \def\collargs@begins@increment{0}%
3876   \fi
```

Whatever the result was, we have to append the gobbled group to the temporary toks.

```
3877   \etoksapp0{\collargs@grabbed@spaces\unexpanded{{#1}}}%
3878   \collargs@init@grabspaces
3879   \collargs@gobbleOneB@vi
3880 }
3881 \def\collargs@gobbleOneB@v#1{\collargs@gobbleOneB@vi}
3882 \def\collargs@gobbleOneB@vi{%
```

Store.

```
3883   \etoksapp\collargsArg{\the\toks0}%
```

Advance the begin-tag counter.

```
3884   \advance\collargs@begins\collargs@begins@increment\relax
```

Find more begin-tags, unless this was the final one.

```
3885   \ifnum\collargs@begins@increment=-1
3886   \else
3887     \expandafter\collargs@gobbleOneB\expandafter\collargs@empty
3888     \fi
3889 }
3890 \def\collargs@gobbleUntilE@i{%
```

Do we have to check the tag argument (i.e. the environment name after `\end`)?

```
3891   \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
3892   \expandafter\collargs@gobbleUntilE@iv
3893   \else
```

Yup, so let's (carefully) collect the tag argument.

```
3894   \expandafter\collargs@gobbleUntilE@ii
3895   \fi
3896 }
3897 \def\collargs@gobbleUntilE@ii{%
3898   \collargs@grabspaces{%
3899     \collargs@letusecollector\collargs@gobbleUntilE@iii
3900   }%
3901 }
```

```

3902 \def\collargs@gobbleUntilE@iii#1{%
3903   \etoksapp0{\collargs@grabbed@spaces}%
3904   \collargs@init@grabspaces
3905   \def\collargs@tempa{#1}%
3906   \ifx\collargs@tempa\collargs@tagarg

```

This is the tag argument we've been waiting for!

```

3907   \expandafter\collargs@gobbleUntilE@iv
3908   \else

```

Nope, this `\end` belongs to someone else. Insert the end tag plus the tag argument, and collect until the next `\end`.

```

3909   \expandafter\toksapp\expandafter0\expandafter{\collargs@endtag{#1}}%
3910   \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
3911   \fi
3912 }
3913 \def\collargs@gobbleUntilE@iv{%

```

Invoke `\collargs@gobbleOneB` with the collected material, plus a fake begin-tag and a quark guard.

```

3914   \ifcollargsIgnoreNesting
3915     \expandafter\collargsArg\expandafter{\the\toks0}%
3916     \expandafter\collargs@commandatend
3917   \else
3918     \expandafter\collargs@gobbleUntilE@v
3919   \fi
3920 }
3921 \def\collargs@gobbleUntilE@v{%
3922   \expanded{%
3923     \noexpand\collargs@letusecollector\noexpand\collargs@gobbleOneB
3924     \noexpand\collargs@empty
3925     \the\toks0

```

Add a fake begin-tag and a quark guard.

```

3926   \expandonce\collargs@begintag
3927   \noexpand\collargs@qend
3928 }%
3929 \ifnum\collargs@begins<0
3930   \expandafter\collargs@commandatend
3931 \else
3932   \etoksapp\collargsArg{%
3933     \expandonce\collargs@endtag
3934     \expandafter\ifx\expandafter\relax\collargs@tagarg\relax\else{%
3935       \expandonce\collargs@tagarg}\fi
3936   }%
3937   \toks0={}%
3938   \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
3939   \expandafter\collargs@empty
3940   \fi
3941 }

```

`\collargs@e` Embellishments. Each embellishment counts as an argument, in the sense that we will execute `\collargs@appendarg`, with all the processors, for each embellishment separately.

```

3942 \def\collargs@e{%

```

We open an extra group, because `\collargs@appendarg` will close a group for each embellishment.

```

3943   \global\collargs@fix@requestedtrue
3944   \begingroup

```

```

3945 \ifcollargs@verbatim
3946   \expandafter\collargs@e@verbatim
3947 \else
3948   \expandafter\collargs@e@i
3949 \fi
3950 }

```

Detokenize the embellishment tokens in the verbatim mode.

```

3951 \def\collargs@e@verbatim#1{%
3952   \expandafter\collargs@e@i\expandafter{\detokenize{#1}}%
3953 }

```

Ungroup the embellishment tokens, separating them from the rest of the argument specification by a dot.

```

3954 \def\collargs@e@i#1{\collargs@e@ii#1.}

```

We now have embellishment tokens in #1 and the rest of the argument specification in #2. Let's grab spaces first.

```

3955 \def\collargs@e@ii#1.#2.{%
3956   \collargs@grabspaces{\collargs@e@iii#1.#2.}%
3957 }

```

What's the argument token?

```

3958 \def\collargs@e@iii#1.#2.{%
3959   \def\collargs@e@iv{\collargs@e@v#1.#2.}%
3960   \futurelet\collargs@temp\collargs@e@iv
3961 }

```

If it is a open or close group character, we surely don't have an embellishment.

```

3962 \def\collargs@e@v{%
3963   \ifcat\noexpand\collargs@temp\bgroup\relax
3964     \let\collargs@marshal\collargs@e@z
3965   \else
3966     \ifcat\noexpand\collargs@temp\egroup\relax
3967       \let\collargs@marshal\collargs@e@z
3968     \else
3969       \let\collargs@marshal\collargs@e@vi
3970     \fi
3971 \fi
3972 \collargs@marshal
3973 }

```

We borrow the “Does #1 occur within #2?” macro from `pgfutil-common`, but we fix it by executing `\collargs@in@@` in a braced group. This will prevent an `&` in an argument to function as an alignment character; the minor price to pay is that we assign the conditional globally.

```

3974 \newif\ifcollargs@in@
3975 \def\collargs@in@#1#2{%
3976   \def\collargs@in@@##1##2##3\collargs@in@@{%
3977     \ifx\collargs@in@@##2\global\collargs@in@false\else\global\collargs@in@true\fi
3978   }%
3979   {\collargs@in@@##2#1\collargs@in@\collargs@in@@}%
3980 }

```

Let's see whether the following token, now #3, is an embellishment token.

```

3981 \def\collargs@e@vi#1.#2.#3{%
3982   \collargs@in@{#3}{#1}%
3983   \ifcollargs@in@

```



```

3984 \expandafter\collargs@e@vii
3985 \else
3986 \expandafter\collargs@e@z
3987 \fi
3988 #1.#2.#3%
3989 }

```

#3 is the current embellishment token. We'll collect its argument using `\collargs@m`, but to do that, we have to (locally) redefine `\collargs@appendarg` and `\collargs@`, which get called by `\collargs@m`.

```
3990 \def\collargs@e@vii#1.#2.#3{%
```

We'll have to execute the original `\collargs@appendarg` later, so let's remember it. The temporary `\collargs@appendarg` simply stores the collected argument into `\collargsArg` — we'll do the processing etc. later.

```

3991 \let\collargs@real@appendarg\collargs@appendarg
3992 \def\collargs@appendarg##1{\collargsArg{##1}}%

```

Once `\collargs@m` is done, it will call the redefined `\collargs@` and thereby get us back into this handler.

```

3993 \def\collargs@{\collargs@e@viii#1.#3}%
3994 \collargs@m#2.%
3995 }

```

The parameters here are as follows. #1 are the embellishment tokens, and #2 is the current embellishment token; these get here via our local redefinition of `\collargs@` in `\collargs@e@vii`. #3 are the rest of the argument specification, which is put behind control sequence `\collargs@` by the `m` handler.

```
3996 \def\collargs@e@viii#1.#2#3.{%
```

Our wrapper puts the current embellishment token in front of the collected embellishment argument. Note that if the embellishment argument was in braces, `\collargs@m` has already set one wrapper (which will apply first).

```
3997 \collargs@wrap{#2##1}%
```

We need to get rid of the current embellishment from embellishments, not to catch the same embellishment twice.

```

3998 \def\collargs@e@ix##1#2{\collargs@e@x##1}%
3999 \collargs@e@ix#1.#3.%
4000 }

```

When this is executed, the input stream starts with the (remaining) embellishment tokens, followed by a dot, then the rest of the argument specification, also followed by a dot.

```
4001 \def\collargs@e@x{%
```

Process the argument and append it to the storage.

```
4002 \expandafter\collargs@real@appendarg\expandafter{\the\collargsArg}%
```

`\collargs@real@appendarg` has closed a group, so we open it again, and start looking for another embellishment token in the input stream.

```

4003 \begingroup
4004 \collargs@e@ii
4005 }

```

The first argument token is not an embellishment token. We finish by consuming the list of embellishment tokens, closing the two groups opened by this handler, and reexecuting the central loop.

```
4006 \def\collargs@e@z#1.{\endgroup\endgroup\collargs@}
```

`\collargs@E` Discard the defaults and execute e.

```
4007 \def\collargs@E#1#2{\collargs@e#{#1}}
```

8.2.5 The verbatim modes

`\collargsVerbatim` These macros set the two verbatim-related conditionals, `\ifcollargs@verbatim` and `\collargsVerb` `\ifcollargs@verbatimbraces`, and then call `\collargs@make@verbatim` to effect the requested category code changes (among other things). A group should be opened prior to executing either of them. After execution, they are redefined to minimize the effort needed to enter into another mode in an embedded group. Below, we first define all the possible transitions.

```
4008 \let\collargs@NoVerbatimAfterNoVerbatim\relax
4009 \def\collargs@VerbAfterNoVerbatim{%
4010   \collargs@verbatimtrue
4011   \collargs@verbatimbracesfalse
4012   \collargs@make@verbatim
4013   \collargs@after{Verb}%
4014 }
4015 \def\collargs@VerbatimAfterNoVerbatim{%
4016   \collargs@verbatimtrue
4017   \collargs@verbatimbracestrue
4018   \collargs@make@verbatim
4019   \collargs@after{Verbatim}%
4020 }
4021 \def\collargs@NoVerbatimAfterVerb{%
4022   \collargs@verbatimfalse
4023   \collargs@verbatimbracesfalse
4024   \collargs@make@other@groups
4025   \collargs@make@no@verbatim
4026   \collargs@after{NoVerbatim}%
4027 }
4028 \def\collargs@VerbAfterVerb{%
4029   \collargs@make@other@groups
4030 }
4031 \def\collargs@VerbatimAfterVerb{%
4032   \collargs@verbatimbracestrue
4033   \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters of category “other”.

```
4034 \def\collargs@do##1{\catcode##1=12 }%
4035 \collargs@bgroups
4036 \collargs@egroups
4037 \collargs@after{Verbatim}%
4038 }%
4039 \let\collargs@NoVerbatimAfterVerbatim\collargs@NoVerbatimAfterVerb
4040 \def\collargs@VerbAfterVerbatim{%
4041   \collargs@verbatimbracesfalse
4042   \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters be of their normal category.

```
4043 \def\collargs@do##1{\catcode##1=1 }%
4044 \collargs@bgroups
```

```

4045 \def\collargs@do##1{\catcode##1=2 }%
4046 \collargs@egroups
4047 \collargs@after{Verb}%
4048 }%
4049 \let\collargs@VerbatimAfterVerbatim\collargs@VerbAfterVerb

```

This macro expects #1 to be the mode just entered (Verbatim, Verb or NoVerbatim), and points macros \collargsVerbatim, \collargsVerb and \collargsNoVerbatim to the appropriate transition macro.

```

4050 \def\collargs@after#1{%
4051 \letcs\collargsVerbatim{collargs@VerbatimAfter#1}%
4052 \letcs\collargsVerb{collargs@VerbAfter#1}%
4053 \letcs\collargsNoVerbatim{collargs@NoVerbatimAfter#1}%
4054 }

```

The first transition is always from the non-verbatim mode.

```

4055 \collargs@after{NoVerbatim}

```

\collargs@bgroups Initialize the lists of the current grouping characters used in the redefinitions of macros **\collargs@egroups** **\collargsVerbatim** and **\collargsVerb** above. Each entry is of form **\collargs@do**{*character code*}. These lists will be populated by **\collargs@make@verbatim**. They may be local, as they only used within the group opened for a verbatim environment.

```

4056 \def\collargs@bgroups{}%
4057 \def\collargs@egroups{}%

```

\collargs@cc This macro recalls the category code of character #1. In LuaTeX, we simply look up the category code in the original category code table; in other engines, we have stored the original category code into **\collargs@cc@***character code* by **\collargs@make@verbatim**. (Note that #1 is a character, not a number.)

```

4058 \ifdefined\luatexversion
4059 \def\collargs@cc#1{%
4060 \directlua{tex.sprint(tex.getcatcode(\collargs@catcodetable@original,
4061 \the\numexpr\expandafter`\csname#1\endcsname\relax))}%
4062 }
4063 \else
4064 \def\collargs@cc#1{%
4065 \ifcsname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
4066 \csname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
4067 \else
4068 12%
4069 \fi
4070 }
4071 \fi

```

\collargs@other@bgroup Macros **\collargs@other@bgroup** and **\collargs@other@egroup** hold the characters **\collargs@other@egroup** of category code “other” which will play the role of grouping characters in the **\collargs@Braces** full verbatim mode. They are usually defined when entering a verbatim mode in **\collargs@make@verbatim**, but may be also set by the user via **\collargs@Braces** (it is not even necessary to select characters which indeed have the grouping function in the outside category code regime). The setting process is indirect: executing **\collargs@Braces** merely sets **\collargs@make@other@groups**, which gets executed by the subsequent **\collargsVerbatim**, **\collargsVerb** or **\collargsNoVerbatim** (either directly or via **\collargs@make@verbatim**).

```

4072 \def\collargs@Braces#1{%
4073 \expandafter\collargs@braces@i\detokenize{#1}\relax
4074 }
4075 \def\collargs@braces@i#1#2#3\relax{%

```

```

4076 \def\collargs@make@other@groups{%
4077   \def\collargs@other@bgroup{#1}%
4078   \def\collargs@other@egroup{#2}%
4079 }%
4080 }
4081 \def\collargs@make@other@groups{}

```

`\collargs@catcodetable@verbatim` We declare several new catcode tables in LuaTeX, the most important `\catcodetable@atletter` one being `\collargs@catcodetable@verbatim`, where all characters have `\collargs@catcodetable@initex` category code 12. We only need the other two tables in some formats: `\collargs@catcodetable@atletter` holds the catcode in effect at the time of loading the package, and `\collargs@catcodetable@initex` is the iniTeX table.

```

4082 \ifdefined\luatexversion
4083 (*latex, context)
4084 \newcatcodetable\collargs@catcodetable@verbatim
4085 (latex) \let\collargs@catcodetable@atletter\catcodetable@atletter
4086 (context) \newcatcodetable\collargs@catcodetable@atletter
4087 (/latex, context)
4088 (*plain)
4089 \ifdefined\collargs@catcodetable@verbatim\else
4090   \chardef\collargs@catcodetable@verbatim=4242
4091   \fi
4092   \chardef\collargs@catcodetable@atletter=%
4093     \number\numexpr\collargs@catcodetable@verbatim+1\relax
4094   \chardef\collargs@catcodetable@initex=%
4095     \number\numexpr\collargs@catcodetable@verbatim+2\relax
4096   \initcatcodetable\collargs@catcodetable@initex
4097 (/plain)
4098 (plain, context) \savecatcodetable\collargs@catcodetable@atletter
4099 \begingroup
4100 \@firstofone{%
4101 (latex) \catcodetable\catcodetable@initex
4102 (plain) \catcodetable\collargs@catcodetable@initex
4103 (context) \catcodetable\inicatcodes
4104 \catcode`\=12
4105 \catcode13=12
4106 \catcode0=12
4107 \catcode32=12
4108 \catcode`\%=12
4109 \catcode127=12
4110 \def\collargs@do#1{\catcode#1=12 }%
4111 \collargs@forrange{`a}{`z}%
4112 \collargs@forrange{`A}{`Z}%
4113 \savecatcodetable\collargs@catcodetable@verbatim
4114 \endgroup
4115 }%
4116 \fi

```

`verbatim ranges` This key and macro set the character ranges to which the verbatim mode will apply (in `\collargsVerbatimRanges` pdfTeX and XeTeX), or which will be inspected for grouping and comment characters `\collargs@verbatim@ranges` (in LuaTeX). In pdfTeX, the default value 0–255 should really remain unchanged.

```

4117 \collargsSet{
4118   verbatim ranges/.store in=\collargs@verbatim@ranges,
4119 }
4120 \def\collargsVerbatimRanges#1{\def\collargs@verbatim@ranges{#1}}
4121 \def\collargs@verbatim@ranges{0-255}

```

`\collargs@make@verbatim` This macro changes the category code of all characters to “other” — except the grouping characters in the partial verbatim mode. While doing that, it also stores (unless we’re in LuaTeX) the current category codes into `\collargs@cc@{character code}` (easily recallable by

`\collargs@cc`), redefines the “primary” grouping characters `\collargs@make@other@bgroup` and `\collargs@make@other@egroup` if necessary, and “remembers” the grouping characters (storing them into `\collargs@bgroups` and `\collargs@egroups`) and the comment characters (storing them into `\collargs@comments`).

In Lua_T_E_X, we can use catcode tables, so we change the category codes by switching to category code table `\collargs@catcodetable@verbatim`. In other engines, we have to change the codes manually. In order to offer some flexibility in X_Y_T_E_X, we perform the change for characters in `verbatim` ranges.

```

4122 \ifdefined\luatexversion
4123   \def\collargs@make@verbatim{%
4124     \directlua{%
4125       for from, to in string.gmatch(
4126         "\luaescapestring{\collargs@verbatim@ranges}",
4127         "(\collargs@percentchar d+)-(\collargs@percentchar d+)"
4128       ) do
4129         for char = tex.round(from), tex.round(to) do
4130           catcode = tex.catcode[char]

```

For category codes 1, 2 and 14, we have to call macros `\collargs@make@verbatim@bgroup`, `\collargs@make@verbatim@egroup` and `\collargs@make@verbatim@comment`, same as for engines other than Lua_T_E_X.

```

4131         if catcode == 1 then
4132           tex.sprint(
4133             \number\collargs@catcodetable@atletter,
4134             "\noexpand\collargs@make@verbatim@bgroup{" .. char .. "}")
4135         elseif catcode == 2 then
4136           tex.sprint(
4137             \number\collargs@catcodetable@atletter,
4138             "\noexpand\collargs@make@verbatim@egroup{" .. char .. "}")
4139         elseif catcode == 14 then
4140           tex.sprint(
4141             \number\collargs@catcodetable@atletter,
4142             "\noexpand\collargs@make@verbatim@comment{" .. char .. "}")
4143         end
4144       end
4145     end
4146   }%
4147   \edef\collargs@catcodetable@original{\the\catcodetable}%
4148   \catcodetable\collargs@catcodetable@verbatim

```

Even in Lua_T_E_X, we switch between the `verbatim` braces regimes by hand.

```

4149   \ifcollargs@verbatimbraces
4150   \else
4151     \def\collargs@do##1{\catcode##1=1\relax}%
4152     \collargs@bgroups
4153     \def\collargs@do##1{\catcode##1=2\relax}%
4154     \collargs@egroups
4155   \fi
4156 }
4157 \else

```

The non-Lua_T_E_X version:

```

4158 \def\collargs@make@verbatim{%
4159   \ifdefempty\collargs@make@other@groups{}-%

```

The user has executed `\collargsBraces`. We first apply that setting by executing macro `\collargs@make@other@groups`, and then disable our automatic setting of the primary grouping characters.

```

4160     \collargs@make@other@groups
4161     \def\collargs@make@other@groups{}%
4162     \let\collargs@make@other@bgroup\@gobble
4163     \let\collargs@make@other@egroup\@gobble
4164     }%

```

Initialize the list of current comment characters. Each entry is of form `\collargs@do{<character code>}`. The definition must be global, because the macro will be used only once we exit the current group (by `\collargs@fix@cc@from@other@comment`, if at all).

```

4165     \gdef\collargs@comments{}%
4166     \let\collargs@do\collargs@make@verbatim@char
4167     \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
4168     }
4169     \def\collargs@make@verbatim@char#1{%

```

Store the current category code of the current character.

```

4170     \ifnum\catcode#1=12
4171     \else
4172         \csedef{collargs@cc@#1}{\the\catcode#1}%
4173     \fi
4174     \ifnum\catcode#1=1
4175         \collargs@make@verbatim@bgroup{#1}%
4176     \else
4177         \ifnum\catcode#1=2
4178             \collargs@make@verbatim@egroup{#1}%
4179         \else
4180             \ifnum\catcode#1=14
4181                 \collargs@make@verbatim@comment{#1}%
4182             \fi

```

Change the category code of the current character (including the comment characters).

```

4183         \ifnum\catcode#1=12
4184         \else
4185             \catcode#1=12\relax
4186         \fi
4187     \fi
4188 \fi
4189 }
4190 \fi

```

`\collargs@make@verbatim@bgroup` This macro changes the category of the opening group character to “other”, but only in the full verbatim mode. Next, it populates `\collargs@bgroups`, to facilitate the potential transition into the other verbatim mode. Finally, it executes `\collargs@make@other@bgroup`, which stores the “other” variant of the current character into `\collargs@other@bgroup`, and automatically disables itself, so that it is only executed for the first encountered opening group character — unless it was already `\relaxed` at the top of `\collargs@make@verbatim` as a consequence of the user executing `\collargsBraces`.

```

4191 \def\collargs@make@verbatim@bgroup#1{%
4192     \ifcollargs@verbatimbraces
4193         \catcode#1=12\relax
4194     \fi
4195     \appto\collargs@bgroups{\collargs@do{#1}}%
4196     \collargs@make@other@bgroup{#1}%
4197 }
4198 \def\collargs@make@other@bgroup#1{%
4199     \collargs@make@char\collargs@other@bgroup{#1}{12}%
4200     \let\collargs@make@other@bgroup\@gobble
4201 }

```

`\collargs@make@verbatim@egroup` Ditto for the closing group character.

```
4202 \def\collargs@make@verbatim@egroup#1{%
4203   \ifcollargs@verbatimbraces
4204     \catcode#1=12\relax
4205   \fi
4206   \appto\collargs@egroups{\collargs@do{#1}}%
4207   \collargs@make@other@egroup{#1}%
4208 }
4209 \def\collargs@make@other@egroup#1{%
4210   \collargs@make@char\collargs@other@egroup{#1}{12}%
4211   \let\collargs@make@other@egroup\@gobble
4212 }
```

`\collargs@make@verbatim@comment` This macro populates `\collargs@make@comments@other`.

```
4213 \def\collargs@make@verbatim@comment#1{%
4214   \gappto\collargs@comments{\collargs@do{#1}}%
4215 }
```

`\collargs@make@no@verbatim` This macro switches back to the non-verbatim mode: in LuaTeX, by switching to the original catcode table; in other engines, by recalling the stored category codes.

```
4216 \ifdefined\luatexversion
4217   \def\collargs@make@no@verbatim{%
4218     \catcodetable\collargs@catcodetable@original\relax
4219   }%
4220 \else
4221   \def\collargs@make@no@verbatim{%
4222     \let\collargs@do\collargs@make@no@verbatim@char
4223     \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
4224   }
4225 \fi
4226 \def\collargs@make@no@verbatim@char#1{%
```

The original category code of a character was stored into `\collargs@cc@character code` by `\collargs@make@verbatim`. (We don't use `\collargs@cc`, because we have a number.)

```
4227   \ifcsname collargs@cc@#1\endcsname
4228     \catcode#1=\csname collargs@cc@#1\endcsname\relax
```

We don't have to restore category code 12.

```
4229   \fi
4230 }
```

8.2.6 Transition between the verbatim and the non-verbatim mode

At the transition from verbatim to non-verbatim mode, and vice versa, we sometimes have to fix the category code of the next argument token. This happens when we have an optional argument type in one mode followed by an argument type in another mode, but the optional argument is absent, or when an optional, but absent, verbatim argument is the last argument in the specification. The problem arises because the presence of optional arguments is determined by looking ahead in the input stream; when the argument is absent, this means that we have fixed the category code of the next token. CollArgs addresses this issue by noting the situations where a token receives the wrong category code, and then does its best to replace that token with the same character of the appropriate category code.

`\ifcollargs@fix@requested` This conditional is set, globally, by the optional argument handlers when the argument is in fact absent, and reset in the central loop after applying the fix if necessary.

```
4231 \newif\ifcollargs@fix@requested
```

`\collargs@fix` This macro selects the fixer appropriate to the transition between the previous verbatim mode (determined by `\ifcollargs@last@verbatim` and `\ifcollargs@last@verbatimbraces`) and the current verbatim mode (which is determined by macros `\ifcollargs@verbatim` and `\ifcollargs@verbatimbraces`); if the category code `fix` was not requested (for this, we check `\ifcollargs@fix@requested`), the macro simply executes the next-code given as the sole argument. The name of the fixer macro has the form `\collargs@fix@<last mode>to<current mode>`, where the modes are given by mnemonic codes: `V` = full verbatim, `v` = partial verbatim, and `N` = non-verbatim.

```
4232 \long\def\collargs@fix#1{%
```

Going through `\edef + \unexpanded` avoids doubling the hashes.

```
4233 \edef\collargs@fix@next{\unexpanded{#1}}%
4234 \ifcollargs@fix@requested
4235 \letcs\collargs@action{collargs@fix@%
4236 \ifcollargs@last@verbatim
4237 \ifcollargs@last@verbatimbraces V\else v\fi
4238 \else
4239 N%
4240 \fi
4241 to%
4242 \ifcollargs@verbatim
4243 \ifcollargs@verbatimbraces V\else v\fi
4244 \else
4245 N%
4246 \fi
4247 }%
4248 \else
4249 \let\collargs@action\collargs@fix@next
4250 \fi
4251 \collargs@action
4252 }
```

`\collargs@fix@NtoN` Nothing to do, continue with the next-code.

`\collargs@fix@vtov`

```
4253 \def\collargs@fix@NtoN{\collargs@fix@next}
4254 \let\collargs@fix@vtov\collargs@fix@NtoN
4255 \let\collargs@fix@VtoV\collargs@fix@NtoN
```

`\collargs@fix@NtoV` We do nothing for the group tokens; for other tokens, we redirect to `\collargs@fix@NtoV`.

```
4256 \def\collargs@fix@Ntov{%
4257 \futurelet\collargs@temp\collargs@fix@cc@to@other@ii
4258 }
4259 \def\collargs@fix@cc@to@other@ii{%
4260 \ifcat\noexpand\collargs@temp\bgroup
4261 \let\collargs@action\collargs@fix@next
4262 \else
4263 \ifcat\noexpand\collargs@temp\egroup
4264 \let\collargs@action\collargs@fix@next
4265 \else
4266 \let\collargs@action\collargs@fix@NtoV
4267 \fi
4268 \fi
4269 \collargs@action
4270 }
```

`\collargs@fix@NtoV` The only complication here is that we might be in front of a control sequence that was a result of a previous fix in the other direction.

```
4271 \def\collargs@fix@NtoV{%
```



```

4272 \ifcollargs@double@fix
4273   \ifcollargs@in@second@fix
4274     \expandafter\expandafter\expandafter\collargs@fix@NtoV@secondfix
4275   \else
4276     \expandafter\expandafter\expandafter\collargs@fix@NtoV@onemore
4277   \fi
4278 \else
4279   \expandafter\collargs@fix@NtoV@singlefix
4280 \fi
4281 }

```

This is the usual situation of a single fix. We just use `\string` on the next token here (but note that some situations can't be saved: noone can bring a comment back to life, or distinguish a newline and a space)

```

4282 \def\collargs@fix@NtoV@singlefix{%
4283   \expandafter\collargs@fix@next\string
4284 }

```

If this is the first fix of two, we know #1 is a control sequence, so it is safe to grab it.

```

4285 \def\collargs@fix@NtoV@onemore#1{%
4286   \collargs@do@one@more@fix{%
4287     \expandafter\collargs@fix@next\string#1%
4288   }%
4289 }

```

If this is the second fix of the two, we have to check whether the next token is a control sequence, and if it is, we need to remember it. Afterwards, we redirect to the single-fix.

```

4290 \def\collargs@fix@NtoV@secondfix{%
4291   \if\noexpand\collargs@temp\relax
4292     \expandafter\collargs@fix@NtoV@secondfix@i
4293   \else
4294     \expandafter\collargs@fix@NtoV@singlefix
4295   \fi
4296 }
4297 \def\collargs@fix@NtoV@secondfix@i#1{%
4298   \gdef\collargs@double@fix@cs@ii{#1}%
4299   \collargs@fix@NtoV@singlefix#1%
4300 }

```

`\collargs@fix@vtoN` Do nothing for the grouping tokens, redirect to `\collargs@fix@VtoN` for other tokens.

```

4301 \def\collargs@fix@vtoN{%
4302   \futurelet\collargs@token\collargs@fix@vtoN@i
4303 }
4304 \def\collargs@fix@vtoN@i{%
4305   \ifcat\noexpand\collargs@token\bgroup
4306     \expandafter\collargs@fix@next
4307   \else
4308     \ifcat\noexpand\collargs@token\egroup
4309     \expandafter\expandafter\expandafter\collargs@fix@next
4310   \else
4311     \expandafter\expandafter\expandafter\collargs@fix@VtoN
4312   \fi
4313 \fi
4314 }

```

`\collargs@fix@vtoV` Redirect group tokens to `\collargs@fix@NtoV`, and do nothing for other tokens.

```

4315 \def\collargs@fix@vtoV{%
4316   \futurelet\collargs@token\collargs@fix@vtoV@i

```

```

4317 }
4318 \def\collargs@fix@vtoV@i{%
4319 \ifcat\noexpand\collargs@token\bgroup
4320 \expandafter\collargs@fix@NtoV
4321 \else
4322 \ifcat\noexpand\collargs@token\egroup
4323 \expandafter\expandafter\expandafter\collargs@fix@NtoV
4324 \else
4325 \expandafter\expandafter\expandafter\collargs@fix@next
4326 \fi
4327 \fi
4328 }

```

`\collargs@fix@Vtov` Redirect group tokens to `\collargs@fix@VtoN`, and do nothing for other tokens. #1 is surely of category 12, so we can safely grab it.

```

4329 \def\collargs@fix@catcode@of@braces@fromverbatim#1{%
4330 \ifnum\catcode`#1=1
4331 \expandafter\collargs@fix@VtoN
4332 \expandafter#1%
4333 \else
4334 \ifnum\catcode`#1=2
4335 \expandafter\expandafter\expandafter\collargs@fix@cc@VtoN
4336 \expandafter\expandafter\expandafter#1%
4337 \else
4338 \expandafter\expandafter\expandafter\collargs@fix@next
4339 \fi
4340 \fi
4341 }

```

`\collargs@fix@VtoN` This is the only complicated part. Control sequences and comments (but not grouping characters!) require special attention. We're fine to grab the token right away, as we know it is of category 12.

```

4342 \def\collargs@fix@VtoN#1{%
4343 \ifnum\catcode`#1=0
4344 \expandafter\collargs@fix@VtoN@escape
4345 \else
4346 \ifnum\catcode`#1=14
4347 \expandafter\expandafter\expandafter\collargs@fix@VtoN@comment
4348 \else
4349 \expandafter\expandafter\expandafter\collargs@fix@VtoN@token
4350 \fi
4351 \fi
4352 #1%
4353 }

```

`\collargs@fix@VtoN@token` We create a new character with the current category code behind the next-code. This works even for grouping characters.

```

4354 \def\collargs@fix@VtoN@token#1{%
4355 \collargs@insert@char\collargs@fix@next{`#1}{\the\catcode`#1}%
4356 }

```

`\collargs@fix@VtoN@comment` This macro defines a macro which will, when placed at a comment character, remove the tokens until the end of the line. The code is adapted from the TeX.SE answer at tex.stackexchange.com/a/10454/16819 by Bruno Le Floch.

```

4357 \def\collargs@defcommentstripper#1#2{%

```

We chuck a parameter into the following definition, to grab the (verbatim) comment character. This is why this macro must be executed precisely before the (verbatim) comment character.

```

4358 \def#1##1{%
4359   \begingroup%
4360   \escapechar=`\%
4361   \catcode\endlinechar=\active%

```

We assign the “other” category code to comment characters. Without this, comment characters behind the first one make trouble: there would be no \^M at the end of the line, so the comment stripper would gobble the following line as well; in fact, it would gobble all subsequent lines containing a comment character. We also make sure to change the category code of *all* comment characters, even if there is usually just one.

```

4362   \def\collargs@do####1{\catcode###1=12 }%
4363   \collargs@comments
4364   \csname\string#1\endcsname%
4365 }%
4366 \begingroup%
4367 \escapechar=`\%
4368 \lccode`\~=\endlinechar%
4369 \lowercase{%
4370   \expandafter\endgroup
4371   \expandafter\def\csname\string#1\endcsname##1~%
4372 }{%

```

I have removed `\space` from the end of the following line. We don’t want it for our application.

```

4373   \endgroup#2%
4374 }%
4375 }
4376 \collargs@defcommentstripper\collargs@fix@VtoN@comment{%
4377   \collargs@fix@next
4378 }

```

We don’t need the generator any more.

```

4379 \let\collargs@defcommentstripper\relax

```

`\collargs@fix@VtoN@escape` An escape character of category code 12 is the most challenging — and we won’t get things completely right — as we have swim further down the input stream to create a control sequence. This macro will throw away the verbatim escape character #1.

```

4380 \def\collargs@fix@VtoN@escape#1{%
4381   \ifcollargs@double@fix

```

We need to do things in a special way if we’re in the double-fix situation triggered by the previous fixing of a control sequence (probably this very one). In that case, we can’t collect it in the usual way because the entire control sequence is spelled out in verbatim.

```

4382   \expandafter\collargs@fix@VtoN@escape@d
4383   \else

```

This here is the usual situation where the escape character was tokenized verbatim, but the control sequence name itself will be collected (right away) in the non-verbatim regime.

```

4384   \expandafter\collargs@fix@VtoN@escape@i
4385   \fi
4386 }
4387 \def\collargs@fix@VtoN@escape@i{%

```

The sole character forming a control symbol name may be of any category. Temporarily redefining the category codes of the craziest characters allows `\collargs@fix@VtoN@escape@ii` to simply grab the following character.

```

4388   \begingroup

```

```

4389 \catcode`\ =12
4390 \catcode`\{=12
4391 \catcode`\}=12
4392 \catcode`\_ =12
4393 \collargs@fix@VtoN@escape@ii
4394 }

```

The argument is the first character of the control sequence name.

```

4395 \def\collargs@fix@VtoN@escape@ii#1{%
4396 \endgroup
4397 \def\collargs@csname{#1}%

```

Only if #1 is a letter may the control sequence name continue.

```

4398 \ifnum\catcode`#1=11
4399 \expandafter\collargs@fix@VtoN@escape@iii
4400 \else

```

In the case of a control space, we have to throw away the following spaces.

```

4401 \ifnum\catcode`#1=10
4402 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@s
4403 \else

```

We have a control symbol. That means that we haven't peeked ahead and can thus skip `\collargs@fix@VtoN@escape@z`.

```

4404 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@z@i
4405 \fi
4406 \fi
4407 }

```

We still have to collect the rest of the control sequence name. Braces have their usual meaning again, so we have to check for them explicitly (and bail out if we stumble upon them).

```

4408 \def\collargs@fix@VtoN@escape@iii{%
4409 \futurelet\collargs@temp\collargs@fix@VtoN@escape@iv
4410 }
4411 \def\collargs@fix@VtoN@escape@iv{%
4412 \ifcat\noexpand\collargs@temp\bgroup
4413 \let\collargs@action\collargs@fix@VtoN@escape@z
4414 \else
4415 \ifcat\noexpand\collargs@temp\egroup
4416 \let\collargs@action\collargs@fix@VtoN@escape@z
4417 \else
4418 \expandafter\ifx\space\collargs@temp
4419 \let\collargs@action\collargs@fix@VtoN@escape@s
4420 \else
4421 \let\collargs@action\collargs@fix@VtoN@escape@v
4422 \fi
4423 \fi
4424 \fi
4425 \collargs@action
4426 }

```

If we have a letter, store it and loop back, otherwise finish.

```

4427 \def\collargs@fix@VtoN@escape@v#1{%
4428 \ifcat\noexpand#1a%
4429 \appto\collargs@csname{#1}%
4430 \expandafter\collargs@fix@VtoN@escape@iii
4431 \else
4432 \expandafter\collargs@fix@VtoN@escape@z\expandafter#1%
4433 \fi
4434 }

```

Throw away the following spaces.

```
4435 \def\collargs@fix@VtoN@escape@s{%
4436   \futurelet\collargs@temp\collargs@fix@VtoN@escape@s@i
4437 }
4438 \def\collargs@fix@VtoN@escape@s@i{%
4439   \expandafter\ifx\space\collargs@temp
4440     \expandafter\collargs@fix@VtoN@escape@s@ii
4441   \else
4442     \expandafter\collargs@fix@VtoN@escape@z
4443   \fi
4444 }
4445 \def\collargs@fix@VtoN@escape@s@ii{%
4446   \expandafter\collargs@fix@VtoN@escape@z\romannumeral-0%
4447 }
```

Once we have collected the control sequence name into `\collargs@csname`, we will create the control sequence behind the next-code. However, we have two complications. The minor one is that `\csname` defines an unexisting control sequence to mean `\relax`, so we have to check whether the control sequence we will create is defined, and if not, “undefine” it in advance.

```
4448 \def\collargs@fix@VtoN@escape@z@i{%
4449   \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
4450   \collargs@fix@VtoN@escape@z@ii
4451 }%
4452 \def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin{%
4453   \ifcsname\collargs@csname\endcsname
4454     \@tempwattrue
4455   \else
4456     \@tempwafalse
4457   \fi
4458 }
4459 \def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end{%
4460   \if@tempswa
4461   \else
4462     \cslet{\collargs@csname}\collargs@undefined
4463   \fi
4464 }
4465 \def\collargs@fix@VtoN@escape@z@ii{%
4466   \expandafter\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
4467   \expandafter\collargs@fix@next\csname\collargs@csname\endcsname
4468 }
```

The second complication is much greater, but it only applies to control words and spaces, and that’s why control symbols went directly to the macro above. Control words and spaces will only get there via a detour through the following macro.

The problem is that collecting the control word/space name peeked ahead in the stream, so the character following the control sequence (name) is already tokenized. We will (at least partially) address this by requesting a “double-fix”: until the control sequence we’re about to create is consumed into some argument, each category code fix will fix two “tokens” rather than one.

```
4469 \def\collargs@fix@VtoN@escape@z{%
4470   \collargs@if@one@more@fix{%
```

Some previous fixing has requested a double fix, so let’s do it. Afterwards, redirect to the control symbol code `\collargs@fix@VtoN@escape@z@i`. It will surely use the correct `\collargs@csname` because we do the second fix in a group.

```
4471   \collargs@do@one@more@fix\collargs@fix@VtoN@escape@z@i
4472 }-%
```

Remember the collected control sequence. It will be used in `\collargs@cancel@double@fix`.

```
4473 \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
4474 \xdef\collargs@double@fix@cs@i{\expandonce{\csname\collargs@csname\endcsname}}%
4475 \collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
```

Request the double-fix.

```
4476 \global\collargs@double@fixtrue
```

The complication is addressed, redirect to the control symbol finish.

```
4477 \collargs@fix@VtoN@escape@z@ii
4478 }%
4479 }
```

When we have to “redo” a control sequence, because it was ping-ponged back into the verbatim mode, we cannot collect it by `\collargs@fix@VtoN@escape@i`, because it is spelled out entirely in verbatim. However, we have seen this control sequence before, and remembered it, so we’ll simply grab it. Another complication is that we might be either at the “first” control sequence, whose fixing created all these double-fix trouble, or at the “second” control sequence, if the first one was immediately followed by another one. But we have remembered both of them: the first one in `\collargs@fix@VtoN@escape@z`, the second one in `\collargs@fix@NtoV@secondfix`.

```
4480 \def\collargs@fix@VtoN@escape@d{%
4481 \ifcollargs@in@second@fix
4482 \expandafter\collargs@fix@VtoN@escape@d@i
4483 \expandafter\collargs@double@fix@cs@ii
4484 \else
4485 \expandafter\collargs@fix@VtoN@escape@d@i
4486 \expandafter\collargs@double@fix@cs@i
4487 \fi
4488 }
```

We have the contents of either `\collargs@double@fix@cs@i` or `\collargs@double@fix@cs@ii` here, a control sequence in both cases.

```
4489 \def\collargs@fix@VtoN@escape@d@i#1{%
4490 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@d@ii
4491 \expandafter\string#1\relax
4492 }
```

We have the verbatimized control sequence name in #2 (#1 is the escape character). By storing it into `\collargs@csname`, we pretend we have collected it. By defining and executing `\collargs@fix@VtoN@escape@d@iii`, we actually gobble it from the input stream. Finally, we reroute to `\collargs@fix@VtoN@escape@z`.

```
4493 \def\collargs@fix@VtoN@escape@d@ii#1#2\relax{%
4494 \def\collargs@csname{#2}%
4495 \def\collargs@fix@VtoN@escape@d@iii#2{%
4496 \collargs@fix@VtoN@escape@z
4497 }%
4498 \collargs@fix@VtoN@escape@d@iii
4499 }
```

This conditional signals a double-fix request. It should be always set globally, because it is cleared by `\collargs@double@fixfalse` in a group.

```
4500 \newif\ifcollargs@double@fix
```

This conditional signals that we’re currently performing the second fix.

```
4501 \newif\ifcollargs@in@second@fix
```

Inspect the two conditionals above to decide whether we have to perform another fix: if so, execute the first argument, otherwise the second one. This macro is called only from `\collargs@fix@VtoN@escape@z` and `\collargs@fix@NtoV`, because these are the only two places where we might need the second fix, ping-ponging a control sequence between the verbatim and the non-verbatim mode.

```

4502 \def\collargs@if@one@more@fix{%
4503   \ifcollargs@double@fix
4504     \ifcollargs@in@second@fix
4505       \expandafter\expandafter\expandafter\@secondoftwo
4506     \else
4507       \expandafter\expandafter\expandafter\@firstoftwo
4508     \fi
4509   \else
4510     \expandafter\@secondoftwo
4511   \fi
4512 }
4513 \def\collargs@do@one@more@fix#1{%

```

We perform the second fix in a group, signalling that we’re performing it.

```

4514   \begingroup
4515   \collargs@in@second@fixtrue

```

Reexecute the fixing routine, at the end, close the group and execute the given code afterwards.

```

4516   \collargs@fix{%
4517     \endgroup
4518     #1%
4519   }%
4520 }

```

This macro is called from `\collargs@appendarg` to cancel the double-fix request.

```

4521 \def\collargs@cancel@double@fix{%

```

`\collargs@appendarg` is only executed when something was actually consumed. We thus know that at least one of the problematic “tokens” is gone, so the double fix is not necessary anymore.

```

4522   \global\collargs@double@fixfalse

```

What we have to figure out, still, is whether both problematic “tokens” we consumed. If so, no more fixing is required. But if only one of them was consumed, we need to request the normal, single, fix for the remaining “token”.

```

4523   \begingroup

```

This will attach the delimiters directly to the argument, so we’ll see what was actually consumed.

```

4524   \collargs@process@arg

```

We compare what was consumed when collecting the current argument with the control word that triggered double-fixing. If they match, only the offending control word was consumed, so we need to set the fix request to true for the following token.

```

4525   \edef\collargs@temp{\the\collargsArg}%
4526   \edef\collargs@tempa{\expandafter\string\collargs@double@fix@cs@i}%
4527   \ifx\collargs@temp\collargs@tempa
4528     \global\collargs@fix@requestedtrue
4529   \fi
4530   \endgroup
4531 }

```

`\collargs@insert@char` These macros create a character of character code #2 and category code #3. The first macro `\collargs@make@char` inserts it into the stream behind the code in #1; the second one defines the control sequence in #1 to hold the created character (clearly, it should not be used for categories 1 and 2).

We use the facilities of Lua_TE_X, X_Y_TE_X and L^AT_EX where possible. In the end, we only have to implement our own macros for plain pdf_TE_X.

```

4532 (!context) \ifdefined\luatexversion
4533   \def\collargs@insert@char#1#2#3{%
4534     \edef\collargs@temp{\unexpanded{#1}}%
4535     \expandafter\collargs@temp\directlua{%
4536       tex.cprint(\number#3,string.char(\number#2))}%
4537   }%
4538   \def\collargs@make@char#1#2#3{%
4539     \edef#1{\directlua{tex.cprint(\number#3,string.char(\number#2))}}%
4540   }%
4541 (*!context)
4542 \else
4543   \ifdefined\XeTeXversion
4544     \def\collargs@insert@char#1#2#3{%
4545       \edef\collargs@temp{\unexpanded{#1}}%
4546       \expandafter\collargs@temp\Ucharcat #2 #3
4547     }%
4548     \def\collargs@make@char#1#2#3{%
4549       \edef#1{\Ucharcat#2 #3}%
4550     }%
4551   \else
4552     (*!latex)
4553     \ExplSyntaxOn
4554     \def\collargs@insert@char#1#2#3{%
4555       \edef\collargs@temp{\unexpanded{#1}}%
4556       \expandafter\expandafter\expandafter\collargs@temp\char_generate:nn{#2}{#3}%
4557     }%
4558     \def\collargs@make@char#1#2#3{%
4559       \edef#1{\char_generate:nn{#2}{#3}}%
4560     }%
4561     \ExplSyntaxOff
4562     (/!latex)
4563     (*!plain)

```

The implementation is inspired by `expl3`'s implementation of `\char_generate:nn`, but our implementation is not expandable, for simplicity. We first store an (arbitrary) character `^^@` of category code `n` into control sequence `\collargs@charofcat@n`, for every (implementable) category code.

```

4564   \begingroup
4565   \catcode`^^@=1 \csgdef{collargs@charofcat@1}{%
4566     \noexpand\expandafter^^@\iffalse}\fi}
4567   \catcode`^^@=2 \csgdef{collargs@charofcat@2}{\iffalse{\fi^^@}}
4568   \catcode`^^@=3 \csgdef{collargs@charofcat@3}{^^@}
4569   \catcode`^^@=4 \csgdef{collargs@charofcat@4}{^^@}

```

As we have grabbed the spaces already, a remaining newline should surely be fixed into a `\par`.

```

4570           \csgdef{collargs@charofcat@5}{\par}
4571   \catcode`^^@=6 \csxdef{collargs@charofcat@6}{\unexpanded{^^@}}
4572   \catcode`^^@=7 \csgdef{collargs@charofcat@7}{^^@}
4573   \catcode`^^@=8 \csgdef{collargs@charofcat@8}{^^@}
4574           \csgdef{collargs@charofcat@10}{\noexpand\space}
4575   \catcode`^^@=11 \csgdef{collargs@charofcat@11}{^^@}
4576   \catcode`^^@=12 \csgdef{collargs@charofcat@12}{^^@}
4577   \catcode`^^@=13 \csgdef{collargs@charofcat@13}{^^@}
4578   \endgroup
4579   \def\collargs@insert@char#1#2#3{%

```


Temporarily change the lowercase code of \char" to the requested character #2.

```
4580 \begingroup
4581 \lccode\text{\char"#2}\relax
```

We'll have to close the group before executing the next-code.

```
4582 \def\collargs@temp{\endgroup#1}%
```

$\text{\collargs@charofcat@}$ *requested category code* is f-expanded first, leaving us to lowercase $\text{\expandafter\collargs@temp\text}$. Clearly, lowercasing $\text{\expandafter\collargs@temp}$ is a no-op, but lowercasing \text gets us the requested character of the requested category. \expandafter is executed next, and this gets rid of the conditional for category codes 1 and 2.

```
4583 \expandafter\lowercase\expandafter{%
4584 \expandafter\expandafter\expandafter\collargs@temp
4585 \romannumeral-`0\csname collargs@charofcat@the\numexpr#3\relax\endcsname
4586 }%
4587 }
```

This macro cannot not work for category code 6 (because we assign the result to a macro), but no matter, we only use it for category code 12 anyway.

```
4588 \def\collargs@make@char#1#2#3{%
4589 \begingroup
4590 \lccode\text{\char"#2}\relax
```

Define \collargs@temp to hold \text of the appropriate category.

```
4591 \edef\collargs@temp{%
4592 \csname collargs@charofcat@the\numexpr#3\relax\endcsname}%
```

Preexpand the second \collargs@temp so that we lowercase $\text{\def\collargs@temp{\text}}$, with \text of the appropriate category.

```
4593 \expandafter\lowercase\expandafter{%
4594 \expandafter\def\expandafter\collargs@temp\expandafter{\collargs@temp}%
4595 }%
4596 \expandafter\endgroup
4597 \expandafter\def\expandafter#1\expandafter{\collargs@temp}%
4598 }
4599 </plain>
4600 \fi
4601 \fi
4602 </!context>
```

```
4603 <plain>\resetatcatcode
4604 <context>\stopmodule
4605 <context>\protect
```

Local Variables: TeX-engine: luatex TeX-master: "doc/memoize-code.tex" TeX-auto-save:
nil End:

9 The scripts

9.1 The Perl extraction script `memoize-extract.pl`

```
4606 my $PROG = 'memoize-extract.pl';
4607 my $VERSION = '2024/12/02 v1.4.1';
4608
4609 use strict;
4610 use File::Basename qw/basename/;
4611 use Getopt::Long;
```

```

4612 use File::Spec::Functions
4613     qw/splitpath catpath splitdir rootdir file_name_is_absolute/;
4614 use File::Path qw(make_path);

```

We will only try to import the PDF processing library once we set up the error log. Declare variables for command-line arguments and for `kpathsea` variables. They are defined here so that they are global in the subs which use them.

```

4615 our ($pdf_file, $prune, $keep, $format, $force, $quiet,
4616     $pdf_library, $print_version, $mkdir, $help,
4617     $openin_any, $openout_any, $texmfoutput, $texmf_output_directory);

```

Messages The messages are written both to the extraction log and the terminal (we output to `stdout` rather than `stderr` so that messages on the TeX terminal and document `.log` appear in chronological order). Messages are automatically adapted to the TeX `--format`. The format of the messages. It depends on the given `--format`; the last entry is for `t` the terminal output.

```

4618 my %ERROR = (
4619     latex => '\PackageError{memoize (perl-based extraction)}{${short}}{${long}}',
4620     plain => '\errhelp{${long}}\errmsgmessage{memoize (perl-based extraction): ${short}}',
4621     context => '\errhelp{${long}}\errmsgmessage{memoize (perl-based extraction): ${short}}',
4622     '' => '$header${short}. ${long}');
4623
4624 my %WARNING = (
4625     latex => '\PackageWarning{memoize (perl-based extraction)}{${texindent}$text}',
4626     plain => '\message{memoize (perl-based extraction) Warning: ${texindent}$text}',
4627     context => '\message{memoize (perl-based extraction) Warning: ${texindent}$text}',
4628     '' => '$header${indent}$text. ');
4629
4630 my %INFO = (
4631     latex => '\PackageInfo{memoize (perl-based extraction)}{${texindent}$text}',
4632     plain => '\message{memoize (perl-based extraction): ${texindent}$text}',
4633     context => '\message{memoize (perl-based extraction): ${texindent}$text}',
4634     '' => '$header${indent}$text. ');

```

Some variables used in the message routines; note that `header` will be redefined once we parse the arguments.

```

4635 my $exit_code = 0;
4636 my $log;
4637 my $header = '';
4638 my $indent = '';
4639 my $texindent = '';

```

The message routines.

```

4640 sub error {
4641     my ($short, $long) = @_;
4642     if (! $quiet) {
4643         $_ = $ERROR{' '};
4644         s/\$header/\$header/;
4645         s/\$short/\$short/;
4646         s/\$long/\$long/;
4647         print(STDOUT "$_\n");
4648     }
4649     if ($log) {
4650         $short =~ s/\\/\\/string\\/\\/g;
4651         $long =~ s/\\/\\/string\\/\\/g;
4652         $_ = $ERROR{$format};
4653         s/\$short/\$short/;
4654         s/\$long/\$long/;
4655         print(LOG "$_\n");
4656     }
4657     $exit_code = 11;
4658     endinput();
4659 }

```

```

4660
4661 sub warning {
4662     my $text = shift;
4663     if (! $quiet) {
4664         $_ = $WARNING{' '};
4665         s/\$header/$header/;
4666         s/\$indent/$indent/;
4667         s/\$text/$text/;
4668         print(STDOUT "$_\n");
4669     }
4670     if ($log) {
4671         $_ = $WARNING{$format};
4672         $text =~ s/\\/\\/string\\/g;
4673         s/\$texindent/$texindent/;
4674         s/\$text/$text/;
4675         print(LOG "$_\n");
4676     }
4677     $exit_code = 10;
4678 }
4679
4680 sub info {
4681     my $text = shift;
4682     if ($text && ! $quiet) {
4683         $_ = $INFO{' '};
4684         s/\$header/$header/;
4685         s/\$indent/$indent/;
4686         s/\$text/$text/;
4687         print(STDOUT "$_\n");
4688         if ($log) {
4689             $_ = $INFO{$format};
4690             $text =~ s/\\/\\/string\\/g;
4691             s/\$texindent/$texindent/;
4692             s/\$text/$text/;
4693             print(LOG "$_\n");
4694         }
4695     }
4696 }

```

Mark the log as complete and exit.

```

4697 sub endinput {
4698     if ($log) {
4699         print(LOG "\\endinput\n");
4700         close(LOG);
4701     }
4702     exit $exit_code;
4703 }
4704
4705 sub die_handler {
4706     stderr_to_warning();
4707     my $text = shift;
4708     chomp($text);
4709     error("Perl error: $text", ' ');
4710 }
4711
4712 sub warn_handler {
4713     my $text = shift;
4714     chomp($text);
4715     warning("Perl warning: $text");
4716 }

```

This is used to print warning messages from PDF::Builder, which are output to STDERR.

```

4717 my $stderr;
4718 sub stderr_to_warning {

```

```

4719     if ($stderr) {
4720         my $w = ' Perl info: ';
4721         my $nl = '';
4722         for (split(/\n/, $stderr)) {
4723             /(^\s*)(.*?)(\s*)$/;
4724             $w .= ($1 ? ' ' : $nl) . $2;
4725             $nl = "\n";
4726         }
4727         warning("$w");
4728         $stderr = '';
4729     }
4730 }

```

[Permission-related functions](#) We will need these variables below. Note that we only support Unix and Windows.

```

4731 my $on_windows = $^O eq 'MSWin32';
4732 my $dirsep = $on_windows ? '\\\ ' : '/\';

```

paranoia_in/out should work exactly as `kpsewhich -safe-in-name/-safe-out-name`.

```

4733 sub paranoia_in {
4734     my ($f, $remark) = @_;
4735     error("I'm not allowed to read from '$f' (openin_any = $openin_any)",
4736         $remark) unless _paranoia($f, $openin_any);
4737 }
4738
4739 sub paranoia_out {
4740     my ($f, $remark) = @_;
4741     error("I'm not allowed to write to '$f' (openin_any = $openout_any)",
4742         $remark) unless _paranoia($f, $openout_any);
4743 }
4744
4745 sub _paranoia {

```

`f` is the path to the file (it should not be empty), and `mode` is the value of `openin_any` or `openout_any`.

```

4746     my ($f, $mode) = @_;
4747     return if (! $f);

```

We split the filename into the directory and the basename part, and the directory into components.

```

4748     my ($volume, $dir, $basename) = splitpath($f);
4749     my @dir = splitdir($dir);
4750     return (

```

In mode ‘any’ (a, y or 1), we may access any file.

```

4751         $mode =~ /^[ay1]$/
4752         || (

```

Otherwise, we are at least in the restricted mode, so we should not open dot files on Unix-like systems (except file called `.tex`).

```

4753             ! ($on_windows && $basename =~ /\.\./ && !($basename =~ /\.\.tex$/))
4754             && (

```

If we are precisely in the restricted mode (r, n, 0), then there are no further restrictions.

```

4755                 $mode =~ /^[rn0]$/

```

Otherwise, we are in the paranoid mode (officially p, but any other value is interpreted as p as well). There are two further restrictions in the paranoid mode.

```

4756                 || (

```

We’re not allowed to go to a parent directory.

```

4757                     ! grep(/^\.\.$/, @dir) && $basename ne '..'
4758                     &&

```

If the given path is absolute, it should be a descendant of either `TEXMF_OUTPUT_DIRECTORY` or `TEXMFOUTPUT`.

```

4759             (!file_name_is_absolute($f)
4760             ||
4761             is_ancestor($texmf_output_directory, $f)
4762             ||
4763             is_ancestor($texmfoutput, $f)
4764             ))));
4765 }

```

Only removes final `"/`s. This is unlike `File::Spec`'s `canonpath`, which also removes `.` components, collapses multiple `/` — and unfortunately also goes up for `..` on Windows.

```

4766 sub normalize_path {
4767     my $path = shift;
4768     my ($v, $d, $n) = splitpath($path);
4769     if ($n eq '' && $d =~ /^[^\\Q$dirsep\E]\\Q$dirsep\E+$/ ) {
4770         $path =~ s/\\Q$dirsep\E+$/ /;
4771     }
4772     return $path;
4773 }

```

On Windows, we disallow “semi-absolute” paths, i.e. paths starting with the `\` but lacking the drive. `File::Spec`'s function `file_name_is_absolute` returns 2 if the path is absolute with a volume, 1 if it's absolute with no volume, and 0 otherwise. After a path was sanitized using this function, `file_name_is_absolute` will work as we want it to.

```

4774 sub sanitize_path {
4775     my $f = normalize_path(shift);
4776     my ($v, $d, $n) = splitpath($f);
4777     if ($on_windows) {
4778         my $a = file_name_is_absolute($f);
4779         if ($a == 1 || ($a == 0 && $v) ) {
4780             error("\Semi-absolute\" paths are disallowed: " . $f,
4781                 "The path must either both contain the drive letter and " .
4782                 "start with '\\', or none of these; paths like 'C:foo\\bar' " .
4783                 "and '\\foo\\bar' are disallowed");
4784         }
4785     }
4786 }
4787
4788 sub access_in {
4789     return -r shift;
4790 }
4791
4792 sub access_out {
4793     my $f = shift;
4794     my $exists;
4795     eval { $exists = -e $f };

```

Presumably, we get this error when the parent directory is not executable.

```

4796     return if ($@);
4797     if ($exists) {

```

An existing file should be writable, and if it's a directory, it should also be executable.

```

4798         my $rw = -w $f; my $rd = -d $f; my $rx = -x $f;
4799         return -w $f && (! -d $f || -x $f);
4800     } else {

```

For a non-existing file, the parent directory should be writable. (This is the only place where function `parent` is used, so it's ok that it returns the logical parent.)

```

4801         my $p = parent($f);
4802         return -w $p;
4803     }
4804 }

```

This function finds the location for an input file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (always), and `TEXMFOUTPUT` directory (if defined), in this order. The first readable file found is returned; if no readable file is found, the file in the current directory is returned.

```

4805 sub find_in {
4806     my $f = shift;
4807     sanitize_path($f);
4808     return $f if file_name_is_absolute($f);
4809     for my $df (
4810         $texmf_output_directory ? join_paths($texmf_output_directory, $f) : undef,
4811         $f,
4812         $texmfoutput ? join_paths($texmfoutput, $f) : undef) {
4813         return $df if $df && -r $df;
4814     }
4815     return $f;
4816 }

```

This function finds the location for an output file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (unless `TEXMF_OUTPUT_DIRECTORY` is defined), and `TEXMFOUTPUT` directory (if defined), in this order. The first writable file found is returned; if no writable file is found, the file in either the current or the output directory is returned.

```

4817 sub find_out {
4818     my $f = shift;
4819     sanitize_path($f);
4820     return $f if file_name_is_absolute($f);
4821     for my $df (
4822         $texmf_output_directory ? join_paths($texmf_output_directory, $f) : undef,
4823         $texmf_output_directory ? undef : $f,
4824         $texmfoutput ? join_paths($texmfoutput, $f) : undef) {
4825         return $df if $df && access_out($df);
4826     }
4827     return $texmf_output_directory ? join_paths($texmf_output_directory, $f) : $f;
4828 }

```

We next define some filename-related utilities matching what Python offers out of the box. We avoid using `File::Spec`'s `canonpath`, because on Windows, which has no concept of symlinks, this function resolves `..` to the parent.

```

4829 sub name {
4830     my $path = shift;
4831     my ($volume, $dir, $filename) = splitpath($path);
4832     return $filename;
4833 }
4834
4835 sub suffix {
4836     my $path = shift;
4837     my ($volume, $dir, $filename) = splitpath($path);
4838     $filename =~ /\.[^.]*/;
4839     return $&;
4840 }
4841
4842 sub with_suffix {
4843     my ($path, $suffix) = @_;
4844     my ($volume, $dir, $filename) = splitpath($path);
4845     if ($filename =~ s/\.[^.]*/$suffix/) {
4846         return catpath($volume, $dir, $filename);
4847     } else {
4848         return catpath($volume, $dir, $filename . $suffix);
4849     }
4850 }

```

```

4851
4852 sub with_name {
4853     my ($path, $name) = @_;
4854     my ($volume, $dir, $filename) = splitpath($path);
4855     my ($v,$d,$f) = splitpath($name);
4856     die "Runtime error in with_name: " .
4857     "'$name' should not contain the directory component"
4858     unless $v eq '' && $d eq '' && $f eq $name;
4859     return catpath($volume, $dir, $name);
4860 }
4861
4862 sub join_paths {
4863     my $path1 = normalize_path(shift);
4864     my $path2 = normalize_path(shift);
4865     return $path2 if !$path1 || file_name_is_absolute($path2);
4866     my ($volume1, $dir1, $filename1) = splitpath($path1, 'no_file');
4867     my ($volume2, $dir2, $filename2) = splitpath($path2);
4868     die if $volume2;
4869     return catpath($volume1,
4870                   join($dirsep, ($dir1 eq $dirsep ? '' : $dir1, $dir2)),
4871                   $filename2);
4872 }

```

The logical parent. The same as `pathlib.parent` in Python.

```

4873 sub parent {
4874     my $f = normalize_path(shift);
4875     my ($v, $dn, $_dummy) = splitpath($f, 1);
4876     my $p_dn = $dn =~ s/[^\Q$dirsep\E]+$/r;
4877     if ($p_dn eq '') {
4878         $p_dn = $dn =~ /^\Q$dirsep\E/ ? $dirsep : '.';
4879     }
4880     my $p = catpath($v, $p_dn, '');
4881     $p = normalize_path($p);
4882     return $p;
4883 }

```

This function assumes that both paths are absolute; ancestor may be "", signaling a non-path.

```

4884 sub is_ancestor {
4885     my $ancestor = normalize_path(shift);
4886     my $descendant = normalize_path(shift);
4887     return if !$ancestor;
4888     $ancestor .= $dirsep unless $ancestor =~ /\Q$dirsep\E$/;
4889     return $descendant =~ /^\Q$ancestor/;
4890 }

```

A paranoid `Path.mkdir`. The given folder is preprocessed by `find_out`.

```

4891 sub make_directory {
4892     my $folder = find_out(shift);
4893     if (! -d $folder) {
4894         paranoia_out($folder);

```

Using `make_path` is fine because we know that `TEXMF_OUTPUT_DIRECTORY/TEXMFOUTPUT`, if given, exists, and that "folder" contains no ...

```

4895     make_path($folder);

```

This does not get logged when the function is invoked via `--mkdir`, as it is not clear what the log name should be.

```

4896         info("Created directory $folder");
4897     }
4898 }
4899
4900 sub unquote {
4901     shift =~ s/"(.*)"\/\1/rg;

```

```
4902 }
```

Kpathsea Get the values of `openin_any`, `openout_any`, `TEXMFOUTPUT` and `TEXMF_OUTPUT_DIRECTORY`.

```
4903 my $maybe_backslash = $on_windows ? '\\' : '\\';
4904 my $query = 'kpsewhich -expand-var=' .
4905     "openin_any=$maybe_backslash\$openin_any," .
4906     "openout_any=$maybe_backslash\$openout_any," .
4907     "TEXMFOUTPUT=$maybe_backslash\$TEXMFOUTPUT";
4908 my $kpsewhich_output = ` $query `;
4909 if (! $kpsewhich_output) {
```

No TeX? (Note that `kpsewhich` should exist in MiKTeX as well.) In absence of `kpathsea` information, we get very paranoid.

```
4910     ($openin_any, $openout_any) = ('p', 'p');
4911     ($texmfoutput, $texmf_output_directory) = ('', '');
```

Unfortunately, this warning can't make it into the log. But then again, the chances of a missing `kpsewhich` are very slim, and its absence would show all over the place anyway.

```
4912     warning('I failed to execute "kpsewhich", is there no TeX system installed? ' .
4913         'Assuming openin_any = openout_any = "p" ' .
4914         '(i.e. restricting all file operations to non-hidden files ' .
4915         'in the current directory of its subdirectories).');
4916 } else {
4917     $kpsewhich_output =~ /^openin_any=(.*),openout_any=(.*),TEXMFOUTPUT=(.*)/;
4918     ($openin_any, $openout_any, $texmfoutput) = @{^CAPTURE};
4919     $texmf_output_directory = $ENV{'TEXMF_OUTPUT_DIRECTORY'};
4920     if ($openin_any =~ '^\$openin_any') {
```

When the `open*_any` variables are not expanded, we assume we're running MiKTeX. The two config settings below correspond to TeXLive's `openin_any` and `openout_any`; afaik, there is no analogue to `TEXMFOUTPUT`.

```
4921     $query = 'initexmf --show-config-value=[Core]AllowUnsafeInputFiles ' .
4922         '--show-config-value=[Core]AllowUnsafeOutputFiles';
4923     my $initexmf_output = ` $query `;
4924     $initexmf_output =~ /^(.*)\n(.*)\n/m;
4925     $openin_any = $1 eq 'true' ? 'a' : 'p';
4926     $openout_any = $2 eq 'true' ? 'a' : 'p';
4927     $texmfoutput = '';
4928     $texmf_output_directory = '';
4929 }
4930 }
```

An output directory should exist, and may not point to the root on Linux. On Windows, it may point to the root, because being absolute also implies containing the drive; see `sanitize_filename`.

```
4931 sub sanitize_output_dir {
4932     return unless my $d = shift;
4933     sanitize_path($d);
```

On Windows, `rootdir` returns `\`, so it cannot possibly match `$d`.

```
4934     return $d if -d $d && $d ne rootdir();
4935 }
4936
4937 $texmfoutput = sanitize_output_dir($texmfoutput);
4938 $texmf_output_directory = sanitize_output_dir($texmf_output_directory);
```

We don't delve into the real script when loaded from the testing code.

```
4939 return 1 if caller;
```

Arguments

```
4940 my $usage = "usage: $PROG [-h] [-P PDF] [-p] [-k] [-F {latex,plain,context}] [-f] " .
4941     "[-L {PDF::API2,PDF::Builder}] [-q] [-m] [-V] mmz\n";
4942 my $Help = <<END;
```



```

4943 Extract extern pages produced by package Memoize out of the document PDF.
4944
4945 positional arguments:
4946   mmz                the record file produced by Memoize:
4947                       doc.mmz when compiling doc.tex
4948                       (doc and doc.tex are accepted as well)
4949
4950 options:
4951   -h, --help          show this help message and exit
4952   -P PDF, --pdf PDF  extract from file PDF
4953   -p, --prune         remove the extern pages after extraction
4954   -k, --keep          do not mark externs as extracted
4955   -F, --format {latex,plain,context}
4956                       the format of the TeX document invoking extraction
4957   -f, --force         extract even if the size-check fails
4958   -q, --quiet         describe what's happening
4959   -L, --library {PDF::API2, PDF::Builder}
4960                       which PDF library to use for extraction (default: PDF::API2)
4961   -m, --mkdir         create a directory (and exit);
4962                       mmz argument is interpreted as directory name
4963   -V, --version       show program's version number and exit
4964
4965 For details, see the man page or the Memoize documentation.
4966 END
4967
4968 my @valid_libraries = ('PDF::API2', 'PDF::Builder');
4969 Getopt::Long::Configure ("bundling");
4970 GetOptions(
4971   "pdf|P=s"    => \$pdf_file,
4972   "prune|p"    => \$prune,
4973   "keep|k"     => \$keep,
4974   "format|F=s" => \$format,
4975   "force|f"    => \$force,
4976   "quiet|q"    => \$quiet,
4977   "library|L=s" => \$pdf_library,
4978   "mkdir|m"    => \$mkdir,
4979   "version|V"  => \$print_version,
4980   "help|h|?"  => \$help,
4981   ) or die $usage;
4982
4983 if ($help) {print("$usage\n$Help"); exit 0}
4984
4985 if ($print_version) { print("$PROG of Memoize $VERSION\n"); exit 0 }
4986
4987 die "${usage}$PROG: error: the following arguments are required: mmz\n"
4988     unless @ARGV == 1;
4989
4990 die "${usage}$PROG: error: argument -F/--format: invalid choice: '$format' " .
4991     "(choose from 'latex', 'plain', 'context')\n"
4992     unless grep $_ eq $format, ('', 'latex', 'plain', 'context');
4993
4994 die "${usage}$PROG: error: argument -L/--library: invalid choice: '$pdf_library' " .
4995     "(choose from " . join(", ", @valid_libraries) . ")\n"
4996     if $pdf_library && ! grep $_ eq $pdf_library, @valid_libraries;
4997
4998 $header = $format ? basename($0) . ': ' : '';

```

start a new line in the TeX terminal output

```

4999 print("\n") if $format;

```

Initialization With `--mkdir`, argument `mmz` is interpreted as the directory to create.

```

5000 if ($mkdir) {
5001     make_directory($ARGV[0]);

```

```

5002     exit 0;
5003 }

```

Normalize the mmz argument into a .mmz filename.

```

5004 my $mmz_file = $ARGV[0];
5005 $mmz_file = with_suffix($mmz_file, '.mmz')
5006     if suffix($mmz_file) eq '.tex';
5007 $mmz_file = with_name($mmz_file, name($mmz_file) . '.mmz')
5008     if suffix($mmz_file) ne '.mmz';

```

Once we have the .mmz filename, we can open the log.

```

5009 if ($format) {
5010     my $_log = find_out(with_suffix($mmz_file, '.mmz.log'));
5011     paranoia_out($_log);
5012     info("Logging to '$_log'");
5013     $log = $_log;
5014     open LOG, ">$log";
5015 }

```

Now that we have opened the log file, we can try loading the PDF processing library.

```

5016 if ($pdf_library) {
5017     eval "use $pdf_library";
5018     error("Perl module '$pdf_library' was not found",
5019         'Have you followed the instructions in section 1.1 of the manual?')
5020     if ($@);
5021 } else {
5022     for (@valid_libraries) {
5023         eval "use $_";
5024         if (!$@) {
5025             $pdf_library = $_;
5026             last;
5027         }
5028     }
5029     if (!$pdf_library) {
5030         error("No suitable Perl module for PDF processing was found, options are " .
5031             join(", ", @valid_libraries),
5032             'Have you followed the instructions in section 1.1 of the manual?');
5033     }
5034 }

```

Catch any errors in the script and output them to the log.

```

5035 $SIG{__DIE__} = \&die_handler;
5036 $SIG{__WARN__} = \&warn_handler;
5037 close(STDERR);
5038 open(STDERR, ">", \&stderr);

```

Find the .mmz file we will read, but retain the original filename in \$given_mmz_file, as we will still need it.

```

5039 my $given_mmz_file = $mmz_file;
5040 $mmz_file = find_in($mmz_file, 1);
5041 if (! -e $mmz_file) {
5042     info("File '$given_mmz_file' does not exist, assuming there's nothing to do");
5043     endinput();
5044 }
5045 paranoia_in($mmz_file);
5046 paranoia_out($mmz_file,
5047     'I would have to rewrite this file unless option --keep is given.')
5048     unless $keep;

```

Determine the PDF filename: it is either given via --pdf, or constructed from the .mmz filename.

```

5049 $pdf_file = with_suffix($given_mmz_file, '.pdf') if !$pdf_file;
5050 $pdf_file = find_in($pdf_file);
5051 paranoia_in($pdf_file);

```

```

5052 paranoia_out($pdf_file,
5053             'I would have to rewrite this file because option --prune was given.')
```

```

5054     if $prune;
```

Various initializations.

```

5055 my $pdf;
5056 my %extern_pages;
5057 my $new_mmz;
5058 my $tolerance = 0.01;
5059 info("Extracting new externs listed in '$mmz_file' " .
5060     "from '$pdf_file' using Perl module $pdf_library");
5061 my $done_message = "Done (there was nothing to extract)";
5062 $indent = ' ';
5063 $texindent = '\space\space ';
5064 my $dir_to_make;
```

Process .mmz We cannot process the .mmz file using in-place editing. It would fail when the file is writable but its parent directory is not.

```

5065 open (MMZ, $mmz_file);
5066 while (<MMZ>) {
5067     my $mmz_line = $_;
5068     if (/^\mmzPrefix *{(?P<prefix>)}\/) {
```

Found \mmzPrefix: create the extern directory, but only later, if an extern file is actually produced. We parse the prefix in two steps because we have to unquote the entire prefix.

```

5069         my $prefix = unquote($+{prefix});
5070         warning("Cannot parse line '$mmz_line'") unless
5071             $prefix =~ /(?P<dir_prefix>.*\/)?(?P<name_prefix>.*?)/;
5072         $dir_to_make = $+{dir_prefix};
5073     } elsif (/^\mmzNewExtern\ *{(?P<extern_path>.*?)}{(?P<page_n>[0-9]+)}#
5074             {(?P<expected_width>[0-9.]*pt)}{(?P<expected_height>[0-9.]*pt)/x) {
```

Found \mmzNewExtern: extract the extern page into an extern file.

```

5075         $done_message = "Done";
5076         my $ok = 1;
5077         my %m_ne = %+;
```

The extern filename, as specified in .mmz:

```

5078     my $extern_file = unquote($m_ne{extern_path});
```

We parse the extern filename in a separate step because we have to unquote the entire path.

```

5079     warning("Cannot parse line '$mmz_line'") unless
5080         $extern_file =~ /(?P<dir_prefix>.*\/)?(?P<name_prefix>.*?)#
5081         (?P<code_md5sum>[0-9A-F]{32})-#
5082         (?P<context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf/x;
```

The actual extern filename:

```

5083     my $extern_file_out = find_out($extern_file);
5084     paranoia_out($extern_file_out);
5085     my $page = $m_ne{page_n};
```

Check whether c-memo and cc-memo exist (in any input directory).

```

5086     my $c_memo = with_name($extern_file,
5087                         $+{name_prefix} . $+{code_md5sum} . '.memo');
5088     my $cc_memo = with_name($extern_file,
5089                         $+{name_prefix} . $+{code_md5sum} .
5090                         '-' . $+{context_md5sum} . '.memo');
5091     my $c_memo_in = find_in($c_memo);
5092     my $cc_memo_in = find_in($cc_memo);
5093     if ((! access_in($c_memo_in) || ! access_in($cc_memo_in)) && !$force) {
5094         warning("I refuse to extract page $page into extern '$extern_file', " .
5095             "because the associated c-memo '$c_memo' and/or " .
5096             "cc-memo '$cc_memo' does not exist");
```

```

5097     $ok = '';
5098 }

```

Load the PDF. We only do this now so that we don't load it if there is nothing to extract.

```

5099     if ($ok && ! $pdf) {
5100         if (!access_in($pdf_file)) {
5101             warning("Cannot open '$pdf_file'", '');
5102             endinput();
5103         }

```

Temporarily disable error handling, so that we can catch the error ourselves.

```

5104     $SIG{__DIE__} = undef; $SIG{__WARN__} = undef;

```

All safe, paranoia_in was already called above.

```

5105     eval { $pdf = $pdf_library->open($pdf_file, msgver => 0) };
5106     $SIG{__DIE__} = \&die_handler; $SIG{__WARN__} = \&warn_handler;
5107     error("File '$pdf_file' seems corrupted. " .
5108         "Perhaps you have to load Memoize earlier in the preamble",
5109         "In particular, Memoize must be loaded before TikZ library " .
5110         "'fadings' and any package deploying it, and in Beamer, " .
5111         "load Memoize by writing \\RequirePackage{memoize} before " .
5112         "\\documentclass{beamer}." .
5113         "This was the error thrown by Perl:" . "\n$@" ) if $@;
5114 }

```

Does the page exist?

```

5115     if ($ok && $page > (my $n_pages = $pdf->page_count())) {
5116         error("I cannot extract page $page from '$pdf_file', " .
5117             "as it contains only $n_pages page" .
5118             ($n_pages > 1 ? 's' : ''), '');
5119     }
5120     if ($ok) {

```

Import the page into the extern PDF (no disk access yet).

```

5121         my $extern = $pdf_library->new(outver => $pdf->version);
5122         $extern->import_page($pdf, $page);
5123         my $extern_page = $extern->open_page(1);

```

Check whether the page size matches the .mmz expectations.

```

5124         my ($x0, $y0, $x1, $y1) = $extern_page->get_mediabox();
5125         my $width_pt = ($x1 - $x0) / 72 * 72.27;
5126         my $height_pt = ($y1 - $y0) / 72 * 72.27;
5127         my $expected_width_pt = $m_ne{expected_width};
5128         my $expected_height_pt = $m_ne{expected_height};
5129         if ((abs($width_pt - $expected_width_pt) > $tolerance
5130             || abs($height_pt - $expected_height_pt) > $tolerance) && !$force) {
5131             warning("I refuse to extract page $page from $pdf_file, " .
5132                 "because its size (${width_pt}pt x ${height_pt}pt) " .
5133                 "is not what I expected " .
5134                 "(${expected_width_pt}pt x ${expected_height_pt}pt)");
5135         } else {

```

All tests were successful, let's create the extern file. First, the containing directory, if necessary.

```

5136         if ($dir_to_make) {
5137             make_directory($dir_to_make);
5138             $dir_to_make = undef;
5139         }

```

Now the extern file. Note that paranoia_out was already called above.

```

5140         info("Page $page --> $extern_file_out");
5141         $extern->saveas($extern_file_out);

```

This page will get pruned.

```

5142             $extern_pages{$page} = 1 if $prune;

    Comment out this \mmzNewExtern.
5143             $new_mmz .= '%' unless $keep;
5144         }
5145     }
5146 }
5147 $new_mmz .= $mmz_line unless $keep;
5148 stderr_to_warning();
5149 }
5150 close(MMZ);
5151 $indent = '';
5152 $texindent = '';
5153 info($done_message);

```

Write out the .mmz file with \mmzNewExtern lines commented out. (All safe, paranoia_out was already called above.)

```

5154 if (!$keep) {
5155     open(MMZ, ">", $mmz_file);
5156     print MMZ $new_mmz;
5157     close(MMZ);
5158 }

```

Remove the extracted pages from the original PDF. (All safe, paranoia_out was already called above.)

```

5159 if ($prune and keys(%extern_pages) != 0) {
5160     my $pruned_pdf = $pdf_library->new();
5161     for (my $n = 1; $n <= $pdf->page_count(); $n++) {
5162         if (! $extern_pages{$n}) {
5163             $pruned_pdf->import_page($pdf, $n);
5164         }
5165     }
5166     $pruned_pdf->save($pdf_file);
5167     info("The following extern pages were pruned out of the PDF: " .
5168         join(", ", sort(keys(%extern_pages))));
5169 }
5170
5171 endinput();

```

9.2 The Python extraction script memoize-extract.py

```

5172 __version__ = '2024/12/02 v1.4.1'
5173
5174 import argparse, re, sys, os, subprocess, itertools, traceback, platform
5175 from pathlib import Path, PurePath

```

[Messages](#) We will only try to import the PDF processing library once we set up the error log. The messages are written both to the extraction log and the terminal (we output to stdout rather than stderr so that messages on the TeX terminal and document .log appear in chronological order). Messages are automatically adapted to the TeX --format. The format of the messages. It depends on the given --format; the last entry is for t the terminal output.

```

5176 ERROR = {
5177     'latex':    r'\PackageError{{{package_name}}}{short}{long}',
5178     'plain':   r'\errhelp{{{long}}}\errmessage{{{package_name}: {short}}}',
5179     'context': r'\errhelp{{{long}}}\errmessage{{{package_name}: {short}}}',
5180     None:      '{header}{short}.\n{long}',
5181 }
5182
5183 WARNING = {
5184     'latex':    r'\PackageWarning{{{package_name}}}{texindent}{text}',
5185     'plain':   r'\message{{{package_name}: {texindent}{text}}}',
5186     'context': r'\message{{{package_name}: {texindent}{text}}}',
5187     None:      r'{header}{indent}{text}.',

```

```

5188 }
5189
5190 INFO = {
5191     'latex': r'\PackageInfo{{{package_name}}}{texindent}{text}}',
5192     'plain': r'\message{{{package_name}: {texindent}{text}}}',
5193     'context': r'\message{{{package_name}: {texindent}{text}}}',
5194     None: r'{header}{indent}{text}.',
5195 }

```

Some variables used in the message routines; note that `header` will be redefined once we parse the arguments.

```

5196 package_name = 'memoize (python-based extraction)'
5197 exit_code = 0
5198 log = None
5199 header = ''
5200 indent = ''
5201 texindent = ''

```

The message routines.

```

5202 def error(short, long):
5203     if not args.quiet:
5204         print(ERROR[None].format(short = short, long = long, header = header))
5205     if log:
5206         short = short.replace('\n', '\\string\\n')
5207         long = long.replace('\n', '\\string\\n')
5208         print(
5209             ERROR[args.format].format(
5210                 short = short, long = long, package_name = package_name),
5211             file = log)
5212     global exit_code
5213     exit_code = 11
5214     endinput()
5215
5216 def warning(text):
5217     if text and not args.quiet:
5218         print(WARNING[None].format(text = text, header = header, indent = indent))
5219     if log:
5220         text = text.replace('\n', '\\string\\n')
5221         print(
5222             WARNING[args.format].format(
5223                 text = text, texindent = texindent, package_name = package_name),
5224             file = log)
5225     global exit_code
5226     exit_code = 10
5227
5228 def info(text):
5229     if text and not args.quiet:
5230         print(INFO[None].format(text = text, header = header, indent = indent))
5231     if log:
5232         text = text.replace('\n', '\\string\\n')
5233         print(
5234             INFO[args.format].format(
5235                 text = text, texindent = texindent, package_name = package_name),
5236             file = log)

```

Mark the log as complete and exit.

```

5237 def endinput():
5238     if log:
5239         print(r'\endinput', file = log)
5240         log.close()
5241     sys.exit(exit_code)

```

[Permission-related functions](#) `paranoia_in/out` should work exactly as `kpsewhich -safe-in-name/-safe-out-name`.

```

5242 def paranoia_in(f, remark = ''):
5243     if f and not _paranoia(f, openin_any):
5244         error(f"I'm not allowed to read from '{f}' (openin_any = {openin_any})",
5245             remark)
5246
5247 def paranoia_out(f, remark = ''):
5248     if f and not _paranoia(f, openout_any):
5249         error(f"I'm not allowed to write to '{f}' (openout_any = {openout_any})",
5250             remark)
5251
5252 def _paranoia(f, mode):

```

mode is the value of openin_any or openout_any. f is a pathlib.Path object.

```

5253     return (

```

In mode 'any' (a, y or 1), we may access any file.

```

5254         mode in 'ay1'
5255     or (

```

Otherwise, we are at least in the restricted mode, so we should not open dot files on Unix-like systems (except file called .tex).

```

5256         not (os.name == 'posix' and f.stem.startswith('.') and f.stem != '.tex')
5257     and (

```

If we are precisely in the restricted mode (r, n, 0), then there are no further restrictions.

```

5258         mode in 'rn0'

```

Otherwise, we are in the paranoid mode (officially p, but any other value is interpreted as p as well). There are two further restrictions in the paranoid mode.

```

5259     or (

```

We're not allowed to go to a parent directory.

```

5260         '..' not in f.parts
5261     and

```

If the given path is absolute, it should be a descendant of either TEXMF_OUTPUT_DIRECTORY or TEXMFOUTPUT.

```

5262         (not f.is_absolute()
5263         or
5264         is_ancestor(texmf_output_directory, f)
5265         or
5266         is_ancestor(texmfoutput, f)
5267         ))))

```

On Windows, we disallow “semi-absolute” paths, i.e. paths starting with the \ but lacking the drive. On Windows, pathlib’s is_absolute returns True only for paths starting with \ and containing the drive.

```

5268 def sanitize_filename(f):
5269     if f and platform.system() == 'Windows' and not (f.is_absolute() or not f.drive):
5270         error(f"\Semi-absolute\" paths are disallowed: '{f}'", r"The path must "
5271             r"either contain both the drive letter and start with '\\, "
5272             r"or none of these; paths like 'C:foo' and '\\foo' are disallowed")
5273
5274 def access_in(f):
5275     return os.access(f, os.R_OK)

```

This function can fail on Windows, reporting a non-writable file or dir as writable, because os.access does not work with Windows’ icaccls permissions. Consequence: we might try to write to a read-only current or output directory instead of switching to the temporary directory. Paranoia is unaffected, as it doesn’t use access_* functions.

```

5276 def access_out(f):
5277     try:

```

```
5278         exists = f.exists()
```

Presumably, we get this error when the parent directory is not executable.

```
5279     except PermissionError:
5280         return
5281     if exists:
```

An existing file should be writable, and if it's a directory, it should also be executable.

```
5282         return os.access(f, os.W_OK) and (not f.is_dir() or os.access(f, os.X_OK))
5283     else:
```

For a non-existing file, the parent directory should be writable. (This is the only place where function `pathlib.parent` is used, so it's ok that it returns the logical parent.)

```
5284         return os.access(f.parent, os.W_OK)
```

This function finds the location for an input file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (always), and `TEXMFOUTPUT` directory (if defined), in this order. The first readable file found is returned; if no readable file is found, the file in the current directory is returned.

```
5285 def find_in(f):
5286     sanitize_filename(f)
5287     if f.is_absolute():
5288         return f
5289     for df in (texmf_output_directory / f if texmf_output_directory else None,
5290              f,
5291              texmfoutput / f if texmfoutput else None):
5292         if df and access_in(df):
5293             return df
5294     return f
```

This function finds the location for an output file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (unless `TEXMF_OUTPUT_DIRECTORY` is defined), and `TEXMFOUTPUT` directory (if defined), in this order. The first writable file found is returned; if no writable file is found, the file in either the current or the output directory is returned.

```
5295 def find_out(f):
5296     sanitize_filename(f)
5297     if f.is_absolute():
5298         return f
5299     for df in (texmf_output_directory / f if texmf_output_directory else None,
5300              f if not texmf_output_directory else None,
5301              texmfoutput / f if texmfoutput else None):
5302         if df and access_out(df):
5303             return df
5304     return texmf_output_directory / f if texmf_output_directory else f
```

This function assumes that both paths are absolute; ancestor may be `None`, signaling a non-path.

```
5305 def is_ancestor(ancestor, descendant):
5306     if not ancestor:
5307         return
5308     a = ancestor.parts
5309     d = descendant.parts
5310     return len(a) < len(d) and a == d[0:len(a)]
```

A paranoid `Path.mkdir`. The given folder is preprocessed by `find_out`.

```
5311 def mkdir(folder):
5312     folder = find_out(Path(folder))
5313     if not folder.exists():
5314         paranoia_out(folder)
```


Using `folder.mkdir` is fine because we know that `TEXMF_OUTPUT_DIRECTORY/TEXMFOUTPUT`, if given, exists, and that “folder” contains no ...

```
5315     folder.mkdir(parents = True, exist_ok = True)
```

This does not get logged when the function is invoked via `--mkdir`, as it is not clear what the log name should be.

```
5316     info(f"Created directory {folder}")
5317
5318     _re_unquote = re.compile(r'\"(.*)\"')
5319     def unquote(fn):
5320         return _re_unquote.sub(r'\1', fn)
```

[Kpathsea](#) Get the values of `openin_any`, `openout_any`, `TEXMFOUTPUT` and `TEXMF_OUTPUT_DIRECTORY`.

```
5321     kpsewhich_output = subprocess.run(['kpsewhich',
5322                                       f'-expand-var='
5323                                       f'openin_any=${openin_any}, '
5324                                       f'openout_any=${openout_any}, '
5325                                       f'TEXMFOUTPUT=${TEXMFOUTPUT}'],
5326                                       capture_output = True
5327                                       ).stdout.decode().strip()
5328     if not kpsewhich_output:
```

No TeX? (Note that `kpsewhich` should exist in MiKTeX as well.) In absence of `kpathsea` information, we get very paranoid, but still try to get `TEXMFOUTPUT` from an environment variable.

```
5329     openin_any, openout_any = 'p', 'p'
5330     texmfoutput, texmf_output_directory = None, None
```

Unfortunately, this warning can't make it into the log. But then again, the chances of a missing `kpsewhich` are very slim, and its absence would show all over the place anyway.

```
5331     warning('I failed to execute "kpsewhich"; , is there no TeX system installed? '
5332             'Assuming openin_any = openout_any = "p" '
5333             '(i.e. restricting all file operations to non-hidden files '
5334             'in the current directory of its subdirectories).')
5335     else:
5336         m = re.fullmatch(r'openin_any=(.*),openout_any=(.*),TEXMFOUTPUT=(.*)',
5337                         kpsewhich_output)
5338         openin_any, openout_any, texmfoutput = m.groups()
5339         texmf_output_directory = os.environ.get('TEXMF_OUTPUT_DIRECTORY', None)
5340         if openin_any == '$openin_any':
```

When the `open*_any` variables are not expanded, we assume we're running MiKTeX. The two config settings below correspond to TeXLive's `openin_any` and `openout_any`; afaik, there is no analogue to `TEXMFOUTPUT`.

```
5341     initexmf_output = subprocess.run(
5342         ['initexmf', '--show-config-value=[Core]AllowUnsafeInputFiles',
5343          '--show-config-value=[Core]AllowUnsafeOutputFiles'],
5344         capture_output = True).stdout.decode().strip()
5345     openin_any, openout_any = initexmf_output.split()
5346     openin_any = 'a' if openin_any == 'true' else 'p'
5347     openout_any = 'a' if openout_any == 'true' else 'p'
5348     texmfoutput = None
5349     texmf_output_directory = None
```

An output directory should exist, and may not point to the root on Linux. On Windows, it may point to the root, because we only allow absolute filenames containing the drive, e.g. `F:\`; see `is_absolute`.

```
5350     def sanitize_output_dir(d_str):
5351         d = Path(d_str) if d_str else None
5352         sanitize_filename(d)
5353         return d if d and d.is_dir() and \
5354             (not d.is_absolute() or len(d.parts) != 1 or d.drive) else None
```

```

5355
5356 texmfoutput = sanitize_output_dir(texmfoutput)
5357 texmf_output_directory = sanitize_output_dir(texmf_output_directory)
5358
5359 class NotExtracted(UserWarning):
5360     pass

```

We don't delve into the real script when loaded from the testing code.

```

5361 if __name__ == '__main__':

```

Arguments

```

5362     parser = argparse.ArgumentParser(
5363         description = "Extract extern pages produced by package Memoize "
5364             "out of the document PDF.",
5365         epilog = "For details, see the man page or the Memoize documentation.",
5366         prog = 'memoize-extract.py',
5367     )
5368     parser.add_argument('-P', '--pdf', help = 'extract from file PDF')
5369     parser.add_argument('-p', '--prune', action = 'store_true',
5370         help = 'remove the extern pages after extraction')
5371     parser.add_argument('-k', '--keep', action = 'store_true',
5372         help = 'do not mark externs as extracted')
5373     parser.add_argument('-F', '--format', choices = ['latex', 'plain', 'context'],
5374         help = 'the format of the TeX document invoking extraction')
5375     parser.add_argument('-f', '--force', action = 'store_true',
5376         help = 'extract even if the size-check fails')
5377     parser.add_argument('-q', '--quiet', action = 'store_true',
5378         help = "describe what's happening")
5379     parser.add_argument('-m', '--mkdir', action = 'store_true',
5380         help = 'create a directory (and exit); '
5381             "'mmz argument is interpreted as directory name')
5382     parser.add_argument('-V', '--version', action = 'version',
5383         version = f"%(prog)s of Memoize " + __version__)
5384     parser.add_argument('mmz', help = 'the record file produced by Memoize: '
5385         'doc.mmz when compiling doc.tex '
5386         '(doc and doc.tex are accepted as well)')
5387
5388     args = parser.parse_args()
5389
5390     header = parser.prog + ': ' if args.format else '

```

Start a new line in the TeX terminal output.

```

5391     if args.format:
5392         print()

```

Initialization With `--mkdir`, argument `mmz` is interpreted as the directory to create.

```

5393     if args.mkdir:
5394         mkdir(args.mmz)
5395         sys.exit()

```

Normalize the `mmz` argument into a `.mmz` filename.

```

5396     mmz_file = Path(args.mmz)
5397     if mmz_file.suffix == '.tex':
5398         mmz_file = mmz_file.with_suffix('.mmz')
5399     elif mmz_file.suffix != '.mmz':
5400         mmz_file = mmz_file.with_name(mmz_file.name + '.mmz')

```

Once we have the `.mmz` filename, we can open the log.

```

5401     if args.format:
5402         log_file = find_out(mmz_file.with_suffix('.mmz.log'))
5403         paranoia_out(log_file)
5404         info(f"Logging to '{log_file}'");
5405         log = open(log_file, 'w')

```

Now that we have opened the log file, we can try loading the PDF processing library.

```
5406     try:
5407         import pdfrw
5408     except ModuleNotFoundError:
5409         error("Python module 'pdfrw' was not found",
5410             'Have you followed the instructions in section 1.1 of the manual?')
```

Catch any errors in the script and output them to the log.

```
5411     try:
```

Find the .mmz file we will read, but retain the original filename in `given_mmz_file`, as we will still need it.

```
5412         given_mmz_file = mmz_file
5413         mmz_file = find_in(mmz_file)
5414         paranoia_in(mmz_file)
5415         if not args.keep:
5416             paranoia_out(mmz_file,
5417                 remark = 'This file is rewritten unless option --keep is given.')
```

```
5418     try:
5419         mmz = open(mmz_file)
5420     except FileNotFoundError:
5421         info(f"File '{given_mmz_file}' does not exist, "
5422             f"assuming there's nothing to do")
5423     endinput()
```

Determine the PDF filename: it is either given via `--pdf`, or constructed from the .mmz filename.

```
5424     pdf_file = find_in(Path(args.pdf)
5425                         if args.pdf else given_mmz_file.with_suffix('.pdf'))
5426     paranoia_in(pdf_file)
5427     if args.prune:
5428         paranoia_out(pdf_file,
5429             remark = 'I would have to rewrite this file '
5430                 'because option --prune was given.')
```

Various initializations.

```
5431     re_prefix = re.compile(r'\\mmzPrefix *{(?P<prefix>.*?)}')
5432     re_split_prefix = re.compile(r'(?P<dir_prefix>.*?)(?P<name_prefix>.*?)')
5433     re_newextern = re.compile(
5434         r'\\mmzNewExtern *{(?P<extern_path>.*?)}{(?P<page_n>[0-9]+)}'
5435         r'{(?P<expected_width>[0-9.]*pt){(?P<expected_height>[0-9.]*pt)}')
5436     re_extern_path = re.compile(
5437         r'(?P<dir_prefix>.*?)(?P<name_prefix>.*?)'
5438         r'(?P<code_md5sum>[0-9A-F]{32})-'
5439         r'(?P<context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf')
5440     pdf = None
5441     extern_pages = []
5442     new_mmz = []
5443     tolerance = 0.01
5444     dir_to_make = None
5445     info(f"Extracting new externs listed in '{mmz_file}' from '{pdf_file}'")
5446     done_message = "Done (there was nothing to extract)"
5447     indent = ' '
5448     texindent = r'\space\space '
```

Process .mmz

```
5449     for line in mmz:
5450         try:
5451             if m_p := re_prefix.match(line):
```

Found `\mmzPrefix`: create the extern directory, but only later, if an extern file is actually produced. We parse the prefix in two steps because we have to unquote the entire prefix.

```
5452         prefix = unquote(m_p['prefix'])
5453         if not (m_sp := re_split_prefix.match(prefix)):
```

```

5454         warning(f"Cannot parse line {line.strip()}")
5455         dir_to_make = m_sp['dir_prefix']
5456     elif m_ne := re_newextern.match(line):

Found \mmzNewExtern: extract the extern page into an extern file.
5457         done_message = "Done"

The extern filename, as specified in .mmz:
5458         unquoted_extern_path = unquote(m_ne['extern_path'])
5459         extern_file = Path(unquoted_extern_path)

We parse the extern filename in a separate step because we have to unquote the entire path.
5460         if not (m_ep := re_extern_path.match(unquoted_extern_path)):
5461             warning(f"Cannot parse line {line.strip()}")

The actual extern filename:
5462         extern_file_out = find_out(extern_file)
5463         paranoia_out(extern_file_out)
5464         page_n = int(m_ne['page_n'])-1

Check whether c-memo and cc-memo exist (in any input directory).
5465         c_memo = extern_file.with_name(
5466             m_ep['name_prefix'] + m_ep['code_md5sum'] + '.memo')
5467         cc_memo = extern_file.with_name(
5468             m_ep['name_prefix'] + m_ep['code_md5sum']
5469             + '-' + m_ep['context_md5sum'] + '.memo')
5470         c_memo_in = find_in(c_memo)
5471         cc_memo_in = find_in(cc_memo)
5472         if not (access_in(c_memo_in) and access_in(cc_memo_in)) \
5473             and not args.force:
5474             warning(f"I refuse to extract page {page_n+1} into extern "
5475                 f"'{extern_file}', because the associated c-memo "
5476                 f"'{c_memo}' and/or cc-memo '{cc_memo}' "
5477                 f"does not exist")
5478             raise NotExtracted()

Load the PDF. We only do this now so that we don't load it if there is nothing to extract.
5479         if not pdf:
5480             if not access_in(pdf_file):
5481                 warning(f"Cannot open '{pdf_file}'")
5482                 endinput()
5483         try:

All safe, paranoia_in was already called above.
5484         pdf = pdfmw.PdfReader(pdf_file)
5485         except pdfmw.errors.PdfParseError as err:
5486             error(rf"File '{pdf_file}' seems corrupted. Perhaps you "
5487                 rf"have to load Memoize earlier in the preamble",
5488                 rf"In particular, Memoize must be loaded before "
5489                 rf"TikZ library 'fadings' and any package "
5490                 rf"deploying it, and in Beamer, load Memoize "
5491                 rf"by writing \RequirePackage{{memoize}} before "
5492                 rf"\documentclass{{beamer}}. "
5493                 rf"This was the error thrown by Python: \n{err}")

Does the page exist?
5494         if page_n >= len(pdf.pages):
5495             error(rf"I cannot extract page {page_n} from '{pdf_file}', "
5496                 rf"as it contains only {len(pdf.pages)} page" +
5497                 ('s' if len(pdf.pages) > 1 else ''), '')

Check whether the page size matches the .mmz expectations.
5498         page = pdf.pages[page_n]

```

```

5499         expected_width_pt = float(m_ne['expected_width'])
5500         expected_height_pt = float(m_ne['expected_height'])
5501         mb = page['/MediaBox']
5502         width_bp = float(mb[2]) - float(mb[0])
5503         height_bp = float(mb[3]) - float(mb[1])
5504         width_pt = width_bp / 72 * 72.27
5505         height_pt = height_bp / 72 * 72.27
5506         if (abs(width_pt - expected_width_pt) > tolerance
5507             or abs(height_pt - expected_height_pt) > tolerance) \
5508             and not args.force:
5509             warning(
5510                 f"I refuse to extract page {page_n+1} from '{pdf_file}' "
5511                 f"because its size ({width_pt}pt x {height_pt}pt) "
5512                 f"is not what I expected "
5513                 f"({expected_width_pt}pt x {expected_height_pt}pt)")
5514             raise NotExtracted()

```

All tests were successful, let's create the extern file. First, the containing directory, if necessary.

```

5515         if dir_to_make:
5516             mkdir(dir_to_make)
5517         dir_to_make = None

```

Now the extern file. Note that `paranoia_out` was already called above.

```

5518         info(f"Page {page_n+1} --> {extern_file_out}")
5519         extern = pdfwr.PdfWriter(extern_file_out)
5520         extern.addpage(page)
5521         extern.write()

```

This page will get pruned.

```

5522         if args.prune:
5523             extern_pages.append(page_n)

```

Comment out this `\mmzNewExtern`.

```

5524         if not args.keep:
5525             line = '%' + line
5526         except NotExtracted:
5527             pass
5528         finally:
5529             if not args.keep:
5530                 new_mmz.append(line)
5531         mmz.close()
5532         indent = ''
5533         texindent = ''
5534         info(done_message)

```

Write out the `.mmz` file with `\mmzNewExtern` lines commented out. (All safe, `paranoia_out` was already called above.)

```

5535         if not args.keep:
5536             with open(mmz_file, 'w') as mmz:
5537                 for line in new_mmz:
5538                     print(line, file = mmz, end = '')

```

Remove the extracted pages from the original PDF. (All safe, `paranoia_out` was already called above.)

```

5539         if args.prune and extern_pages:
5540             pruned_pdf = pdfwr.PdfWriter(pdf_file)
5541             pruned_pdf.addpages(
5542                 page for n, page in enumerate(pdf.pages) if n not in extern_pages)
5543             pruned_pdf.write()
5544             info(f"The following extern pages were pruned out of the PDF: " +
5545                 ", ".join(str(page+1) for page in extern_pages))

```

Report that extraction was successful.

```
5546         endinput()
```

Catch any errors in the script and output them to the log.

```
5547     except Exception as err:
```

```
5548         error(f'Python error: {err}', traceback.format_exc())
```

9.3 The Perl clean-up script memoize-clean.pl

```
5549 my $PROG = 'memoize-clean.pl';
```

```
5550 my $VERSION = '2024/12/02 v1.4.1';
```

```
5551
```

```
5552 use strict;
```

```
5553 use Getopt::Long;
```

```
5554 use Cwd 'realpath';
```

```
5555 use File::Spec;
```

```
5556 use File::Basename;
```

```
5557
```

```
5558 my $usage = "usage: $PROG [-h] [--yes] [--all] [--quiet] [--prefix PREFIX] " .
```

```
5559     "[mmz ...]\n";
```

```
5560 my $Help = <<END;
```

```
5561 Remove (stale) memo and extern files produced by package Memoize.
```

```
5562
```

```
5563 positional arguments:
```

```
5564   mmz                .mmz record files
```

```
5565
```

```
5566 options:
```

```
5567   -h, --help          show this help message and exit
```

```
5568   --version, -V       show version and exit
```

```
5569   --yes, -y           Do not ask for confirmation.
```

```
5570   --all, -a           Remove *all* memos and externs.
```

```
5571   --quiet, -q
```

```
5572   --prefix PREFIX, -p PREFIX
```

```
5573                       A path prefix to clean;
```

```
5574                       this option can be specified multiple times.
```

```
5575
```

```
5576 For details, see the man page or the Memoize documentation.
```

```
5577 END
```

```
5578
```

```
5579 my ($yes, $all, @prefixes, $quiet, $help, $print_version);
```

```
5580 GetOptions(
```

```
5581     "yes|y" => \$yes,
```

```
5582     "all|a" => \$all,
```

```
5583     "prefix|p=s" => \@prefixes,
```

```
5584     "quiet|q" => \$quiet,
```

```
5585     "help|h|?" => \$help,
```

```
5586     "version|V" => \$print_version,
```

```
5587     ) or die $usage;
```

```
5588 $help and die "$usage\n$Help";
```

```
5589 if ($print_version) { print("memoize-clean.pl of Memoize $VERSION\n"); exit 0 }
```

```
5590
```

```
5591 my (%keep, %prefixes);
```

```
5592
```

```
5593 my $curdir = Cwd::getcwd();
```

```
5594
```

```
5595 for my $prefix (@prefixes) {
```

```
5596     $prefixes{Cwd::realpath(File::Spec->catfile(($curdir), $prefix))} = '';
```

```
5597 }
```

```
5598
```

```
5599 my @mmzs = @ARGV;
```

```
5600
```

```
5601 for my $mmz (@mmzs) {
```

```
5602     my ($mmz_filename, $mmz_dir) = File::Basename::fileparse($mmz);
```

```
5603     @ARGV = ($mmz);
```

```
5604     my $endinput = 0;
```

```

5605     my $empty = -1;
5606     my $prefix = "";
5607     while (<>) {
5608     if (/^ *$/) {
5609     } elsif ($endinput) {
5610         die "Bailing out, \\endinput is not the last line of file $mmz.\n";
5611     } elsif (/^ *\\mmzPrefix *{(.*)}/) {
5612         $prefix = $1;
5613         $prefixes{Cwd::realpath(File::Spec->catfile(($curdir,$mmz_dir), $prefix))} = '';
5614         $empty = 1 if $empty == -1;
5615     } elsif (/^%? *\\mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*)}/) {
5616         my $fn = $1;
5617         if ($prefix eq '') {
5618             die "Bailing out, no prefix announced before file $fn.\n";
5619         }
5620         $keep{Cwd::realpath(File::Spec->catfile(($mmz_dir), $fn))} = 1;
5621         $empty = 0;
5622         if (rindex($fn, $prefix, 0) != 0) {
5623             die "Bailing out, prefix of file $fn does not match " .
5624                 "the last announced prefix ($prefix).\n";
5625         }
5626     } elsif (/^ *\\endinput *$/) {
5627         $endinput = 1;
5628     } else {
5629         die "Bailing out, file $mmz contains an unrecognized line: $_.\n";
5630     }
5631     }
5632     die "Bailing out, file $mmz is empty.\n" if $empty && !$all;
5633     die "Bailing out, file $mmz does not end with \\endinput; this could mean that " .
5634     "the compilation did not finish properly. You can only clean with --all.\n"
5635     if $endinput == 0 && !$all;
5636 }
5637
5638 my @tbdeleted;
5639 sub populate_tbdeleted {
5640     my ($basename_prefix, $dir, $suffix_dummy) = @_ ;
5641     opendir(MD, $dir) or die "Cannot open directory '$dir'";
5642     while( (my $fn = readdir(MD)) ) {
5643     my $path = File::Spec->catfile(($dir),$fn);
5644     if ($fn =~
5645         /\Q$basename_prefix\E[0-9A-F]{32}(?:-[0-9A-F]{32})?(?:-[0-9]+)?#
5646         (\.memo|(?:-[0-9]+)?\.pdf|\.log)/x
5647         and ($all || !exists($keep{$path}))) {
5648         push @tbdeleted, $path;
5649     }
5650     }
5651     closedir(MD);
5652 }
5653 for my $prefix (keys %prefixes) {
5654     my ($basename_prefix, $dir, $suffix);
5655     if (-d $prefix) {
5656     populate_tbdeleted('', $prefix, '');
5657     }
5658     populate_tbdeleted(File::Basename::fileparse($prefix));
5659 }
5660 @tbdeleted = sort(@tbdeleted);
5661
5662 my @allowed_dirs = ($curdir);
5663 my @deletion_not_allowed;
5664 for my $f (@tbdeleted) {
5665     my $f_allowed = 0;
5666     for my $dir (@allowed_dirs) {
5667     if ($f =~ /\Q$dir\E/) {

```

```

5668     $f_allowed = 1;
5669     last;
5670 }
5671 }
5672 push(@deletion_not_allowed, $f) if ! $f_allowed;
5673 }
5674 die "Bailing out, I was asked to delete these files outside the current directory:\n" .
5675     join("\n", @deletion_not_allowed) if (@deletion_not_allowed);
5676
5677 if (scalar(@tbdeleted) != 0) {
5678     my $a;
5679     unless ($yes) {
5680         print("I will delete the following files:\n" .
5681             join("\n",@tbdeleted) . "\n" .
5682             "Proceed (y/n)? ");
5683         $a = lc(<>);
5684         chomp $a;
5685     }
5686     if ($yes || $a eq 'y' || $a eq 'yes') {
5687         foreach my $fn (@tbdeleted) {
5688             print "Deleting ", $fn, "\n" unless $quiet;
5689             unlink $fn;
5690         }
5691     } else {
5692         die "Bailing out.\n";
5693     }
5694 } elsif (!$quiet) {
5695     print "Nothing to do, the directory seems clean.\n";
5696 }

```

9.4 The Python clean-up script memoize-clean.py

```

5697 __version__ = '2024/12/02 v1.4.1'
5698
5699 import argparse, re, sys, pathlib, os
5700
5701 parser = argparse.ArgumentParser(
5702     description="Remove (stale) memo and extern files.",
5703     epilog = "For details, see the man page or the Memoize documentation "
5704             "(https://ctan.org/pkg/memoize).")
5705 )
5706 parser.add_argument('--yes', '-y', action = 'store_true',
5707                     help = 'Do not ask for confirmation.')
5708 parser.add_argument('--all', '-a', action = 'store_true',
5709                     help = 'Remove *all* memos and externs.')
5710 parser.add_argument('--quiet', '-q', action = 'store_true')
5711 parser.add_argument('--prefix', '-p', action = 'append', default = [],
5712                     help = 'A path prefix to clean; this option can be specified multiple times.')
5713 parser.add_argument('mmz', nargs= '*', help='mmz record files')
5714 parser.add_argument('--version', '-V', action = 'version',
5715                     version = f"%(prog)s of Memoize " + __version__)
5716 args = parser.parse_args()
5717
5718 re_prefix = re.compile(r'\\.mmzPrefix *{(.*)}')
5719 re_memo = re.compile(r'%? *\\.mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*)}')
5720 re_endinput = re.compile(r'\\.endinput *$')
5721
5722 prefixes = set(pathlib.Path(prefix).resolve() for prefix in args.prefix)
5723 keep = set()

```

We loop through the given .mmz files, adding prefixes to whatever manually specified by the user, and collecting the files to keep.

```

5724 for mmz_fn in args.mmz:

```



```

5725 mmz = pathlib.Path(mMZ_fn)
5726 mmz_parent = mmz.parent.resolve()
5727 try:
5728     with open(mmz) as mmz_fh:
5729         prefix = ''
5730         endinput = False
5731         empty = None
5732         for line in mmz_fh:
5733             line = line.strip()
5734
5735             if not line:
5736                 pass
5737
5738             elif endinput:
5739                 raise RuntimeError(
5740                     rf'Bailing out, '
5741                     rf'\endinput is not the last line of file {mmz_fn}.')
5742
5743             elif m := re_prefix.match(line):
5744                 prefix = m[1]
5745                 prefixes.add( (mmz_parent/prefix).resolve() )
5746                 if empty is None:
5747                     empty = True
5748
5749             elif m := re_memo.match(line):
5750                 if not prefix:
5751                     raise RuntimeError(
5752                         f'Bailing out, no prefix announced before file "{m[1]}".')
5753                 if not m[1].startswith(prefix):
5754                     raise RuntimeError(
5755                         f'Bailing out, prefix of file "{m[1]}" does not match '
5756                         f'the last announced prefix ({prefix}).')
5757                 keep.add((mmz_parent / m[1]))
5758                 empty = False
5759
5760             elif re_endinput.match(line):
5761                 endinput = True
5762                 continue
5763
5764             else:
5765                 raise RuntimeError(fr"Bailing out, "
5766                                     fr"file {mmz_fn} contains an unrecognized line: {line}")
5767
5768         if empty and not args.all:
5769             raise RuntimeError(fr'Bailing out, file {mmz_fn} is empty.')
5770
5771         if not endinput and empty is not None and not args.all:
5772             raise RuntimeError(
5773                 fr'Bailing out, file {mmz_fn} does not end with \endinput; '
5774                 fr'this could mean that the compilation did not finish properly. '
5775                 fr'You can only clean with --all.'
5776             )

```

It is not an error if the file doesn't exist. Otherwise, cleaning from scripts would be cumbersome.

```

5777     except FileNotFoundError:
5778         pass
5779
5780 tbdeleted = []
5781 def populate_tbdeleted(folder, basename_prefix):
5782     re_aux = re.compile(
5783         re.escape(basename_prefix) +
5784         r'[0-9A-F]{32}(?:-[0-9A-F]{32})?'
5785         r'(?:-[0-9]+)?(?:\.memo|(?:-[0-9]+)?\.pdf|\.log)$')

```

```

5786     try:
5787         for f in folder.iterdir():
5788             if re_aux.match(f.name) and (args.all or f not in keep):
5789                 tbdeleted.append(f)
5790     except FileNotFoundError:
5791         pass
5792
5793 for prefix in prefixes:
5794
5795     "prefix" is interpreted both as a directory (if it exists) and a basename prefix.
5796     if prefix.is_dir():
5797         populate_tbdeleted(prefix, '')
5798     populate_tbdeleted(prefix.parent, prefix.name)
5799
5800 allowed_dirs = [pathlib.Path().absolute()] # todo: output directory
5801 deletion_not_allowed = [f for f in tbdeleted if not f.is_relative_to(*allowed_dirs)]
5802 if deletion_not_allowed:
5803     raise RuntimeError("Bailing out, "
5804                        "I was asked to delete these files outside the current directory:\n" +
5805                        "\n".join(str(f) for f in deletion_not_allowed))
5806
5807 _cwd_absolute = pathlib.Path().absolute()
5808 def relativize(path):
5809     try:
5810         return path.relative_to(_cwd_absolute)
5811     except ValueError:
5812         return path
5813
5814 if tbdeleted:
5815     tbdeleted.sort()
5816     if not args.yes:
5817         print('I will delete the following files:')
5818         for f in tbdeleted:
5819             print(relativize(f))
5820         print("Proceed (y/n)? ")
5821         a = input()
5822     if args.yes or a == 'y' or a == 'yes':
5823         for f in tbdeleted:
5824             if not args.quiet:
5825                 print("Deleting", relativize(f))
5826             try:
5827                 f.unlink()
5828             except FileNotFoundError:
5829                 print(f"Cannot delete {f}")
5830     else:
5831         print("Bailing out.")
5832 elif not args.quiet:
5833     print('Nothing to do, the directory seems clean.')

```

Index

Numbers written in red refer to the code line where the corresponding entry is defined; numbers in blue refer to the code lines where the entry is used.

Symbols	
.install advice (/handlers)	2301
/handlers/.meaning to context (/mmz)	831
/handlers/.meaning to salt (/mmz)	841
/handlers/.value to context (/mmz)	831
/handlers/.value to salt (/mmz)	841
/advice/install keys:	
activation	2313
setup key	2313
/advice keys:	
activation/deferred	2462
activation/immediate	2462
/collargs keys:	
alias	3102
append expandable postprocessor	3057
append expandable preprocessor	3057
append postprocessor	3039
append preprocessor	3039
begin tag	3023
braces	3015
caller	2949
clear args	3087
clear postprocessors	3049
clear preprocessors	3049
end tag	3023
environment	3018
fix from no verbatim	2996
fix from verb	2996
fix from verbatim	2996
ignore nesting	3031
ignore other tags	3035
no delimiters	3083
no verbatim	2986
prepend expandable postprocessor	3057
prepend expandable preprocessor	3057
prepend postprocessor	3039
prepend preprocessor	3039
return	3092
tags	3023
verb	2986
verbatim	2986
verbatim ranges	4117
/handlers keys:	
.install advice	2301
/mmz/auto keys:	
after setup	2385
apply options	1714
args	2360
bailout handler	2360
biblatex ccmemo cite	2030
cite	2000
cites	2000
clear collector options	2360
clear options	2360
clear raw collector options	2360
collector	2360
collector options	2360
force multiref	1816
force ref	1789
force reframe	1803
inner handler	2360
integrated driver	698
memoize	1727
multiref	1816
noop	1735
options	2360
outer handler	2360
raw collector options	2360
ref	1789
refrange	1803
replicate	1749
reset	2384
run conditions	2360
run if memoization is possible	1698
run if memoizing	1706
to context	1764
volcite	2000
volcites	2000
/mmz keys:	
/handlers/.meaning to context	831
/handlers/.meaning to salt	841
/handlers/.value to context	831
/handlers/.value to salt	841
activate	2473
activate csname	2348
activate deferred	2347
activate key	2350
activation	2345
after memoization	645
at begin memoization	645
at end memoization	645
auto	2322
auto csname	2322
auto csname'	2322
auto key	2322
auto key'	2322
auto'	2322
bat	1419
begindocument	2198
begindocument/before	2198
begindocument/end	2198
capture	686
clear context	813
clear salt	841
context	813
csname meaning to context	831
csname meaning to salt	841
deactivate	2473
deactivate csname	2348
deactivate key	2350
direct ccmemo input	952
disable	238
driver	674
enable	238

enddocument/afterlastpage	2198	\advice@begin@env@outer	2682, 2689, 2689
extract	1285	\advice@begin@env@rc	2672, 2678, 2678
extract/perl	1290	\advice@begin@rc	2386, 2670, 2670
extract/python	1290	\advice@CollectArgumentsRaw	2378, 2419, 2609, 2609
extract/tex	1475	\advice@deactivate	2480, 2512, 2518
force activate	2473	\advice@deactivate@cmd	2520, 2543, 2543, 2647
ignore spaces	340	\advice@deactivate@cmd@do	2547, 2571, 2571
include context in ccmemo	948	\advice@deactivate@env	2522, 2637, 2645, 2662
include source in cmemo	895	\advice@handle	2559, 2568, 2576, 2576
key meaning to context	831	\advice@handle@outer	2584, 2591, 2591
key meaning to salt	841	\advice@handle@rc	2580, 2582, 2582, 2670
key value to context	831	\advice@if@our@definition	2527, 2546, 2558, 2558
key value to salt	841	\advice@init@I	2438, 2439, 2441, 2597, 2695, 2739, 2743
makefile	1447	\advice@init@i	2438, 2438, 2440, 2577, 2679, 2738, 2742
manual	1650	\advice@original@cs	2490, 2491, 2587, 2674, 2685, 2755
meaning to context	831	\advice@original@csname	2490, 2490, 2526, 2545, 2567, 2572, 2573, 2655, 2658, 2663, 2664, 2671
meaning to salt	841	\advice@pgfkeys@collector	2338, 2343, 2634, 2634
memo dir	305	\advice@setup@save	2413, 2442, 2442
mkdir	287	\advice@trace	2745, 2751, 2753, 2754, 2759, 2767, 2776, 2782, 2785, 2788, 2791, 2801, 2804, 2807
mkdir command	287	\advice@trace@init@I	2739, 2752, 2773
no memo dir	305	\advice@trace@init@i	2738, 2752, 2752
no record	1364	\advice@typeout	719, 2745, 2746, 2748, 2749, 2751
no verbatim	344	\AdviceArgs	
normal	252	\AdviceBailoutHandler	
options	250	\AdviceCollector	1717, 2058, 2098, 2113, 2363, 2377, 2418, 2419, 2451, 2596, 2781, 2784, 2798
padding	1052	\AdviceCollectorOptions	
padding bottom	1047	\AdviceCsnameGetOriginal	1770, 2499
padding left	1047	\AdviceGetOriginal	2490, 2596, 2694
padding right	1047	\AdviceIfArgs	2610, 2626
padding to context	1057	\AdviceInnerHandler	2369, 2421, 2456, 2596, 2623, 2635, 2803, 2810, 2812, 2870
padding top	1047	\AdviceName	1742, 2090, 2389, 2432, 2590, 2593, 2595, 2606, 2611, 2616, 2691, 2735
prefix	261	\AdviceNamespace	1767, 1770, 2590, 2592, 2690, 2734
readonly	252	\AdviceOptions	1723, 1752, 2370, 2371, 2373, 2422, 2457, 2596, 2809
recompile	252	\AdviceOriginal	1731, 1736, 1746, 1758, 1775, 1792, 1814, 1826, 1832, 1855, 2041, 2069, 2121, 2590, 2596, 2694
record	1348	\AdviceOuterHandler	
record/bat/...	1435	\AdviceRawCollectorOptions	1725, 2366, 2367, 2426, 2431, 2452, 2453, 2596, 2617, 2793, 2796
record/makefile/...	1454	\AdviceReplaced	1730, 1757, 2590, 2595, 2693
record/mmz/...	1386	\AdviceRunConditions	
record/sh/...	1423	\AdviceSetup	2323, 2327, 2389
salt	841	\AdviceTracingOff	2737
sh	1419	\AdviceTracingOn	2737
tex extraction command	1510	\AdviceType	1674, 2384, 2389
tex extraction options	1510	after memoization (/mmz)	645
tex extraction script	1510	after setup (/mmz/auto)	2385
tracing	764	alias (/collargs)	3102
try activate	2473	append expandable postprocessor (/collargs)	3057
verb	344	append expandable preprocessor (/collargs)	3057
verbatim	344	append postprocessor (/collargs)	3039
		append preprocessor (/collargs)	3039
A			
abortOnError (Lua function)	566		
activate (/mmz)	2473		
activate csname (/mmz)	2348		
activate deferred (/mmz)	2347		
activate key (/mmz)	2350		
activation (/advice/install)	2313		
activation (/mmz)	2345		
activation/deferred (/advice)	2462		
activation/immediate (/advice)	2462		
\advice@activate	2475, 2512, 2512		
\advice@activate@cmd	2514, 2524, 2524, 2640		
\advice@activate@cmd@do	2531, 2537, 2566, 2566		
\advice@activate@env	2516, 2637, 2638, 2654		

apply options (/mmz/auto)	1714	\collargs@fix	3122, 3177, 4232, 4232, 4516
args (/mmz/auto)	2360	\collargs@fix@NtoN	4253, 4253–4255
at begin memoization (/mmz)	645	\collargs@fix@NtoV	4266, 4271, 4271, 4320, 4323
at end memoization (/mmz)	645	\collargs@fix@Ntov	4256, 4256
auto (/mmz)	2322	\collargs@fix@VtoN	4311, 4331, 4342, 4342
auto csname (/mmz)	2322	\collargs@fix@vtoN	4301, 4301
auto csname' (/mmz)	2322	\collargs@fix@VtoN@comment	4347, 4357, 4376
auto key (/mmz)	2322	\collargs@fix@VtoN@escape	4344, 4380, 4380
auto key' (/mmz)	2322	\collargs@fix@VtoN@token	4349, 4354, 4354
auto' (/mmz)	2322	\collargs@fix@VtoV	4253, 4255
		\collargs@fix@Vtov	4329
		\collargs@fix@vtoV	4315, 4315
		\collargs@fix@vtov	4253, 4254
		\collargs@forrange	3349, 3349, 3367, 4111, 4112
		\collargs@forranges	3365, 3365, 4167, 4223
		\collargs@G	3663, 3663
		\collargs@g	3654, 3654, 3663
		\collargs@grabbed@spaces	
		\collargs@grabspaces	3199, 3313, 3313, 3326, 3505, 3877, 3903
			3233, 3233, 3308,
			3440, 3499, 3542, 3569, 3668, 3787, 3867, 3898, 3956
		\collargs@ifnextcat	3329, 3329, 3695
		\collargs@init@grabspaces	
			2924, 3200, 3228, 3228, 3323, 3504, 3878, 3904
		\collargs@insert@char	4355, 4532, 4533, 4544, 4554, 4579
		\collargs@l	3372, 3372
		\collargs@letusecollector	
			3213, 3222, 3840, 3868, 3899, 3910, 3923, 3938
		\collargs@m	3534, 3534, 3661, 3994
		\collargs@make@char	
			4199, 4210, 4532, 4538, 4548, 4558, 4588
		\collargs@make@no@verbatim	4025, 4216, 4217, 4221
		\collargs@make@verbatim	4012, 4018, 4122, 4123, 4158
		\collargs@make@verbatim@bgroup	4175, 4191, 4191
		\collargs@make@verbatim@comment	4181, 4213, 4213
		\collargs@make@verbatim@egroup	4178, 4202, 4202
		\collargs@maybegrabspaces	3306, 3306, 3474, 3528
		\collargs@O	3478, 3478
		\collargs@o	3477, 3477
		\collargs@other@bgroup	3391, 3395, 3583, 3641,
			3649, 3671, 3672, 3677, 3731, 3742, 4072, 4077, 4199
		\collargs@other@egroup	3586,
			3642, 3649, 3673, 3677, 3731, 3742, 4072, 4078, 4210
		\collargs@percentchar	904, 906, 962, 974, 3368, 3370, 4127
		\collargs@R	3442, 3442
		\collargs@r	3421, 3421, 3442
		\collargs@readContent	3645, 3683, 3719, 3826, 3826
		\collargs@reinsert@spaces	3320, 3320, 3373, 3410, 3719
		\collargs@s	3533, 3533
		\collargs@t	3479, 3479, 3533
		\collargs@u	3399, 3399
		\collargs@v	3664, 3664
		\collargs@verbatim@ranges	
			4117, 4118, 4120, 4121, 4126, 4167, 4223
		\collargs@verbatim@wrap	2923, 2986, 2991, 3154, 3158
		\collargs@wrap	3203, 3203,
			3414, 3434, 3461, 3557, 3650, 3678, 3686, 3824, 3997
		\collargsAlias	2061, 3102
		\collargsAppendExpandablePostprocessor	
			325, 439, 1995, 3057
		\collargsAppendExpandablePreprocessor	3057
		\collargsAppendPostprocessor	3039
		\collargsAppendPreprocessor	2064, 3039
B			
bailout handler (/mmz/auto)	2360		
bat (/mmz)	1419		
begin tag (/collargs)	3023		
begindocument (/mmz)	2198		
begindocument/before (/mmz)	2198		
begindocument/end (/mmz)	2198		
biblatex ccmemo cite (/mmz/auto)	2030		
braces (/collargs)	3015		
C			
caller (/collargs)	2949		
capture (/mmz)	686		
\catcodetable@atletter	4082, 4085		
\cite	2124		
cite (/mmz/auto)	2000		
\cites	2135		
cites (/mmz/auto)	2000		
clear args (/collargs)	3087		
clear collector options (/mmz/auto)	2360		
clear context (/mmz)	813		
clear options (/mmz/auto)	2360		
clear postprocessors (/collargs)	3049		
clear preprocessors (/collargs)	3049		
clear raw collector options (/mmz/auto)	2360		
clear salt (/mmz)	841		
\collargs@	2929, 3108, 3386, 3394,		
	3416, 3436, 3457, 3463, 3508, 3515, 3519, 3559, 3565,		
	3617, 3637, 3652, 3658, 3681, 3689, 3805, 3993, 4006		
\collargs@&	3142		
\collargs@!	3167		
\collargs@+	3161		
\collargs@.	3173		
\collargs@@	3106, 3110, 3112, 3155, 3159, 3166		
\collargs@appendarg	3188, 3188,		
	3385, 3392, 3415, 3435, 3462, 3501, 3505, 3518, 3558,		
	3564, 3616, 3635, 3651, 3679, 3687, 3804, 3991, 3992		
\collargs@b	3694, 3694		
\collargs@bgroups	4035, 4044, 4056, 4056, 4152, 4195		
\collargs@catcodetable@initex	4082, 4094, 4096, 4102		
\collargs@catcodetable@verbatim			
	4082, 4084, 4089, 4090, 4093, 4095, 4113, 4148		
\collargs@cc	3277, 3280, 3600, 3608, 3625, 4058, 4059, 4064		
\collargs@cs@cases	860, 2393, 2513, 2519, 2932, 2932, 2953		
\collargs@D	3476, 3476		
\collargs@d	3443, 3443, 3476, 3477, 3478		
\collargs@defcollector	3213, 3213, 3432, 3459, 3685		
\collargs@defusecollector			
	3213, 3217, 3384, 3390, 3413, 3433, 3460, 3556, 3563		
\collargs@E	4007, 4007		
\collargs@e	3942, 3942, 4007		
\collargs@egroups	4036, 4046, 4056, 4057, 4154, 4206		

<code>\collargsArg</code>	325, 1995, 2064, 3065, 3070, 3075, 3080, 3172, 3189, 3199, 3206, 3208, 3620, 3626, 3635, 3651, 3679, 3804, 3838, 3883, 3915, 3932, 3992, 4002, 4525	<code>force refrange (/mmz/auto)</code>	1803
<code>\collargsArgs</code>	2069, 2090, 2096, 2121, 2829, 2927, 3171, 3180, 3182, 3199	G	
<code>\collargsBraces</code>	3016, 3666, 4072	<code>\gtoksapp</code>	616, 816, 1756, 1950, 1969, 2895
<code>\collargsCaller</code>	388, 435, 2616, 2949	I	
<code>\collargsClearPostprocessors</code>	3049	<code>\ifAdviceRun</code>	2583, 2608, 2681, 2763
<code>\collargsClearPreprocessors</code>	3049	<code>\ifcollargs@fix@requested</code>	4231, 4231, 4234
<code>\collargsEnvironment</code>	2432, 3018	<code>\ifcollargs@verbatim</code>	2920, 2984, 2984, 3126, 3257, 3314, 3400, 3422, 3444, 3480, 3535, 3702, 3724, 3735, 3750, 3761, 3945, 4242
<code>\collargsFixFromNoVerbatim</code>	367, 400, 420, 2996	<code>\ifcollargs@verbatimbraces</code>	2921, 2984, 2985, 3127, 3377, 3572, 3707, 3730, 3741, 3776, 3813, 3820, 4149, 4192, 4203, 4243
<code>\collargsFixFromVerb</code>	2996	<code>\ifcollargsAddTags</code>	3023
<code>\collargsFixFromVerbatim</code>	2996	<code>\ifcollargsBeginTag</code>	3023, 3810
<code>\collargsNoVerbatim</code>	356, 2989, 4008	<code>\ifcollargsClearArgs</code>	2926, 3087
<code>\collargsPrependExpandablePostprocessor</code>	3057	<code>\ifcollargsEndTag</code>	3023, 3817
<code>\collargsPrependExpandablePreprocessor</code>	3057	<code>\ifcollargsIgnoreNesting</code>	3031, 3914
<code>\collargsPrependPostprocessor</code>	3039	<code>\ifcollargsIgnoreOtherTags</code>	3035, 3709, 3779
<code>\collargsPrependPreprocessor</code>	3039	<code>\ifcollargsNoDelimiters</code>	3083, 3194
<code>\collargsReturn</code>	2815, 3092, 3179	<code>\ifinmemoize</code>	490, 563, 1700
<code>\collargsSet</code>	2619, 2917, 2931, 2949, 2986, 2996, 3015, 3018, 3023, 3031, 3035, 3039, 3051, 3057, 3083, 3087, 3092, 3102, 3158, 4117	<code>\ifmemoize</code>	238, 524, 725, 1699
<code>\collargsVerb</code>	352, 1994, 2988, 4008	<code>\IfMemoizing</code>	707, 2264
<code>\collargsVerbatim</code>	348, 2987, 3667, 4008	<code>\ifmemoizing</code>	488, 522, 650, 658, 666, 815, 823, 1707, 1940, 2044, 2052
<code>\collargsVerbatimRanges</code>	4117	<code>\ifmmz@abort</code>	539, 621, 642, 642, 1028
<code>\CollectArguments</code>	2914, 2983	<code>\ifmmz@direct@ccmemo@input</code>	755, 952, 952, 978, 997, 1007, 1018
<code>\CollectArgumentsRaw</code>	322, 387, 434, 2615, 2914	<code>\ifmmz@include@context</code>	948, 948, 963
<code>collector (/mmz/auto)</code>	2360	<code>\ifmmz@include@source</code>	895, 898, 907
<code>collector options (/mmz/auto)</code>	2360	<code>\ifmmzkeepexterns</code>	678, 1099
<code>context (/mmz)</code>	813	<code>\ifmmzUnmemoizable</code>	536, 614, 644
<code>csname meaning to context (/mmz)</code>	831	<code>ignore nesting (/collargs)</code>	3031
<code>csname meaning to salt (/mmz)</code>	841	<code>ignore other tags (/collargs)</code>	3035
D		<code>ignore spaces (/mmz)</code>	340
<code>deactivate (/mmz)</code>	2473	<code>include context in ccmemo (/mmz)</code>	948
<code>deactivate csname (/mmz)</code>	2348	<code>include source in cmemo (/mmz)</code>	895
<code>deactivate key (/mmz)</code>	2350	<code>inner handler (/mmz/auto)</code>	2360
<code>direct ccmemo input (/mmz)</code>	952	<code>integrated driver (/mmz/auto)</code>	698
<code>disable (/mmz)</code>	238	K	
<code>driver (/mmz)</code>	674	<code>key meaning to context (/mmz)</code>	831
E		<code>key meaning to salt (/mmz)</code>	841
<code>enable (/mmz)</code>	238	<code>key value to context (/mmz)</code>	831
<code>end tag (/collargs)</code>	3023	<code>key value to salt (/mmz)</code>	841
<code>enddocument/afterlastpage (/mmz)</code>	2198	L	
<code>environment (/collargs)</code>	3018	<code>Lua functions:</code>	
<code>environments:</code>		<code>abortOnError</code>	566
<code>memoize</code>	394	M	
<code>nomemoize</code>	452	<code>makefile (/mmz)</code>	1447
<code>\etoksapp</code>	2895, 3855, 3877, 3883, 3903, 3932	<code>manual (/mmz)</code>	1650
<code>extract (/mmz)</code>	1285	<code>meaning to context (/mmz)</code>	831
<code>extract/perl (/mmz)</code>	1290	<code>meaning to salt (/mmz)</code>	841
<code>extract/python (/mmz)</code>	1290	<code>memo dir (/mmz)</code>	305
<code>extract/tex (/mmz)</code>	1475	<code>\Memoize</code>	383, 431, 507, 724, 1729
F		<code>\memoize</code>	409, 482
<code>\filetotoks</code>	214, 1021	<code>memoize (/mmz/auto)</code>	1727
<code>fix from no verbatim (/collargs)</code>	2996	<code>memoize (env.)</code>	394
<code>fix from verb (/collargs)</code>	2996	<code>\memoizinggrouplevel</code>	597, 640, 709
<code>fix from verbatim (/collargs)</code>	2996	<code>mkdir (/mmz)</code>	287
<code>force activate (/mmz)</code>	2473	<code>mkdir command (/mmz)</code>	287
<code>force multiref (/mmz/auto)</code>	1816		
<code>force ref (/mmz/auto)</code>	1789		

<code>\mmz</code>	360, 451	<code>\mmzNewCCMemo</code>	1404, 1482
<code>\mmz@auto@bailout</code>	1665, 1677, 1714, 1719	<code>\mmzNewCMemo</code>	1401, 1482
<code>\mmz@auto@label</code>	1786, 1828, 1828	<code>\mmzNewExtern</code>	1396, 1488
<code>\mmz@auto@memoize</code>	1671, 1727, 1727	<code>\mmznxt</code>	1709, 1720, 1724
<code>\mmz@auto@multiref</code>	1816, 1818, 1820, 1823	<code>\mmzNoRef</code>	1780, 1794, 1805, 1818, 2265
<code>\mmz@auto@noop</code>	1675, 1679, 1735, 1735	<code>\mmzPrefix</code>	1392, 1482
<code>\mmz@auto@noop@env</code>	1677, 1735, 1741	<code>\mmzRegularPages</code>	129, 1152, 1196
<code>\mmz@auto@outer</code>	1666, 1714, 1714	<code>\mmzResource</code>	974, 976
<code>\mmz@auto@rc@if@memoization@possible</code>	1661, 1698, 1698	<code>\mmzSalt</code>	526, 841
<code>\mmz@auto@rc@if@memoizing</code>	1663, 1706, 1706	<code>\mmzset</code> 228, 242, 256, 261, 281, 288, 305, 341, 346, 372,	
<code>\mmz@auto@ref</code>	1780, 1781, 1789, 1789	403, 425, 648, 674, 686, 698, 764, 813, 842, 895, 949,	
<code>\mmz@auto@refrange</code>	1803, 1805, 1807, 1810	953, 1047, 1156, 1178, 1200, 1286, 1348, 1364, 1378,	
<code>\mmz@auto@replicate</code>	1681, 1749, 1749	1380, 1386, 1454, 1475, 1515, 1646, 1651, 1684, 1687,	
<code>\mmz@auto@tocontext</code>	1682, 1764, 1764	1691, 1693, 1723, 1724, 1778, 1783, 1803, 1816, 1842,	
<code>\mmz@biblatex@cite@inner</code>	2005, 2012, 2066, 2066	1854, 1862, 1893, 1900, 1907, 1929, 1936, 1976, 1982,	
<code>\mmz@biblatex@cite@outer</code>	2002, 2009, 2056, 2056	1989, 2123, 2199–2202, 2205, 2210, 2217, 2227, 2234	
<code>\mmz@biblatex@cite@rc</code>	2001, 2043, 2043	<code>\mmzSingleExternDriver</code>	676, 679
<code>\mmz@biblatex@cites@outer</code>	2016, 2024, 2095, 2095	<code>\mmzSource</code>	908, 914
<code>\mmz@biblatex@cites@rc</code>	2015, 2043, 2051	<code>\mmzThisContext</code>	964, 988
<code>\mmz@biblatex@entry</code>	1991, 2038, 2038	<code>\mmzTracingOff</code>	764, 2274
<code>\mmz@biblatex@mark@citation@key</code>	2060, 2061, 2063	<code>\mmzTracingOn</code>	764, 2273
<code>\mmz@ccmemo@append@resource</code>	971, 971, 1084	<code>\mmzUnmemoizable</code>	643, 1688, 2263
<code>\mmz@ccmemo@resources</code>	602, 959, 1083, 1090, 1090	<code>\mmzUsedCCMemo</code>	1413, 1482
<code>\mmz@compile</code>	521, 557, 557, 1031	<code>\mmzUsedCMemo</code>	1410, 1482
<code>\mmz@compute@context@mdfivesum</code> 624, 627, 927, 929, 929		<code>\mmzUsedExtern</code>	1407, 1482
<code>\mmz@if@roughly@equal</code> 1231, 1232, 1576, 1579, 1606, 1607		<code>multiref (/mmz/auto)</code>	1816
<code>\mmz@include@extern</code>	994, 1209, 1209		
<code>\mmz@include@extern@from@tbe@box</code> ...	636, 1268, 1268	N	
<code>\mmz@insertpdfpage</code>	1220, 1249, 1250, 1257, 1261	<code>no delimiters (/collargs)</code>	3083
<code>\mmz@inverse@mag</code>		<code>no memo dir (/mmz)</code>	305
..... 1161, 1166, 1168, 1170, 1172–1175, 1177, 1179		<code>no record (/mmz)</code>	1364
<code>\mmz@lua@atbeginmemoization</code>	566, 578, 590, 608	<code>no verbatim (/collargs)</code>	2986
<code>\mmz@lua@atendmemoization</code>	566, 581, 591, 613	<code>no verbatim (/mmz)</code>	344
<code>\mmz@maybe@scantokens</code>	491, 493, 518, 1736	<code>\nomemoize</code>	461, 482
<code>\mmz@memoize</code>	532, 544, 593, 593, 1033	<code>nomemoize (env.)</code>	452
<code>\mmz@process@ccmemo</code>	549, 988, 988	<code>\nommz</code>	442
<code>\mmz@process@cmemo</code>	535, 915, 915	<code>\nommzkeys</code>	233
<code>\mmz@remove@quotes</code>	299, 312, 312	<code>noop (/mmz/auto)</code>	1735
<code>\mmz@shipout@extern</code>	1102, 1106, 1106	<code>normal (/mmz)</code>	252
<code>\mmz@shipout@externs</code>	630, 1092, 1092		
<code>\mmz@shipout@mag</code>	1114, 1159, 1159	O	
<code>\mmz@shipout@unrotate</code> ...	1142, 1181, 1182, 1185, 1190	<code>options (/mmz/auto)</code>	2360
<code>\mmz@split@prefix</code>	264	<code>options (/mmz)</code>	250
<code>\mmz@tbe@box</code>	601, 1082, 1091, 1091, 1099, 1272	<code>outer handler (/mmz/auto)</code>	2360
<code>\mmz@write@ccmemo</code>	629, 956, 956		
<code>\mmz@write@cmemo</code>	617, 625, 628, 900, 900	P	
<code>\mmzAbort</code>	641, 1685, 1800, 2079, 2262	<code>padding (/mmz)</code>	1052
<code>\mmzAfterMemoization</code>	637, 645	<code>padding bottom (/mmz)</code>	1047
<code>\mmzAtBeginMemoization</code>	609, 645, 1968	<code>padding left (/mmz)</code>	1047
<code>\mmzAtEndMemoization</code>	611, 645	<code>padding right (/mmz)</code>	1047
<code>\mmzAutoInit</code>	1716, 1722, 1743	<code>padding to context (/mmz)</code>	1057
<code>\mmzCCMemo</code>	604, 680, 683, 947,	<code>padding top (/mmz)</code>	1047
962, 1756, 1829, 1887, 1950, 1954, 2082, 2087, 2270		<code>prefix (/mmz)</code>	261
<code>\mmzCMemo</code>	603, 616, 894, 906, 1948, 1969, 2269	<code>prepend expandable postprocessor (/collargs)</code> ..	3057
<code>\mmzContext</code>	811, 818, 826, 930, 2267	<code>prepend expandable preprocessor (/collargs)</code> ..	3057
<code>\mmzContextExtra</code>		<code>prepend postprocessor (/collargs)</code>	3039
..... 605, 615, 811, 816, 824, 904, 920, 930, 2268		<code>prepend preprocessor (/collargs)</code>	3039
<code>\mmzEndMemo</code>	967, 981, 988		
<code>\mmzExternalizeBox</code>	682, 1065	R	
<code>\mmzExternPages</code>	1149, 1154, 1199, 1202–1204, 2271	<code>raw collector options (/mmz/auto)</code>	2360
<code>\mmzExtraPages</code>	1154, 1208, 1852, 1853, 1855, 2272	<code>readonly (/mmz)</code>	252
<code>\mmzForceNoRef</code>	1781, 1794, 1807, 1820, 2266	<code>recompile (/mmz)</code>	252
<code>\mmzLabel</code>	1830, 1834	<code>record (/mmz)</code>	1348
		<code>record/bat/... (/mmz)</code>	1435

record/makefile/... (/mmz)	1454	tex extraction script (/mmz)	1510
record/mmz/... (/mmz)	1386	to context (/mmz/auto)	1764
record/sh/... (/mmz)	1423	\toksapp	220, 818, 843, 2857, 2861, 2867, 2868, 2895, 3626, 3834, 3909
\ref	1784	tracing (/mmz)	764
ref (/mmz/auto)	1789	try activate (/mmz)	2473
refrange (/mmz/auto)	1803		
replicate (/mmz/auto)	1749		
reset (/mmz/auto)	2384		
return (/collargs)	3092	V	
run conditions (/mmz/auto)	2360	verb (/collargs)	2986
run if memoization is possible (/mmz/auto)	1698	verb (/mmz)	344
run if memoizing (/mmz/auto)	1706	verbatim (/collargs)	2986
		verbatim (/mmz)	344
S		verbatim ranges (/collargs)	4117
salt (/mmz)	841	\volcite	2159
setup key (/advice/install)	2313	volcite (/mmz/auto)	2000
sh (/mmz)	1419	\volcites	2161
		volcites (/mmz/auto)	2000
T			
tags (/collargs)	3023	X	
tex extraction command (/mmz)	1510	\xtoksapp	680, 683, 1083, 1468, 1829, 1887, 1948, 1954, 2082, 2087, 2895, 3199
tex extraction options (/mmz)	1510		