The documented source of Memoize, Advice and CollArgs

Memoize v1.0.0, Advice v1.0.0, CollArgs v1.0.0
October 10, 2023

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 Ti*k*Z 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{memoize-extract-one.tex}{\from{memoize.dtx}{extract-one}}%
}
```

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`). The code is listed in §9, I believe it is self-explanatory enough to lack more than a occasional comment.

Contents

1 First things first

Identification of memoize, memoizable and nomemoize.

```
⟨*mmz⟩
⟨latex⟩\ProvidesPackage{memoize}[2023/10/10 v1.0.0 Fast and flexible externalization]
⟨context⟩%D \module[
⟨context⟩%D         file=t-memoize.tex,
⟨context⟩%D      version=1.0.0,
⟨context⟩%D        title=Memoize,
⟨context⟩%D     subtitle=Fast and flexible externalization,
⟨context⟩%D       author=Saso Zivanovic,
⟨context⟩%D         date=2023-10-10,
⟨context⟩%D    copyright=Saso Zivanovic,
⟨context⟩%D      license=LPPL,
⟨context⟩%D ]
⟨context⟩\writestatus{loading}{ConTeXt User Module / memoize}
⟨context⟩\unprotect
⟨context⟩\startmodule[memoize]
⟨plain⟩% Package memoize 2023/10/10 v1.0.0
⟨/mmz⟩
⟨*mmzable⟩
⟨latex⟩\ProvidesPackage{memoizable}[2023/10/10 v1.0.0 A programmer's stub for Memoize]
⟨context⟩%D \module[
⟨context⟩%D         file=t-memoizable.tex,
⟨context⟩%D      version=1.0.0,
⟨context⟩%D        title=Memoizable,
⟨context⟩%D     subtitle=A programmer's stub for Memoize,
⟨context⟩%D       author=Saso Zivanovic,
⟨context⟩%D         date=2023-10-10,
⟨context⟩%D    copyright=Saso Zivanovic,
⟨context⟩%D      license=LPPL,
⟨context⟩%D ]
⟨context⟩\writestatus{loading}{ConTeXt User Module / memoizable}
⟨context⟩\unprotect
⟨context⟩\startmodule[memoizable]
⟨plain⟩% Package memoizable 2023/10/10 v1.0.0
⟨/mmzable⟩
⟨*nommz⟩
⟨latex⟩\ProvidesPackage{nomemoize}[2023/10/10 v1.0.0 A no-op stub for Memoize]
⟨context⟩%D \module[
⟨context⟩%D         file=t-nomemoize.tex,
⟨context⟩%D      version=1.0.0,
⟨context⟩%D        title=Memoize,
⟨context⟩%D     subtitle=A no-op stub for Memoize,
⟨context⟩%D       author=Saso Zivanovic,
⟨context⟩%D         date=2023-10-10,
⟨context⟩%D    copyright=Saso Zivanovic,
⟨context⟩%D      license=LPPL,
⟨context⟩%D ]
⟨context⟩\writestatus{loading}{ConTeXt User Module / nomemoize}
⟨context⟩\unprotect
⟨context⟩\startmodule[nomemoize]
⟨mmz⟩% Package nomemoize 2023/10/10 v1.0.0
⟨/nommz⟩
```

Required packages and LaTeXization of plain TeX and ConTeXt.

```
⟨*(mmz, mmzable, nommz) & (plain, context)⟩
\input miniltx
⟨/(mmz, mmzable, nommz) & (plain, context)⟩
```

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

```
⟨*mmz & (plain, context)⟩
\def\PackageWarning#1#2{{%
    \newlinechar`\^^J\def\MessageBreak{^^J\space\space#1: }%
    \message{#1: #2}}}
⟨/mmz & (plain, context)⟩
```

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

```
⟨*mmz & plain⟩
\def\newtoks{\alloc@5\toks\toksdef\@cclvi}
\def\newwrite{\alloc@7\write\chardef\sixt@@n}
⟨/mmz & plain⟩
```

`Nomemoize` has to load `pgfopts` as well, and process package options (right away, why not), otherwise LaTeX will complain.

```
⟨*latex⟩
⟨mmz, nommz⟩\RequirePackage{pgfopts}  % pgfkeys-based package options
⟨nommz⟩\pgfkeys{/memoize/package options/.unknown/.code={}}
⟨nommz⟩\ProcessPgfPackageOptions{/memoize/package options}
⟨/latex⟩
```

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

```
⟨*mmz⟩
⟨latex⟩\RequirePackage{etoolbox}
⟨plain, context⟩\input etoolbox-generic
```

Setup the `memoize` namespace in LuaTeX.

```
\ifdefined\luatexversion
  \directlua{memoize = {}}
\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`.

```
⟨latex⟩\RequirePackage{pdftexcmds}
⟨plain⟩\input pdftexcmds.sty
  ⟨*context⟩
\directlua{%
  require("pdftexcmds")
  tex.enableprimitives('pdf@', {'draftmode'})
}
\long\def\pdf@mdfivesum#1{%
  \directlua{%
    oberdiek.pdftexcmds.mdfivesum("\luaescapestring{#1}", "byte")%
  }%
}%
\def\pdf@system#1{%
  \directlua{%
    oberdiek.pdftexcmds.system("\luaescapestring{#1}")%
  }%
}
\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.

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

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

```
    if filehandle == nil then
    else
      tex.write(filehandle:seek("end"))
      io.close(filehandle)
    end
  end
}%
\def\pdf@filesize#1{%
  \directlua{memoize.filesize("\luaescapestring{#1}")}%
}
  ⟨/context⟩
```

Take care of some further differences between the engines.

```
\ifdef\pdftexversion{%
}{%
  \def\pdfhorigin{1true in}%
  \def\pdfvorigin{1true in}%
  \ifdef\XeTeXversion{%
    \let\quitvmode\leavevmode
  }{%
    \ifdef\luatexversion{%
      \let\pdfpagewidth\pagewidth
      \let\pdfpageheight\pageheight
      \def\pdfmajorversion{\pdfvariable majorversion}%
      \def\pdfminorversion{\pdfvariable minorversion}%
    }{%
      \PackageError{memoize}{Support for this TeX engine is not implemented}{}%
    }%
  }%
}
```

In ConTEXt, `\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 ConTEXt files to replace all its occurrences by `\normalunexpanded`. Oh yeah!

Shipout We will next load our own auxiliary package, CollArgs, but before we do that, we need to grab `\shipout` in plain TEX. 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.

```
  ⟨*plain⟩
\def\mmz@regular@shipout{%
  \global\advance\mmzRegularPages1\relax
  \mmz@primitive@shipout
}
\edef\mmz@temp{\string\shipout}%
\edef\mmz@tempa{\meaning\shipout}%
\ifx\mmz@temp\mmz@tempa
  \let\mmz@primitive@shipout\shipout
  \let\shipout\mmz@regular@shipout
\else
  \PackageError{memoize}{Cannot grab \string\shipout, it is already redefined}{}%
\fi
  ⟨/plain⟩
```

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

```
⟨latex⟩\RequirePackage{advice}
⟨plain⟩\input advice
⟨context⟩\input t-advice
⟨/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.

```
⟨*(plain, context) & (mmz, nommz, mmzable)⟩
\def\ifmmz@loadstatus#1{%
  \ifnum#1=0\csname mmz@loadstatus\endcsname\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
}
⟨/(plain, context) & (mmz, nommz, mmzable)⟩
⟨*mmz⟩
⟨latex⟩\@ifpackageloaded{nomemoize}%
⟨plain, context⟩\ifmmz@loadstatus{3}%
{%
  \PackageError{memoize}{Cannot load the package, as "nomemoize" is already
    loaded. Memoization will NOT be in effect}{Packages "memoize" and
    "nomemoize" are mutually exclusive, please load either one or the other.}%
⟨latex⟩  \pgfkeys{/memoize/package options/.unknown/.code={}}
⟨latex⟩  \ProcessPgfPackageOptions{/memoize/package options}
    \endinput
}{}%
⟨latex⟩\@ifpackageloaded{memoizable}%
⟨plain, context⟩\ifmmz@loadstatus{2}%
{%
  \PackageError{memoize}{Cannot load the package, as "memoizable" is already
    loaded}{Package "memoizable" is loaded by packages which support
    memoization.  Memoize must be loaded before all such packages.  The
    compilation log can help you figure out which package loaded "memoizable";
    please move
⟨latex⟩    "\string\usepackage{memoize}"
⟨plain⟩    "\string\input memoize"
⟨context⟩    "\string\usemodule[memoize]"
    before the
⟨latex⟩    "\string\usepackage"
⟨plain⟩    "\string\input"
⟨context⟩    "\string\usemodule"
    of that package.}%
⟨latex⟩    \pgfkeys{/memoize/package options/.unknown/.code={}}
⟨latex⟩    \ProcessPgfPackageOptions{/memoize/package options}
  \endinput
}{}%
⟨plain, context⟩\ifmmz@loadstatus{1}{\endinput}{}%
⟨plain, context⟩\def\mmz@loadstatus{1}%
⟨/mmz⟩
⟨*mmzable⟩
⟨latex⟩\@ifpackageloaded{memoize}%
⟨plain, context⟩\ifmmz@loadstatus{1}%
{\endinput}{}
⟨latex⟩\@ifpackageloaded{nomemoize}%
⟨plain, context⟩\ifmmz@loadstatus{3}%
{\endinput}{}%
⟨plain, context⟩\ifmmz@loadstatus{2}{\endinput}{}%
⟨plain, context⟩\def\mmz@loadstatus{2}%
```

```
⟨/mmzable⟩
⟨*nommz⟩
⟨latex⟩\@ifpackageloaded{memoize}
⟨plain, context⟩\ifmmz@loadstatus{1}%
{%
  \PackageError{nomemoize}{Cannot load the package, as "memoize" is already
    loaded; memoization will remain in effect}{Packages "memoize" and
    "nomemoize" are mutually exclusive, please load either one or the other.}%
  \endinput }{}%
⟨latex⟩\@ifpackageloaded{memoizable}%
⟨plain, context⟩\ifmmz@loadstatus{2}%
{%
  \PackageError{nomemoize}{Cannot load the package, as "memoizable" is already
    loaded}{Package "memoizable" is loaded by packages which support
    memoization.  (No)Memoize must be loaded before all such packages.  The
    compilation log can help you figure out which package loaded
    "memoizable"; please move
⟨latex⟩    "\string\usepackage{nomemoize}"
⟨plain⟩    "\string\input memoize"
⟨context⟩    "\string\usemodule[memoize]"
    before the
⟨latex⟩    "\string\usepackage"
⟨plain⟩    "\string\input"
⟨context⟩    "\string\usemodule"
    of that package.}%
  \endinput
}{}%
⟨plain, context⟩\ifmmz@loadstatus{3}{\endinput}{}%
⟨plain, context⟩\def\mmz@loadstatus{3}%
⟨/nommz⟩

⟨*mmz⟩
```

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

```
\def\filetotoks#1#2{%
  \immediate\openin0{#2}%
  #1={}%
  \loop
  \unless\ifeof0
    \read0 to \totoks@temp
```

We need the `\expandafter`s for our `\toksapp` macro.

```
    \expandafter\toksapp\expandafter#1\expandafter{\totoks@temp}%
  \repeat
  \immediate\closein0
}
```

Other little things.

```
\newif\ifmmz@temp
\newtoks\mmz@temptoks
\newbox\mmz@box
\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`, is exists as a no-op.

```
\def\mmzset#1{\pgfqkeys{/mmz}{#1}\ignorespaces}
```

```
⟨/mmz⟩
⟨nommz, mmzable⟩\def\mmzset#1{\ignorespaces}
```

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

```
⟨mmz⟩\def\nommzkeys#1{}
⟨*nommz, mmzable⟩
\def\nommzkeys{\pgfqkeys{/mmz}}
\pgfqkeys{/mmz}{.unknown/.code={\pgfkeysdef{\pgfkeyscurrentkey}{}}}
⟨/nommz, mmzable⟩
```

enable disable \ifmemoize 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 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.

```
⟨mmz, nommz, mmzable⟩\newif\ifmemoize
⟨*mmz⟩
\mmzset{%
  enable/.style={begindocument/.append code=\memoizetrue},
  disable/.style={begindocument/.append code=\memoizefalse},
  begindocument/.append style={
    enable/.code=\memoizetrue,
    disable/.code=\memoizefalse,
  },
```

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

```
  enable,
}
```

normal readonly recompile 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 matters in \Memoize (and \mmz@process@ccmemo).[1]

```
\def\mmz@mode@normal{0}
\def\mmz@mode@readonly{1}
\def\mmz@mode@recompile{2}
\let\mmz@mode\mmz@mode@normal
\mmzset{%
  normal/.code={\let\mmz@mode\mmz@mode@normal},
  readonly/.code={\let\mmz@mode\mmz@mode@readonly},
  recompile/.code={\let\mmz@mode\mmz@mode@recompile},
}
```

path path/relative path/dir path/prefix Key path executes the given keylist in path /mmz/path, to determine the full *path prefix* to memo and extern files (M§2.5,4.2): relative, true by default, determines whether the location of these files is relative to the current directory; dir sets their directory; and prefix sets the first, fixed part of their basename; the second part containing the MD5 sum(s) is not under user control, and neither is the suffix. These subkeys will be initialized a bit later, via no memo dir.

```
\mmzset{%
  path/.code={\pgfqkeys{/mmz/path}{#1}},
  path/.cd,
```

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

```
  relative/.is if=mmz@relativepath,
  dir/.store in=\mmz@dir,
  dir/.value required,
  prefix/.store in=\mmz@prefix,
  prefix/.value required,
}
\newif\ifmmz@relativepath
```

Key `path` concludes by performing two post-path-setting actions: creating the given directory if `mkdir` is in effect, and noting the new path prefix in record files (by eventually executing `record/prefix`, which typically puts a `\mmzPrefix` line in the `.mmz` file). These actions are the reason for having the path setting keys grouped under `path` — we don't want them to be triggered by changes of the individual components of the path. Similarly, we don't want them triggered by multiple invocations of `path` in the preamble; only the final setting matters, so `path` is only equipped with the action-triggering code at the beginning of the document.

```
\mmzset{%
  begindocument/.append style={
    path/.append code=\mmz@maybe@mkmemodir\mmz@record@prefix,
  },
```

Consequently, the post-path-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.

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

Define the paths to the memo directory and the prefix.

```
\def\mmz@dir@path{\ifmmz@relativepath.\fi/\mmz@dir}
\def\mmz@prefix@path{\mmz@dir@path/\mmz@prefix}
```

`mkdir`
`mkdir command`
Should we create the memo/extern directory if it doesn't exist? And which command should we use to create it? Of course, shell escape must be properly configured for this to work (^{M}§??).

```
\mmzset{
  mkdir/.is if=mmz@mkdir,
  mkdir command/.code={\def\mmz@mkdir@command##1{#1}},
  mkdir command={mkdir "#1"},
}
```

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

```
\newif\ifmmz@mkdir
\def\mmz@maybe@mkmemodir{%
  \ifmmz@mkdir
    \pdf@system{\mmz@mkdir@command{\mmzOutputDirectory\mmz@dir@path}}%
  \fi
}
```

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

```
\mmzset{%
  memo dir/.style={
    mkdir,
    path={
      relative,
      dir={#1.memo.dir},
      prefix={},
    },
  },
  memo dir/.default=\mmzUnquote\jobname,
  no memo dir/.style={
    mkdir=false,
    path={
      relative,
      dir={},
      prefix={#1.},
    },
  },
  no memo dir/.default=\mmzUnquote\jobname,
  no memo dir,
}
```

`\mmzUnquote` If the expanded argument is surrounded by double quotes, remove them. This relies on `#1` containing no quotes other than the potential surrounding quotes, which should be the case when applying the macro to `\jobname`. An empty `#1` is dealt with correctly, even if `\jobname` can hardly ever be empty (needs `openout_any=a`).

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

```
\def\mmzUnquote#1{\expanded{\noexpand\mmz@unquote#1}\mmz@unquote@end}
\def\mmz@unquote#1{%
  \ifx\mmz@unquote@end#1%
  \else
    \ifx"#1%
      \expandafter\expandafter\expandafter\mmz@unquote@quotes
    \else
      \expandafter\expandafter\expandafter\mmz@unquote@noquotes
      \expandafter\expandafter\expandafter#1%
    \fi
  \fi
}
\def\mmz@unquote@quotes#1"\mmz@unquote@end{#1}
\def\mmz@unquote@noquotes#1\mmz@unquote@end{#1}
```

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

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

`verbatim` `verb` `no verbatim` These keys are tricky. For one, there's `verbatim`, which sets all characters' category codes to other, and there's `verb`, which leaves braces untouched (well, honestly, it redefines them). But 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.

```
\newif\ifmmz@verbatim
\def\mmzRawCollectorOptions{}
\mmzset{
  verbatim/.code={%
    \def\mmzRawCollectorOptions{\collargsVerbatim}%
    \mmz@verbatimtrue
  },
  verb/.code={%
    \def\mmzRawCollectorOptions{\collargsVerb}%
    \mmz@verbatimtrue
  },
  no verbatim/.code={%
    \def\mmzRawCollectorOptions{\collargsNoVerbatim}%
    \mmz@verbatimfalse
  },
}
```

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.

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

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

```
  \begingroup
```

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

```
  \ifx\mmz@temp[%]
    \def\mmz@verbatim@fix{}%
    \expandafter\mmz@ii
  \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.)

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

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

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

Apply the options given in the optional argument.

```
  \mmzset{#1}%
  \mmz@iii
}
\def\mmz@iii{%
```

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

```
  \ifmmz@verbatim
    \expandafter\mmz@do@verbatim
  \else
    \expandafter\mmz@do
  \fi
}
```

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

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

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).

```
\def\mmz@do@verbatim#1{%
  \expanded{%
    \noexpand\CollectArgumentsRaw{%
      \noexpand\collargsCaller{\noexpand\mmz}%
      \expandonce\mmzRawCollectorOptions
      \mmz@verbatim@fix
    }%
  }{+m}\mmz@do
}
```

`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).

```
  ⟨*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.

```
\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.

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

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

```
  \begingroup
```

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.

```
  \def\mmz@temp{#1}%
  \ifx\mmz@temp\mmz@noarg
    \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
  \else
```

```
    \def\mmz@verbatim@fix{}%
    \mmzset{#1}%
  \fi
  \mmz@env@iii
}{}
\def\mmz@noarg{\mmz@noarg}
  ⟨/latex⟩
⟨plain⟩ \def\memoize{%
⟨context⟩ \def\startmemoize{%
  ⟨*plain, context⟩
  \begingroup
```

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.

```
  \futurelet\mmz@temp\mmz@env@i
}
\def\mmz@env@i{%
  \ifx\mmz@temp[%]
    \def\mmz@verbatim@fix{}%
    \expandafter\mmz@env@ii
  \else
    \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
    \expandafter\mmz@env@iii
  \fi
}
\def\mmz@env@ii[#1]{%
  \mmzset{#1}%
  \mmz@env@iii
}
  ⟨/plain, context⟩
\def\mmz@env@iii{%
  \long\edef\mmz@do##1{%
```

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

```
    \noexpand\Memoize{##1}{##1\unskip}%
  }%
  \expanded{%
    \noexpand\CollectArgumentsRaw{%
```

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

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

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

```
      \expandonce\mmzRawCollectorOptions
```

The category code fix, if needed.

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

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

```
  }{&&{\collargsAppendPostwrap{}}!t{ }+b{memoize}}{\mmz@do}%
}%
⟨/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`.

```
⟨*mmz, nommz⟩
\protected\def\nommz#1#{%
  \afterassignment\nommz@i
  \let\mmz@temp
}
\def\nommz@i{%
  \bgroup
  \memoizefalse
}
⟨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.

```
  ⟨*latex⟩
\newenvironment{nomemoize}[1][]{%
  \memoizefalse
  \ignorespaces
}{%
  \unskip
}
  ⟨/latex⟩
  ⟨*plain, context⟩
⟨plain⟩\def\nomemoize{%
⟨context⟩\def\startnomemoize{%
```

Start a group to delimit `\memoizefalse`.

```
  \begingroup
  \memoizefalse
  \futurelet\mmz@temp\nommz@env@i
}
\def\nommz@env@i{%
  \ifx\mmz@temp[%]
    \expandafter\nommz@env@ii
```

No optional argument, no problems with spaces.

```
  \fi
}
\def\nommz@env@ii[#1]{%
  \ignorespaces
}
⟨plain⟩\def\endnomemoize{%
⟨context⟩\def\stopnomemoize{%
  \endgroup
  \unskip
}
  ⟨/plain, context⟩
  ⟨*nommz⟩
⟨plain, latex⟩\let\memoize\nomemoize
⟨plain, latex⟩\let\endmemoize\endnomemoize
⟨context⟩\let\startmemoize\startnomemoize
⟨context⟩\let\stopmemoize\stopnomemoize
  ⟨/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.

```
\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.

```
\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.

```
⟨/mmz, nommz⟩
⟨*mmz⟩
\def\mmz@maybe@scantokens{%
  \ifmmz@verbatim
    \expandafter\mmz@scantokens
  \else
    \expandafter\@firstofone
  \fi
}
```

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.)

```
\long\def\mmz@scantokens#1{%
  \expanded{%
    \newlinechar=13
    \unexpanded{\scantokens{#1\endinput}}%
    \newlinechar=\the\newlinechar
  }%
}
```

`\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 ($^{\mathrm{M}}$§4.1).

`\Memoize` takes two arguments, which contain two potentially different versions of the code submitted to memoization: `#1` contains the code which ⟨*code MD5 sum*⟩ 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.

```
\newtoks\mmz@mdfive@source
\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.

```
\let\Memoize\@secondoftwo
\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`).

```
  \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{...}`.

```
  \expandafter\expandafter\expandafter\expandafter
  \expandafter\expandafter\expandafter
  \mmz@exec@source
  \expandafter\expandafter\expandafter\expandafter
  \expandafter\expandafter\expandafter
  {%
    \mmz@maybe@scantokens{#2}%
  }%
  \mmz@trace@Memoize
```

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

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

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

```
  \ifmemoizing
  \else
    \ifmemoize
```

Compute ⟨*code md5sum*⟩ off of the first argument, 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.

```
      \xdef\mmz@code@mdfivesum{\pdf@mdfivesum{\the\mmz@mdfive@source}}%
      \mmz@trace@code@mdfive
```

Recompile mode forces memoization.

```
      \ifnum\mmz@mode=\mmz@mode@recompile\relax
        \ifnum\pdf@draftmode=0
          \let\mmz@action\mmz@memoize
        \fi
      \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.

```
        \mmz@process@cmemo
        \ifmmzUnmemoizable
        \mmz@trace@cmemo@unmemoizable
      \else
        \ifmmz@abort
```

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

```
          \mmz@trace@process@cmemo@fail
          \ifnum\mmz@mode=\mmz@mode@readonly\relax
          \else
            \ifnum\pdf@draftmode=0
              \let\mmz@action\mmz@memoize
            \fi
          \fi
        \else
          \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.

```
          \let\mmz@action\mmz@process@ccmemo
        \fi
      \fi
    \fi
  \fi
\fi
\mmz@action
}
```

`\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`.

```
\def\mmz@compile{%
  \mmz@trace@compile
  \expanded{%
    \endgroup
    \noexpand\inmemoizetrue
    \the\mmz@exec@source
    \ifinmemoize\noexpand\inmemoizetrue\else\noexpand\inmemoizefalse\fi
  }%
}
```

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

```
  ⟨*!context⟩
\ifdefined\luatexversion
  \directlua{%
```

```
    luatexbase.add_to_callback(
      "show_error_message",
      function()
        memoize.abort = true
        texio.write_nl(status.lasterrorstring)
      end,
      "Abort memoization on error"
    )
  }%
  \def\mmz@lua@atbeginmemoization{%
    \directlua{memoize.abort = false}%
  }%
  \def\mmz@lua@atendmemoization{%
    \directlua{%
      if memoize.abort then
        tex.print("\noexpand\\mmzAbort")
      end
    }%
  }%
\else
  ⟨/!context⟩
  \let\mmz@lua@atbeginmemoization\relax
  \let\mmz@lua@atendmemoization\relax
⟨!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.

```
\def\mmz@memoize{%
  \mmz@trace@memoize
  \memoizingtrue
  \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.

```
  \edef\memoizinggrouplevel{\the\currentgrouplevel}%
  \global\mmz@abortfalse
  \global\mmzUnmemoizablefalse
  \global\mmz@seq 0
  \global\setbox\mmz@tbe@box\vbox{}%
  \global\mmz@ccmemo@resources{}%
  \global\mmzCMemo{}%
  \global\mmzCCMemo{}%
  \global\mmzContextExtra{}%
  \gdef\mmzAtEndMemoizationExtra{}%
  \gdef\mmzAfterMemoizationExtra{}%
  \mmz@lua@atbeginmemoization
```

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

```
  \mmzAtBeginMemoization
  \mmzDriver{\the\mmz@exec@source}%
  \mmzAtEndMemoization
  \mmzAtEndMemoizationExtra
  \mmz@lua@atendmemoization
  \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.

```
    \global\mmzContextExtra{}%
    \gtoksapp\mmzCMemo{\global\mmzUnmemoizabletrue}%
    \mmz@write@cmemo
    \mmz@trace@endmemoize@unmemoizable
    \PackageWarning{memoize}{Marking this code as unmemoizable}%
  \else
    \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.

```
      \mmz@trace@endmemoize@aborted
      \PackageWarning{memoize}{Memoization was aborted}%
      \mmz@write@empty@cmemo
    \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).

```
      \mmz@compute@context@mdfivesum
      \mmz@write@cmemo
      \mmz@write@ccmemo
      \mmz@shipout@externs
      \mmz@trace@endmemoize@ok
    \fi
  \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.

```
  \expandafter\endgroup
  \expandafter\let
    \expandafter\mmzIncludeExtern\expandafter\mmz@include@extern@from@tbe@box
  \mmzAfterMemoization
  \mmzAfterMemoizationExtra
}
```

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

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

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

```
\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).

```
\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`.

```
\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`.

```
\newif\ifmmzUnmemoizable
```

`\mmzAtBeginMemoization`
`\mmzAtEndMemoization`
`\mmzAfterMemoization`
`at begin memoization`
`at end memoization`
`after memoization`

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 macro (`\mmzAtBeginMemoization`, `\mmzAtEndMemoization`, `\mmzAfterMemoization`), and the assignment should be local. In the latter case, one should modify the extra macro (`\mmzAtEndMemoizationExtra`, `\mmzAfterMemoizationExtra`; there is no `\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.

```
\def\mmzAtBeginMemoization{}
\def\mmzAtEndMemoization{}
\def\mmzAfterMemoization{}
\mmzset{
  at begin memoization/.code={%
    \ifmemoizing
      \expandafter\@firstofone
    \else
      \expandafter\appto\expandafter\mmzAtBeginMemoization
    \fi
    {#1}%
  },
  at end memoization/.code={%
    \ifmemoizing
      \expandafter\gappto\expandafter\mmzAtEndMemoizationExtra
    \else
      \expandafter\appto\expandafter\mmzAtEndMemoization
    \fi
    {#1}%
  },
  after memoization/.code={%
    \ifmemoizing
      \expandafter\gappto\expandafter\mmzAfterMemoizationExtra
    \else
      \expandafter\appto\expandafter\mmzAfterMemoization
    \fi
    {#1}%
  },
}
```

`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.

```
\mmzset{
  driver/.store in=\mmzDriver,
  driver=\mmzSingleExternDriver,
}
```

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

```
\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.)

```
\long\def\mmzSingleExternDriver#1{%
  \xtoksapp\mmzCCMemo{\mmz@maybe@quitvmode}%
  \setbox\mmz@box\mmz@capture{#1}%
  \mmzExternalizeBox\mmz@box\mmz@temptoks
  \xtoksapp\mmzCCMemo{\the\mmz@temptoks}%
  \mmz@maybe@quitvmode\box\mmz@box
}
```

`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 XƎTEX, 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.

```
\mmzset{
  capture/.is choice,
  capture/hbox/.code={%
    \let\mmz@capture\hbox
    \def\mmz@maybe@quitvmode{\noexpand\quitvmode}%
  },
  capture/vbox/.code={%
    \let\mmz@capture\vbox
    \def\mmz@maybe@quitvmode{}%
  },
  capture=hbox,
}
```

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 $^{\mathrm{M}}$§4.4.4 for details.

`integrated driver` This is an advice key, residing in `/mmz/auto`. Given ⟨*suffix*⟩ as the only argument, it declares conditional `\ifmemoizing`⟨*suffix*⟩, 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.

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

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

```
      \@firstofone
    }%
  },
}
```

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

```
\newcommand\IfMemoizing[2][\mmz@Ifmemoizing@nogrouplevel]{%>\fi
\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.

```
  \ifnum\currentgrouplevel=\the\numexpr\memoizinggrouplevel+#1\relax\relax
    \expandafter\expandafter\expandafter\@firstoftwo
  \else
    \expandafter\expandafter\expandafter\@secondoftwo
  \fi
\else
  \expandafter\@secondoftwo
\fi
}
\def\mmz@Ifmemoizing@nogrouplevel{0\relax\relax\fi\iftrue}
```

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

```
\def\mmz@trace#1{\immediate\write16{[tracing memoize] #1}}
\def\mmz@trace@context{\mmz@trace{\space\space
    Context: "\expandonce{\mmz@context@key}" --> \mmz@context@mdfivesum}}
\def\mmz@trace@Memoize@on{%
  \mmz@trace{%
    Entering \noexpand\Memoize (%
    \ifmemoize enabled\else disabled\fi,
    \ifnum\mmz@mode=\mmz@mode@recompile recompile\fi
    \ifnum\mmz@mode=\mmz@mode@readonly readonly\fi
    \ifnum\mmz@mode=\mmz@mode@normal normal\fi
    \space mode) on line \the\inputlineno
  }%
  \mmz@trace{\space\space Code: \the\mmz@mdfive@source}%
}
\def\mmz@trace@code@mdfive@on{\mmz@trace{\space\space
    Code md5sum: \mmz@code@mdfivesum}}
\def\mmz@trace@compile@on{\mmz@trace{\space\space Compiling}}
\def\mmz@trace@memoize@on{\mmz@trace{\space\space Memoizing}}
\def\mmz@trace@endmemoize@ok@on{\mmz@trace{\space\space
    Memoization completed}}%
\def\mmz@trace@endmemoize@aborted@on{\mmz@trace{\space\space
    Memoization was aborted}}
\def\mmz@trace@endmemoize@unmemoizable@on{\mmz@trace{\space\space
    Marking this code as unmemoizable}}
```

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

```
\def\mmz@trace@process@cmemo@on{\mmz@trace{\space\space
    Attempting to utilize c-memo \mmz@cmemo@path}}
\def\mmz@trace@process@no@cmemo@on{\mmz@trace{\space\space
    C-memo does not exist}}
\def\mmz@trace@process@cmemo@ok@on{\mmz@trace{\space\space
    C-memo was processed successfully}\mmz@trace@context}
\def\mmz@trace@process@cmemo@fail@on{\mmz@trace{\space\space
    C-memo input failed}}
\def\mmz@trace@cmemo@unmemoizable@on{\mmz@trace{\space\space
```

```
    This code was marked as unmemoizable}}
\def\mmz@trace@process@ccmemo@on{\mmz@trace{\space\space
    Attempting to utilize cc-memo \mmz@ccmemo@path\space
    (\ifmmz@direct@ccmemo@input\else in\fi direct input)}}
\def\mmz@trace@resource@on#1{\mmz@trace{\space\space
    Extern file does not exist: #1}}
\def\mmz@trace@process@ccmemo@ok@on{%
  \mmz@trace{\space\space Utilization successful}}
\def\mmz@trace@process@no@ccmemo@on{%
  \mmz@trace{\space\space CC-memo does not exist}}
\def\mmz@trace@process@ccmemo@fail@on{%
  \mmz@trace{\space\space Cc-memo input failed}}
```

tracing
\mmzTracingOn
\mmzTracingOff

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

```
\mmzset{%
  trace/.is choice,
  trace/.default=true,
  trace/true/.code=\mmzTracingOn,
  trace/false/.code=\mmzTracingOff,
}
\def\mmzTracingOn{%
  \let\mmz@trace@Memoize\mmz@trace@Memoize@on
  \let\mmz@trace@code@mdfive\mmz@trace@code@mdfive@on
  \let\mmz@trace@compile\mmz@trace@compile@on
  \let\mmz@trace@memoize\mmz@trace@memoize@on
  \let\mmz@trace@process@cmemo\mmz@trace@process@cmemo@on
  \let\mmz@trace@endmemoize@ok\mmz@trace@endmemoize@ok@on
  \let\mmz@trace@endmemoize@unmemoizable\mmz@trace@endmemoize@unmemoizable@on
  \let\mmz@trace@endmemoize@aborted\mmz@trace@endmemoize@aborted@on
  \let\mmz@trace@process@cmemo\mmz@trace@process@cmemo@on
  \let\mmz@trace@process@cmemo@ok\mmz@trace@process@cmemo@ok@on
  \let\mmz@trace@process@no@cmemo\mmz@trace@process@no@cmemo@on
  \let\mmz@trace@process@cmemo@fail\mmz@trace@process@cmemo@fail@on
  \let\mmz@trace@cmemo@unmemoizable\mmz@trace@cmemo@unmemoizable@on
  \let\mmz@trace@process@ccmemo\mmz@trace@process@ccmemo@on
  \let\mmz@trace@resource\mmz@trace@resource@on
  \let\mmz@trace@process@ccmemo@ok\mmz@trace@process@ccmemo@ok@on
  \let\mmz@trace@process@no@ccmemo\mmz@trace@process@no@ccmemo@on
  \let\mmz@trace@process@ccmemo@fail\mmz@trace@process@ccmemo@fail@on
}
\def\mmzTracingOff{%
  \let\mmz@trace@Memoize\relax
  \let\mmz@trace@code@mdfive\relax
  \let\mmz@trace@compile\relax
  \let\mmz@trace@memoize\relax
  \let\mmz@trace@process@cmemo\relax
  \let\mmz@trace@endmemoize@ok\relax
  \let\mmz@trace@endmemoize@unmemoizable\relax
  \let\mmz@trace@endmemoize@aborted\relax
  \let\mmz@trace@process@cmemo\relax
  \let\mmz@trace@process@cmemo@ok\relax
  \let\mmz@trace@process@no@cmemo\relax
  \let\mmz@trace@process@cmemo@fail\relax
  \let\mmz@trace@cmemo@unmemoizable\relax
  \let\mmz@trace@process@ccmemo\relax
  \let\mmz@trace@resource\@gobble
  \let\mmz@trace@process@ccmemo@ok\relax
  \let\mmz@trace@process@no@ccmemo\relax
  \let\mmz@trace@process@ccmemo@fail\relax
}
```

```
\mmzTracingOff
⟨/mmz⟩
⟨nommz, mmzable⟩\newcommand\IfMemoizing[2][]{\@secondoftwo}
⟨*mmz⟩
```

3.3 Context

\mmzContext
\mmzContextExtra

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

```
\newtoks\mmzContext
\newtoks\mmzContextExtra
```

context
clear context

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

```
\mmzset{%
  context/.code={%
    \ifmemoizing
      \expandafter\gtoksapp\expandafter\mmzContextExtra
    \else
      \expandafter\toksapp\expandafter\mmzContext
    \fi
```

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

```
    {#1,}%
  },
  clear context/.code={%
    \ifmemoizing
      \expandafter\global\expandafter\mmzContextExtra
    \else
      \expandafter\mmzContext
    \fi
    {}%
  },
  clear context/.value forbidden,
```

meaning to context
csname meaning to context
key meaning to context
key value to context
/handlers/.meaning to context
/handlers/.value to context

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

```
  meaning to context/.code={\forcsvlist\mmz@mtoc{#1}},
  csname meaning to context/.code={\mmz@mtoc@cs{#1}},
  key meaning to context/.code={\forcsvlist\mmz@mtoc\mmz@mtoc@keycmd{#1}},
  key value to context/.code={\forcsvlist\mmz@mtoc@key{#1}},
  /handlers/.meaning to context/.code={%
    \expanded{\noexpand\mmz@mtoc@cs{pgfk@\pgfkeyscurrentpath/.@cmd}}},
  /handlers/.value to context/.code={%
    \expanded{\noexpand\mmz@mtoc@cs{pgfk@\pgfkeyscurrentpath}}},
}
\def\mmz@mtoc#1{%
  \collargs@cs@cases{#1}%
    {\mmz@mtoc@cmd{#1}}%
    {\mmz@mtoc@error@notcsorenv{#1}}%
    {%
      \mmz@mtoc@cs{%
⟨context⟩        start%
        #1}%
      \mmz@mtoc@cs{%
⟨latex, plain⟩        end%
⟨context⟩        stop%
        #1}%
    }%
```

```
}
\def\mmz@mtoc@cmd#1{%
  \begingroup
  \escapechar=-1
  \expandafter\endgroup
  \expandafter\mmz@mtoc@cs\expandafter{\string#1}%
}
\def\mmz@mtoc@cs#1{%
  \pgfkeysvalueof{/mmz/context/.@cmd}%
  \expandafter\string\csname#1\endcsname={\expandafter\meaning\csname#1\endcsname}%
  \pgfeov
}
\def\mmz@mtoc@key#1{\mmz@mtoc@cs{pgfk@#1}}
\def\mmz@mtoc@key#1{\mmz@mtoc@cs{pgfk@#1/.@cmd}}
\def\mmz@mtoc@error@notcsorenv#1{%
  \PackageError{memoize}{'\detokenize{#1}' passed to key 'meaning to context' is neither
}
```

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`.

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

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

```
\newtoks\mmzCMemo
```

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

```
\mmzset{%
  include source in cmemo/.is if=mmz@include@source,
}
\newif\ifmmz@include@source
\mmz@include@sourcetrue
```

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

```
\def\mmz@write@cmemo{%
```

Open the file for writing.

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

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

```
  \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.

```
  \immediate\write\mmz@out{%
    \global\mmzContextExtra{\the\mmzContextExtra}\collargs@percentchar
  }%
```

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

```
  \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.

```
  \ifmmz@include@source
    \immediate\write\mmz@out{\noexpand\mmzSource}%
    \immediate\write\mmz@out{\the\mmz@mdfive@source}%
  \fi
```

Close the file.

```
  \immediate\closeout\mmz@out
```

Record that we wrote a new c-memo.

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

`\mmz@write@empty@cmemo` This macro is used to create an empty c-memo on aborted memoization, to make sure that no leftover c-memo tricks Memoize into thinking that the code was successfully memoized.

```
\def\mmz@write@empty@cmemo{%
  \immediate\openout\mmz@out{\mmz@cmemo@path}%
  \immediate\closeout\mmz@out
}
```

`\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.

```
\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.

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

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

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

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

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

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

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

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

```
  \IfFileExists{\mmz@cmemo@path}{%
    \input{\mmz@cmemo@path}%
    \pgfkeysalso{/mmz/record/used cmemo={\mmz@cmemo@path}}%
  }{%
    \mmz@trace@process@no@cmemo
  }%
```

Compute the context MD5 sum.

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

`\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.

```
\def\mmz@compute@context@mdfivesum{%
  \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.

```
  \begingroup
  \begingroup
  \def\width{\string\width}%
  \def\height{\string\height}%
  \def\depth{\string\depth}%
  \edef\mmz@paddings{\mmz@paddings}%
  \expandafter\endgroup
  \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.

```
⟨latex⟩   \protected@xdef
⟨!latex⟩  \xdef
  \mmz@context@key{\mmz@context@key}%
  \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`.

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

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`.

```
\def\mmz@ccmemo@path{%
  \mmz@prefix@path\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.

```
\newtoks\mmzCCMemo
```

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

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

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

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

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

```
\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`.

```
  \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.

```
  \begingroup
  \the\mmz@ccmemo@resources
  \endgroup
```

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

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

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

```
  \ifmmz@include@context
    \immediate\write\mmz@out{\noexpand\mmzThisContext}%
    \immediate\write\mmz@out{\expandonce{\mmz@context@key}}%
  \fi
```

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

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

Record that we wrote a new cc-memo.

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

\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.

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

\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.

```
\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.

```
  \ifnum0\pdf@filesize{\mmz@dir@path/#1}=0
    \ifmmz@direct@ccmemo@input
      \let\mmzMemo\endinput
    \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).

```
      \long\def\mmzMemo##1\mmzEndMemo\par{}%
    \fi
    \mmz@trace@resource{#1}%
  \fi
}
```

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

```
\def\mmz@process@ccmemo{%
  \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!

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

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

```
  \def\mmzMemo{%
    \endgroup
    \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.

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

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.

```
  \xdef\mmzEndMemo{%
    \ifmmz@direct@ccmemo@input
      \noexpand\endinput
    \else
```

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

```
      \unexpanded{%
        \def\mmz@temp\par{}%
        \mmz@temp
      }%
    \fi
  }%
```

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).

```
  \xdef\mmzThisContext{%
    \ifmmz@direct@ccmemo@input
      \noexpand\endinput
    \else
      \unexpanded{%
        \long\def\mmz@temp##1\mmzEndMemo\par{}%
        \mmz@temp
      }%
    \fi
  }%
```

Input the cc-memo if it exists.

```
  \IfFileExists{\mmz@ccmemo@path}{%
    \ifmmz@direct@ccmemo@input
      \input{\mmz@ccmemo@path}%
    \else
```

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

```
      \filetotoks\toks@{\mmz@ccmemo@path}%
      \the\toks@
    \fi
```

Record that we have used the cc-memo.

```
    \pgfkeysalso{/mmz/record/used ccmemo={\mmz@ccmemo@path}}%
  }{%
    \mmz@trace@process@no@ccmemo
  }%
  \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`.)

```
    \mmz@trace@process@ccmemo@fail
    \ifnum\mmz@mode=\mmz@mode@readonly\relax
      \expandafter\expandafter\expandafter\mmz@compile
    \else
      \expandafter\expandafter\expandafter\mmz@memoize
    \fi
  \else
    \mmz@trace@process@ccmemo@ok
  \fi
}
```

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.

```
\newcount\mmz@seq
\def\mmz@extern@basename{%
  \mmz@prefix\mmz@code@mdfivesum-\mmz@context@mdfivesum
  \ifnum\mmz@seq>0 -\the\mmz@seq\fi
}
\def\mmz@extern@name{\mmz@extern@basename.pdf}
\def\mmz@extern@basepath{\mmz@dir@path/\mmz@extern@basename}
\def\mmz@extern@path{\mmz@extern@basepath.pdf}
```

`padding left` `padding right` `padding top` `padding bottom` 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 Ti*k*Z overlays, `\rlap` and `\llap`, etc.

```
\mmzset{
  padding left/.store in=\mmz@padding@left,
  padding right/.store in=\mmz@padding@right,
  padding top/.store in=\mmz@padding@top,
  padding bottom/.store in=\mmz@padding@bottom,
```

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

```
  padding/.style={
    padding left=#1, padding right=#1,
    padding top=#1, padding bottom=#1
  },
```

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.

```
  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.

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

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

```
  padding to context,
}
\def\mmz@paddings{%
  \mmz@padding@left,\mmz@padding@bottom,\mmz@padding@right,\mmz@padding@top
}
```

`\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 $^{\text{M}}$§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*⟩.

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

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

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

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

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

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

```
     \noexpand\mmzIncludeExtern
```

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

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

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

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

The dimensions of the extern.

```
      {\the\wd#1}%
      {\the\ht#1}%
      {\the\dp#1}%
```

The padding values.

```
      {\the\dimexpr\mmz@padding@left}%
      {\the\dimexpr\mmz@padding@bottom}%
      {\the\dimexpr\mmz@padding@right}%
      {\the\dimexpr\mmz@padding@top}%
  }%
```

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.

```
  \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.

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

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

```
  \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*⟩.

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

\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.

```
\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.

```
\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.

```
\def\mmz@shipout@externs{%
  \global\mmz@seq 0
  \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).

```
    \def\width{\wd\mmz@box}%
    \def\height{\ht\mmz@box}%
    \def\depth{\dp\mmz@box}%
    \vskip1pt
    \ifmmzkeepexterns\expandafter\unvcopy\else\expandafter\unvbox\fi\mmz@tbe@box
    \@whilesw\ifdim0pt=\lastskip\fi{%
      \setbox\mmz@box\lastbox
      \mmz@shipout@extern
    }%
  }%
}
```

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

```
\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.

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

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

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

```
\begingroup
\ifnum\mag=1000
\else
  \mmz@shipout@mag
\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 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.

```
⟨*plain, latex⟩
\pdfpagewidth\dimexpr
  (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)\relax
\pdfpageheight\dimexpr
  (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box+ (\mmz@padding@bottom)\relax
⟨/plain, latex⟩
⟨*context⟩
\edef\mmz@temp{%
  \noexpand\directlua{
    backends.codeinjections.setupcanvas({
      paperwidth=\the\numexpr\pagewidth,
      paperheight=\the\numexpr\pageheight
    })
  }%
}%
\expanded{%
  \noexpand\directlua{
    backends.codeinjections.setupcanvas({
      paperwidth=\the\numexpr\dimexpr
        \mmz@padding@left + \wd\mmz@box + \mmz@padding@right\relax,
      paperheight=\the\numexpr\dimexpr
        \mmz@padding@top + \ht\mmz@box + \dp\mmz@box+ \mmz@padding@bottom\relax
    })
  }%
}%
⟨/context⟩
```

We complete the page setup by setting the content offset.

```
\hoffset\dimexpr\mmz@padding@left - \pdfhorigin\relax
\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.

```
⟨latex⟩   \let\protect\noexpand
          \pdf@primitive\shipout\box\mmz@box
⟨context⟩ \mmz@temp
          \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.

```
  \global\advance\mmzExternPages1
```

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

```
  \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.

```
  \edef\pagenumber{\the\numexpr\mmzRegularPages
```

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

```
⟨context⟩     -1%
              +\mmzExternPages+\mmzExtraPages}%
```

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`.

```
  \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`.

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

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

```
\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 $(ax + cy + e, bx + dy + f)$.)

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

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

```
  \dimen0=\dimexpr\mmz@padding@left\relax
  \edef\mmz@padding@left{\the\dimexpr\mmz@inverse@mag\dimen0}%
```

```
\dimen0=\dimexpr\mmz@padding@bottom\relax
\edef\mmz@padding@bottom{\the\dimexpr\mmz@inverse@mag\dimen0}%
\dimen0=\dimexpr\mmz@padding@right\relax
\edef\mmz@padding@right{\the\dimexpr\mmz@inverse@mag\dimen0}%
\dimen0=\dimexpr\mmz@padding@top\relax
\edef\mmz@padding@top{\the\dimexpr\mmz@inverse@mag\dimen0}%
```

Scale the extern box.

```
\wd\mmz@box=\mmz@inverse@mag\wd\mmz@box\relax
\ht\mmz@box=\mmz@inverse@mag\ht\mmz@box\relax
\dp\mmz@box=\mmz@inverse@mag\dp\mmz@box\relax
}
```

`\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).

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

`\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.

```
⟨latex⟩\let\mmzRegularPages\ReadonlyShipoutCounter
⟨context⟩\let\mmzRegularPages\realpageno
⟨plain⟩\newcount\mmzRegularPages
```

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

```
\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.

```
\mmzset{
  enddocument/afterlastpage/.append code={%
    \ifnum\mmzExternPages>0
      \PackageWarning{memoize}{The compilation produced \the\mmzExternPages\space
        new extern\ifnum\mmzExternPages>1 s\fi.}%
    \fi
  },
}
```

`\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.

```
\newcount\mmzExtraPages
```

`\mmz@include@extern` This macro, called from cc-memos as `\mmzIncludeExtern`, inserts an extern file into the document. `#1` is the sequential number, `#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).

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

Set the extern sequential number, so that we open the correct extern file (`\mmz@extern@basename`).

```
  \mmz@seq=#1\relax
```

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.

```
  \setbox\mmz@box=#2{%
    \setbox0=\hbox{%
      \lower\dimexpr #5+#7\relax\hbox{%
        \hskip -#6\relax
        \setbox0=\hbox{%
          \mmz@insertpdfpage{\mmz@extern@path}{1}%
        }%
        \unhbox0
      }%
    }%
    \wd0 \dimexpr\wd0-#8\relax
    \ht0 \dimexpr\ht0-#9\relax
    \dp0 #5\relax
    \box0
  }%
```

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.)

```
  \mmz@tempfalse
  \mmz@if@roughly@equal{\mmz@tolerance}{#3}{\wd\mmz@box}{%
    \mmz@if@roughly@equal{\mmz@tolerance}{#4}{\ht\mmz@box}{%
      \mmz@temptrue
    }{}}{}%
  \ifmmz@temp
  \else
    \mmz@use@memo@warning{\mmz@extern@path}{#3}{#4}{#5}%
  \fi
```

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

```
  \wd\mmz@box=#3\relax
  \ht\mmz@box=#4\relax
  \box\mmz@box
```

Record that we have used this extern.

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

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

`\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 ConTEXt always uses LuaTEX.

```
⟨latex, plain⟩\ifdef\luatexversion{%
  \def\mmz@insertpdfpage#1#2{% #1 = filename, #2 = page number
    \saveimageresource page #2 mediabox {#1}%
    \useimageresource\lastsavedimageresourceindex
  }%
  ⟨*latex, plain⟩
}{%
  \ifdef\XeTeXversion{%
```

```
    \def\mmz@insertpdfpage#1#2{%
      \XeTeXpdffile #1 page #2 media
    }%
  }{% pdfLaTeX
    \def\mmz@insertpdfpage#1#2{%
      \pdfximage page #2 mediabox {#1}%
      \pdfrefximage\pdflastximage
    }%
  }%
}
⟨/latex, plain⟩
```

`\mmz@include@extern@from@tbe@box` Include the extern number `#1` residing in `\mmz@tbe@box` into the document. This helper macro makes it possible for a complex memoization driver to process the cc-memo right after memoization — by using the `\mmzkeepexternstrue\xtoksapp\mmzAfterMemoizationExtra{\the\mm` trick — to ensure that the result of the memoizing compilation matches the result of inputting the cc-memo. The rest of the arguments are gobbled, as we don't have to do any size adjustment or checking here, and the box is of the correct type.

```
\def\mmz@include@extern@from@tbe@box#1#2#3#4#5#6#7#8#9{%
  \setbox0\vbox{%
    \@tempcnta#1\relax
    \vskip1pt
    \unvcopy\mmz@tbe@box
    \@whilenum\@tempcnta>0\do{%
      \setbox0\lastbox
      \advance\@tempcnta-1\relax
    }%
    \global\setbox1\lastbox
    \@whilesw\ifdim0pt=\lastskip\fi{%
      \setbox0\lastbox
    }%
    \box\mmz@box
  }%
  \box1
}
```

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.

```
\def\mmzvalueof#1{\pgfkeysvalueof{/mmz/#1}}
\mmzset{
  extract/.estore in=\mmz@extraction@method,
  extract/.value required,
  begindocument/.append style={extract/.code=\mmz@preamble@only@warning},
```

`extract/perl` `extract/python` 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, i.e. we're not commenting out the `\mmzNewExtern` lines, because we're about to overwrite it anyway. We also request the log file, which will contain `\mmzExtractionSuccessful` if extraction was successful.

```
  extract/perl/.code={%
    \mmz@clear@extraction@log
    \pdf@system{%
      \mmzvalueof{perl extraction command}\space
      \mmzvalueof{perl extraction options}%
    }%
    \mmz@check@extraction@log{perl}%
    \def\mmz@mkdir@command##1{\mmzvalueof{perl extraction command}\space --mkdir "##1"}%
  },
  perl extraction command/.initial=memoize-extract.pl,
  perl extraction options/.initial={%
    -e -l "\mmzOutputDirectory\mmzUnquote\jobname.mmz.log" -w
⟨latex⟩    "\string\PackageWarning{memoize (perl-based extraction)}{\string\warningtext}"
⟨plain⟩    "\string\warning{memoize (perl-based extraction): \string\warningtext}"
⟨context⟩    "\string\warning{memoize (perl-based extraction): \string\warningtext}"
    "\mmzOutputDirectory\mmzUnquote\jobname.mmz"
  },
  extract=perl,
  extract/python/.code={%
    \mmz@clear@extraction@log
    \pdf@system{%
      \mmzvalueof{python extraction command}\space
      \mmzvalueof{python extraction options}%
    }%
    \mmz@check@extraction@log{python}%
    \def\mmz@mkdir@command##1{\mmzvalueof{python extraction command}\space --mkdir "##1"
  },
  python extraction command/.initial=memoize-extract.py,
  python extraction options/.initial={%
  -e -l "\mmzOutputDirectory\mmzUnquote\jobname.mmz.log" -w
⟨latex⟩  "\string\PackageWarning{memoize (python-based extraction)}{\string\warningtext}"
⟨plain⟩  "\string\warning{memoize (python-based extraction): \string\warningtext}"
⟨context⟩  "\string\warning{memoize (python-based extraction): \string\warningtext}"
  "\mmzOutputDirectory\mmzUnquote\jobname.mmz"
  },
}
\def\mmz@preamble@only@warning{%
  \PackageWarning{memoize}{%
    Ignoring the invocation of "\pgfkeyscurrentkey".
    This key may only be executed in the preamble}%
}
```

The extraction log — As we cannot access the exit status of a system command in TEX, we communicate with the system command via the "extraction log file," produced by both TEX-based extraction and the Perl and Python extraction script. This file signals whether the embedded extraction was successful — if it is, the file contains `\mmzExtractionSuccessful` — and also contains any size-mismatch warnings (these are currently only thrown by the TEX-based extraction). As the log is really a TEX file, the idea is to simply input it after extracting each extern (for TEX-based extraction) or after the extraction of all externs (for the external scripts).

```
\def\mmz@clear@extraction@log{%
  \begingroup
  \immediate\openout0{\mmzUnquote\jobname.mmz.log"}%
  \immediate\closeout0
  \endgroup
}
```

`#1` is the extraction method.

```
\def\mmz@check@extraction@log#1{%
  \begingroup \def\extractionmethod{#1}%
  \mmz@tempfalse \let\mmz@orig@endinput\endinput
  \def\endinput{\mmz@temptrue\mmz@orig@endinput}%
  \@input{\jobname.mmz.log}%
  \ifmmz@temp \else \mmz@extraction@error \fi \endgroup }
\def\mmz@extraction@error{%
  \PackageWarning{memoize}{Extraction of externs from document "\mmzUnquote\jobname.pdf"
```

```
    using method "\extractionmethod" was unsuccessful. Have you set the
    shell escape mode as suggested in chapter 1 of the manual?}{}}
```

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.

```
\mmzset{
  record/.style={%
    record/begin/.append style={
      /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`.

```
      /mmz/record/#1/prefix/.try/.expanded=\mmz@prefix@path,
    },
    record/prefix/.append style={/mmz/record/#1/prefix/.try={##1}},
    record/new extern/.append style={/mmz/record/#1/new extern/.try={##1}},
    record/used extern/.append style={/mmz/record/#1/used extern/.try={##1}},
    record/new cmemo/.append style={/mmz/record/#1/new cmemo/.try={##1}},
    record/new ccmemo/.append style={/mmz/record/#1/new ccmemo/.try={##1}},
    record/used cmemo/.append style={/mmz/record/#1/used cmemo/.try={##1}},
    record/used ccmemo/.append style={/mmz/record/#1/used ccmemo/.try={##1}},
    record/end/.append style={/mmz/record/#1/end/.try},
  },
}
```

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.

```
\mmzset{
  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.

```
    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.

```
    record/prefix/.code={\aftergroup\mmz@record@prefix},
    record/new extern/.code={},
    record/used extern/.code={},
    record/new cmemo/.code={},
    record/new ccmemo/.code={},
    record/used cmemo/.code={},
    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.

```
    record/end/.style={record/end/.code={}},
  }
}
```

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

```
\def\mmz@record@prefix{%
  \mmzset{/mmz/record/prefix/.expanded=\mmz@prefix@path}%
}
```

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

```
\mmzset{
  no record,
  record=mmz,
  begindocument/.append style={record/begin},
  enddocument/afterlastpage/.append style={record/end},
}
```

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.

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`).

`record/mmz/...`

```
\mmzset{
  record/mmz/begin/.code={%
    \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).

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

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.

```
  record/mmz/prefix/.code={%
    \immediate\write\mmz@mmzout{\noexpand\mmzPrefix{#1}}%
  },
  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).

```
    \immediate\write\mmz@mmzout{%
      \noexpand\mmzNewExtern{#1}{\pagenumber}{\expectedwidth}{\expectedheight}%
    }%
  },
  record/mmz/new cmemo/.code={%
    \immediate\write\mmz@mmzout{\noexpand\mmzNewCMemo{#1}}%
  },
  record/mmz/new ccmemo/.code={%
    \immediate\write\mmz@mmzout{\noexpand\mmzNewCCMemo{#1}}%
  },
  record/mmz/used extern/.code={%
    \immediate\write\mmz@mmzout{\noexpand\mmzUsedExtern{#1}}%
  },
  record/mmz/used cmemo/.code={%
    \immediate\write\mmz@mmzout{\noexpand\mmzUsedCMemo{#1}}%
  },
  record/mmz/used ccmemo/.code={%
    \immediate\write\mmz@mmzout{\noexpand\mmzUsedCCMemo{#1}}%
  },
  record/mmz/end/.code={%
```

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

```
    \immediate\write\mmz@mmzout{\noexpand\endinput}%
    \immediate\closeout\mmz@mmzout
  },
```

4.2.2 The shell scripts

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

`sh` `bat` These keys set the shell script filenames.

```
  sh/.store in=\mmz@shname,
  sh=memoize-extract.\jobname.sh,
  bat/.store in=\mmz@batname,
  bat=memoize-extract.\jobname.bat,
```

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

```
  record/sh/begin/.code={%
    \newwrite\mmz@shout
    \immediate\openout\mmz@shout{\mmz@shname}%
  },
  record/sh/new extern/.code={%
    \begingroup
```

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

```
    \immediate\write\mmz@shout{\mmz@tex@extraction@systemcall}%
    \endgroup
  },
  record/sh/end/.code={%
    \immediate\closeout\mmz@shout
  },
```

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

```
  record/bat/begin/.code={%
    \newwrite\mmz@batout
    \immediate\openout\mmz@batout{\mmz@batname}%
  },
  record/bat/new extern/.code={%
    \begingroup
    \immediate\write\mmz@batout{\mmz@tex@extraction@systemcall}%
    \endgroup
  },
  record/bat/end/.code={%
    \immediate\closeout\mmz@batout
  },
```

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.

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

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

```
\begingroup
\catcode`\^^I=12
\gdef\mmz@makefile@recipe@prefix{^^I}%
\endgroup
```

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

```
\mmzset{
  record/makefile/begin/.code={%
```

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

```
    \newwrite\mmz@makefileout
    \newtoks\mmz@makefile@externs
    \immediate\openout\mmz@makefileout{\mmz@makefilename}%
    \immediate\write\mmz@makefileout{.DEFAULT_GOAL = externs}%
    \immediate\write\mmz@makefileout{.PHONY: externs}%
  },
```

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`.

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

The target extern file:

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

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

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

Append the extern file to list of targets.

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

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

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

4.3 TEX-based extraction

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

```
\mmzset{
  extract/tex/.code={%
    \begingroup
    \@input{\jobname.mmz}%
    \endgroup
  },
}
```

\mmzUsedCMemo \mmzUsedCCMemo \mmzUsedExtern \mmzNewCMemo \mmzNewCCMemo \mmzPrefix We can ignore everything but \mmzNewExterns. All these macros receive a single argument.

```
\def\mmzUsedCMemo#1{}
\def\mmzUsedCCMemo#1{}
\def\mmzUsedExtern#1{}
\def\mmzNewCMemo#1{}
\def\mmzNewCCMemo#1{}
\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 TEX 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.

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

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

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

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

```
  \def\externbasepath{#1}%
  \def\pagenumber{#2}%
  \def\expectedwidth{#3}%
  \def\expectedheight{#4}%
```

Empty out the extraction log.

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

Extract.

```
  \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.

```
  \let\mmz@extraction@error\mmz@pageextraction@error
  \mmz@check@extraction@log{tex}%
  \endgroup
}
```

```
\def\mmz@pageextraction@error{%
  \PackageError{memoize}{Extraction of extern page \pagenumber\space from
    document "\mmzUnquote\jobname.pdf" using method "\extractionmethod" was unsuccessful
    Have you set the shell escape mode as suggested in chapter 1 of the
    manual?}{If "\mmzvalueof{tex extraction command}" was executed,
    shell escape mode is not the problem, and inspecting "\externbasepath.log"
    might give you a clue what's wrong}}
```

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

```
\def\mmz@tex@extraction@systemcall{%
  \mmzvalueof{tex extraction command}\space
  \mmzvalueof{tex extraction options}\space
  "\mmzvalueof{tex extraction script}"%
}
```

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 pdfTEX.[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`.

```
\mmzset{
  tex extraction command/.initial=pdftex,
  tex extraction options/.initial={%
    -halt-on-error
    -interaction=batchmode
    -jobname "\externbasepath"
    \ifdefempty\mmzOutputDirectory{}{-output-directory "\mmzOutputDirectory"}
  },
  tex extraction script/.initial={%
    \def\noexpand\fromdocument{"\mmzOutputDirectory"\jobname.pdf}%
    \def\noexpand\pagenumber{\pagenumber}%
    \def\noexpand\expectedwidth{\expectedwidth}%
    \def\noexpand\expectedheight{\expectedheight}%
    \def\noexpand\logfile{\jobname.mmz.log}%
    \unexpanded{%
      \def\warningtemplate{%
⟨latex⟩        \noexpand\PackageWarning{memoize}{\warningtext}%
⟨plain⟩        \warning{memoize: \warningtext}%
⟨context⟩        \warning{memoize: \warningtext}%
      }}%
    \ifdef\XeTeXversion{}{%
      \def\noexpand\mmzpdfmajorversion{\the\pdfmajorversion}%
      \def\noexpand\mmzpdfminorversion{\the\pdfminorversion}%
    }%
    \noexpand\input memoize-extract-one
  },
}
⟨/mmz⟩
```

[3] I implemented the first version of TEX-based extraction using LATEX and package **graphicx**, and it was (running with pdfTEX engine) almost four times slower than the current plain TEX implementation.

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 pdfTeX engine by default, but we nevertheless take care that it will work with other (supported) engines as well.

```
⟨*extract-one⟩
\catcode`\@11\relax
\def\@firstoftwo#1#2{#1}
\def\@secondoftwo#1#2{#2}
```

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

```
\ifdefined\XeTeXversion
\else
  \ifdefined\luatexversion
    \def\pdfmajorversion{\pdfvariable majorversion}%
    \def\pdfminorversion{\pdfvariable minorversion}%
  \fi
  \ifdefined\mmzpdfmajorversion
    \pdfmajorversion\mmzpdfmajorversion\relax
  \fi
  \ifdefined\mmzpdfminorversion
    \pdfminorversion\mmzpdfminorversion\relax
  \fi
\fi
```

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

```
\newwrite\extractionlog
```

Are we requested to produce a log file?

```
\ifdefined\logfile
  \immediate\openout\extractionlog{\logfile}%
```

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

```
  \def\doublewarning#1{%
    \message{#1}%
    \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.

```
    \immediate\write\extractionlog{%
      \ifdefined\warningtemplate\warningtemplate\else\warningtext\fi
    }%
  }%
\else
  \let\doublewarning\message
\fi
\newif\ifforce
\ifdefined\force
  \csname force\force\endcsname
\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 XƎTEX.)

```
⟨/extract-one⟩
⟨*mmz, extract-one⟩
\def\mmz@tolerance{0.01pt}
\def\mmz@if@roughly@equal#1#2#3{%
  \dimen0=\dimexpr#2-#3\relax
  \ifdim\dimen0<0pt
    \dimen0=-\dimen0\relax
  \fi
  \ifdim\dimen0>#1\relax
    \expandafter\@secondoftwo
  \else
```

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

```
    \expandafter\@firstoftwo
  \fi
}%
⟨/mmz, extract-one⟩
⟨*extract-one⟩
```

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

```
\ifdefined\XeTeXversion
  \setbox0=\hbox{\XeTeXpdffile \fromdocument\space page \pagenumber media}%
\else
  \ifdefined\luatexversion
    \saveimageresource page \pagenumber mediabox {\fromdocument}%
    \setbox0=\hbox{\useimageresource\lastsavedimageresourceindex}%
  \else
    \pdfximage page \pagenumber mediabox {\fromdocument}%
    \setbox0=\hbox{\pdfrefximage\pdflastximage}%
  \fi
\fi
```

Check whether the extern page is of the expected size.

```
\newif\ifbaddimensions
\ifdefined\expectedwidth
  \ifdefined\expectedheight
    \mmz@if@roughly@equal{\mmz@tolerance}{\wd0}{\expectedwidth}{%
      \mmz@if@roughly@equal{\mmz@tolerance}{\ht0}{\expectedheight}%
        {}%
        {\baddimensionstrue}%
    }{\baddimensionstrue}%
  \fi
\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.

```
\ifdefined\luatexversion
  \let\pdfpagewidth\pagewidth
  \let\pdfpageheight\pageheight
  \def\pdfhorigin{\pdfvariable horigin}%
  \def\pdfvorigin{\pdfvariable vorigin}%
\fi
\def\do@shipout{%
  \pdfpagewidth=\wd0
  \pdfpageheight=\ht0
  \ifdefined\XeTeXversion
```

```
    \hoffset -1 true in
    \voffset -1 true in
  \else
    \pdfhorigin=0pt
    \pdfvorigin=0pt
  \fi
  \shipout\box0
}
\ifbaddimensions
  \doublewarning{I refuse to extract page \pagenumber\space from
    "\fromdocument", because its size (\the\wd0 \space x \the\ht0) is not
    what I expected (\expectedwidth\space x \expectedheight)}%
  \ifforce\do@shipout\fi
\else
  \do@shipout
\fi
```

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

```
\ifdefined\logfile
  \immediate\write\extractionlog{\noexpand\endinput}%
  \immediate\closeout\extractionlog
\fi
\bye
⟨/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.

```
⟨*mmz⟩
\mmzset{
  .install advice={setup key=auto, activation=deferred},
```

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

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

`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 LaTeX, we will use the latest possible hook, `begindocument/end`, as we want to hack into commands defined by other packages. (The TeX conditional needs to be defined before using it in `.append code` below.

```
\newif\ifmmz@manual
\mmzset{
  manual/.is if=mmz@manual,
  begindocument/end/.append code={%
    \ifmmz@manual
    \else
      \pgfkeysalso{activate deferred,activate deferred/.code={}}%
    \fi
  },
```

Announce Memoize's run conditions and handlers.

```
  auto/.cd,
  run if memoization is possible/.style={
    run conditions=\mmz@auto@rc@if@memoization@possible
  },
  run if memoizing/.style={run conditions=\mmz@auto@rc@if@memoizing},
  apply options/.style={
    bailout handler=\mmz@auto@bailout,
    outer handler=\mmz@auto@outer,
  },
  memoize/.style={
    run if memoization is possible,
    apply options,
    inner handler=\mmz@auto@memoize
  },
  ⟨*latex⟩
  noop/.style={run if memoization is possible, noop \AdviceType},
  noop command/.style={apply options, inner handler=\mmz@auto@noop},
  noop environment/.style={
    outer handler=\mmz@auto@noop@env, bailout handler=\mmz@auto@bailout},
  ⟨/latex⟩
⟨plain, context⟩  noop/.style={inner handler=\mmz@auto@noop},
  nomemoize/.style={noop, options=disable},
  replicate/.style={run if memoizing, inner handler=\mmz@auto@replicate},
}
```

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.)

```
\mmzset{
  auto/abort/.style={run conditions=\mmzAbort},
}
```

And the same for `unmemoizable`:

```
\mmzset{
  auto/unmemoizable/.style={run conditions=\mmzUnmemoizable},
}
```

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

```
\ifdef\luatexversion{%
  \mmzset{auto=\savepos{abort}}
}{%
  \mmzset{
    auto=\pdfsavepos{abort},
    auto=\errmessage{abort},
  }
}
```

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

```
\def\mmz@auto@rc@if@memoization@possible{%
  \ifmemoize
    \ifinmemoize
    \else
      \AdviceRuntrue
    \fi
  \fi
}
```

run if memoizing
\mmz@auto@rc@if@memoizing

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

```
\def\mmz@auto@rc@if@memoizing{%
  \ifmemoizing\AdviceRuntrue\fi
}
```

\mmznext

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 `\mmznext`, being a user command, must also be defined in package `nomemoize`.

```
⟨/mmz⟩
⟨nommz⟩\def\mmznext#1{\ignorespaces}
⟨*mmz⟩
\def\mmznext#1{\gdef\mmz@next{#1}\ignorespaces}
\mmznext{}%
```

apply options
\mmz@auto@outer
\mmz@auto@bailout

The outer and the bailout handler defined here work as a team. The outer handler's job is to apply the auto- and the next-options; therefore, the bailout handler must consume the next-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`.

```
\def\mmz@auto@outer{%
  \begingroup
  \mmzAutoInit
  \AdviceCollector
}
\def\mmz@auto@bailout{%
  \mmznext{}%
}
```

\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.

```
\def\mmzAutoInit{%
  \ifdefempty\AdviceOptions{}{\expandafter\mmzset\expandafter{\AdviceOptions}}%
  \ifdefempty\mmz@next{}{\expandafter\mmzset\expandafter{\mmz@next}\mmznext{}}%
  \eappto\AdviceRawCollectorOptions{\expandonce\mmzRawCollectorOptions}%
}
```

memoize
\mmz@auto@memoize

This key installs the inner handler for memoization. If you compare this handler to the definition 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.)

```
\long\def\mmz@auto@memoize#1{%
  \expanded{%
    \noexpand\Memoize
      {\expandonce\AdviceReplaced\unexpanded{#1}}%
      {\expandonce\AdviceOriginal\unexpanded{#1}}%
    \ifmmz@ignorespaces\ignorespaces\fi
  }%
}
```

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

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

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.

```
  ⟨*latex⟩
\def\mmz@auto@noop@env{%
  \AddToHookNext{env/\AdviceName/begin}{%
    \mmzAutoInit
    \ifmmz@ignorespaces\ignorespacesafterend\fi
  }%
  \AdviceOriginal
}
  ⟨/latex⟩
```

`replicate` `\mmz@auto@replicate` This inner handler writes a copy of the handled command or environment's invocation into 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.

```
\def\mmz@auto@replicate#1{%
  \begingroup
  \let\mmz@auto@replicate@expansion\unexpanded
  \expandafter\pgfqkeys\expanded{{/mmz/auto/replicate}{\AdviceOptions}}%
⟨latex⟩  \let\protect\noexpand
  \expanded{%
    \endgroup
    \noexpand\gtoksapp\noexpand\mmzCCMemo{%
      \expandonce\AdviceReplaced\mmz@auto@replicate@expansion{#1}}%
    \expandonce\AdviceOriginal\unexpanded{#1}%
  }%
}
\pgfqkeys{/mmz/auto/replicate}{
  expanded/.code={\let\mmz@auto@replicate@expansion\@firstofone},
}
```

5.1 LaTeX-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.

```
  ⟨*latex⟩
\mmzset{
  auto/.cd,
  ref/.style={outer handler=\mmz@auto@ref\mmzNoRef, run if memoizing},
  force ref/.style={outer handler=\mmz@auto@ref\mmzForceNoRef, run if memoizing},
}
\mmzset{
  auto=\ref{ref},
  auto=\pageref{ref},
  auto=\label{run if memoizing, outer handler=\mmz@auto@label},
  auto=\index{replicate, args=m, expanded},
}
```

`ref`
`force ref`
`\mmz@auto@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 — 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.

```
\def\mmz@auto@ref#1#2#{\mmz@auto@ref@i#1{#2}}
\def\mmz@auto@ref@i#1#2#3{%
  #1{#3}%
  \AdviceOriginal#2{#3}%
}
```

`\mmzForceNoRef`
`\mmzNoRef`

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.

```
\def\mmzForceNoRef#1{%
  \ifmemoizing
    \expandafter\gtoksapp\expandafter\mmzContextExtra
  \else
    \expandafter\toksapp\expandafter\mmzContext
  \fi
  {r@#1={\csname r@#1\endcsname}}%
  \ignorespaces
}
\def\mmzNoRef#1{%
  \ifcsundef{r@#1}{\mmzAbort}{\mmzForceNoRef{#1}}%
  \ignorespaces
}
```

refrange
force refrange
\mmz@auto@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.

```
\mmzset{
  auto/.cd,
  refrange/.style={outer handler=\mmz@auto@refrange\mmzNoRef,
    bailout handler=\relax, run if memoizing},
  force refrange/.style={outer handler=\mmz@auto@refrange\mmzForceNoRef,
    bailout handler=\relax, run if memoizing},
}
```

```
\def\mmz@auto@refrange#1#2#{\mmz@auto@refrange@i#1{#2}}
\def\mmz@auto@refrange@i#1#2#3#4{%
  #1{#3}%
  #1{#4}%
  \AdviceOriginal#2{#3}{#4}%
}
```

multiref
force multiref
\mmz@auto@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, this time distributing #1 (\mmzNoRef or \mmzForceNoRef) over #3 by \forcsvlist.

```
\mmzset{
  auto/.cd,
  multiref/.style={outer handler=\mmz@auto@multiref\mmzNoRef,
    bailout handler=\relax, run if memoizing},
  force multiref/.style={outer handler=\mmz@auto@multiref\mmzForceNoRef,
    bailout handler=\relax, run if memoizing},
}
\def\mmz@auto@multiref#1#2#{\mmz@auto@multiref@i#1{#2}}
\def\mmz@auto@multiref@i#1#2#3{%
  \forcsvlist{#1}{#3}%
  \AdviceOriginal#2{#3}%
}
```

\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.

```
\def\mmz@auto@label#1{%
  \xtoksapp\mmzCCMemo{%
    \noexpand\mmzLabel{#1}{\expandonce\@currentlabel}%
  }%
  \AdviceOriginal{#1}%
}
```

\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.

```
\def\mmzLabel#1#2{%
  \begingroup
  \def\@currentlabel{#2}%
  \label{#1}%
  \endgroup
}
  ⟨/latex⟩
⟨/mmz⟩
```

6 Support for various classes and packages

6.1 Ti*k*Z

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

```
⟨*mmz⟩
\mmzset{
  begindocument/before/.append code={%
⟨latex⟩    \@ifpackageloaded{tikz}{%
⟨plain, context⟩    \ifdefined\tikz
      \input advice-tikz.code.tex
⟨latex⟩    }{}%
⟨plain, context⟩    \fi
```

We define and activate the automemoization handlers for the Ti*k*Z command and environment.

```
    \mmzset{%
/utils/exec=,
      auto=\tikz{memoize, collector=\AdviceCollectTikZArguments},
      auto={tikzpicture}{memoize},
```

A hack to prevent memoizing pictures which are accidentally marked as remembered — accidentally in the sense that because the document changed, the `.aux` file contains a `\pgfsyspdfmark` command which erroneously refers to the picture being memoized. We know that memoizing a remembered picture can't be right, as we always abort on `\pdfsavepos`. This is implemented by hacking into PGF's `\pgfsys@getposition`, and aborting memoization if the mark does not equal `\relax`. (We have to duplicate `#` because of `.append code`.)

```
      auto=\pgfsys@getposition{
        run if memoizing, outer handler=\mmz@pgfsys@getposition},
    }%
    \def\mmz@pgfsys@getposition##1{%
      \expandafter\ifx\csname pgf@sys@pdf@mark@pos@##1\endcsname\relax
      \else
        \mmzAbort
      \fi
      \AdviceOriginal{##1}%
    }%
  },
}
⟨/mmz⟩
```

6.2 Forest

Forest will soon feature extensive memoization support, but for now, let's just enable the basic, single extern externalization.

```
⟨*mmz⟩
  ⟨*latex⟩
\mmzset{
  begindocument/before/.append code={%
    \@ifpackageloaded{forest}{%
      \mmzset{
        auto={forest}{memoize},
```

Yes, `\Forest` is defined using `xparse`.

```
        auto=\Forest{memoize},
      }%
    }{}%
```

```
  },
}
  ⟨/latex⟩
```

6.3 Beamer

The Beamer code is explained in M§4.2.4.

```
  ⟨*latex⟩
\AddToHook{begindocument/before}{\@ifclassloaded{beamer}{%
  \mmzset{per overlay/.style={
    /mmz/context={%
      overlay=\csname beamer@overlaynumber\endcsname,
      pauses=\ifmemoizing
               \mmzBeamerPauses
             \else
               \expandafter\the\csname c@beamerpauses\endcsname
             \fi
    },
    /mmz/at begin memoization={%
      \xdef\mmzBeamerPauses{\the\c@beamerpauses}%
      \xtoksapp\mmzCMemo{%
        \noexpand\mmzSetBeamerOverlays{\mmzBeamerPauses}{\beamer@overlaynumber}}%
      \gtoksapp\mmzCCMemo{%
        \only<\mmzBeamerOverlays>{}}%
    },
    /mmz/at end memoization={%
      \xtoksapp\mmzCCMemo{%
        \noexpand\setcounter{beamerpauses}{\the\c@beamerpauses}}%
    },
    /mmz/per overlay/.code={},
  }}
  \def\mmzSetBeamerOverlays#1#2{%
    \ifnum\c@beamerpauses=#1\relax
      \gdef\mmzBeamerOverlays{#2}%
      \ifnum\beamer@overlaynumber<#2\relax \mmz@temptrue \else \mmz@tempfalse \fi
    \else
      \mmz@temptrue
    \fi
    \ifmmz@temp
      \appto\mmzAtBeginMemoization{%
        \gtoksapp\mmzCMemo{\mmzSetBeamerOverlays{#1}{#2}}}%
    \fi
  }%
}{}}
  ⟨/latex⟩
```

7 Initialization

`begindocument/before` `begindocument` `begindocument/end` `enddocument/afterlastpage` 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:

```
  ⟨*latex⟩
\AddToHook{begindocument/before}{\mmzset{begindocument/before}}
\AddToHook{begindocument}{\mmzset{begindocument}}
\AddToHook{begindocument/end}{\mmzset{begindocument/end}}
```

```
\AddToHook{enddocument/afterlastpage}{\mmzset{enddocument/afterlastpage}}
  ⟨/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!

```
  ⟨*plain, context⟩
\mmzset{
  begin document/.style={begindocument/before, begindocument, begindocument/end},
  end document/.style={enddocument/afterlastpage},
}
  ⟨/plain, context⟩
```

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).

```
⟨/mmz⟩
⟨mmz, nommz⟩\InputIfFileExists{memoize.cfg}{}{}
⟨*mmz⟩
```

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

```
\mmzset{
⟨!plain⟩  extract/no/.code={},
```

output directory
mmzOutputDirectory Set the `-output-directory` — manually, as there is no other way.

```
  output-directory/.store in=\mmzOutputDirectory,
}
```

Process the package options (except in plain TEX).

```
⟨latex⟩\ProcessPgfPackageOptions{/mmz}
⟨context⟩\expandafter\mmzset\expandafter{\currentmoduleparameters}
```

Define `\mmzOutputDirectory` if `output-directory` was not given.

```
\ifdefined\mmzOutputDirectory
\else
  \def\mmzOutputDirectory{}%
\fi
\mmzset{output directory/.code={\PackageError{memoize}{Key "output-directory"
      may only be used as a package option}{}}}
\mmzset{
```

Extract the externs using the method given by `memoize.cfg` or the package options — unless we're running plain TEX.

```
  ⟨*!plain⟩
  extract/\mmz@extraction@method,
```

In non-plain TEX formats, also disable `extract` in the preamble.

```
  extract/.code={\PackageError{memoize}{Key "extract" is only allowed as a
      package option.}{If you really want to extract in the preamble, execute
      "extract/<method>".}},
  ⟨/!plain⟩
  ⟨*plain⟩
```

In plain TEX, where extraction must be invoked after loading the package, we now have to redefine `extract`, so that it will immediately trigger extraction.

```
  extract/.is choice,
  extract/.default=\mmz@extraction@method,
```

But only once:

```
  extract/.append style={
    extract/.code={\PackageError{memoize}{Key "extract" is only allowed to
        be used once.}{If you really want to extract again, execute
        "extract/<method>".}},
  },
  ⟨/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 is still perfectly valid in this mode, so let's just warn the user.

```
\ifnum\pdf@draftmode=1
  \PackageWarning{memoize}{No memoization will be performed in the draft mode}%
\fi
⟨/mmz⟩
```

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

```
⟨*mmz, nommz, mmzable⟩
⟨plain⟩\resetatcatcode
⟨context⟩\stopmodule
⟨context⟩\protect
⟨/mmz, nommz, mmzable⟩
```

That's all, folks!

8 Auxiliary packages

8.1 Extending commands and environments with Advice

```
⟨*main⟩
⟨latex⟩\ProvidesPackage{advice}[2023/10/10 v1.0.0 Extend commands and environments]
⟨context⟩%D \module[
⟨context⟩%D         file=t-advice.tex,
⟨context⟩%D      version=1.0.1,
⟨context⟩%D        title=Advice,
⟨context⟩%D     subtitle=Extend commands and environments,
⟨context⟩%D       author=Saso Zivanovic,
⟨context⟩%D         date=2023-09-10,
⟨context⟩%D    copyright=Saso Zivanovic,
⟨context⟩%D      license=LPPL,
⟨context⟩%D ]
⟨context⟩\writestatus{loading}{ConTeXt User Module / advice}
⟨context⟩\unprotect
⟨context⟩\startmodule[advice]
```

Required packages

```
⟨latex⟩\RequirePackage{collargs}
⟨plain⟩\input collargs
⟨context⟩\input t-collargs
```

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.

```
\pgfkeys{
  /handlers/.install advice/.code={%
    \edef\auto@install@namespace{\pgfkeyscurrentpath}%
    \def\advice@install@setupkey{advice}%
    \def\advice@install@activation{immediate}%
    \pgfqkeys{/advice/install}{#1}%
    \expanded{\noexpand\advice@install
      {\auto@install@namespace}%
      {\advice@install@setupkey}%
      {\advice@install@activation}%
    }%
  },
```

setup key
activation

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

```
  /advice/install/.cd,
  setup key/.store in=\advice@install@setupkey,
  activation/.is choice,
  activation/.append code=\def\advice@install@activation{#1},
  activation/immediate/.code={},
  activation/deferred/.code={},
}
```

`#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.

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

Switch to the installation keypath.

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

auto
auto csname
auto key
auto'
auto csname'
auto key'

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

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

Call the internal setup macro, wrapping the received keylist into a `pgfkeys` invocation.

```
      \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.)

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

A variant without activation.

```
    #2'/.code 2 args={%
      \AdviceSetup{#1}{#2}{##1}{\pgfqkeys{#1/#2}{##2}}%
    },
    #2 csname/.style 2 args={
      #2/.expand once=\expandafter{\csname ##1\endcsname}{##2},
    },
    #2 csname'/.style 2 args={
      #2'/.expand once=\expandafter{\csname ##1\endcsname}{##2},
    },
    #2 key/.style 2 args={
      #2/.expand once=%
        \expandafter{\csname pgfk@##1/.@cmd\endcsname}%
        {collector=\advice@pgfkeys@collector,##2},
    },
    #2 key'/.style 2 args={
      #2'/.expand once=%
        \expandafter{\csname pgfk@##1/.@cmd\endcsname}%
        {collector=\advice@pgfkeys@collector,##2},
    },
```

`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`).

```
activation/.style={/advice/activation/##1={#1}},
activation=#3,
```

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

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

`activate csname` `deactivate 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).

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

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

```
activate key/.style={activate@key={#1/activate}{##1}},
deactivate key/.style={activate@key={#1/deactivate}{##1}},
activate@key/.code n args=2{%
  \def\advice@temp{}%
  \def\advice@do####1{%
    \eappto\advice@temp{,\expandonce{\csname pgfk@####1/.@cmd\endcsname}}}%
  \forcsvlist\advice@do{##2}%
  \pgfkeysalso{##1/.expand once=\advice@temp}%
},
```

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

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

`run conditions` `outer handler` `bailout handler` `collector` `args` `collector options` `clear collector options` `raw collector options` `clear raw collector options` `inner handler` `options` `clear options` These keys are used to setup the handling of the command or environment. The storage macros (`\AdviceRunConditions` etc.) have public names as they also play a crucial role in the handler definitions, see section 8.1.3.

```
run conditions/.store in=\AdviceRunConditions,
bailout handler/.store in=\AdviceBailoutHandler,
outer handler/.store in=\AdviceOuterHandler,
collector/.store in=\AdviceCollector,
collector options/.code={\appto\AdviceCollectorOptions{,##1}},
clear collector options/.code={\def\AdviceCollectorOptions{}},
raw collector options/.code={\appto\AdviceRawCollectorOptions{##1}},
clear raw collector options/.code={\def\AdviceRawCollectorOptions{}},
args/.store in=\AdviceArgs,
inner handler/.store in=\AdviceInnerHandler,
options/.code={\appto\AdviceOptions{,##1}},
clear options/.code={\def\AdviceOptions{}},
```

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

```
.unknown/.code={%
  \eappto{\AdviceOptions}{,\pgfkeyscurrentname={\unexpanded{##1}}}%
},
```

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

```
    run conditions/.default=\AdviceRuntrue,
    bailout handler/.default=\relax,
    outer handler/.default=\advice@default@outer@handler,
    collector/.default=\advice@CollectArgumentsRaw,
    collector options/.value required,
    raw collector options/.value required,
    args/.default=\advice@noargs,
    inner handler/.default=\advice@error@noinnerhandler,
    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.

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

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

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

In LaTeX, 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 deffered.

```
⟨latex⟩    #1/#2=\begin{run conditions=\advice@begin@rc},
  }%
}
```

8.1.2 Submitting a command or environment

`\AdviceSetup` `\AdviceName` `\AdviceType` Macro `\advice@setup` is called by key `auto` to submit a command or environment to advising. 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.)

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

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

```
  \begingroup
  \def\AdviceName{#3}%
```

Command, complain, or environment?

```
  \collargs@cs@cases{#3}{%
    \def\AdviceType{command}%
    \advice@setup@init@command
    \advice@setup@i{#3}{#1}{#3}%
  }{%
```

```
    \advice@error@advice@notcs{#1/#2}{#3}%
  }{%
    \def\AdviceType{environment}%
    \advice@setup@init@environment
⟨latex⟩    \advice@setup@i{#3}%
⟨plain⟩    \expandafter\advice@setup@i\expandafter{\csname #3\endcsname}%
⟨context⟩    \expandafter\advice@setup@i\expandafter{\csname start#3\endcsname}%
      {#1}{#3}%
  }%
}
```

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 `\expandafter`s 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 (`\string`ified) 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 (`\string`ified) 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.)

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

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

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

Apply the setup code/keys.

```
  #4%
```

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

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

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.

```
\def\advice@setup@init@common{%
  \def\AdviceRunConditions{\AdviceRuntrue}%
  \def\AdviceBailoutHandler{\relax}%
  \def\AdviceOuterHandler{\advice@default@outer@handler}%
  \def\AdviceCollector{\advice@CollectArgumentsRaw}%
  \def\AdviceCollectorOptions{}%
  \def\AdviceInnerHandler{\advice@error@noinnerhandler}%
  \def\AdviceOptions{}%
}
\def\advice@setup@init@command{%
  \advice@setup@init@common
  \def\AdviceRawCollectorOptions{}%
  \def\AdviceArgs{\advice@noargs}%
}
\def\advice@setup@init@environment{%
  \advice@setup@init@common
  \edef\AdviceRawCollectorOptions{%
    \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.

```
    \noexpand\collargsEndTagtrue
  }%
  \def\AdviceArgs{+b}%
}
```

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

```
\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` `\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).

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

```
\let\advice@setup@init@i\advice@init@i
\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.

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

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

```
    \endgroup
    \noexpand\csdef{advice@i#1//\string#2}{%
      \def\noexpand\AdviceRunConditions{\expandonce\AdviceRunConditions}%
      \def\noexpand\AdviceBailoutHandler{\expandonce\AdviceBailoutHandler}%
    }%
    \noexpand\csdef{advice@I#1//\string#2}{%
      \def\noexpand\AdviceOuterHandler{\expandonce\AdviceOuterHandler}%
      \def\noexpand\AdviceCollector{\expandonce\AdviceCollector}%
      \def\noexpand\AdviceRawCollectorOptions{\expandonce\AdviceRawCollectorOptions}%
      \def\noexpand\AdviceCollectorOptions{\expandonce\AdviceCollectorOptions}%
      \def\noexpand\AdviceArgs{\expandonce\AdviceArgs}%
      \def\noexpand\AdviceInnerHandler{\expandonce\AdviceInnerHandler}%
      \def\noexpand\AdviceOptions{\expandonce\AdviceOptions}%
    }%
    \expandonce{\AdviceAfterSetup}%
  }%
}
```

`activation/immediate`
`activation/deferred`

The two subkeys of `/advice/activation` install the immediate and the deferred activation code into the installation keypath. They are invoked by ⟨*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.)

```
\pgfkeys{
  /advice/activation/deferred/.style={
    #1/activate/.style={%
      activate deferred/.append style={#1/activate={##1}}},
    #1/deactivate/.style={%
      activate deferred/.append style={#1/deactivate={##1}}},
    #1/force activate/.style={%
      activate deferred/.append style={#1/force activate={##1}}},
    #1/try activate/.style={%
      activate deferred/.append style={#1/try activate={##1}}},
  },
```

`activate`
`deactivate`
`force activate`
`try 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. 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.)

```
  /advice/activation/immediate/.style={
    #1/activate/.code={%
      \forcsvlist{\advice@activate{#1}}{##1}%
      \advice@activate@forcefalse
      \advice@activate@tryfalse
    },
    #1/deactivate/.code={%
      \forcsvlist{\advice@deactivate{#1}}{##1}%
      \advice@activate@forcefalse
      \advice@activate@tryfalse
    },
    #1/force activate/.is if=advice@activate@force,
    #1/try activate/.is if=advice@activate@try,
  },
}
\newif\ifadvice@activate@force
\newif\ifadvice@activate@try
```

`\advice@original@csname`
`\advice@original@cs`
`\AdviceGetOriginal`

Activation replaces the original meaning of the handled command with our definition. We store the original definition into control sequence `\advice@o`⟨*namespace*⟩`//`⟨*storage name*⟩ (with a `\string`ified ⟨*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.

```
\def\advice@original@csname#1#2{advice@o#1//\string#2}
\def\advice@original@cs#1#2{\csname advice@o#1//\string#2\endcsname}
\def\AdviceGetOriginal#1#2{%
  \ifcsname advice@o#1//\string#2\endcsname
    \csname advice@o#1//\string#2\expandafter\endcsname
  \else
    \expandafter#2%
  \fi
}
```

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

```
\def\advice@activate#1#2{%
  \collargs@cs@cases{#2}%
    {\advice@activate@cmd{#1}{#2}}%
    {\advice@error@activate@notcsorenv{}{#1}}%
    {\advice@activate@env{#1}{#2}}%
}
\def\advice@deactivate#1#2{%
  \collargs@cs@cases{#2}%
    {\advice@deactivate@cmd{#1}{#2}}%
    {\advice@error@activate@notcsorenv{de}{#1}}%
    {\advice@deactivate@env{#1}{#2}}%
}
```

`\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.)

```
\def\advice@activate@cmd#1#2{%
```

Is the command defined?

```
  \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).

```
    \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?

```
      \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.

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

We don't recognize the current definition as our own code (despite the fact that we have surely activated the commmand 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.

```
        \ifadvice@activate@force
          \advice@activate@cmd@do{#1}{#2}%
        \else
          \advice@activate@error@activated{#1}{#2}{Command}{already}%
        \fi
      }%
    }{%
```

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

```
      \advice@activate@cmd@do{#1}{#2}%
    }%
  }{%
    \advice@activate@error@undefined{#1}{#2}{Command}{}%
  }%
}
```

`\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).

```
\def\advice@deactivate@cmd#1#2{%
  \ifdef{#2}{%
    \ifcsdef{\advice@original@csname{#1}{#2}}{%
      \advice@if@our@definition{#1}{#2}{%
        \advice@deactivate@cmd@do{#1}{#2}%
      }{%
        \advice@deactivate@error@changed{#1}{#2}%
      }%
    }{%
      \advice@activate@error@activated{#1}{#2}{Command}{not yet}%
    }%
  }{%
    \advice@activate@error@undefined{#1}{#2}{Command}{de}%
  }%
}
```

`\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).

```
\def\advice@if@our@definition#1#2{%
  \protected\def\advice@temp{\advice@handle{#1}{#2}}%
  \ifx#2\advice@temp
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
}
```

\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 LaTeX's robust commands by redefining their "inner" command. Protecting our replacement suffices.)

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

\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.

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

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 LaTeX'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 not 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`.

```
\def\advice@handle#1#2{%
  \advice@init@i{#1}{#2}%
  \AdviceRunfalse
  \AdviceRunConditions
  \advice@handle@rc{#1}{#2}%
}
```

\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.

```
\def\advice@handle@rc#1#2{%
  \ifAdviceRun
    \expandafter\advice@handle@outer
  \else
```

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

```
    \AdviceBailoutHandler
    \expandafter\advice@original@cs
```

```
  \fi
  {#1}{#2}%
}
```

\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 LaTeX, \foo in plain TeX and \startfoo in ConTeXt.
- \AdviceOriginal executes the original definition of the handled command or environment.

```
\def\advice@handle@outer#1#2{%
  \def\AdviceNamespace{#1}%
  \def\AdviceName{#2}%
  \let\AdviceReplaced\AdviceName
  \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.

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

All prepared, we execute the outer handler.

```
  \AdviceOuterHandler
}
```

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

```
\newif\ifAdviceRun
```

\advice@default@outer@handler The default outer handler merely executes the argument collector. Note that it works for both commands and environments.

```
\def\advice@default@outer@handler{%
  \AdviceCollector
}
```

\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`.

```
\def\advice@CollectArgumentsRaw{%
  \AdviceIfArgs{}{%
    \expandafter\GetDocumentCommandArgSpec\expandafter{\AdviceName}%
    \let\AdviceArgs\ArgumentSpecification
  }%
  \expanded{%
    \noexpand\CollectArgumentsRaw{%
      \noexpand\collargsCaller{\expandonce\AdviceName}%
      \expandonce\AdviceRawCollectorOptions
      \ifdefempty\AdviceCollectorOptions{}{%
        \noexpand\collargsSet{\expandonce\AdviceCollectorOptions}%
      }%
    }%
    {\expandonce\AdviceArgs}%
    {\expandonce\AdviceInnerHandler}%
  }%
}
```

`\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".)

```
\def\advice@noargs@text{\advice@noargs}
\def\AdviceIfArgs{%
  \ifx\AdviceArgs\advice@noargs@text
    \expandafter\@secondoftwo
  \else
    \expandafter\@firstoftwo
  \fi
}
```

`\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`.

```
\def\advice@pgfkeys@collector#1\pgfeov{%
  \AdviceInnerHandler{#1}%
}
```

8.1.4 Environments

`\advice@activate@env` `\advice@deactivate@env` Things are simple in TEX and ConTEXt, as their environments are really commands. So rather than activating environment name `#2`, we (de)activate command `\#2` or `\start#2`, depending on the format.

```
  ⟨*plain, context⟩
\def\advice@activate@env#1#2{%
  \expanded{%
    \noexpand\advice@activate@cmd{#1}{\expandonce{\csname
⟨context⟩      start%
      #2\endcsname}}%
  }%
}
\def\advice@deactivate@env#1#2{%
  \expanded{%
    \noexpand\advice@deactivate@cmd{#1}{\expandonce{\csname
⟨context⟩         start%
         #2\endcsname}}%
  }%
}
  ⟨/plain, context⟩
```

We activate commands by redefining them; that's the only way to do it. But we won't activate a LaTeX 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 LaTeX environments by defining an outer handler for `\begin` (consequently, LaTeX 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.

```
  ⟨*latex⟩
\def\advice@activate@env#1#2{%
  \ifcsdef{\advice@original@csname{#1}{#2}}{%
    \advice@activate@error@activated{#1}{#2}{Environment}{already}%
  }{%
    \csdef{\advice@original@csname{#1}{#2}}{}%
  }%
}
\def\advice@deactivate@env#1#2{%
  \ifcsdef{\advice@original@csname{#1}{#2}}{%
    \csundef{\advice@original@csname{#1}{#2}}{}%
  }{%
    \advice@activate@error@activated{#1}{#2}{Environment}{not yet}%
  }%
}
```

`\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`).

```
\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.

```
  \ifcsname\advice@original@csname{#2}{#4}\endcsname
    \expandafter\advice@begin@env@rc
  \else
    \expandafter\advice@original@cs
  \fi
  {#2}{#3}{#4}%
}
```

`\advice@begin@env@rc` Starting from this point, we essentially replicate the workings of `\advice@handle`, adapted to LaTeX environments.

```
\def\advice@begin@env@rc#1#2#3{%
```

We first load the stage-one configuration for environment `#3` in namespace `#1`.

```
  \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`).

```
  \AdviceRunConditions
  \ifAdviceRun
    \expandafter\advice@begin@env@outer
  \else
    \AdviceBailoutHandler
    \expandafter\advice@original@cs
  \fi
  {#1}{#2}{#3}%
}
```

`\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.

```
\def\advice@begin@env@outer#1#2#3{%
  \def\AdviceNamespace{#1}%
  \def\AdviceName{#3}%
  \def\AdviceReplaced{#2{#3}}%
  \def\AdviceOriginal{\AdviceGetOriginal{#1}{#2}{#3}}%
  \advice@init@I{#1}{#3}%
  \AdviceOuterHandler
}
  ⟨/latex⟩
```

8.1.5 Error messages

Define error messages for the entire package. Note that `\advice@(de)activate@error@...` implement `try activate`.

```
\def\advice@activate@error@activated#1#2#3#4{%
  \ifadvice@activate@try
  \else
    \PackageError{advice (#1)}{#3 "\string#2" is #4 activated}{}%
  \fi
}
\def\advice@activate@error@undefined#1#2#3#4{%
  \ifadvice@activate@try
  \else
    \PackageError{advice (#1)}{%
      #3 "\string#2" you are trying to #4activate is not defined}{}%
  \fi
}
\def\advice@deactivate@error@changed#1#2{%
  \ifadvice@activate@try
  \else
    \PackageError{advice (#1)}{The definition of "\string#2" has changed since we
      have activated it. Has somebody overridden our command?}{If you have tried
      to deactivate so that you could immediately reactivate, you may want to try
      "force activate".}%
  \fi
}
\def\advice@error@advice@notcs#1#2{%
  \PackageError{advice}{The first argument of key "#1" should be either a single
    control sequence or an environment name, not "#2"}{}%
}
\def\advice@error@activate@notcsorenv#1#2{%
  \PackageError{advice}{Each item in the value of key "#1activate" should be
    either a control sequence or an environment name, not "#2".}{}%
}
\def\advice@error@storecs@notcs#1#2{%
```

```
  \PackageError{advice}{The value of key "#1" should be a single control sequence,
    not "\string#2"}{}%
}
\def\advice@error@noinnerhandler#1{%
  \PackageError{advice (\AdviceNamespace)}{The inner handler for
    "\expandafter\string\AdviceName" is not defined}{}%
}
```

8.1.6 Tracing

\AdviceTracingOn \AdviceTracingOff 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.

```
\def\AdviceTracingOn{%
  \let\advice@init@i\advice@trace@init@i
  \let\advice@init@I\advice@trace@init@I
}
\def\AdviceTracingOff{%
  \let\advice@init@i\advice@setup@init@i
  \let\advice@init@I\advice@setup@init@I
}
```

```
\def\advice@trace#1{\immediate\write16{[tracing advice] #1}}
\def\advice@trace@init@i#1#2{%
  \advice@trace{Advising \detokenize\expandafter{\string#2} (\detokenize{#1})}%
  \advice@setup@init@i{#1}{#2}%
  \edef\AdviceRunConditions{%
```

We first execute the original run conditions, so that we can show the result.

```
    \expandonce\AdviceRunConditions
    \noexpand\advice@trace{\space\space
      Executing run conditions:
      \detokenize\expandafter{\AdviceRunConditions}
      -->
      \noexpand\ifAdviceRun true\noexpand\else false\noexpand\fi
    }%
  }%
  \edef\AdviceBailoutHandler{%
    \noexpand\advice@trace{\space\space
      Executing bailout handler:
      \detokenize\expandafter{\AdviceBailoutHandler}}%
    \expandonce\AdviceBailoutHandler
  }%
}
\def\advice@trace@init@I#1#2{%
  \advice@setup@init@I{#1}{#2}%
  \edef\AdviceOuterHandler{%
    \noexpand\advice@trace{\space\space
      Executing outer handler:
      \detokenize\expandafter{\AdviceOuterHandler}}%
    \expandonce\AdviceOuterHandler
  }%
  \edef\AdviceCollector{%
    \noexpand\advice@trace{\space\space
      Executing collector:
      \detokenize\expandafter{\AdviceCollector}}%
    \noexpand\advice@trace{\space\space\space\space
      argument specification:
      \detokenize\expandafter{\AdviceArgs}}%
    \noexpand\advice@trace{\space\space\space\space
      options:
```

```
      \detokenize\expandafter{\AdviceCollectorOptions}}%
    \noexpand\advice@trace{\space\space\space\space
      raw options:
      \detokenize\expandafter{\AdviceRawCollectorOptions}}%
    \expandonce\AdviceCollector
  }%
```

The tracing inner handler must grab the provided argument, if it's to show what it is.

```
  \edef\advice@inner@handler@trace##1{%
    \noexpand\advice@trace{\space\space
      Executing inner handler:
      \detokenize\expandafter{\AdviceInnerHandler}}%
    \noexpand\advice@trace{\space\space\space\space
      Received arguments:
      \noexpand\detokenize{##1}}%
    \noexpand\advice@trace{\space\space\space\space
      options:
      \detokenize\expandafter{\AdviceOptions}}%
    \expandonce{\AdviceInnerHandler}{##1}%
  }%
  \def\AdviceInnerHandler{\advice@inner@handler@trace}%
}
⟨plain⟩\resetatcatcode
⟨context⟩\stopmodule
⟨context⟩\protect
⟨/main⟩
```

8.1.7 The Ti*k*Z collector

In this section, we implement the argument collector for command `\tikz`, which has idiosyncratic syntax, see §12.2.2 of the Ti*k*Z & PGF manual:

- `\tikz`⟨*animation spec*⟩`[`⟨*options*⟩`]{`⟨*picture code*⟩`}`
- `\tikz`⟨*animation spec*⟩`[`⟨*options*⟩`]`⟨*picture command*⟩`;`

where ⟨*animation spec*⟩ = (`:`⟨*key*⟩`={`⟨*value*⟩`}`)*.

The Ti*k*Z code resides in a special file. It is meant to be `\input` at any time, so we need to temporarily assign `@` category code 11.

```
⟨*tikz⟩
\edef\mmzresetatcatcode{\catcode`\noexpand\@\the\catcode`\@\relax}%
\catcode`\@=11
\def\AdviceCollectTikZArguments{%
```

We initialize the token register which will hold the collected arguments, and start the collection. Nothing of note happens until …

```
  \mmz@temptoks={}%
  \mmz@tikz@anim
}
\def\mmz@tikz@anim{%
  \pgfutil@ifnextchar[{\mmz@tikz@opt}{%
      \pgfutil@ifnextchar:{\mmz@tikz@anim@a}{%
        \mmz@tikz@code}}%]
}
\def\mmz@tikz@anim@a#1=#2{%
  \toksapp\mmz@temptoks{#1={#2}}%
  \mmz@tikz@anim
}
\def\mmz@tikz@opt[#1]{%
  \toksapp\mmz@temptoks{[#1]}%
  \mmz@tikz@code
}
\def\mmz@tikz@code{%
```

```
  \pgfutil@ifnextchar\bgroup\mmz@tikz@braced\mmz@tikz@single
}
\long\def\mmz@tikz@braced#1{\toksapp\mmz@temptoks{{#1}}\mmz@tikz@done}
\def\mmz@tikz@single#1;{\toksapp\mmz@temptoks{#1;}\mmz@tikz@done}
```

… we finish collecting the arguments, when we execute the inner handler, with the (braced) collected arguments is its sole argument.

```
\def\mmz@tikz@done{%
  \expandafter\AdviceInnerHandler\expandafter{\the\mmz@temptoks}%
}
\mmzresetatcatcode
⟨/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

```
⟨latex⟩\ProvidesPackage{collargs}[2023/10/07 v1.0.0 Collect arguments of any command]
⟨context⟩%D \module[
⟨context⟩%D         file=t-collargs.tex,
⟨context⟩%D      version=1.0.0,
⟨context⟩%D        title=CollArgs,
⟨context⟩%D     subtitle=Collect arguments of any command,
⟨context⟩%D       author=Saso Zivanovic,
⟨context⟩%D         date=2023-10-07,
⟨context⟩%D    copyright=Saso Zivanovic,
⟨context⟩%D      license=LPPL,
⟨context⟩%D ]
⟨context⟩\writestatus{loading}{ConTeXt User Module / collargs}
⟨context⟩\unprotect
⟨context⟩\startmodule[collargs]
```

Required packages

```
⟨latex⟩\RequirePackage{pgfkeys}
⟨plain⟩\input pgfkeys
⟨context⟩\input t-pgfkey
⟨latex⟩\RequirePackage{etoolbox}
⟨plain, context⟩\input etoolbox-generic
⟨plain⟩\edef\resetatcatcode{\catcode`\noexpand\@\the\catcode`\@\relax}
⟨plain⟩\catcode`\@11\relax
```

`\toksapp` `\gtoksapp` `\etoksapp` `\xtoksapp` Our macros for appending to a token register only accept a control sequence defined by `\toksdef` (like `\mytoks`) but not an explicit register designation like `\toks0`, so we only define them if noone else did; the names of the macros match the LuaTEX primitives, so they surely won't be defined there.

```
\ifdefined\toksapp\else
  \long\def\toksapp#1#2{#1\expandafter{\the#1#2}}%
\fi
\ifdefined\etoksapp\else
  \long\def\etoksapp#1#2{#1\expandafter{\expanded{\the#1#2}}}%
\fi
\ifdefined\gtoksapp\else
  \long\def\gtoksapp#1#2{\global#1\expandafter{\the#1#2}}%
\fi
\ifdefined\xtoksapp\else
  \long\def\xtoksapp#1#2{\global#1\expandafter{\expanded{\the#1#2}}}%
\fi
\ifdefined\toks@\else\toksdef\toks@=0 \fi
```

`\CollectArguments` `\CollectArgumentsRaw` These are the only public commands provided by the package. `\CollectArguments` takes three arguments: the optional `#1` is the option list, processed 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.)

```
\protected\def\CollectArguments{%
  \pgf@keys@utilifnextchar[\CollectArguments@i{\CollectArgumentsRaw{}}%]
}
\def\CollectArguments@i[#1]{\CollectArgumentsRaw{\collargsSet{#1}}}
\protected\def\CollectArgumentsRaw#1#2#3{%
```

This group will be closed by `\collargs@.` once we grinded through the argument specification.

```
  \begingroup
```

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.

```
  \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
  \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
  \global\collargs@double@fixfalse
```

Apply the settings.

```
  \collargs@verbatim@wrap{#1}%
```

Initialize the space-grabber.

```
  \collargs@init@grabspaces
```

Remember the code to execute after collection.

```
  \def\collargs@next{#3}%
```

Initialize the token register holding the collected arguments.

```
  \global\collargs@toks{}%
```

Execute the central loop macro, which expects the argument specification `#2` to be delimited from the following argument tokens by a dot.

```
  \collargs@#2.%
}
```

`\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.

```
\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.

```
\def\collargs@cs@cases#1{\collargs@cs@cases@i#1\collargs@cs@cases@end}
\let\collargs@cs@cases@end\relax
\def\collargs@cs@cases@i{\futurelet\collargs@temp\collargs@cs@cases@ii}
\def\collargs@cs@cases@ii#1#2\collargs@cs@cases@end{%
  \ifcat\noexpand\collargs@temp\relax
    \ifx\relax#2\relax
      \expandafter\expandafter\expandafter\@firstofthree
    \else
      \expandafter\expandafter\expandafter\@secondofthree
    \fi
  \else
    \expandafter\@thirdofthree
  \fi
}
\def\@firstofthree#1#2#3{#1}
\def\@secondofthree#1#2#3{#2}
\def\@thirdofthree#1#2#3{#3}
```

`caller`
`\collargsCaller` 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}`.

```
\collargsSet{
  caller/.code={\collargsCaller{#1}},
}
\def\collargsCaller#1{%
  \collargs@cs@cases{#1}{%
    \let\collargs@temp\collargs@caller@cs
```

[4]The idea is borrowed from package `environ`, which is in turn based on code from `amsmath`.

```
  }{%
    \let\collargs@temp\collargs@caller@csandmore
  }{%
    \let\collargs@temp\collargs@caller@env
  }%
  \collargs@temp{#1}%
}
\def\collargs@caller@cs#1{%
```

If `#1` is a single control sequence, just use that as the caller.

```
  \def\collargs@caller{#1}%
}
\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.

```
  \begingroup
  \escapechar -1
  \expandafter\endgroup
  \expandafter\def\expandafter\collargs@caller\expandafter{%
    \csname\string#1\endcsname
  }%
}
\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.

```
             \expandafter\def\expandafter\collargs@caller\expandafter{%
               \csname
⟨context⟩      start%
⟨latex⟩        begin{%
               #1%
⟨latex⟩        }%
               \endcsname
             }%
           }
           \collargsCaller\CollectArguments
```

`\ifcollargs@verbatim` `\ifcollargs@verbatimbraces` The first of these conditional signals that we're collecting the arguments in one of the verbatim modes; the second one signals the `verb` mode in particular.

```
\newif\ifcollargs@verbatim
\newif\ifcollargs@verbatimbraces
```

`verbatim` `verb` `no verbatim` `\collargs@verbatim@wrap` These keys set the verbatim mode macro which will be executed by `\collargsSet` after processing all keys. The verbatim mode macros `\collargsVerbatim`, `\collargsVerb` 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 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`.

```
\collargsSet{
  verbatim/.code={\let\collargs@apply@verbatim\collargsVerbatim},
  verb/.code={\let\collargs@apply@verbatim\collargsVerb},
  no verbatim/.code={\let\collargs@apply@verbatim\collargsNoVerbatim},
}
\def\collargs@verbatim@wrap#1{%
  \let\collargs@apply@verbatim\relax
  #1%
  \collargs@apply@verbatim
}
```

fix from verbatim
fix from verb
fix from no verbatim
\collargsFixFromVerbatim
\collargsFixFromVerb
\collargsFixFromNoVerbatim

These keys and macros should be used to request a category code fix, when the offending tokenization took place prior to invoking `\CollectArguments`; see section 8.2.6 for details. While I assume that only `\collargsFixFromNoVerbatim` will ever be used (and it is used by `\mmz`), we provide macros for all three transitions, for completeness.

```
\collargsSet{
  fix from verbatim/.code={\collargsFixFromVerbatim},
  fix from verb/.code={\collargsFixFromVerb},
  fix from no verbatim/.code={\collargsFixFromNoVerbatim},
}
```

```
\def\collargsFixFromNoVerbatim{%
  \global\collargs@fix@requestedtrue
  \global\let\ifcollargs@last@verbatim\iffalse
}
\def\collargsFixFromVerbatim{%
  \global\collargs@fix@requestedtrue
  \global\let\ifcollargs@last@verbatim\iftrue
  \global\let\ifcollargs@last@verbatimbraces\iftrue
}
\def\collargsFixFromVerb{%
  \global\collargs@fix@requestedtrue
  \global\let\ifcollargs@last@verbatim\iftrue
  \global\let\ifcollargs@last@verbatimbraces\iffalse
}
```

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.

```
\collargsSet{
  braces/.code={\collargsBraces{#1}}%
}
```

environment
\collargsEnvironment

Set the environment name.

```
\collargsSet{
  environment/.estore in=\collargs@b@envname
}
\def\collargsEnvironment#1{\edef\collargs@b@envname{#1}}
\collargsEnvironment{}
```

begin tag
end tag
tags
\ifcollargsBeginTag
\ifcollargsEndTag
\ifcollargsAddTags

When `begin tag`/`end tag` is in effect, the begin/end-tag will be will be prepended/appended to the environment body. `tags` is a shortcut for setting `begin tag` and `end tag` simultaneously.

```
\collargsSet{
  begin tag/.is if=collargsBeginTag,
  end tag/.is if=collargsEndTag,
  tags/.style={begin tag=#1, end tag=#1},
  tags/.default=true,
}
\newif\ifcollargsBeginTag
\newif\ifcollargsEndTag
```

ignore nesting
\ifcollargsIgnoreNesting

When this key is in effect, we will ignore any `\begin{`⟨*name*⟩`}`s and simply grab everything up to the first `\end{`⟨*name*⟩`}` (again, the markers are automatically adapted to the format).

```
\collargsSet{
  ignore nesting/.is if=collargsIgnoreNesting,
}
\newif\ifcollargsIgnoreNesting
```

ignore other tags
\ifcollargsIgnoreOtherTags

This key is only relevant in the non-verbatim and partial verbatim modes in LaTeX. 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`.

```
\collargsSet{
  ignore other tags/.is if=collargsIgnoreOtherTags,
}
\newif\ifcollargsIgnoreOtherTags
```

(append/prepend) (pre/post)processor
\collargs(Append/Prepend)(Pre/Post)processor

These keys and macros populate the list of preprocessors, `\collargs@preprocess@arg`, and the list of post-processors, `\collargs@postprocess@arg`, executed in `\collargs@appendarg`.

```
\collargsSet{
  append preprocessor/.code={\collargsAppendPreprocessor{#1}},
  prepend preprocessor/.code={\collargsPrependPreprocessor{#1}},
  append postprocessor/.code={\collargsAppendPostprocessor{#1}},
  prepend postprocessor/.code={\collargsPrependPostprocessor{#1}},
}
\def\collargsAppendPreprocessor{%
  \collargs@addprocessor\appto\collargs@preprocess@arg}
\def\collargsPrependPreprocessor{%
  \collargs@addprocessor\preto\collargs@preprocess@arg}
\def\collargsAppendPostprocessor{%
  \collargs@addprocessor\appto\collargs@postprocess@arg}
\def\collargsPrependPostprocessor{%
  \collargs@addprocessor\preto\collargs@postprocess@arg}
```

Here, `#1` will be either `\appto` or `\preto`, and `#2` will be either `\collargs@preprocess@arg` or `\collargs@postprocess@arg`. `#3` is the processor code.

```
\def\collargs@addprocessor#1#2#3{%
  #1#2{%
    \expanded{%
      \unexpanded{#3}{\the\collargsArg}%
    }%
  }%
}
```

clear (pre/post)processors
\collargsClear(Pre/Post)processors

These keys and macros clear the pre- and post-processor lists, which are initially empty as well.

```
\def\collargs@preprocess@arg{}
\def\collargs@postprocess@arg{}
\collargsSet{
  clear preprocessors/.code={\collargsClearPreprocessors},
  clear postprocessors/.code={\collargsClearPostprocessors},
}
\def\collargsClearPreprocessors{\def\collargs@preprocess@arg{}}%
\def\collargsClearPostprocessors{\def\collargs@postprocess@arg{}}%
```

(append/prepend) expandable (pre/post)processor
\collargs(Append/Prepend)Expandable(Pre/Post)processor These keys and macros simplify the definition of fully expandable processors. Note that expandable processors are added to the same list as non-expandable processors.

```
\collargsSet{
  append expandable preprocessor/.code={%
    \collargsAppendExpandablePreprocessor{#1}},
  prepend expandable preprocessor/.code={%
    \collargsPrependExpandablePreprocessor{#1}},
  append expandable postprocessor/.code={%
    \collargsAppendExpandablePostprocessor{#1}},
  prepend expandable postprocessor/.code={%
    \collargsPrependExpandablePostprocessor{#1}},
}
\def\collargsAppendExpandablePreprocessor{%
  \collargs@addeprocessor\appto\collargs@preprocess@arg}
\def\collargsPrependExpandablePreprocessor{%
  \collargs@addeprocessor\preto\collargs@preprocess@arg}
\def\collargsAppendExpandablePostprocessor{%
  \collargs@addeprocessor\appto\collargs@postprocess@arg}
\def\collargsPrependExpandablePostprocessor{%
  \collargs@addeprocessor\preto\collargs@postprocess@arg}
\def\collargs@addeprocessor#1#2#3{%
  #1#2{%
    \expanded{%
      \edef\noexpand\collargs@temp{\unexpanded{#3}{\the\collargsArg}}%
      \unexpanded{\expandafter\collargsArg\expandafter{\collargs@temp}}%
    }%
  }%
}
```

(append/prepend) (pre/post)wrap
\collargs(Append/Prepend)(Pre/Post)wrap These keys and macros simplify the definition of processors which yield the result after a single expansion. Again, they are added to the same list as other processors.

```
\collargsSet{
  append prewrap/.code={\collargsAppendPrewrap{#1}},
  prepend prewrap/.code={\collargsPrependPrewrap{#1}},
  append postwrap/.code={\collargsAppendPostwrap{#1}},
  prepend postwrap/.code={\collargsPrependPostwrap{#1}},
}
\def\collargsAppendPrewrap{\collargs@addwrap\appto\collargs@preprocess@arg}
\def\collargsPrependPrewrap{\collargs@addwrap\preto\collargs@preprocess@arg}
\def\collargsAppendPostwrap{\collargs@addwrap\appto\collargs@postprocess@arg}
\def\collargsPrependPostwrap{\collargs@addwrap\preto\collargs@postprocess@arg}
\def\collargs@addwrap#1#2#3{%
  #1#2{%
    \long\def\collargs@temp##1{#3}%
    \expandafter\expandafter\expandafter\collargsArg
    \expandafter\expandafter\expandafter{%
      \expandafter\collargs@temp\expandafter{\the\collargsArg}%
    }%
  }%
}
```

no delimiters
\ifcollargsNoDelimiters When this conditional is in effect, the delimiter wrappers set by \collargs@wrap are ignored by \collargs@appendarg.

```
\collargsSet{%
  no delimiters/.is if=collargsNoDelimiters,
}
\newif\ifcollargsNoDelimiters
```

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`.

```
\def\collargs@{%
  \begingroup
  \collargs@@@
}
```

`\collargs@@@` This macro is where modifier handlers reenter the central loop — we don't want modifers to open a group, because their settings should remain in effect until the next argument. Furthermore, modifiers do not trigger category code fixes.

```
\def\collargs@@@#1{%
  \collargs@in@{#1}{&+!>.}%
  \ifcollargs@in@
    \expandafter\collargs@@@iii
  \else
    \expandafter\collargs@@@i
  \fi
  #1%
}
\def\collargs@@@i#1.{%
```

Fix the category code of the next argument token, if necessary, and then proceed with the main loop.

```
  \collargs@fix{\collargs@@@ii#1.}%
}
```

Reset the fix request and set the last verbatim conditionals to the current state.

```
\def\collargs@@@ii{%
  \global\collargs@fix@requestedfalse
  \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
  \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
  \collargs@@@iii
}
```

Call the modifier or argument type handler denoted by the first token of the remainder of the argument specification.

```
\def\collargs@@@iii#1{%
  \ifcsname collargs@#1\endcsname
    \csname collargs@#1\expandafter\endcsname
  \else
```

We throw an error if the token refers to no argument type or modifier.

```
    \collargs@error@badtype{#1}%
  \fi
}
```

Throwing an error stops the processing of the argument specification, and closes the group opened in `\collargs@i`.

```
\def\collargs@error@badtype#1#2.{%
  \PackageError{collargs}{Unknown xparse argument type or modifier "#1"
    for "\expandafter\string\collargs@caller\space"}{}%
  \endgroup
}
```

`\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) `&`.

```
\csdef{collargs@&}{%
  \futurelet\collargs@temp\collargs@amp@i
}
\def\collargs@amp@i{%
```

In ConTEXt, `&` has character code “other” in the text.

```
⟨!context⟩  \ifx\collargs@temp&%
⟨context⟩   \expandafter\ifx\detokenize{&}\collargs@temp
    \expandafter\collargs@amp@raw
  \else
    \expandafter\collargs@amp@set
  \fi
}
\def\collargs@amp@raw#1#2{%
  \collargs@verbatim@wrap{#2}%
  \collargs@@@
}
\def\collargs@amp@set#1{%
  \collargs@verbatim@wrap{\collargsSet{#1}}%
  \collargs@@@
}
```

`\collargs@+` This modifier makes the next argument long, i.e. accept paragraph tokens.

```
\csdef{collargs@+}{%
  \collargs@longtrue
  \collargs@@@
}
\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.)

```
\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.

```
\csdef{collargs@!}{%
  \collargs@grabspacesfalse
  \collargs@@@
}
```

\collargs@toks 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.

```
\newtoks\collargs@toks
```

\collargsArg An auxiliary, but publicly available token register, used for processing the argument, and by some argument type handlers.

```
\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.

```
\csdef{collargs@.}{%
```

Close the group opened in `\collargs@`.

```
  \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.

```
  \expanded{%
    \endgroup
    \noexpand\collargs@fix{%
      \expandonce\collargs@next{\the\collargs@toks}%
      \collargs@spaces
    }%
  }%
}
```

8.2.3 Auxiliary macros

\collargs@appendarg This macro is used by the argument type handlers to append the collected argument to the storage (`\collargs@toks`).

```
\long\def\collargs@appendarg#1{%
```

Temporarily store the collected argument into a token register. The processors will manipulate the contents of this register.

```
  \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.

```
  \ifcollargs@double@fix
    \collargs@cancel@double@fix
  \fi
```

Process the argument with user-definable preprocessors, the wrapper defined by the argument type, and user-definable postprocessors.

```
  \collargs@preprocess@arg
  \ifcollargsNoDelimiters
  \else
    \collargs@process@arg
  \fi
  \collargs@postprocess@arg
```

Append the processed argument, preceded by any grabbed spaces (in the correct mode), to the storage.

```
  \xtoksapp\collargs@toks{\collargs@grabbed@spaces\the\collargsArg}%
```

Initialize the space-grabber.

```
  \collargs@init@grabspaces
```

Once the argument was appended to the list, we can close its group, opened by `\collargs@`.

```
  \endgroup
}
```

`\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.

```
\def\collargs@wrap{\collargs@addwrap\appto\collargs@process@arg}
\def\collargs@process@arg{}
```

`\collargs@defcollector` `\collargs@defusecollector` `\collargs@letusecollector` These macros streamline the usage of the "caller" control sequence. They are like a `\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.

```
\def\collargs@defcollector#1#{%
  \ifcollargs@long\long\fi
  \expandafter\def\collargs@caller#1%
}
\def\collargs@defusecollector#1#{%
  \afterassignment\collargs@caller
  \ifcollargs@long\long\fi
  \expandafter\def\collargs@caller#1%
}
\def\collargs@letusecollector#1{%
  \expandafter\let\collargs@caller#1%
  \collargs@caller
}
\newif\ifcollargs@grabspaces
\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.

```
\def\collargs@init@grabspaces{%
  \gdef\collargs@gs@state{0}%
  \gdef\collargs@spaces{}%
  \gdef\collargs@otherspaces{}%
}
```

`\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.

```
\def\collargs@grabspaces#1{%
  \edef\collargs@gs@next{\unexpanded{#1}}%
  \ifnum\collargs@gs@state=0
    \gdef\collargs@gs@state{1}%
    \expandafter\collargs@gs@i
  \else
    \expandafter\collargs@gs@next
  \fi
}
\def\collargs@gs@i{%
  \futurelet\collargs@temp\collargs@gs@g
}
```

We check for grouping characters even in the verbatim mode, because we might be in the partial verbatim.

```
\def\collargs@gs@g{%
  \ifcat\noexpand\collargs@temp\bgroup
    \expandafter\collargs@gs@next
  \else
    \ifcat\noexpand\collargs@temp\egroup
      \expandafter\expandafter\expandafter\collargs@gs@next
    \else
      \expandafter\expandafter\expandafter\collargs@gs@ii
    \fi
  \fi
}
\def\collargs@gs@ii{%
  \ifcollargs@verbatim
    \expandafter\collargs@gos@iii
  \else
    \expandafter\collargs@gs@iii
  \fi
}
```

This works because the character code of a space token is always 32.

```
\def\collargs@gs@iii{%
  \expandafter\ifx\space\collargs@temp
    \expandafter\collargs@gs@iv
  \else
    \expandafter\collargs@gs@next
  \fi
}
\expandafter\def\expandafter\collargs@gs@iv\space{%
  \gappto\collargs@spaces{ }%
  \xappto\collargs@otherspaces{\collargs@otherspace}%
  \collargs@gs@i
}
```

We need the space of category 12 above.

```
\begingroup\catcode`\ =12\relax\gdef\collargs@otherspace{ }\endgroup
\def\collargs@gos@iii#1{%
```

Macro `\collargs@cc` recalls the “outside” category code of character `#1`; see section 8.2.5.

```
  \ifnum\collargs@cc{#1}=10
```

We have a space.

```
    \expandafter\collargs@gos@iv
  \else
    \ifnum\collargs@cc{#1}=5
```

We have a newline.

```
      \expandafter\expandafter\expandafter\collargs@gos@v
    \else
      \expandafter\expandafter\expandafter\collargs@gs@next
    \fi
  \fi
  #1%
}
\def\collargs@gos@iv#1{%
  \gappto\collargs@otherspaces{#1}%
```

No matter how many verbatim spaces we collect, they equal a single non-verbatim space.

```
  \gdef\collargs@spaces{ }%
  \collargs@gs@i
}
\def\collargs@gos@v{%
```

Only add the first newline.

```
  \ifnum\collargs@gs@state=2
    \expandafter\collargs@gs@next
  \else
    \expandafter\collargs@gs@vi
  \fi
}
\def\collargs@gs@vi#1{%
  \gdef\collargs@gs@state{2}%
  \gappto\collargs@otherspaces{#1}%
  \gdef\collargs@spaces{ }%
  \collargs@gs@i
}
```

`\collargs@maybegrabspaces` This macro grabs any following spaces, but it will do so only when conditional `\ifcollargs@grabspaces`, which can be *un*set by modifier `!`, is in effect. The macro is used by handlers for types `d` and `t`.

```
\def\collargs@maybegrabspaces{%
  \ifcollargs@grabspaces
    \expandafter\collargs@grabspaces
  \else
    \expandafter\@firstofone
  \fi
}
```

`\collargs@grabbed@spaces` 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.

```
\def\collargs@grabbed@spaces{%
  \ifcollargs@verbatim
    \collargs@otherspaces
  \else
    \collargs@spaces
  \fi
}
```

\collargs@reinsert@spaces 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.

```
\def\collargs@reinsert@spaces#1{%
  \expanded{%
    \unexpanded{%
      \collargs@init@grabspaces
      #1%
    }%
    \collargs@grabbed@spaces
  }%
}
```

\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`.

```
\long\def\collargs@ifnextcat#1#2#3{%
  \let\pgf@keys@utilreserved@d=#1%
  \def\pgf@keys@utilreserved@a{#2}%
  \def\pgf@keys@utilreserved@b{#3}%
  \futurelet\pgf@keys@utillet@token\collargs@ifncat}
\def\collargs@ifncat{%
  \ifx\pgf@keys@utillet@token\pgf@keys@utilsptoken
    \let\pgf@keys@utilreserved@c\collargsxifnch
  \else
    \ifcat\noexpand\pgf@keys@utillet@token\pgf@keys@utilreserved@d
      \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@a
    \else
      \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@b
    \fi
  \fi
  \pgf@keys@utilreserved@c}
{%
  \def\:{\collargs@xifncat}
  \expandafter\gdef\: {\futurelet\pgf@keys@utillet@token\collargs@ifncat}
}
```

\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.

```
\def\collargs@forrange#1#2{%
  \expanded{%
     \noexpand\collargs@forrange@i{\number#1}{\number#2}%
   }%
 }
\def\collargs@forrange@i#1#2{%
  \ifnum#1>#2 %
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {%
    \collargs@do{#1}%
    \expandafter\collargs@forrange@i\expandafter{\number\numexpr#1+1\relax}{#2}%
  }%
}
```

\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`.

```
\def\collargs@forranges{\forcsvlist\collarg@forrange@i}
\def\collarg@forrange@i#1{\collarg@forrange@ii#1-}
\def\collarg@forrange@ii#1-#2-{\collargs@forrange{#1}{#2}}
```

\collargs@percentchar This macro holds the percent character of category 12.

```
\begingroup
\catcode`\%=12
\gdef\collargs@percentchar{%}
\endgroup
```

8.2.4 The handlers

\collargs@l We will first define the handler for the very funky argument type `l`, which corresponds to TEX'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.

```
\def\collargs@l#1.{%
```

Any pre-grabbed spaces in fact belong into the argument.

```
  \collargs@reinsert@spaces{\collargs@l@i#1.}%
}
\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.

```
  \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.

```
  \ifcollargs@verbatimbraces
    \expandafter\collargs@l@verb
  \else
    \expandafter\collargs@l@ii
  \fi
}
```

We grab the rest of the argument specification (`#1`), to be reinserted into the token stream when we reexecute the central loop.

```
\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.

```
  \collargs@defusecollector##1##{%
```

We append the collected argument, `##1`, to `\collargs@toks`, the token register holding the collected argument tokens.

```
    \collargs@appendarg{##1}%
```

Back to the central loop, with the rest of the argument specification reinserted.

```
    \collargs@#1.%
  }%
}
\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.

```
  \expandafter\collargs@defusecollector
  \expandafter##\expandafter1\collargs@other@bgroup{%
```

Appending the argument works the same as in the non-verbatim case.

```
    \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.

```
    \expanded{%
      \noexpand\collargs@\unexpanded{#1.}%
      \collargs@other@bgroup
    }%
  }%
}
```

`\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.

```
\def\collargs@u{%
  \ifcollargs@verbatim
    \expandafter\collargs@u@verb
  \else
    \expandafter\collargs@u@i
  \fi
}
```

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`.

```
\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.

```
  \expandafter\collargs@u@i\expandafter{\detokenize\expandafter{\string#1}}%
}
```

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.

```
\def\collargs@u@i#1#2.{%
  \collargs@reinsert@spaces{\collargs@u@ii{#1}#2.}%
}
\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`).

```
  \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).

```
    \collargs@wrap{####1#1}%
```

Expand the first token in `##1`, which we know to be `\collargs@empty`, with empty expansion.

```
    \expandafter\collargs@appendarg\expandafter{##1}%
    \collargs@#2.%
  }%
```

Insert `\collargs@empty` into the input stream, in front of the "real" argument tokens.

```
  \collargs@empty
}
\def\collargs@empty{}
```

`\collargs@r` Finally, a real argument type: required delimited argument.

```
\def\collargs@r{%
  \ifcollargs@verbatim
    \expandafter\collargs@r@verb
  \else
    \expandafter\collargs@r@i
  \fi
}
\def\collargs@r@verb#1#2{%
  \expandafter\collargs@r@i\detokenize{#1#2}%
}
\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.

```
  \collargs@defcollector#1{%
```

The inner collector will grab everything up to the closing delimiter.

```
    \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`.

```
      \collargs@wrap{#1########1#2}%
      \expandafter\collargs@appendarg\expandafter{####1}%
      \collargs@#3.%
    }%
    \collargs@empty
  }%
```

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.

```
  \collargs@grabspaces\collargs@caller
}
```

\collargs@R Discard the default and execute `r`.

```
\def\collargs@R#1#2#3{\collargs@r#1#2}
```

\collargs@d Optional delimited argument. Very similar to `r`.

```
\def\collargs@d{%
  \ifcollargs@verbatim
    \expandafter\collargs@d@verb
  \else
    \expandafter\collargs@d@i
  \fi
}
\def\collargs@d@verb#1#2{%
  \expandafter\collargs@d@i\detokenize{#1#2}%
}
\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.

```
  \def\collargs@d@noopt{%
    \global\collargs@fix@requestedtrue
    \endgroup
    \collargs@#3.%
  }%
```

The collector(s) are exactly as for `r`.

```
  \collargs@defcollector#1{%
    \collargs@defusecollector####1#2{%
      \collargs@wrap{#1########1#2}%
      \expandafter\collargs@appendarg\expandafter{####1}%
      \collargs@#3.%
    }%
    \collargs@empty
  }%
```

This macro will check, in conjunction with `\futurelet` below, whether the optional argument is present or not.

```
  \def\collargs@d@ii{%
    \ifx#1\collargs@temp
      \expandafter\collargs@caller
    \else
      \expandafter\collargs@d@noopt
    \fi
  }%
```

Whether spaces are allowed in front of this type of argument depends on the presence of modifier `!`.

```
  \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@d@ii}%
}
```

\collargs@D Discard the default and execute `d`.

```
\def\collargs@D#1#2#3{\collargs@d#1#2}
```

`\collargs@o` o is just d with delimiters [and].

```
\def\collargs@o{\collargs@d[]}
```

`\collargs@O` O is just d with delimiters [and] and the discarded default.

```
\def\collargs@O#1{\collargs@d[]}
```

`\collargs@t` An optional token. Similar to d.

```
\def\collargs@t{%
  \ifcollargs@verbatim
    \expandafter\collargs@t@verb
  \else
    \expandafter\collargs@t@i
  \fi
}
\def\collargs@t@space{ }
\def\collargs@t@verb#1{%
  \let\collargs@t@space\collargs@otherspace
  \expandafter\collargs@t@i\expandafter{\detokenize{#1}}%
}
\def\collargs@t@i#1{%
  \expandafter\ifx\space#1%
    \expandafter\collargs@t@s
  \else
    \expandafter\collargs@t@I\expandafter#1%
  \fi
}
\def\collargs@t@s#1.{%
  \collargs@grabspaces{%
    \ifcollargs@grabspaces
      \collargs@appendarg{}%
    \else
      \expanded{%
        \noexpand\collargs@init@grabspaces
        \noexpand\collargs@appendarg{\collargs@grabbed@spaces}%
      }%
    \fi
    \collargs@#1.%
  }%
}
\def\collargs@t@I#1#2.{%
  \def\collargs@t@noopt{%
    \global\collargs@fix@requestedtrue
    \endgroup
    \collargs@#2.%
  }%
  \def\collargs@t@opt##1{%
    \collargs@appendarg{#1}%
    \collargs@#2.%
  }%
  \def\collargs@t@ii{%
    \ifx#1\collargs@temp
      \expandafter\collargs@t@opt
    \else
      \expandafter\collargs@t@noopt
    \fi
  }%
  \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@t@ii}%
}
\def\collargs@t@opt@space{%
  \expanded{\noexpand\collargs@t@opt{\space}\expandafter}\romannumeral-0%
}%
```

\collargs@s The optional star is just a special case of t.

```
\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.

```
\def\collargs@m{%
  \ifcollargs@verbatim
    \expandafter\collargs@m@verb
  \else
    \expandafter\collargs@m@i
  \fi
}
```

The non-verbatim mode. First, collect any spaces in front of the argument.

```
\def\collargs@m@i#1.{%
  \collargs@grabspaces{\collargs@m@checkforgroup#1.}%
}
```

Is the argument in braces or not?

```
\def\collargs@m@checkforgroup#1.{%
  \edef\collargs@action{\unexpanded{\collargs@m@checkforgroup@i#1.}}%
  \futurelet\collargs@token\collargs@action
}
\def\collargs@m@checkforgroup@i{%
  \ifcat\noexpand\collargs@token\bgroup
    \expandafter\collargs@m@group
  \else
    \expandafter\collargs@m@token
  \fi
}
```

The argument is given in braces, so we put them back around it (\collargs@wrap) when appending to the storage.

```
\def\collargs@m@group#1.{%
  \collargs@defusecollector##1{%
    \collargs@wrap{{####1}}%
    \collargs@appendarg{##1}%
    \collargs@#1.%
  }%
}
```

The argument is a single token, we append it to the storage as is.

```
\def\collargs@m@token#1.{%
  \collargs@defusecollector##1{%
    \collargs@appendarg{##1}%
    \collargs@#1.%
  }%
}
```

The verbatim mode. Again, we first collect any spaces in front of the argument.

```
\def\collargs@m@verb#1.{%
  \collargs@grabspaces{\collargs@m@verb@checkforgroup#1.}%
}
```

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.

```
\def\collargs@m@verb@checkforgroup{%
  \ifcollargs@verbatimbraces
    \expandafter\collargs@m@verb@checkforgroup@i
  \else
    \expandafter\collargs@m@checkforgroup
  \fi
}
```

Is the argument in verbatim braces?

```
\def\collargs@m@verb@checkforgroup@i#1.{%
  \def\collargs@m@verb@checkforgroup@ii{\collargs@m@verb@checkforgroup@iii#1.}%
  \futurelet\collargs@temp\collargs@m@verb@checkforgroup@ii
}
\def\collargs@m@verb@checkforgroup@iii#1.{%
  \expandafter\ifx\collargs@other@bgroup\collargs@temp
```

Yes, the argument is in (verbatim) braces.

```
    \expandafter\collargs@m@verb@group
  \else
```

We need to manually check whether the following token is a (verbatim) closing brace, and throw an error if it is.

```
    \expandafter\ifx\collargs@other@egroup\collargs@temp
      \expandafter\expandafter\expandafter\collargs@m@verb@egrouperror
    \else
```

The argument is a single token.

```
      \expandafter\expandafter\expandafter\collargs@m@v@token
    \fi
  \fi
  #1.%
}
\def\collargs@m@verb@egrouperror#1.{%
  \PackageError{collargs}{%
    Argument of \expandafter\string\collargs@caller\space has an extra
    \iffalse{\else\string}}{}%
}
```

A single-token verbatim argument.

```
\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.)

```
  \ifnum\collargs@cc{#2}=0
    \expandafter\collargs@m@v@token@cs
  \else
    \expandafter\collargs@m@token
  \fi
  #1.#2%
}
```

Is it a one-character control sequence?

```
\def\collargs@m@v@token@cs#1.#2#3{%
  \ifnum\collargs@cc{#3}=11
    \expandafter\collargs@m@v@token@cs@letter
  \else
    \expandafter\collargs@m@v@token@cs@nonletter
  \fi
  #1.#2#3%
}
```

Store `\<token>`.

```
\def\collargs@m@v@token@cs@nonletter#1.#2#3{%
  \collargs@appendarg{#2#3}%
  \collargs@#1.%
}
```

Store `\` to a temporary register, we'll parse the control sequence name now.

```
\def\collargs@m@v@token@cs@letter#1.#2{%
  \collargsArg{#2}%
  \def\collargs@tempa{#1}%
  \collargs@m@v@token@cs@letter@i
}
```

Append a letter to the control sequence.

```
\def\collargs@m@v@token@cs@letter@i#1{%
  \ifnum\collargs@cc{#1}=11
    \toksapp\collargsArg{#1}%
    \expandafter\collargs@m@v@token@cs@letter@i
  \else
```

Finish, returning the non-letter to the input stream.

```
    \expandafter\collargs@m@v@token@cs@letter@ii\expandafter#1%
  \fi
}
```

Store the verbatim control sequence.

```
\def\collargs@m@v@token@cs@letter@ii{%
  \expanded{%
    \unexpanded{%
      \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
    }%
    \noexpand\collargs@\expandonce\collargs@tempa.%
  }%
}
```

The verbatim mandatory argument is delimited by verbatim braces. We have to use the heavy machinery adapted from `cprotect`.

```
\def\collargs@m@verb@group#1.#2{%
  \let\collargs@begintag\collargs@other@bgroup
  \let\collargs@endtag\collargs@other@egroup
  \def\collargs@tagarg{}%
  \def\collargs@commandatend{\collargs@m@verb@group@i#1.}%
  \collargs@readContent
}
```

This macro appends the result given by the heavy machinery, waiting for us in macro `\collargsArg`, to `\collargs@toks`, but not before dressing it up (via `\collargs@wrap`) in a pair of verbatim braces.

```
\def\collargs@m@verb@group@i{%
  \edef\collargs@temp{%
    \collargs@other@bgroup\unexpanded{##1}\collargs@other@egroup}%
  \expandafter\collargs@wrap\expandafter{\collargs@temp}%
  \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
  \collargs@
}
```

\collargs@g An optional group: same as m, but we simply bail out if we don't find the group character.

```
\def\collargs@g{%
  \def\collargs@m@token{%
    \global\collargs@fix@requestedtrue
    \endgroup
    \collargs@
  }%
  \let\collargs@m@v@token\collargs@m@token
  \collargs@m
}
```

\collargs@G Discard the default and execute g.

```
\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 mimick `xparse` perfectly.

```
\def\collargs@v#1.{%
  \begingroup
  \collargsBraces{{}}%
  \collargsVerbatim
  \collargs@grabspaces{\collargs@v@i#1.}%
}
\def\collargs@v@i#1.#2{%
  \expandafter\ifx\collargs@other@bgroup#2%
```

If the first token we see is an opening brace, use the `cprotect` adaptation to grab the group.

```
    \let\collargs@begintag\collargs@other@bgroup
    \let\collargs@endtag\collargs@other@egroup
    \def\collargs@tagarg{}%
    \def\collargs@commandatend{%
      \edef\collargs@temp{%
        \collargs@other@bgroup\unexpanded{####1}\collargs@other@egroup}%
      \expandafter\collargs@wrap\expandafter{\collargs@temp}%
      \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
      \endgroup
      \collargs@#1.%
    }%
    \expandafter\collargs@readContent
  \else
```

Otherwise, the verbatim argument is delimited by two identical characters (#2).

```
    \collargs@defcollector##1#2{%
      \collargs@wrap{#2####1#2}%
      \collargs@appendarg{##1}%
      \endgroup
      \collargs@#1.%
    }%
    \expandafter\collargs@caller
  \fi
}
```

\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).

```
\def\collargs@b{%
```

```
  \collargs@ifnextcat\bgroup\collargs@bg\collargs@bi
}
\def\collargs@bg#1{%
  \edef\collargs@b@envname{#1}%
  \collargs@bi
}
\def\collargs@bi#1.{%
```

Convert the environment name to verbatim if necessary.

```
  \ifcollargs@verbatim
    \edef\collargs@b@envname{\detokenize\expandafter{\collargs@b@envname}}%
  \fi
```

This is a format-specific macro which sets up `\collargs@begintag` and `\collargs@endtag`.

```
  \collargs@bi@defCPTbeginend
  \edef\collargs@tagarg{%
    \ifcollargs@verbatimbraces
    \else
      \ifcollargsIgnoreOtherTags
        \collargs@b@envname
      \fi
    \fi
  }%
```

Run this after collecting the body.

```
  \def\collargs@commandatend{%
```

In LaTeX, we might, depending on the verbatim mode, need to check whether the environment name is correct.

```
⟨latex⟩    \collargs@bii
```

In plain TeX and ConTeXt, we can skip directly to `\collargs@biii`.

```
⟨plain, context⟩    \collargs@biii
    #1.%
  }%
```

Collect the environment body, but first, put any grabbed spaces back into the input stream.

```
  \collargs@reinsert@spaces\collargs@readContent
}
⟨*latex⟩
```

In LaTeX 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*⟩`}`.

```
\def\collargs@bi@defCPTbeginend{%
  \edef\collargs@begintag{%
    \ifcollargs@verbatim
      \expandafter\string
    \else
      \expandafter\noexpand
    \fi
    \begin
    \ifcollargs@verbatimbraces
      \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
    \fi
  }%
  \edef\collargs@endtag{%
    \ifcollargs@verbatim
      \expandafter\string
    \else
      \expandafter\noexpand
    \fi
    \end
    \ifcollargs@verbatimbraces
      \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
    \fi
  }%
}
⟨/latex⟩
⟨*plain, context⟩
```

We can search for the entire `\`⟨*name*⟩/`\end`⟨*name*⟩ (in TEX) or `\start`⟨*name*⟩/`\stop`⟨*name*⟩ (in ConTEXt), either as a control sequence (in the regular mode), or as a string (in the verbatim modes).

```
\def\collargs@bi@defCPTbeginend{%
  \edef\collargs@begintag{%
    \ifcollargs@verbatim
      \expandafter\expandafter\expandafter\string
    \else
      \expandafter\expandafter\expandafter\noexpand
    \fi
    \csname
⟨context⟩      start%
      \collargs@b@envname
    \endcsname
  }%
  \edef\collargs@endtag{%
    \ifcollargs@verbatim
      \expandafter\expandafter\expandafter\string
    \else
      \expandafter\expandafter\expandafter\noexpand
    \fi
    \csname
⟨plain⟩      end%
⟨context⟩      stop%
      \collargs@b@envname
    \endcsname
  }%
}
⟨/plain, context⟩
⟨*latex⟩
```

Check whether we're in front of the (braced) environment name (in LATEX), and consume it.

```
\def\collargs@bii{%
  \ifcollargs@verbatimbraces
    \expandafter\collargs@biii
  \else
    \ifcollargsIgnoreOtherTags
```

We shouldn't check the name in this case, because it was already checked, and consumed.

```
      \expandafter\expandafter\expandafter\collargs@biii
    \else
      \expandafter\expandafter\expandafter\collargs@b@checkend
    \fi
  \fi
}
```

```
\def\collargs@b@checkend#1.{%
  \collargs@grabspaces{\collargs@b@checkend@i#1.}%
}
\def\collargs@b@checkend@i#1.#2{%
  \def\collargs@temp{#2}%
  \ifx\collargs@temp\collargs@b@envname
  \else
    \collargs@b@checkend@error
  \fi
  \collargs@biii#1.%
}
\def\collargs@b@checkend@error{%
  \PackageError{collargs}{Environment "\collargs@b@envname" ended as
    "\collargs@temp"}{}%
}
⟨/latex⟩
```

This macro stores the collected body.

```
\def\collargs@biii{%
```

Define the wrapper macro (`\collargs@temp`).

```
  \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.

```
  \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
```

Reexecute the central loop.

```
  \collargs@
}
\def\collargs@b@def@wrapper{%
⟨latex⟩  \edef\collargs@temp{{\collargs@b@envname}}%
  \edef\collargs@temp{%
```

Was the begin-tag requested?

```
    \ifcollargsBeginTag
```

`\collargs@begintag` is already adapted to the format and the verbatim mode.

```
      \expandonce\collargs@begintag
```

Add the braced environment name in LaTeX in the regular and partial verbatim mode.

```
⟨*latex⟩
      \ifcollargs@verbatimbraces\else\collargs@temp\fi
⟨/latex⟩
    \fi
```

This is the body.

```
    ####1%
```

Rinse and repeat for the end-tag.

```
    \ifcollargsEndTag
      \expandonce\collargs@endtag
⟨*latex⟩
      \ifcollargs@verbatimbraces\else\collargs@temp\fi
⟨/latex⟩
    \fi
  }%
  \expandafter\collargs@wrap\expandafter{\collargs@temp}%
}
```

\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.

```
\def\collargs@readContent{%
```

Define macro which will search for the first begin-tag.

```
  \ifcollargs@long\long\fi
  \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.

```
    \expandafter\toks@\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.

```
    \futurelet\collargs@temp\collargs@gobbleOneB@i
  }%
```

Define macro which will search for the first end-tag. We make it long if so required (by +).

```
  \ifcollargs@long\long\fi
  \collargs@CPT@def\collargs@gobbleUntilE\collargs@endtag{%
```

Expand \collargs@empty at the start of ##1.

```
    \expandafter\toksapp\expandafter\toks@\expandafter{##1}%
    \collargs@gobbleUntilE@i
  }%
```

Initialize.

```
  \collargs@begins=0\relax
  \collargsArg{}%
  \toks@{}%
```

We will call \collargs@gobbleUntilE via the caller control sequence.

```
  \collargs@letusecollector\collargs@gobbleUntilE
```

We insert \collargs@empty to avoid the potential debracing problem.

```
  \collargs@empty
}
```

How many begin-tags do we have opened?

```
\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.

```
\def\collargs@CPT@def#1#2{%
  \expandafter\def\expandafter#1%
  \expandafter##\expandafter1#2%
}
```

A quark quard.

```
\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.

```
\def\collargs@gobbleOneB@i{%
  \def\collargs@begins@increment{1}%
  \ifx\collargs@qend\collargs@temp
```

We have reached the fake begin-tag. Note that we found the end-tag.

```
    \def\collargs@begins@increment{-1}%
```

Gobble the quark guard.

```
    \expandafter\collargs@gobbleOneB@v
  \else
```

Append the real begin-tag to the temporary tokens.

```
    \etoksapp\toks@{\expandonce\collargs@begintag}%
    \expandafter\collargs@gobbleOneB@ii
  \fi
}%
```

Do we have to check the tag argument (i.e. the environment name after `\begin`)?

```
\def\collargs@gobbleOneB@ii{%
  \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
    \expandafter\collargs@gobbleOneB@vi
  \else
```

Yup, so let's (carefully) collect the tag argument.

```
    \expandafter\collargs@gobbleOneB@iii
  \fi
}
\def\collargs@gobbleOneB@iii{%
  \collargs@grabspaces{%
    \collargs@letusecollector\collargs@gobbleOneB@iv
  }%
}
\def\collargs@gobbleOneB@iv#1{%
  \def\collargs@temp{#1}%
  \ifx\collargs@temp\collargs@tagarg
```

This is the tag argument we've been waiting for!

```
  \else
```

Nope, this `\begin` belongs to someone else.

```
    \def\collargs@begins@increment{0}%
  \fi
```

Whatever the result was, we have to append the gobbled group to the temporary toks.

```
  \etoksapp\toks@{\collargs@grabbed@spaces\unexpanded{{#1}}}%
  \collargs@init@grabspaces
  \collargs@gobbleOneB@vi
}
\def\collargs@gobbleOneB@v#1{\collargs@gobbleOneB@vi}
\def\collargs@gobbleOneB@vi{%
```

Store.

```
  \etoksapp\collargsArg{\the\toks@}%
```

Advance the begin-tag counter.

```
  \advance\collargs@begins\collargs@begins@increment\relax
```

Find more begin-tags, unless this was the final one.

```
  \ifnum\collargs@begins@increment=-1
  \else
    \expandafter\collargs@gobbleOneB\expandafter\collargs@empty
  \fi
}
\def\collargs@gobbleUntilE@i{%
```

Do we have to check the tag argument (i.e. the environment name after `\end`)?

```
  \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
    \expandafter\collargs@gobbleUntilE@iv
  \else
```

Yup, so let's (carefully) collect the tag argument.

```
    \expandafter\collargs@gobbleUntilE@ii
  \fi
}
\def\collargs@gobbleUntilE@ii{%
  \collargs@grabspaces{%
    \collargs@letusecollector\collargs@gobbleUntilE@iii
  }%
}
\def\collargs@gobbleUntilE@iii#1{%
  \etoksapp\toks@{\collargs@grabbed@spaces}%
  \collargs@init@grabspaces
  \def\collargs@tempa{#1}%
  \ifx\collargs@tempa\collargs@tagarg
```

This is the tag argument we've been waiting for!

```
    \expandafter\collargs@gobbleUntilE@iv
  \else
```

Nope, this `\end` belongs to someone else. Insert the end tag plus the tag argument, and collect until the next `\end`.

```
    \expandafter\toksapp\expandafter\toks@\expandafter{\collargs@endtag{#1}}%
    \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
  \fi
}
\def\collargs@gobbleUntilE@iv{%
```

Invoke `\collargs@gobbleOneB` with the collected material, plus a fake begin-tag and a quark guard.

```
  \ifcollargsIgnoreNesting
    \expandafter\collargsArg\expandafter{\the\toks@}%
    \expandafter\collargs@commandatend
  \else
    \expandafter\collargs@gobbleUntilE@v
  \fi
}
\def\collargs@gobbleUntilE@v{%
  \expanded{%
```

```
    \noexpand\collargs@letusecollector\noexpand\collargs@gobbleOneB
    \noexpand\collargs@empty
    \the\toks@
```

Add a fake begin-tag and a quark guard.

```
    \expandonce\collargs@begintag
    \noexpand\collargs@qend
  }%
  \ifnum\collargs@begins<0
    \expandafter\collargs@commandatend
  \else
    \etoksapp\collargsArg{%
      \expandonce\collargs@endtag
      \expandafter\ifx\expandafter\relax\collargs@tagarg\relax\else{%
          \expandonce\collargs@tagarg}\fi
    }%
    \toks@={}%
    \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
    \expandafter\collargs@empty
  \fi
}
```

`\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.

```
\def\collargs@e{%
```

We open an extra group, because `\collargs@appendarg` will close a group for each embellishment.

```
  \global\collargs@fix@requestedtrue
  \begingroup
  \ifcollargs@verbatim
    \expandafter\collargs@e@verbatim
  \else
    \expandafter\collargs@e@i
  \fi
}
```

Detokenize the embellishment tokens in the verbatim mode.

```
\def\collargs@e@verbatim#1{%
  \expandafter\collargs@e@i\expandafter{\detokenize{#1}}%
}
```

Ungroup the embellishment tokens, separating them from the rest of the argument specification by a dot.

```
\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.

```
\def\collargs@e@ii#1.#2.{%
  \collargs@grabspaces{\collargs@e@iii#1.#2.}%
}
```

What's the argument token?

```
\def\collargs@e@iii#1.#2.{%
  \def\collargs@e@iv{\collargs@e@v#1.#2.}%
  \futurelet\collargs@temp\collargs@e@iv
}
```

If it is a open or close group character, we surely don't have an embellishment.

```
\def\collargs@e@v{%
  \ifcat\noexpand\collargs@temp\bgroup\relax
    \let\collargs@marshal\collargs@e@z
  \else
    \ifcat\noexpand\collargs@temp\egroup\relax
      \let\collargs@marshal\collargs@e@z
    \else
      \let\collargs@marshal\collargs@e@vi
    \fi
  \fi
  \collargs@marshal
}
```

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.

```
\newif\ifcollargs@in@
\def\collargs@in@#1#2{%
 \def\collargs@in@@##1#1##2##3\collargs@in@@{%
   \ifx\collargs@in@##2\global\collargs@in@false\else\global\collargs@in@true\fi
 }%
 {\collargs@in@@#2#1\collargs@in@\collargs@in@@}%
}
```

Let' see whether the following token, now `#3`, is an embellishment token.

```
\def\collargs@e@vi#1.#2.#3{%
  \collargs@in@{#3}{#1}%
  \ifcollargs@in@
    \expandafter\collargs@e@vii
  \else
    \expandafter\collargs@e@z
  \fi
  #1.#2.#3%
}
```

`#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`.

```
\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.

```
  \let\collargs@real@appendarg\collargs@appendarg
  \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.

```
  \def\collargs@{\collargs@e@viii#1.#3}%
  \collargs@m#2.%
}
```

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.

```
\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).

```
  \collargs@wrap{#2##1}%
```

We need to get rid of the current embellishment from embellishments, not to catch the same embellishment twice.

```
  \def\collargs@e@ix##1#2{\collargs@e@x##1}%
  \collargs@e@ix#1.#3.%
}
```

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.

```
\def\collargs@e@x{%
```

Process the argument and append it to the storage.

```
  \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.

```
  \begingroup
  \collargs@e@ii
}
```

The first argument token in 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.

```
\def\collargs@e@z#1.{\endgroup\endgroup\collargs@}
```

`\collargs@E` Discard the defaults and execute `e`.

```
\def\collargs@E#1#2{\collargs@e{#1}}
```

8.2.5 The verbatim modes

`\collargsVerbatim` `\collargsVerb` `\collargsNoVerbatim` These macros set the two verbatim-related conditionals, `\ifcollargs@verbatim` and `\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.

```
\let\collargs@NoVerbatimAfterNoVerbatim\relax
\def\collargs@VerbAfterNoVerbatim{%
  \collargs@verbatimtrue
  \collargs@verbatimbracesfalse
  \collargs@make@verbatim
  \collargs@after{Verb}%
}
\def\collargs@VerbatimAfterNoVerbatim{%
  \collargs@verbatimtrue
  \collargs@verbatimbracestrue
  \collargs@make@verbatim
  \collargs@after{Verbatim}%
}
\def\collargs@NoVerbatimAfterVerb{%
  \collargs@verbatimfalse
  \collargs@verbatimbracesfalse
  \collargs@make@other@groups
  \collargs@make@no@verbatim
  \collargs@after{NoVerbatim}%
}
\def\collargs@VerbAfterVerb{%
  \collargs@make@other@groups
}
\def\collargs@VerbatimAfterVerb{%
  \collargs@verbatimbracestrue
  \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters of category “other”.

```
  \def\collargs@do##1{\catcode##1=12 }%
  \collargs@bgroups
  \collargs@egroups
  \collargs@after{Verbatim}%
}%
\let\collargs@NoVerbatimAfterVerbatim\collargs@NoVerbatimAfterVerb
\def\collargs@VerbAfterVerbatim{%
  \collargs@verbatimbracesfalse
  \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters be of their normal category.

```
  \def\collargs@do##1{\catcode##1=1 }%
  \collargs@bgroups
  \def\collargs@do##1{\catcode##1=2 }%
  \collargs@egroups
  \collargs@after{Verb}%
}%
\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.

```
\def\collargs@after#1{%
  \letcs\collargsVerbatim{collargs@VerbatimAfter#1}%
  \letcs\collargsVerb{collargs@VerbAfter#1}%
  \letcs\collargsNoVerbatim{collargs@NoVerbatimAfter#1}%
}
```

The first transition is always from the non-verbatim mode.

```
\collargs@after{NoVerbatim}
```

`\collargs@bgroups` `\collargs@egroups` Initialize the lists of the current grouping characters used in the redefinitions of macros `\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.

```
\def\collargs@bgroups{}%
\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.)

```
\ifdefined\luatexversion
```

```
  \def\collargs@cc#1{%
    \directlua{tex.sprint(tex.getcatcode(\collargs@catcodetable@original,
      \the\numexpr\expandafter`\csname#1\endcsname\relax))}%
  }
\else
  \def\collargs@cc#1{%
    \ifcsname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
      \csname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
    \else
      12%
    \fi
  }
\fi
```

\collargs@other@bgroup
\collargs@other@egroup
\collargsBraces

Macros `\collargs@other@bgroup` and `\collargs@other@egroup` hold the characters of category code "other" which will play the role of grouping characters in the 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 `\collargsBraces` (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 `\collargsBraces` merely sets `\collargs@make@other@groups`, which gets executed by the subsequent `\collargsVerbatim`, `\collargsVerb` or `\collargsNoVerbatim` (either directly or via `\collargs@make@verbatim`).

```
\def\collargsBraces#1{%
  \expandafter\collargs@braces@i\detokenize{#1}\relax
}
\def\collargs@braces@i#1#2#3\relax{%
  \def\collargs@make@other@groups{%
    \def\collargs@other@bgroup{#1}%
    \def\collargs@other@egroup{#2}%
  }%
}
\def\collargs@make@other@groups{}
```

\collargs@catcodetable@verbatim
\catcodetable@atletter
\collargs@catcodetable@initex

We declare several new catcode tables in LuaTEX, the most important one being `\collargs@catcodetable@verbatim`, where all characters have 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.

```
          \ifdefined\luatexversion
          ⟨*latex, context⟩
            \newcatcodetable\collargs@catcodetable@verbatim
⟨latex⟩     \let\collargs@catcodetable@atletter\catcodetable@atletter
⟨context⟩   \newcatcodetable\collargs@catcodetable@atletter
          ⟨/latex, context⟩
          ⟨*plain⟩
            \ifdefined\collargs@catcodetable@verbatim\else
              \chardef\collargs@catcodetable@verbatim=4242
            \fi
            \chardef\collargs@catcodetable@atletter=%
              \number\numexpr\collargs@catcodetable@verbatim+1\relax
            \chardef\collargs@catcodetable@initex=%
              \number\numexpr\collargs@catcodetable@verbatim+2\relax
              \initcatcodetable\collargs@catcodetable@initex
          ⟨/plain⟩
⟨plain, context⟩ \savecatcodetable\collargs@catcodetable@atletter
            \begingroup
            \@firstofone{%
⟨latex⟩       \catcodetable\catcodetable@initex
⟨plain⟩       \catcodetable\collargs@catcodetable@initex
⟨context⟩     \catcodetable\inicatcodes
```

```
    \catcode`\\=12
    \catcode13=12
    \catcode0=12
    \catcode32=12
    \catcode`\%=12
    \catcode127=12
    \def\collargs@do#1{\catcode#1=12 }%
    \collargs@forrange{`\a}{`\z}%
    \collargs@forrange{`\A}{`\Z}%
    \savecatcodetable\collargs@catcodetable@verbatim
    \endgroup
  }%
\fi
```

`verbatim ranges` `\collargsVerbatimRanges` `\collargs@verbatim@ranges` This key and macro set the character ranges to which the verbatim mode will apply (in pdfTEX and XƎTEX), or which will be inspected for grouping and comment characters (in LuaTEX). In pdfTEX, the default value `0-255` should really remain unchanged.

```
\collargsSet{
  verbatim ranges/.store in=\collargs@verbatim@ranges,
}
\def\collargsVerbatimRanges#1{\def\collargs@verbatim@ranges{#1}}
\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 LuaTEX, 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 XƎTEX, we perform the change for characters in `verbatim ranges`.

```
\ifdefined\luatexversion
  \def\collargs@make@verbatim{%
    \directlua{%
      for from, to in string.gmatch(
        "\luaescapestring{\collargs@verbatim@ranges}",
        "(\collargs@percentchar d+)-(\collargs@percentchar d+)"
      ) do
        for char = tex.round(from), tex.round(to) do
          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 LuaTEX.

```
          if catcode == 1 then
            tex.sprint(
              \number\collargs@catcodetable@atletter,
              "\noexpand\\collargs@make@verbatim@bgroup{" .. char .. "}")
          elseif catcode == 2 then
            tex.sprint(
              \number\collargs@catcodetable@atletter,
              "\noexpand\\collargs@make@verbatim@egroup{" .. char .. "}")
          elseif catcode == 14 then
            tex.sprint(
              \number\collargs@catcodetable@atletter,
              "\noexpand\\collargs@make@verbatim@comment{" .. char .. "}")
          end
        end
      end
    }%
    \edef\collargs@catcodetable@original{\the\catcodetable}%
    \catcodetable\collargs@catcodetable@verbatim
```

Even in LuaTEX, we switch between the verbatim braces regimes by hand.

```
    \ifcollargs@verbatimbraces
    \else
      \def\collargs@do##1{\catcode##1=1\relax}%
      \collargs@bgroups
      \def\collargs@do##1{\catcode##1=2\relax}%
      \collargs@egroups
    \fi
  }
\else
```

The non-LuaTEX version:

```
  \def\collargs@make@verbatim{%
    \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.

```
      \collargs@make@other@groups
      \def\collargs@make@other@groups{}%
      \let\collargs@make@other@bgroup\@gobble
      \let\collargs@make@other@egroup\@gobble
    }%
```

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).

```
    \gdef\collargs@comments{}%
    \let\collargs@do\collargs@make@verbatim@char
    \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
  }
  \def\collargs@make@verbatim@char#1{%
```

Store the current category code of the current character.

```
    \ifnum\catcode#1=12
    \else
      \csedef{collargs@cc@#1}{\the\catcode#1}%
    \fi
    \ifnum\catcode#1=1
      \collargs@make@verbatim@bgroup{#1}%
    \else
      \ifnum\catcode#1=2
        \collargs@make@verbatim@egroup{#1}%
      \else
        \ifnum\catcode#1=14
          \collargs@make@verbatim@comment{#1}%
        \fi
```

Change the category code of the current character (including the comment characters).

```
        \ifnum\catcode#1=12
        \else
          \catcode#1=12\relax
        \fi
      \fi
    \fi
  }
\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 `\relax`ed at the top of `\collargs@make@verbatim` as a consequence of the user executing `\collargsBraces`.

```
\def\collargs@make@verbatim@bgroup#1{%
  \ifcollargs@verbatimbraces
    \catcode#1=12\relax
  \fi
  \appto\collargs@bgroups{\collargs@do{#1}}%
  \collargs@make@other@bgroup{#1}%
}
\def\collargs@make@other@bgroup#1{%
  \collargs@make@char\collargs@other@bgroup{#1}{12}%
  \let\collargs@make@other@bgroup\@gobble
}
```

`\collargs@make@verbatim@egroup` Ditto for the closing group character.

```
\def\collargs@make@verbatim@egroup#1{%
  \ifcollargs@verbatimbraces
    \catcode#1=12\relax
  \fi
  \appto\collargs@egroups{\collargs@do{#1}}%
  \collargs@make@other@egroup{#1}%
}
\def\collargs@make@other@egroup#1{%
  \collargs@make@char\collargs@other@egroup{#1}{12}%
  \let\collargs@make@other@egroup\@gobble
}
```

`\collargs@make@verbatim@comment` This macro populates `\collargs@make@comments@other`.

```
\def\collargs@make@verbatim@comment#1{%
  \gappto\collargs@comments{\collargs@do{#1}}%
}
```

`\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.

```
\ifdefined\luatexversion
  \def\collargs@make@no@verbatim{%
    \catcodetable\collargs@catcodetable@original\relax
  }%
\else
\def\collargs@make@no@verbatim{%
  \let\collargs@do\collargs@make@no@verbatim@char
  \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
}
\fi
\def\collargs@make@no@verbatim@char#1{%
```

The original category code of a characted was stored into `\collargs@cc@`⟨*character code*⟩ by `\collargs@make@verbatim`. (We don't use `\collargs@cc`, because we have a number.)

```
  \ifcsname collargs@cc@#1\endcsname
    \catcode#1=\csname collargs@cc@#1\endcsname\relax
```

We don't have to restore category code 12.

```
  \fi
}
```

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.

```
\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.

```
\long\def\collargs@fix#1{%
```

Going through `\edef` + `\unexpanded` avoids doubling the hashes.

```
  \edef\collargs@fix@next{\unexpanded{#1}}%
  \ifcollargs@fix@requested
    \letcs\collargs@action{collargs@fix@%
      \ifcollargs@last@verbatim
        \ifcollargs@last@verbatimbraces V\else v\fi
      \else
        N%
      \fi
      to%
      \ifcollargs@verbatim
        \ifcollargs@verbatimbraces V\else v\fi
      \else
        N%
      \fi
    }%
  \else
    \let\collargs@action\collargs@fix@next
  \fi
  \collargs@action
}
```

\collargs@fix@NtoN \collargs@fix@vtov \collargs@fix@VtoV Nothing to do, continue with the next-code.

```
\def\collargs@fix@NtoN{\collargs@fix@next}
\let\collargs@fix@vtov\collargs@fix@NtoN
\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`.

```
\def\collargs@fix@Ntov{%
  \futurelet\collargs@temp\collargs@fix@cc@to@other@ii
}
\def\collargs@fix@cc@to@other@ii{%
  \ifcat\noexpand\collargs@temp\bgroup
    \let\collargs@action\collargs@fix@next
  \else
    \ifcat\noexpand\collargs@temp\egroup
      \let\collargs@action\collargs@fix@next
    \else
      \let\collargs@action\collargs@fix@NtoV
    \fi
  \fi
  \collargs@action
}
```

\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.

```
\def\collargs@fix@NtoV{%
  \ifcollargs@double@fix
    \ifcollargs@in@second@fix
      \expandafter\expandafter\expandafter\collargs@fix@NtoV@secondfix
    \else
      \expandafter\expandafter\expandafter\collargs@fix@NtoV@onemore
    \fi
  \else
    \expandafter\collargs@fix@NtoV@singlefix
  \fi
}
```

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)

```
\def\collargs@fix@NtoV@singlefix{%
  \expandafter\collargs@fix@next\string
}
```

If this is the first fix of two, we know `#1` is a control sequence, so it is safe to grab it.

```
\def\collargs@fix@NtoV@onemore#1{%
  \collargs@do@one@more@fix{%
    \expandafter\collargs@fix@next\string#1%
  }%
}
```

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.

```
\def\collargs@fix@NtoV@secondfix{%
  \if\noexpand\collargs@temp\relax
    \expandafter\collargs@fix@NtoV@secondfix@i
  \else
    \expandafter\collargs@fix@NtoV@singlefix
  \fi
}
\def\collargs@fix@NtoV@secondfix@i#1{%
  \gdef\collargs@double@fix@cs@ii{#1}%
  \collargs@fix@NtoV@singlefix#1%
}
```

\collargs@fix@vtoN Do nothing for the grouping tokens, redirect to `\collargs@fix@VtoN` for other tokens.

```
\def\collargs@fix@vtoN{%
  \futurelet\collargs@token\collargs@fix@vtoN@i
}
\def\collargs@fix@vtoN@i{%
  \ifcat\noexpand\collargs@token\bgroup
    \expandafter\collargs@fix@next
  \else
    \ifcat\noexpand\collargs@token\egroup
      \expandafter\expandafter\expandafter\collargs@fix@next
    \else
      \expandafter\expandafter\expandafter\collargs@fix@VtoN
    \fi
  \fi
}
```

\collargs@fix@vtoV Redirect group tokens to `\collargs@fix@NtoV`, and do nothing for other tokens.

```
\def\collargs@fix@vtoV{%
  \futurelet\collargs@token\collargs@fix@vtoV@i
}
\def\collargs@fix@vtoV@i{%
  \ifcat\noexpand\collargs@token\bgroup
    \expandafter\collargs@fix@NtoV
  \else
    \ifcat\noexpand\collargs@token\egroup
      \expandafter\expandafter\expandafter\collargs@fix@NtoV
    \else
      \expandafter\expandafter\expandafter\collargs@fix@next
    \fi
  \fi
}
```

\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.

```
\def\collargs@fix@catcode@of@braces@fromverbatim#1{%
  \ifnum\catcode`#1=1
    \expandafter\collargs@fix@VtoN
    \expandafter#1%
  \else
    \ifnum\catcode`#1=2
      \expandafter\expandafter\expandafter\collargs@fix@cc@VtoN
      \expandafter\expandafter\expandafter#1%
    \else
      \expandafter\expandafter\expandafter\collargs@fix@next
    \fi
  \fi
}
```

\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.

```
\def\collargs@fix@VtoN#1{%
```

```
  \ifnum\catcode`#1=0
    \expandafter\collargs@fix@VtoN@escape
  \else
    \ifnum\catcode`#1=14
      \expandafter\expandafter\expandafter\collargs@fix@VtoN@comment
    \else
      \expandafter\expandafter\expandafter\collargs@fix@VtoN@token
    \fi
  \fi
  #1%
}
```

`\collargs@fix@VtoN@token` We create a new character with the current category code behing the next-code. This works even for grouping characters.

```
\def\collargs@fix@VtoN@token#1{%
  \collargs@insert@char\collargs@fix@next{`#1}{\the\catcode`#1}%
}
```

`\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.

```
\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.

```
  \def#1##1{%
    \begingroup%
    \escapechar=`\\%
    \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.

```
    \def\collargs@do####1{\catcode####1=12 }%
    \collargs@comments
    \csname\string#1\endcsname%
  }%
  \begingroup%
  \escapechar=`\\%
  \lccode`\~=\endlinechar%
  \lowercase{%
    \expandafter\endgroup
    \expandafter\def\csname\string#1\endcsname##1~%
  }{%
```

I have removed `\space` from the end of the following line. We don't want it for our application.

```
    \endgroup#2%
  }%
}
\collargs@defcommentstripper\collargs@fix@VtoN@comment{%
  \collargs@fix@next
}
```

We don't need the generator any more.

```
\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`.

```
\def\collargs@fix@VtoN@escape#1{%
  \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.

```
    \expandafter\collargs@fix@VtoN@escape@d
  \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.

```
    \expandafter\collargs@fix@VtoN@escape@i
  \fi
}
\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.

```
  \begingroup
  \catcode`\\=12
  \catcode`\{=12
  \catcode`\}=12
  \catcode`\ =12
  \collargs@fix@VtoN@escape@ii
}
```

The argument is the first character of the control sequence name.

```
\def\collargs@fix@VtoN@escape@ii#1{%
  \endgroup
  \def\collargs@csname{#1}%
```

Only if `#1` is a letter may the control sequence name continue.

```
  \ifnum\catcode`#1=11
    \expandafter\collargs@fix@VtoN@escape@iii
  \else
```

In the case of a control space, we have to throw away the following spaces.

```
    \ifnum\catcode`#1=10
      \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@s
    \else
```

We have a control symbol. That means that we haven't peeked ahead and can thus skip `\collargs@fix@VtoN@escape@z`.

```
      \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@z@i
    \fi
  \fi
}
```

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).

```
\def\collargs@fix@VtoN@escape@iii{%
```

```
  \futurelet\collargs@temp\collargs@fix@VtoN@escape@iv
}
\def\collargs@fix@VtoN@escape@iv{%
  \ifcat\noexpand\collargs@temp\bgroup
    \let\collargs@action\collargs@fix@VtoN@escape@z
  \else
    \ifcat\noexpand\collargs@temp\egroup
      \let\collargs@action\collargs@fix@VtoN@escape@z
    \else
      \expandafter\ifx\space\collargs@temp
        \let\collargs@action\collargs@fix@VtoN@escape@s
      \else
        \let\collargs@action\collargs@fix@VtoN@escape@v
      \fi
    \fi
  \fi
  \collargs@action
}
```

If we have a letter, store it and loop back, otherwise finish.

```
\def\collargs@fix@VtoN@escape@v#1{%
  \ifcat\noexpand#1a%
    \appto\collargs@csname{#1}%
    \expandafter\collargs@fix@VtoN@escape@iii
  \else
    \expandafter\collargs@fix@VtoN@escape@z\expandafter#1%
  \fi
}
```

Throw away the following spaces.

```
\def\collargs@fix@VtoN@escape@s{%
  \futurelet\collargs@temp\collargs@fix@VtoN@escape@s@i
}
\def\collargs@fix@VtoN@escape@s@i{%
  \expandafter\ifx\space\collargs@temp
    \expandafter\collargs@fix@VtoN@escape@s@ii
  \else
    \expandafter\collargs@fix@VtoN@escape@z
  \fi
}
\def\collargs@fix@VtoN@escape@s@ii{%
  \expandafter\collargs@fix@VtoN@escape@z\romannumeral-0%
}
```

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.

```
\def\collargs@fix@VtoN@escape@z@i{%
  \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
  \collargs@fix@VtoN@escape@z@ii
}%
\def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin{%
  \ifcsname\collargs@csname\endcsname
    \@tempswatrue
  \else
    \@tempswafalse
  \fi
}
\def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end{%
```

```
  \if@tempswa
  \else
    \cslet{\collargs@csname}\collargs@undefined
  \fi
}
\def\collargs@fix@VtoN@escape@z@ii{%
  \expandafter\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
  \expandafter\collargs@fix@next\csname\collargs@csname\endcsname
}
```

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.

```
\def\collargs@fix@VtoN@escape@z{%
  \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.

```
    \collargs@do@one@more@fix\collargs@fix@VtoN@escape@z@i
  }{%
```

Remember the collected control sequence. It will be used in `\collargs@cancel@double@fix`.

```
    \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
    \xdef\collargs@double@fix@cs@i{\expandonce{\csname\collargs@csname\endcsname}}%
    \collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
```

Request the double-fix.

```
    \global\collargs@double@fixtrue
```

The complication is addressed, redirect to the control symbol finish.

```
    \collargs@fix@VtoN@escape@z@ii
  }%
}
```

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`.

```
\def\collargs@fix@VtoN@escape@d{%
  \ifcollargs@in@second@fix
    \expandafter\collargs@fix@VtoN@escape@d@i
      \expandafter\collargs@double@fix@cs@ii
  \else
    \expandafter\collargs@fix@VtoN@escape@d@i
      \expandafter\collargs@double@fix@cs@i
  \fi
}
```

We have the contents of either `\collargs@double@fix@cs@i` or `\collargs@double@fix@cs@ii` here, a control sequence in both cases.

```
\def\collargs@fix@VtoN@escape@d@i#1{%
  \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@d@ii
    \expandafter\string#1\relax
}
```

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`.

```
\def\collargs@fix@VtoN@escape@d@ii#1#2\relax{%
  \def\collargs@csname{#2}%
  \def\collargs@fix@VtoN@escape@d@iii#2{%
    \collargs@fix@VtoN@escape@z
  }%
  \collargs@fix@VtoN@escape@d@iii
}
```

This conditional signals a double-fix request. It should be always set globally, because it is cleared by `\collargs@double@fixfalse` in a group.

```
\newif\ifcollargs@double@fix
```

This conditional signals that we're currently performing the second fix.

```
\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.

```
\def\collargs@if@one@more@fix{%
  \ifcollargs@double@fix
    \ifcollargs@in@second@fix
      \expandafter\expandafter\expandafter\@secondoftwo
    \else
      \expandafter\expandafter\expandafter\@firstoftwo
    \fi
  \else
    \expandafter\@secondoftwo
  \fi
}
\def\collargs@do@one@more@fix#1{%
```

We perform the second fix in a group, signalling that we're performing it.

```
  \begingroup
  \collargs@in@second@fixtrue
```

Reexecute the fixing routine, at the end, close the group and execute the given code afterwards.

```
  \collargs@fix{%
    \endgroup
    #1%
  }%
}
```

This macro is called from `\collargs@appendarg` to cancel the double-fix request.

```
\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.

```
  \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".

```
  \begingroup
```

This will attach the delimiters directly to the argument, so we'll see what was actually consumed.

```
  \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.

```
  \edef\collargs@temp{\the\collargsArg}%
  \edef\collargs@tempa{\expandafter\string\collargs@double@fix@cs@i}%
  \ifx\collargs@temp\collargs@tempa
    \global\collargs@fix@requestedtrue
  \fi
  \endgroup
}
```

`\collargs@insert@char` `\collargs@make@char` These macros create a character of character code `#2` and category code `#3`. The first macro 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 LuaTEX, XƎTEX and LATEX where possible. In the end, we only have to implement our own macros for plain pdfTEX.

```
⟨!context⟩\ifdefined\luatexversion
  \def\collargs@insert@char#1#2#3{%
    \edef\collargs@temp{\unexpanded{#1}}%
    \expandafter\collargs@temp\directlua{%
      tex.cprint(\number#3,string.char(\number#2))}%
  }%
  \def\collargs@make@char#1#2#3{%
    \edef#1{\directlua{tex.cprint(\number#3,string.char(\number#2))}}%
  }%
⟨*!context⟩
\else
  \ifdefined\XeTeXversion
    \def\collargs@insert@char#1#2#3{%
      \edef\collargs@temp{\unexpanded{#1}}%
      \expandafter\collargs@temp\Ucharcat #2 #3
    }%
    \def\collargs@make@char#1#2#3{%
      \edef#1{\Ucharcat#2 #3}%
    }%
  \else
  ⟨*latex⟩
    \ExplSyntaxOn
    \def\collargs@insert@char#1#2#3{%
      \edef\collargs@temp{\unexpanded{#1}}%
      \expandafter\expandafter\expandafter\collargs@temp\char_generate:nn{#2}{#3}%
    }%
    \def\collargs@make@char#1#2#3{%
      \edef#1{\char_generate:nn{#2}{#3}}%
    }%
    \ExplSyntaxOff
```

```
⟨/latex⟩
⟨*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.

```
\begingroup
\catcode`\^^@=1  \csgdef{collargs@charofcat@1}{%
  \noexpand\expandafter^^@\iffalse}\fi}
\catcode`\^^@=2  \csgdef{collargs@charofcat@2}{\iffalse{\fi^^@}
\catcode`\^^@=3  \csgdef{collargs@charofcat@3}{^^@}
\catcode`\^^@=4  \csgdef{collargs@charofcat@4}{^^@}
```

As we have grabbed the spaces already, a remaining newline should surely be fixed into a `\par`.

```
                 \csgdef{collargs@charofcat@5}{\par}
\catcode`\^^@=6  \csxdef{collargs@charofcat@6}{\unexpanded{^^@}}
\catcode`\^^@=7  \csgdef{collargs@charofcat@7}{^^@}
\catcode`\^^@=8  \csgdef{collargs@charofcat@8}{^^@}
                 \csgdef{collargs@charofcat@10}{\noexpand\space}
\catcode`\^^@=11 \csgdef{collargs@charofcat@11}{^^@}
\catcode`\^^@=12 \csgdef{collargs@charofcat@12}{^^@}
\catcode`\^^@=13 \csgdef{collargs@charofcat@13}{^^@}
\endgroup
\def\collargs@insert@char#1#2#3{%
```

Temporarily change the lowercase code of `^^@` to the requested character `#2`.

```
  \begingroup
  \lccode`\^^@=#2\relax
```

We'll have to close the group before executing the next-code.

```
  \def\collargs@temp{\endgroup#1}%
```

`\collargs@charofcat@`⟨*requested category code*⟩ is f-expanded first, leaving us to lowercase `\expandafter\collargs@temp^^@`. Clearly, lowercasing `\expandafter\collargs@temp` is a no-op, but lowercasing `^^@` 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.

```
  \expandafter\lowercase\expandafter{%
    \expandafter\expandafter\expandafter\collargs@temp
    \romannumeral-`0\csname collargs@charofcat@\the\numexpr#3\relax\endcsname
  }%
}
```

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.

```
\def\collargs@make@char#1#2#3{%
  \begingroup
  \lccode`\^^@=#2\relax
```

Define `\collargs@temp` to hold `^^@` of the appropriate category.

```
  \edef\collargs@temp{%
    \csname collargs@charofcat@\the\numexpr#3\relax\endcsname}%
```

Preexpand the second `\collargs@temp` so that we lowercase `\def\collargs@temp{^^@}`, with `^^@` of the appropriate category.

```
  \expandafter\lowercase\expandafter{%
```

```
        \expandafter\def\expandafter\collargs@temp\expandafter{\collargs@temp}%
      }%
      \expandafter\endgroup
      \expandafter\def\expandafter#1\expandafter{\collargs@temp}%
    }
  ⟨/plain⟩
  \fi
\fi
⟨/!context⟩
```

```
⟨plain⟩\resetatcatcode
⟨context⟩\stopmodule
⟨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

```
use strict;
use Getopt::Long;
use Path::Class;
use File::Spec;
use File::Basename;
use PDF::API2;

my $VERSION = '2023/10/10␣v1.0.0';

my $usage = "usage:␣memoize-extract.pl␣[-h]␣[--pdf␣PDF]␣[--prune]␣[--keep]␣[--force]␣[--log␣LOG]␣[--
    →warning-template␣WARNING_TEMPLATE]␣[--quiet]␣[--mkdir]␣mmz\n";
my $Help = <<END;
Extract extern pages out of the document PDF.

positional arguments:
  mmz the record file produced by Memoize: doc.mmz when compiling doc.tex

options:
  --help, -h show this help message and exit
  --version, -V show version and exit
  --pdf PDF, -P PDF extract from file PDF
  --prune, -p remove the extern pages after extraction
  --keep, -k do not mark externs as extracted
  --force, -f extract even if the size-check fails
  --log LOG, -l LOG the log file
  --warning-template WARNING_TEMPLATE, -w WARNING_TEMPLATE
                     \warningtext in the template will be replaced by the warning message
  --quiet, -q don't␣describe␣what's happening
  --embedded, -e prefix all messages to the standard output with the script name
  --mkdir, -m create a directory (and exit)

For details, see the man page or the Memoize documentation.
END

my ($pdf_arg, $prune, $keep, $log, $quiet, $embedded, $force, $mkdir, $help, $print_version);
my $warning_template = '\warningtext';
Getopt::Long::Configure ("bundling");
GetOptions(
    "pdf|P=s" => \$pdf_arg,
    "keep|k" => \$keep,
    "prune|p" => \$prune,
    "log|l=s" => \$log,
    "quiet|q" => \$quiet,
    "embedded|e" => \$embedded,
    "force|f" => \$force,
    "warning-template|w=s" => \$warning_template,
    "mkdir|m" => \$mkdir,
    "help|h|?" => \$help,
    "version|V" => \$print_version,
    ) or die $usage;
if ($help) {print("$usage\n$Help"); exit 0}
if ($print_version) { print("memoize-extract.pl␣of␣Memoize␣$VERSION\n"); exit 0 }
die $usage unless @ARGV == 1;

my $message_prefix = $embedded ? basename($0) . ':␣' : '';
print("\n") if ($embedded);

my @output_paths = (dir()->absolute->resolve);
my $texmfoutput = `kpsewhich --var-value=TEXMFOUTPUT`;
$texmfoutput =~ s/^\s+|\s+$//g;
if ($texmfoutput) {
   my $texmfoutput_dir = dir($texmfoutput)->absolute->resolve;
   push(@output_paths, $texmfoutput_dir) unless $texmfoutput_dir->dir_list == 1 && ! $texmfoutput_dir
      →->volume;
}

sub paranoia {
   my $file = $_[0];
   die "${message_prefix}Cannot␣create␣a␣hidden␣file␣or␣dir:␣$file"
      if $file->basename =~ /^\./;
   my $parent = $file->parent->absolute->resolve;
   die "${message_prefix}Cannot␣write␣outside␣the␣current␣working␣or␣output␣directory␣tree:␣$file"
      unless grep($_->contains($parent), @output_paths);
}

my $mmz_arg = $ARGV[0];
my $mmz_file = file($mmz_arg);
my $mmz_dir = $mmz_file->parent;

if ($mkdir) {
   my $dir = dir($mmz_arg)->cleanup;
   my $current = dir(File::Spec->catpath($dir->volume,
                              $dir->is_absolute ? File::Spec->rootdir : File::Spec->curdir,
                              ''))->absolute;
   for my $c ($dir->components) {
      $current = $current->subdir($c);
      if (-d $current) {
         $current = $current->resolve;
      } else {
         paranoia($current);
         mkdir($current);
         print("${message_prefix}Created␣directory␣$current\n") unless $quiet;
      }
   }
   exit;
} else {
   die "${message_prefix}The␣'mmz'␣argument␣should␣be␣a␣file␣with␣suffix␣'.mmz',␣not␣'$mmz_file'\n"
      →unless $mmz_file->basename =~ /\.mmz$/;
}

# Enable in-place editing (of the .mmz file).
paranoia($mmz_file) unless $keep;
$^I = "" unless $keep;

my $pdf_file = $pdf_arg ? file($pdf_arg) : $mmz_dir->file($mmz_file->basename =~ s/\.mmz$/\.pdf/r)->
    →cleanup;
paranoia($pdf_file) if $prune;

if ($log) {
   paranoia(file($log));
   open LOG, ">$log";
} else {
   *LOG = *STDERR;
}

my ($pdf, %extern_pages);
print "${message_prefix}Extracting␣externs␣from␣$pdf_file:\n" unless $quiet;

my $tolerance = 0.01;
my $done_message = "${message_prefix}Done␣(there␣was␣nothing␣to␣extract).\n";
```

```
while (<>) {
    if (/^\\mmzNewExtern *{(?P<extern_path>(?P<prefix>.*?)(?P<code_md5sum>[0-9A-F]{32})-(?P<
        →context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf)}{(?P<page_n>[0-9]+)}{(?P<expected_width
        →>[0-9.]*)pt}{(?P<expected_height>[0-9.]*)pt}/) {
        my $extern_file = file($+{extern_path});
        if (! $extern_file->is_absolute) {
            $extern_file = $mmz_dir->file($+{extern_path});
        }
        paranoia($extern_file);
        my $page = $+{page_n};
        my $expected_width_pt = $+{expected_width};
        my $expected_height_pt = $+{expected_height};
        my $c_memo_file = $mmz_dir->file($+{prefix} . $+{code_md5sum} . '.memo');
        my $cc_memo_file = $mmz_dir->file($+{prefix} . $+{code_md5sum} . '-' . $+{context_md5sum} . '.
            →memo');
        if (!(-e $c_memo_file && -e $cc_memo_file)) {
            print LOG ($warning_template =~ s/\\warningtext/Not extracting page $page into extern
                →$extern_file, because the associated (c)c-memo does not exist/gr), "\n\\endinput\n";
            last;
        }
        eval { $pdf = PDF::API2->open($pdf_file->stringify) unless $pdf; };
        if ($@) {
            print LOG ($warning_template =~ s/\\warningtext/Cannot read file "$pdf_file". Perhaps you
                →have to load Memoize earlier in the preamble?/gr), "\n\\endinput\n";
            close LOG;
            die "${message_prefix}File␣'$pdf_file'␣cannot␣be␣read,␣bailing␣out.\n";
        }
        my $extern = PDF::API2->new();
        $extern->version($pdf->version);
        $extern->import_page($pdf, $page);
        my $extern_page = $extern->open_page(1);
        my ($x0, $y0, $x1, $y1) = $extern_page->get_mediabox();
        my $width_pt = ($x1 - $x0) / 72 * 72.27;
        my $height_pt = ($y1 - $y0) / 72 * 72.27;
        my $warning = '';
        if (abs($width_pt - $expected_width_pt) > $tolerance
            || abs($height_pt - $expected_height_pt) > $tolerance)
        {
            $warning = "I␣refuse␣to␣extract␣page␣$page␣from␣$pdf_file,␣because␣its␣size␣(${width_pt}pt␣x
                →␣${height_pt}pt)␣is␣not␣what␣I␣expected␣(${expected_width_pt}pt␣x␣${
                →expected_height_pt}pt)";
            print LOG ($warning_template =~ s/\\warningtext/$warning/gr), "\n";
        }
        if ($warning && !$force) {
            unlink $extern_file;
        } else {
            $extern->saveas($extern_file->stringify);
            $done_message = "${message_prefix}Done.\n";
            print STDOUT "${message_prefix}␣␣Page␣$page␣-->␣$extern_file\n" unless $quiet;
            $extern_pages{$page} = 1 if $prune;
            print("%") unless $keep;
        }
    }
    print unless $keep;
}

print $done_message unless $quiet;

if ($pdf and $prune) {
    paranoia($pdf_file);
    my $pruned_pdf = PDF::API2->new();
    for (my $n = 1; $n <= $pdf->page_count(); $n++) {
        if (! $extern_pages{$n}) {
            $pruned_pdf->import_page($pdf, $n);
        }
    }
    $pruned_pdf->save($pdf_file->stringify);
    print("${message_prefix}The␣following␣extern␣pages␣were␣pruned␣out␣of␣the␣PDF:␣",
          join(",", sort(keys(%extern_pages))) . "\n") unless $quiet;
}

if ($log) {
    print LOG "\\endinput\n";
    close LOG;
}
```

9.2 The Python extraction script `memoize-extract.py`

```
__version__ = '2023/10/10␣v1.0.0'

import argparse, re, sys, os, pdfrw, subprocess, itertools
from pathlib import Path

parser = argparse.ArgumentParser(
    description = "Extract␣extern␣pages␣out␣of␣the␣document␣PDF.",
    epilog = "For␣details,␣see␣the␣man␣page␣or␣the␣Memoize␣documentation.",
    prog = 'memoize-extract.py',
)
parser.add_argument('--pdf', '-P', help = 'extract␣from␣file␣PDF')
parser.add_argument('--prune', '-p', action = 'store_true',
    help = 'remove␣the␣extern␣pages␣after␣extraction')
parser.add_argument('--keep', '-k', action = 'store_true',
    help = 'do␣not␣mark␣externs␣as␣extracted')
parser.add_argument('--force', '-f', action = 'store_true',
    help = 'extract␣even␣if␣the␣size-check␣fails')
parser.add_argument('--log', '-l', default = os.devnull, help = 'the␣log␣file')
parser.add_argument('--warning-template', '-w', default = '\warningtext',
    help = '\warningtext␣in␣the␣template␣will␣be␣replaced␣by␣the␣warning␣message')
parser.add_argument('--quiet', '-q', action = 'store_true',
    help = "describe␣what's␣happening")
parser.add_argument('--embedded', '-e', action = 'store_true',
    help = "prefix␣all␣messages␣to␣the␣standard␣output␣with␣the␣script␣name")
parser.add_argument('--mkdir', '-m', action = 'store_true',
    help = 'create␣a␣directory␣(and␣exit)')
parser.add_argument('mmz',
    help = 'the␣record␣file␣produced␣by␣Memoize:␣doc.mmz␣when␣compiling␣doc.tex')
parser.add_argument('--version', '-V', action = 'version',
                    version = f"%(prog)s␣of␣Memoize␣" + __version__)

args = parser.parse_args()

message_prefix = parser.prog + ':␣' if args.embedded else ''
if args.embedded:
    print()

# Even a bit more paranoid than required:
# (a) disallowing TEXMFOUTPUT=/ (while allowing C:\ on Windows)
# (b) waiting for access to '-output-directory'.
output_paths = [Path.cwd()]
if texmfoutput := subprocess.run(
        ['kpsewhich', '--var-value=TEXMFOUTPUT'],
        capture_output = True).stdout.decode().strip():
    texmfoutput_dir = Path(texmfoutput).resolve()
    if len(texmfoutput_dir.parts) != 1 or texmfoutput_dir.drive:
        output_paths.append(texmfoutput_dir)
```

```
def paranoia(f):
   p = Path(f).resolve()
   if p.stem.startswith('.'):
      raise RuntimeError(f"{message_prefix}Cannot␣create␣a␣hidden␣file␣or␣dir:␣{f}")
   if not any(p.is_relative_to(op) for op in output_paths):
      raise RuntimeError(f"{message_prefix}Cannot␣write␣outside␣the␣current␣working␣or␣output␣
           →directory␣tree:␣{f}")

mmz_file = Path(args.mmz)

if args.mkdir: # Here, argument "mmz" is interpreted as the directory to create.
   # We cannot simply say
   # paranoia(mmz_file)
   # mmz_file.mkdir(parents = True, exist_ok = True)
   # because have be paranoid about the intermediate directories as well:
   # memoize-extract.py -m a/../../BAD/../<cwd-name>/b
   # Note that paranoia might kick in only after a part of the given path was
   # already created. This is in accord to how "mkdir" behaves wrt existing
   # files.
   for folder in itertools.chain(reversed(mmz_file.parents), (mmz_file,)):
      if not folder.is_dir():
         paranoia(folder)
         folder.mkdir(exist_ok = True)
         if not args.quiet:
            print(f"{message_prefix}Created␣directory␣{folder}")
   sys.exit()
elif mmz_file.suffix != '.mmz':
   raise RuntimeError(f"{message_prefix}The␣'mmz'␣argument␣should␣be␣a␣file␣with␣suffix␣'.mmz',␣not␣'{
        →mmz_file}'")

mmz_dir = mmz_file.parent
pdf_file = Path(args.pdf) if args.pdf else mmz_file.with_suffix('.pdf')
paranoia(pdf_file)
pdf = None
extern_pages = []
new_mmz = []
args.log is os.devnull or paranoia(Path(args.log))
re_newextern = re.compile(r'\\mmzNewExtern␣*{(?P<extern_fn>(?P<prefix>.*?)(?P<code_md5sum>[0-9A-F]{32})
      →-(?P<context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf)}{(?P<page_n>[0-9]+)}{(?P<expected_width
      →>[0-9.]*)pt}{(?P<expected_height>[0-9.]*)pt}')
tolerance = 0.01
done_message = f"{message_prefix}Done␣(there␣was␣nothing␣to␣extract)."

# Complication: I want to use 'with', but don't have to open stderr.
with open(args.log, 'w') as log:
   log = sys.stderr if args.log is os.devnull else log
   try:
      mmz = mmz_file.open()
   except FileNotFoundError:
      print(f'''{message_prefix}File "{mmz_file}" does not exist, assuming there's nothing to do.''',
           file = sys.stderr)
   else:
      if not args.quiet:
         print(f"{message_prefix}Extracting␣externs␣from␣{pdf_file}")
      for line in mmz:
         if m := re_newextern.match(line):
            extern_file = mmz_dir / m['extern_fn']
            paranoia(extern_file)
            page_n = int(m['page_n'])-1
            c_memo = mmz_dir / (m['prefix'] + m['code_md5sum'] + '.memo')
            cc_memo = mmz_dir / (m['prefix'] + m['code_md5sum'] + '-' + m['context_md5sum'] + '.memo'
                 →)
        if not (c_memo.exists() and cc_memo.exists()):
            print(args.warning_template.replace('\warningtext', f'Not␣extracting␣page␣{page_n}␣
                  →into␣extern␣{extern_file},␣because␣the␣associated␣(c)c-memo␣does␣not␣exist'),
                  →file = log)
            continue
        if not pdf:
            try:
                pdf = pdfrw.PdfReader(pdf_file)
            except pdfrw.errors.PdfParseError:
                print(f'{message_prefix}File␣"{pdf_file}"␣cannot␣be␣read,␣bailing␣out.', file =
                      →sys.stderr)
                print(args.warning_template.replace('\warningtext', f'Cannot␣read␣file␣"{pdf_file
                      →}".␣Perhaps␣you␣have␣to␣load␣Memoize␣earlier␣in␣the␣preamble?'), file =
                      →log)
                args.keep = True
                break
        extern = pdfrw.PdfWriter(extern_file)
        page = pdf.pages[page_n]
        expected_width_pt, expected_height_pt = float(m['expected_width']), float(m['
              →expected_height'])
        mb = page['/MediaBox']
        width_bp, height_bp = float(mb[2]) - float(mb[0]), float(mb[3]) - float(mb[1])
        width_pt = width_bp / 72 * 72.27
        height_pt = height_bp / 72 * 72.27
        warning = None
        if abs(width_pt - expected_width_pt) > tolerance \
           or abs(height_pt - expected_height_pt) > tolerance:
            warning = (
                f'I␣refuse␣to␣extract␣page␣{page_n+1}␣from␣"{pdf_file}",␣'
                f'because␣its␣size␣({width_pt}pt␣x␣{height_pt}pt)␣is␣not␣'
                f'what␣I␣expected␣({expected_width_pt}pt␣x␣{expected_height_pt}pt)')
            print(args.warning_template.replace('\warningtext', warning), file = log)
        if warning and not args.force:
            extern_file.unlink(missing_ok = True)
        else:
            extern.addpage(page)
            if not args.quiet:
                print(f"{message_prefix}␣␣Page␣{page_n+1}␣-->␣{extern_file}", file = sys.
                      →__stdout__)
            extern.write()
            done_message = f"{message_prefix}Done."
            if args.prune:
                extern_pages.append(page_n)
            if not args.keep:
                line = '%' + line
    if not args.keep:
        new_mmz.append(line)
mmz.close()
if not args.quiet:
    print(done_message)
if not args.keep:
    paranoia(mmz_file)
    with open(mmz_file, 'w') as mmz:
        for line in new_mmz:
            print(line, file = mmz, end = '')
if args.prune and extern_pages:
    pruned_pdf = pdfrw.PdfWriter(pdf_file)
    pruned_pdf.addpages(
        page for n, page in enumerate(pdf.pages) if n not in extern_pages)
    pruned_pdf.write()
    if not args.quiet:
        print(f"{message_prefix}The␣following␣extern␣pages␣were␣pruned␣out␣of␣the␣PDF:",
              ",".join(str(page+1) for page in extern_pages))
    if args.log is not os.devnull:
        print(r'\endinput', file = log)
```

9.3 The Perl clean-up script memoize-clean.pl

```
use strict;
use Getopt::Long;
# I intend to rewrite this script using Path::Class.
use Cwd 'realpath';
use File::Spec;
use File::Basename;

my $VERSION = '2023/10/10␣v1.0.0';

my $usage = "usage:␣memoize-clean.py␣[-h]␣[--yes]␣[--all]␣[--quiet]␣[--prefix␣PREFIX]␣[mmz␣...]\n";
my $Help = <<END;
Remove (stale) memo and extern files.

positional arguments:
  mmz .mmz record files

options:
  -h, --help show this help message and exit
  --version, -V show version and exit
  --yes, -y Do not ask for confirmation.
  --all, -a Remove *all* memos and externs.
  --quiet, -q
  --prefix PREFIX, -p PREFIX
                    A path prefix to clean; this option can be specified multiple times.

For details, see the man page or the Memoize documentation.
END

my ($yes, $all, @prefixes, $quiet, $help, $print_version);
GetOptions(
    "yes|y" => \$yes,
    "all|a" => \$all,
    "prefix|p=s" => \@prefixes,
    "quiet|q|?" => \$quiet,
    "help|h|?" => \$help,
    "version|V" => \$print_version,
    ) or die $usage;
$help and die "$usage\n$Help";
if ($print_version) { print("memoize-clean.pl␣of␣Memoize␣$VERSION\n"); exit 0 }

my (%keep, %prefixes);

my $curdir = Cwd::getcwd();

for my $prefix (@prefixes) {
    $prefixes{Cwd::realpath(File::Spec->catfile(($curdir), $prefix))} = '';
}

my @mmzs = @ARGV;

for my $mmz (@mmzs) {
    my ($mmz_filename, $mmz_dir) = File::Basename::fileparse($mmz);
    @ARGV = ($mmz);
    my $endinput = 0;
    my $empty = -1;
    my $prefix = "";
    while (<>) {
      if (/^ *$/) {
      } elsif ($endinput) {
         die "Bailing␣out,␣\\endinput␣is␣not␣the␣last␣line␣of␣file␣$mmz.\n";
      } elsif (/^ *\\mmzPrefix *{(.*?)}/) {
         $prefix = $1;
         $prefixes{Cwd::realpath(File::Spec->catfile(($curdir,$mmz_dir), $prefix))} = '';
         $empty = 1 if $empty == -1;
      } elsif (/^%? *\\mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*?)}/) {
         my $fn = $1;
         if ($prefix eq '') {
            die "Bailing␣out,␣no␣prefix␣announced␣before␣file␣$fn.\n";
         }
         $keep{Cwd::realpath(File::Spec->catfile(($mmz_dir), $fn))} = 1;
         $empty = 0;
         if (rindex($fn, $prefix, 0) != 0) {
            die "Bailing␣out,␣prefix␣of␣file␣$fn␣does␣not␣match␣" .
               "the␣last␣announced␣prefix␣($prefix).\n";
         }
      } elsif (/^ *\\endinput *$/) {
         $endinput = 1;
      } else {
         die "Bailing␣out,␣file␣$mmz␣contains␣an␣unrecognized␣line:␣$_\n";
      }
   }
   die "Bailing␣out,␣file␣$mmz␣is␣empty.\n" if $empty && !$all;
   die "Bailing␣out,␣file␣$mmz␣does␣not␣end␣with␣\\endinput;␣this␣could␣mean␣that␣" .
      "the␣compilation␣did␣not␣finish␣properly.␣You␣can␣only␣clean␣with␣--all.\n"
      if $endinput == 0 && !$all;
}

my @tbdeleted;
sub populate_tbdeleted {
   my ($basename_prefix, $dir, $suffix_dummy) = @_;
   opendir(MD, $dir) or die "Cannot␣open␣directory␣'$dir'";
   while( (my $fn = readdir(MD)) ) {
      my $path = File::Spec->catfile(($dir),$fn);
      if ($fn =~ /\Q$basename_prefix\E[0-9A-F]{32}(?:-[0-9A-F]{32})?(?:-[0-9]+)?(\.memo|(?:-[0-9]+)?\.
           →pdf|\.log)/ and ($all || !exists($keep{$path}))) {
         push @tbdeleted, $path;
      }
   }
   closedir(MD);
}
for my $prefix (keys %prefixes) {
   my ($basename_prefix, $dir, $suffix);
   if (-d $prefix) {
      populate_tbdeleted('', $prefix, '');
   }
   populate_tbdeleted(File::Basename::fileparse($prefix));
}
@tbdeleted = sort(@tbdeleted);

my @allowed_dirs = ($curdir);
my @deletion_not_allowed;
for my $f (@tbdeleted) {
   my $f_allowed = 0;
   for my $dir (@allowed_dirs) {
      if ($f =~ /^\Q$dir\E/) {
         $f_allowed = 1;
         last;
      }
   }
   push(@deletion_not_allowed, $f) if ! $f_allowed;
```

```
}
die "Bailing␣out,␣I␣was␣asked␣to␣delete␣these␣files␣outside␣the␣current␣directory:\n" .
   join("\n", @deletion_not_allowed) if (@deletion_not_allowed);

if (scalar(@tbdeleted) != 0) {
   my $a;
   unless ($yes) {
      print("I␣will␣delete␣the␣following␣files:\n" .
           join("\n",@tbdeleted) . "\n" .
           "Proceed␣(y/n)?␣");
      $a = lc(<>);
      chomp $a;
   }
   if ($yes || $a eq 'y' || $a eq 'yes') {
      foreach my $fn (@tbdeleted) {
         print "Deleting␣", $fn, "\n" unless $quiet;
         unlink $fn;
      }
   } else {
      die "Bailing␣out.\n";
   }
} elsif (!$quiet) {
   print "Nothing␣to␣do,␣the␣directory␣seems␣clean.\n";
}
```

9.4 The Python clean-up script `memoize-clean.py`

```
__version__ = '2023/10/10␣v1.0.0'

import argparse, re, sys, pathlib, os

parser = argparse.ArgumentParser(
   description="Remove␣(stale)␣memo␣and␣extern␣files.",
   epilog = "For␣details,␣see␣the␣man␣page␣or␣the␣Memoize␣documentation."
)
parser.add_argument('--yes', '-y', action = 'store_true',
                    help = 'Do␣not␣ask␣for␣confirmation.')
parser.add_argument('--all', '-a', action = 'store_true',
                    help = 'Remove␣*all*␣memos␣and␣externs.')
parser.add_argument('--quiet', '-q', action = 'store_true')
parser.add_argument('--prefix', '-p', action = 'append', default = [],
   help = 'A␣path␣prefix␣to␣clean;␣this␣option␣can␣be␣specified␣multiple␣times.')
parser.add_argument('mmz', nargs= '*', help='.mmz␣record␣files')
parser.add_argument('--version', '-V', action = 'version',
                    version = f"%(prog)s␣of␣Memoize␣" + __version__)
args = parser.parse_args()

re_prefix = re.compile(r'\\mmzPrefix␣*{(.*?)}')
re_memo = re.compile(r'%?␣*\\mmz(?:New|Used)(?:CC?Memo|Extern)␣*{(.*?)}')
re_endinput = re.compile(r'␣*\\endinput␣*$')

prefixes = set(pathlib.Path(prefix).resolve() for prefix in args.prefix)
keep = set()

# We loop through the given .mmz files, adding prefixes to whatever manually
# specified by the user, and collecting the files to keep.
for mmz_fn in args.mmz:
   mmz = pathlib.Path(mmz_fn)
   mmz_parent = mmz.parent.resolve()
   try:
      with open(mmz) as mmz_fh:
         prefix = ''
```

```
            endinput = False
            empty = None
            for line in mmz_fh:
                line = line.strip()

                if not line:
                    pass

                elif endinput:
                    raise RuntimeError(
                        r'Bailing␣out,␣\endinput␣is␣not␣the␣last␣line␣of␣file␣$mmz_fn.')

                elif m := re_prefix.match(line):
                    prefix = m[1]
                    prefixes.add( (mmz_parent/prefix).resolve() )
                    if empty is None:
                        empty = True

                elif m := re_memo.match(line):
                    if not prefix:
                        raise RuntimeError(
                            f'Bailing␣out,␣no␣prefix␣announced␣before␣file␣"{m[1]}".')
                    if not m[1].startswith(prefix):
                        raise RuntimeError(
                            f'Bailing␣out,␣prefix␣of␣file␣"{m[1]}"␣does␣not␣match␣'
                            f'the␣last␣announced␣prefix␣({prefix}).')
                    keep.add((mmz_parent / m[1]))
                    empty = False

                elif re_endinput.match(line):
                    endinput = True
                    continue

                else:
                    raise RuntimeError(fr"Bailing␣out,␣"
                        fr"file␣{mmz_fn}␣contains␣an␣unrecognized␣line:␣{line}")

        if empty and not args.all:
            raise RuntimeError(fr'Bailing␣out,␣file␣{mmz_fn}␣is␣empty.')

        if not endinput and empty is not None and not args.all:
            raise RuntimeError(
                fr'Bailing␣out,␣file␣{mmz_fn}␣does␣not␣end␣with␣\endinput;␣'
                fr'this␣could␣mean␣that␣the␣compilation␣did␣not␣finish␣properly.␣'
                fr'You␣can␣only␣clean␣with␣--all.'
            )

    # It is not an error if the file doesn't exist.
    # Otherwise, cleaning from scripts would be cumbersome.
    except FileNotFoundError:
        pass

tbdeleted = []
def populate_tbdeleted(folder, basename_prefix):
    re_aux = re.compile(
        re.escape(basename_prefix) +
        '[0-9A-F]{32}(?:-[0-9A-F]{32})?(?:-[0-9]+)?(?:\.memo|(?:-[0-9]+)?\.pdf|\.log)$')
    try:
        for f in folder.iterdir():
            if re_aux.match(f.name) and (args.all or f not in keep):
                tbdeleted.append(f)
    except FileNotFoundError:
        pass
```

```
for prefix in prefixes:
    # "prefix" is interpreted both as a directory (if it exists) and a basename prefix.
    if prefix.is_dir():
        populate_tbdeleted(prefix, '')
    populate_tbdeleted(prefix.parent, prefix.name)

allowed_dirs = [pathlib.Path().absolute()] # todo: output directory
deletion_not_allowed = [f for f in tbdeleted if not f.is_relative_to(*allowed_dirs)]
if deletion_not_allowed:
    raise RuntimeError("Bailing␣out,␣"
        "I␣was␣asked␣to␣delete␣these␣files␣outside␣the␣current␣directory:\n" +
        "\n".join(str(f) for f in deletion_not_allowed))

_cwd_absolute = pathlib.Path().absolute()
def relativize(path):
    try:
        return path.relative_to(_cwd_absolute)
    except ValueError:
        return path

if tbdeleted:
    tbdeleted.sort()
    if not args.yes:
        print('I␣will␣delete␣the␣following␣files:')
        for f in tbdeleted:
            print(relativize(f))
        print("Proceed␣(y/n)?␣")
        a = input()
    if args.yes or a == 'y' or a == 'yes':
        for f in tbdeleted:
            if not args.quiet:
                print("Deleting", relativize(f))
            try:
                f.unlink()
            except FileNotFoundError:
                print(f"Cannot␣delete␣{f}")
    else:
        print("Bailing␣out.")
elif not args.quiet:
    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.