The CodeDoc class
v.0.3
2010/03/30

Paul Isambert
zappathustra@free.fr

CodeDoc is a class designed to produce LaTeX files such as packages and classes along with their documentations. It does not depart from LaTeX’s ordinary syntax, unlike e.g. DocStrip, allows any existing class to be loaded with its options and offers various fully customizable verbatim environments that allows authors to typeset the code and documentation of their files as they want. To create the documentation, we compile the document as usual; to create the external file(s), we simply put \produce in the class options and compile as before.

Despite my earliest expectations, CodeDoc is not better than DocStrip. It is simply different. If you want a well-delimited approach to literate programming, use DocStrip. On the other hand, CodeDoc is more natural, in the sense that it is ordinary LaTeX all the way down. Note that you can ‘mimick’ DocStrip, either by putting any character at the beginning of each line of your code and setting the \Gobble parameter to 1 (this would be ‘inverted DocStrip’), or by setting the comment character to be of category 9 (‘ignored’) and beginning each line of the documentation with this character. In this latter case, only commands that are considered by CodeDoc when producing a le should not be commented out... but I’m going too fast here, and you should learn the basics first...

CodeDoc is still in its infancy, as indicated by its version number. Although it has passed the test of producing this documentation, countless bugs will probably be reported, and meaningful suggestions will be made. Be patient, and send them to me. Any reported bug and meaningful suggestion will be rewarded by a musical note, played by a virtual instrument, and sent in the mp3 format. Isn’t it amazing? I know it is. I will have to hire medically educated secretaries to face the consequences of such a reckless proposition. But it is worth it. Once a stable version is reached, I might even write a symphony.\footnote{Some of the ideas of this class are not mine; some were inspired by others; some are mine but were independantly implemented in other places; may all these people be thanked, as well as all the verbatim wizards around the world. And, oh, yeah, some ideas are mine, too.}

Some of the ideas of this class are not mine; some were inspired by others; some are mine but were independantly implemented in other places; may all these people be thanked, as well as all the verbatim wizards around the world. And, oh, yeah, some ideas are mine, too.

\footnote{\textquoteleft Meaningful suggestion\textquotefr...}
<table>
<thead>
<tr>
<th>Changes in version 0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ Fixed \ProduceFile, which ruined everything in produce mode when used without optional arguments.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Changes in version 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ Bug fix to make \ref work properly in the unmodified code environment.</td>
</tr>
<tr>
<td>⇒ Files \input and read in produce mode won't produce error messages anymore... at least with \TeX.</td>
</tr>
</tbody>
</table>
Contents

I  User’s manual  4

1  Code & Documentation  4
   1.1 Writing code  4
   1.2 Macros to describe macros  7
   1.3 Choosing the class  9
   1.4 Dangerous strings  9

2  Verbatim Madness  10
   2.1 Example environments  10
   2.2 \verb and friends  14
   2.3 Using fancyverb  16

3  Summary of commands  17
   3.1 Class options  17
   3.2 Environments  17
   3.3 Commands  17

II  Implementation  21

4  Options and basic definitions  21

5  Normal mode  22
   5.1 Describing macros  22
   5.2 \verb and associates  25
   5.3 Verbatim definitions  30
   5.4 The default code environment  32
   5.5 Example environments  34
      5.5.1 Examples without \TeX  39
      5.5.2 Examples with \TeX  40
   5.6 File management  43

6  Produce mode  45
   6.1 Messages  45
   6.2 Testing strings  45
   6.3 Macros executed in produce mode  48
   6.4 Writing environments  54
   6.5 File management  56

Index  60
Part I
User’s manual

1 Code & Documentation

The source of this documentation looks roughly like this:

```latex
\documentclass[article(a4paper),
%produce,
]{codedoc}

\begin{document}
\section{Code \& Documentation}
The source of this document...

\ProduceFile{codedoc.cls}[codedoc][v.0.3][2010/03/30]
\begin{code}
Material here will be written to codedoc.cls
and typeset verbatim in the documentation.
\end{code}
\ShortCode/
/ This too...
/
\begin{invisible}
This material will be written to codedoc.cls
but not typeset in the documentation.
\end{invisible}
\end{document}
```

Everything between `\begin{code}` and `\end{code}` is written verbatim to the dvi file. It is also the case for everything between two `\ShortCode` symbol, in this example `/`. Finally, if the comment sign at the beginning of the second line were removed, thus enabling the produce option, then this code would be written to codedoc.cls and no documentation would be produced. This is CodeDoc’s basic mechanism. Let’s review it more precisely. In what follows, I will say ‘normal mode’ if the produce option is not turned on, that is when we’re typesetting the documentation, and ‘produce mode’ otherwise, that is when produce is present among the class options and CodeDoc is used to create an external file.

The first two sections of this manual explain how CodeDoc works and provide many examples. The third section lists all commands in alphabetical order, and explains what they do in each mode in a more systematic fashion.

1.1 Writing code

- \ProduceFile{(File)}[(File name)][(File version)][(File date)]

  In normal mode, this macro provides four commands: \FileSource stores \textit{(File)}, and the next three arguments are stored in \textit{FileName}, \textit{FileVersion} and \textit{FileDate} respectively. Those are optional, as indicated by their syntax.
In produce mode, CodeDoc opens \texttt{\langle File\rangle} and writes to it everything in a code environment. \texttt{\FileName}, \texttt{\FileVersion} and \texttt{\FileDate} are also provided and may be used in \texttt{\Header} (see below) or in the file itself with \texttt{\CodeEscape} (see page 16). Thus, you can avoid mismatches between your documentation and the \texttt{\ProvidesPackage} declaration, for instance.

- \texttt{\CloseFile{\langle File\rangle}}

In produce mode, when the autoclose option is on, \texttt{\ProduceFile} closes the file that was currently under production, if any. But you might want to keep a file open, in case you’re writing to several files at the same time. That’s why CodeDoc’s basic behavior is to keep all files open. Thus

\begin{verbatim}
\ProduceFile{myfile} \begin{code}
\def\foo{\%}
\end{code}
\ProduceFile{myotherfile} \begin{code}
\relax
\end{code}
\ProduceFile{myfile} \begin{code}
FOO}
\end{code}
\end{verbatim}

will write \texttt{\begin{verbatim} \def\foo{\%}
\end{verbatim}} to \texttt{myfile} and \texttt{\relax} to \texttt{myotherfile}. This might not be very good practice, but who knows? That might be useful when building a complicated package.

But \LaTeX{} cannot keep open as many files as one wants. Actually, CodeDoc will start complaining when more than 16 files are simultaneously in production. \texttt{\CloseFile} is used to close those whose production is over and allocate their streams to new files.

- \texttt{\CodeFont{\langle Font specifications\rangle}}

The font of the code environment may be changed with \texttt{\CodeFont} (by default, it’s \texttt{\ttfamily}). Since everything is in a group, you can use ‘spreading commands’.

- \texttt{\LineNumber{\langle code\rangle}{\langle Font specifications\rangle}{\langle Width\rangle}{\langle Number\rangle}}

This sets the style of the line number, the width of the box it is put in (by default, it’s 0pt, so numbers are in the left margin), and the starting value. The first argument is \texttt{code} and not \texttt{\langle code\rangle}, because \texttt{\LineNumber} is a macro that applies to all example environments (see the next section), and its first argument is the name of the environment to modify. By default, code is not an example environment (although it might be redefined as such) but this command is nonetheless available.

\begin{verbatim}
\CodeFont{\color{red}\itshape}
\LineNumber{\langle code\rangle}{\ttfamily}\footnotesize
\color{green}\langle 2cm\rangle}
\begin{code} This will be gobbled
\def\foo{\%
\end{code}
\end{verbatim}
Note that \LineNumber inherits the specification of \CodeFont that it doesn’t override, in this example the italic shape. The \color command does not belong to CodeDoc, but to the xcolor package. If you want to do really interesting things with code, it is better to redefine it as an example (see next section).

As usual with verbatim environments, there exists a starred version of code that shows spaces.

- **invisible**
  In normal mode, everything in a invisible environment is skipped. In produce mode, however, the material is written to the file in production. This is useful to write code you don’t want to comment in the documentation, like specifications at the beginning of the file or repetitive macro definitions. As you might imagine, there is no starred variant.

- **\Header{⟨Text⟩}**
  In produce mode, unless the noheader option is on, CodeDoc writes the following at the beginning of every file:
  
  ```%
  This is ⟨\FileName⟩, produced by the CodeDoc class
  %
  % To create the documentation, compile ⟨jobname.tex⟩
  % without the ‘produce’ option.
  %
  % SOURCE: ⟨File (\input in File...)⟩
  % DATE: ⟨\FileDate⟩
  % VERSION: ⟨\FileVersion⟩
  ```

  where \FileName, \FileDate and \FileVersion are set by \ProduceFile. The ‘\input in file’ part is optional and recursive, depending on files \input in your document. With \Header, you can change this and print ⟨Text⟩ instead. In ⟨Text⟩, ends of line are obeyed, and a comment sign followed by a space will start every line. Comment signs are normal signs. \Header should appear before \ProduceFile.

- **\AddBlankLine**
  In produce mode, CodeDoc writes a blank line to the file under production. Useful to delimit macros.

- **\TabSize{⟨Number⟩}**
  This is the number of spaces by which a tabulation will be represented in verbatim context. Default is 2. In produce mode, however, tabs are written as tabs, so this parameter has no effect.

- **\Gobble{⟨Number⟩}**
  The number of characters that will be gobbled at the beginning of each line. This works both in normal mode and in produce mode. This might be useful to indent code lines to make them more visible. When gobbling, a tab is considered as a single character and not as n characters, n being the value of \TabSize. A totally blank line is written as a totally blank line in both modes, i.e. CodeDoc does not fill its need for gobbled characters on the next line. The \end{code} line doesn’t need to be indented, although it can be. If there are more characters than the value of \Gobble before \end{code}, then a new line is created.

\begin{code*}
\begin{verbatim}
\TabSize{3}
\Gobble{2}
My own value for \TabSize is 2, hence the 2-space tab here, but in the right panel it's 3
12\foo \foo
\end{verbatim}
\end{code*}
1.2 Macros to describe macros

Most of the commands in this section are similar to those in DocStrip. CodeDoc has an indexing mechanism that simply uses MakeIndex; if the index option is on, the makeidx package is loaded and \makeindex is executed. This also means that \printindex is available. CodeDoc does not require a special style file for MakeIndex. Thus, users can compile a documentation made with CodeDoc with MakeIndex’s default specifications.

• \DescribeMacro{⟨Macro⟩}
• \DefineMacro{⟨Macro⟩}

These commands print their argument according to \PrintMacro (see below). The first token is \string'ed, so it can be a control sequence. They also create an index entry with the first token, and here lies their difference: they print the page number differently to distinguish whether a macro is described or defined (in the implementation). By default described macros have normal page numbers while defined macros have theirs in italics. This is not conventional, I agree, but it can be changed.\footnote{Since CodeDoc doesn’t index macros when used in the code, I’ve found this choice more readable.}

• \DescribeEnvironment{⟨Environment⟩}
• \DefineEnvironment{⟨Environment⟩}

This is similar to the macro version above, except that the entry is followed by ‘(environment)’ in the index.

• \DescribeIndexFont{⟨Font specifications⟩}
• \DefineIndexFont{⟨Font specifications⟩}

This sets how the page numbers are printed for described and defined macros (and environments) respectively. ⟨Font specifications⟩ should be commands like \ttfamily and not argument-taking commands like \texttt. You know that if you use MakeIndex.

• \PrintMacro{⟨Macro or environment⟩}

This is the command that typeset the \string'ed macro. It takes one argument. It is shown here not to use it as is but to redefine it. Its default definition is:

\begin{verbatim}
def\PrintMacro#1{% 
  \noindent% 
  \marginpar{\raggedleft\strut\ttfamily#1}% 
  \ignorespaces}
\end{verbatim}

That is, it puts the macro in the margin. (Obviously, it was redefined in this documentation.) To achieve the same effect as with DocStrip, the following command is needed.

• \DocStripMarginpar

This reverses marginpar and sets the right value for \marginparpush and \marginparwidth. They weren’t included by default because you have the right to do what you want with your margins.

\footnote{Verbatim text does not break by itself. I’ve used \VerbCommand here (see below) to include a discretionary.}
Many package and class authors prefix their internal commands with a string of letters to avoid clashes with other packages. For instance, if one writes a package `mypack`, one may name all internal commands `\mp@foo`, `\mp@boo`, `\mp@moo`, etc. Unfortunately, when indexed, they will all end up in the ‘M’ letter, whereas one might want to have them sorted without the prefix, with `\mp@foo` indexed as if it was `\foo`, etc. This is what \IgnorePrefix does; when sorting entries produced by \DescribeMacro and \DefineMacro, (Macro prefix) is ignored, although it is printed of course as part of the name. In our example, one would say \IgnorePrefix{mp@}. This command has two restrictions: first, (Macro prefix) should be no more than 8 characters long; second, any macro described with \DescribeMacro or \DefineMacro should have as many characters as \IgnorePrefix, 3 in our example. A simple way to circumvent the latter shortcoming is to temporarily define (Macro prefix) as an empty string:

```
\IgnorePrefix{mp@}
\DefineMacro\mp@foo \textit{This will cause an error message}
\IgnorePrefix{}
\DefineMacro\fo \textit{This is perfectly ok}
\IgnorePrefix{mp@}
```

You can have several \IgnorePrefix specifications, they are effective for the macros that follow them. For instance, some macros in CodeDoc are prefixed with `cd@`, and when I define them in this documentation I specify \IgnorePrefix{cd@} and then immediately \IgnorePrefix{cd1}, which is the normal prefix.

Like \PrintMacro, this command is not shown here to be used but to be redefined. It is put just before (Macro prefix) when printing the index, so that you can typeset it differently. For instance, most CodeDoc’s internal macros are prefixed with `cd@`. I have specified \IgnorePrefix{cd@} for this documentation and defined \PrintPrefix as \def\PrintPrefix{\textcolor{gray}} so that all prefixes are printed in gray (thanks to the xcolor package). For instance, `\cd@BadChar` is printed `\textcolor{gray}{\cd@BadChar}` in the index (which you can verify if the \obeystop option is commented out, thus including the implementation in the documentation). Obviously, `\def\PrintPrefix#1{\textcolor{gray}{#1}}` would have been equally efficient. Just note that since \PrintPrefix is \let to \relax by default, you have to use \newcommand and not \renewcommand when defining it for the first time, in case you prefer \LaTeX’s command definitions.

\meta{(Argument)}
\marg{(Argument)}
\oarg{(Argument)}
\parg{(Argument)}
These are well-known. In case you’ve forgotten:
\meta{(Argument)} \Rightarrow (Argument)
\marg{(Mandatory argument)} \Rightarrow \{(Mandatory argument)}
\oarg{(Optional argument)} \Rightarrow \[
\parg{(Picture argument)} \Rightarrow (\langle Picture argument\rangle)
```
\bslash
Everybody needs a backslash. This one is meant to print equally well in usual contexts and in PDF bookmarks created by hyperref, if any. So it can be used in titles without restriction.

\StopHere{\Code}
If the \obeystop command is on, CodeDoc will execute (Code) and then \end{document}, otherwise nothing happens. If the index option is also on, \printindex will be automatically executed after (Code). This is useful to let the user print a version of the documentation with some part(s) left out, typically the implementation.
1.3 Choosing the class

CodeDoc by itself defines nothing that one wants a class to define. It lets the user call the desired class. To do so, just add the name of the class in the options of the `\documentclass` declaration. If you want the class to load options itself, put them after the name of the class, between parenthesis, and separated by semicolons. Thus, `\documentclass[\texttt{memoir}]{\texttt{codedoc}}` loads the `\texttt{memoir}` class without options while `\documentclass[\texttt{memoir}(\texttt{a4paper};\texttt{oneside})]{\texttt{codedoc}}` loads it with the `\texttt{a4paper}` and `\texttt{oneside}` options.\footnote{This means that if you specify an unknown option for CodeDoc, it will try to load an (probably) unknown class, and you will get the corresponding error message.}

By default, CodeDoc loads the `\texttt{article}` class without options.

1.4 Dangerous strings

In produce mode, CodeDoc becomes a string tester and nothing else. Hence, there are strings you don’t want it to see because you don’t want it to execute them. For instance, you don’t want `\texttt{\textbackslash end\{document\}}` to be executed unless at the end of the document. So when you say `\texttt{\textbackslash verb\{+\textbackslash end\{document\}\+}}`, you want CodeDoc to identify that `\texttt{\textbackslash end\{document\}}` is not for real. Fortunately, CodeDoc does so. To some extent, CodeDoc identifies its own verbatim commands (described in the next section), \LaTeX’s `\texttt{\textbackslash verb}` and `\texttt{\textbackslash verbatim}` environment, as well as verbatim environments created with the `\texttt{fancyvrb}` package and the ‘short verb’ characters defined with `\texttt{\textbackslash DefineShortVerb}` from the same package. Thus, you can safely use `\texttt{fancyvrb}` and its companion `fvrb-ex`.

However, `\texttt{\textbackslash begin\{\}}` and `\texttt{\textbackslash end\{\}}`’s are not the only strings that must be used carefully. The most important things you want CodeDoc to ignore in case they shouldn’t be executed are its own macros. For instance, you don’t want `\texttt{\textbackslash Produce\{File\}}` to be executed when there’s no reason to do so. But, unless you’re documenting CodeDoc itself, what might be the situation where `\texttt{\textbackslash Produce\{File\}}` is executed wrongly? Simply if you use it in a statement with `\texttt{\textbackslash let\{\textbackslash def\{\\textbackslash newcommand\{\}}`, etc. In produce mode, CodeDoc does not recognize these commands and for instance in `\texttt{\textbackslash let\{\textbackslash Produce\{File\}\{my\textbackslash command\}}`, `\texttt{\textbackslash let}` will be skipped and `\texttt{\textbackslash Produce\{File\}}` executed. Hence the following.

\noindent \texttt{\textbackslash Dangerous\{Environment\}}{\texttt{\{\texttt{\textbackslash list\{of\textbackslash environments\}\}}}}

Whenever you want CodeDoc to skip an environment in produce mode, for instance because it’s a verbatim environment designed by yourself, you can add its name to `\texttt{\textbackslash Dangerous\{Environment\}}`. If you add more than one name, use commas as separators.

\noindent \texttt{\textbackslash Start\{Ignore\}}

\noindent \texttt{\textbackslash Stop\{Ignore\}}

In produce mode, when CodeDoc encounters `\texttt{\textbackslash Start\{Ignore\}}`, everything is skipped until `\texttt{\textbackslash Stop\{Ignore\}}` is found. This is useful to hide parts of your document that are irrelevant to the file you’re building in produce mode (which is probably contained in the ‘implementation’ section). You should be careful to define your `\texttt{example}` environments and other verbatim devices outside the skipped material, if you want CodeDoc to identify them properly when it stops ignoring things.

\begin{codebox}
\texttt{\textbackslash Dangerous\{Environment\}\{my\textbackslash env,my\textbackslash other\textbackslash env\}}
\texttt{\textbackslash begin\{my\textbackslash env\}}
\texttt{\textbackslash end\{document\}} \texttt{This will be skipped by CodeDoc}
\texttt{\textbackslash end\{my\textbackslash env\}}

\texttt{\textbackslash Start\{Ignore\}}
\texttt{\textbackslash let\{\textbackslash Produce\{File\}\{my\textbackslash produce\}} \texttt{This too, but that will be taken into account in normal mode}
\texttt{\textbackslash Stop\{Ignore\}}
\end{codebox}

However, you should be aware of the following points:

\noindent \texttt{Any command that has some effect in produce mode should appear verbatim in your document.}
Conversely,

- **Commands that have some effect in produce mode cannot be redefined for that mode.**

And when I say ‘cannot’, I mean ‘you can try, it won’t work’. This leads to the final principle:

- **You can redefine a command to have the desired effect in normal mode as long as you respect its arguments, so that it can work properly in produce mode. And this should be done between \texttt{\StartIgnore} and \texttt{\StopIgnore}, of course.**

For instance, you can say:

\begin{verbatim}
\StartIgnore
\renewcommand\CloseFile[1]{End of #1\clearpage}
\StopIgnore
\end{verbatim}

and when you say \texttt{\CloseFile{myfile}}, ‘End of myfile’ will be printed to the documentation, and a new page will be created, while in produce mode CodeDoc will do its usual job. On the other hand, although \texttt{\let\cf\CloseFile} is meaningful in normal mode, in produce mode it won’t take effect, i.e. CodeDoc won’t close anything. Finally, the previous example would have been catastrophic without \texttt{\StartIgnore} and \texttt{\StopIgnore}, because in produce mode CodeDoc would have tried to execute \texttt{\CloseFile}.

\texttt{\StartIgnore} and \texttt{\StopIgnore} are also useful to make CodeDoc go faster and avoid errors, if you use it with \texttt{\input}. For instance, the following file would be perfect, provided everything that should be written to an external file is contained in implementation.tex

\begin{verbatim}
\documentclass{codedoc}
write your verbatim definitions here, so that CodeDoc can see them
\begin{document}
\StartIgnore
\input{documentation}
\StopIgnore
\input{implementation}
\end{document}
\end{verbatim}

This example leads us to the final restriction:

- **You should use \texttt{\input} in the \LaTeX’s way, i.e. \texttt{\input{myfile}}, and not in \TeX’s original way, i.e. \texttt{\input myfile}, if the file in question is to be read in produce mode.** In the example above, documentation can be \texttt{\input} as you want, but implementation should be \texttt{\input} as shown.

To know what commands have some effect in produce mode, see the summary of commands.

## 2 Verbatim Madness

### 2.1 Example environments

- \texttt{\begin{example}}
- \texttt{\CodeInput}
- \texttt{\CodeOutput}

At first sight, the \texttt{\example} environment is totally useless. Indeed, the following code does nothing:

\begin{verbatim}
\begin{example}
\TeX
\end{example}
\end{verbatim}
However, it provides two commands \CodeInput and \CodeOutput. The former prints the code verbatim (and in typewriter font), and the latter executes it. So in the end it’s very useful to document your package or class, because it avoids typing the code twice (and therefore errors are avoided).

The example environment is just one instance of a family of environments that you can create by yourself with the following commands.

- \NewExample[(Options)]{(Name)}{(Code input)}{(Code output)}{(Immediate execution)}
- \RenewExample[(Options)]{(Name)}{(Code input)}{(Code output)}{(Immediate execution)}

These two macros (whose difference is similar to the one between \newcommand and \renewcommand) create an environment (Name) that will provide two commands, \CodeInput and \CodeOutput, whose effect is defined by (Code input) and (Code output). Moreover, (Name) will execute (Immediate execution).

(Code input) and (Code output) have a peculiar syntax. The code to be processed is represented by #. For instance, the example environment is defined as:

```latex
\NewExample{example} This is (Name)
{\ttfamily#} \CodeInput yields but the code in typewriter font
{#} \CodeOutput simply executes the code
{\ttfamily#} Nothing is done when example is called
```

You can do whatever you want. The code, represented by #, may be the argument of a macro. For instance:

```latex
\NewExample{myex}
{\ttfamily\underline{#}}
{}\CodeInput
\begin{myex}
Hello, world!
\end{myex}
```

What does myex do? It sets the verbatim code in typewriter font and underlines it (which is admittedly not the most interesting thing you can do). (Code output) is empty, so \CodeOutput will yield nothing. Finally, (Immediate execution) calls \CodeInput, so there’s no need to call it after the environment.

The following points apply:

- All environments thus defined have a starred variant that shows spaces as characters.
- \CodeInput, \CodeOutput and (Immediate execution) are groups, so you can put any command in them, they won’t spread. For instance, in myex above, there’s no need to add a group to restrict the application of \ttfamily.
- \CodeOutput really executes your code. Any error will appear as such.
- Since \CodeOutput is a group, the definition you make won’t work for the rest of your document, unless you make them global. For instance:
will yield an error message, because \texttt{\code{foo}} was only locally defined in \texttt{\code{CodeOutput}}.

- Everything on the same line after \texttt{\begin{myex}} statement of an environment will be gobbled.
- By default, CodeDoc does not add any space or \texttt{\par} before \texttt{\code{CodeInput}}, \texttt{\code{CodeOutput}} and \texttt{⟨Immediate execution⟩}. A \par is added after \texttt{\CodeInput} if and only if the \texttt{\end{myex}} statement appears on its own line. Here's an illustration:

\begin{myex}
\texttt{\TeX}
\end{myex}
+\texttt{\CodeInput} yields +\texttt{\CodeOutput}+

\begin{myex}
\texttt{\TeX}\end{myex}
+\texttt{\CodeInput} yields +\texttt{\TeX}+

\begin{myex}
\texttt{\TeX}\end{myex}
+\texttt{\CodeInput} yields +\texttt{\TeX}+

- The code environment can be freely redefined as an example environment.
- All example environments obey \texttt{\TabSize} and \texttt{\Gobble} as defined in the previous section, as well as \texttt{\LineNumber} if they are numbered (see below). See the description of \texttt{\TeXOff} and \texttt{\TeXOn} below for a comment on \Gobble.

\begin{itemize}
  \item ⟨\texttt{Options}⟩ may be one or several of the following (separated by commas):
  \begin{itemize}
    \item \texttt{numbered}
      \begin{itemize}
        \item Each line of \texttt{\CodeInput} is numbered. The count starts back to 1 at each occurrence of the environment.
      \end{itemize}
    \item \texttt{continuous}
      \begin{itemize}
        \item Each line of \texttt{\CodeInput} is numbered. The count starts where the last occurrence of the same environment left. As an (utterly boring) example:
      \end{itemize}
  \end{itemize}
\end{itemize}
visibleEOL
This is more complicated and requires some knowledge of how CodeDoc builds examples.

Although you might not know it, your \TeX\ distribution is very probably running on \TeX{}\!. That's the reason why CodeDoc can process some code verbatim and executes it at the same time, as in the example environments, without the need for an external file. This is completely impossible with the original implementation of \TeX. If, for some reason, you don't have \TeX\!, or you're not running on it, then CodeDoc will use an external file.

However, \TeX\!'s `virtual external file' mechanism is not perfect, and CodeDoc has to cope with it. What happens is that when you use \CodeOutput, CodeDoc hacks your code a little in order to simulate a real \TeX\! code; namely, before anything is processed, CodeDoc removes ends of lines and commented parts of lines. For instance, if you say:

```
def\foo{FOO!}
def\foo{FOO!}
```

what CodeDoc really processes with \CodeInput is `\def\foo{FOO!}`. Most of the time, that's exactly what you want. But it might happen that you're toying around with ends of lines or comment characters, and in that case everything will go wrong, as in:

```
catcode`\%=12
catcode`\%=12
```

This will not produce `I'm writing a \% sign', because CodeDoc will remove everything from the comment sign to the end of the line, so that what \CodeOutput will try to execute is:

```
catcode`\='
catcode`\='
```

This will not produce `I'm writing a \%'.
and of course the aborted \catcode declaration will yield an error message. To avoid this problem, the \visibleEOL option makes CodeDoc keeps everything. But now there’s another issue: comments and end of line characters are processed at the same time as other macros and aren’t interpreted independently as in normal \TeX. For instance, the following code, if the \visibleEOL option is on for the environment in question, will apply \emph to the end of line character and not to A.

So you should be sure that comments and line ends occur where they won’t hinder anything. If you find this utterly complicated, then you can use an external file whenever you’re hacking ends of line, thanks to the following two macros.

- \eTeXOff
- \eTeXOn

The former makes CodeDoc process all examples environments with an external file (whose extension is .exp). The latter makes everything back to normal. If \eTeXOff applies, the \visibleEOL option is of course irrelevant. Note that these two macros apply to examples that follow them and not to example definitions. For instance, \eTeXOff and then \NewExample{myex}{#}{#}{} will not lead CodeDoc to use an external file whenever myex is called, but simply as long as no \eTeXOn appears. To put simply, these two macros have no effect on \NewExample.

If \Gobble is positive, examples with \varepsilon-\TeX and examples without behave differently. The latter gobble characters before writing to the external file. Thus, \CodeOutput will execute line with the first characters gobbled. With \varepsilon-\TeX, however, nothing is gobbled in \CodeOutput. This means that first characters, if meant to be gobbled, will be executed. Most of the time, such characters are spaces, and the difference won’t be noticed. If, for some reason, you use other characters instead, and if you want to call \CodeOutput nonetheless, then a switch to an external file may be a good idea.

### 2.2 \ShortVerb and friends

CodeDoc provides a number of facilities to act on verbatim contexts. They declare one or more character(s) to have a special effect under certain circumstances.

- \ShortVerb{⟨Character⟩}
- \UndoShortVerb

This is well-known. ⟨Character⟩ is turned into a shorthand for \verb. You can define only one such character, and that’s why \UndoShortVerb doesn’t take an argument (like all \Undo... below). In CodeDoc verbatim contexts, this character returns to its normal value.

\begin{example}
And the star appeared:*
\end{example}

\CodeInput

The command \TeX gives \TeX.

And the star appeared:*
\ShortVerb and \ShortCode have one caveat. If you \undo... them and the next character (disregarding spaces, comments and ends of lines) is a short verb or a short code respectively, in produce mode it will fire as if it was still active. A pair of braces after the \undo... statement prevents this.

- \VerbBreak{⟨Character⟩}
- \UndoVerbBreak

Every once in a while, breaking a verbatim line may be useful. In verbatim contexts, ⟨Character⟩ breaks the line, creates an unnumbered new one and indents it to the indentation of the original line. When \CodeOutput is processed, the \VerbBreak character is ignored. However, you should not break in the middle of a control sequence (admittingly a strange idea), or it won’t form. It is also ignored, of course, when writing to a file in produce mode.

- \VerbBreak{\=} An escape character is ok
- \TeX
- \emph=\TeX

\begin{example*}
\TeX
\emph=\TeX
\end{example*}
\CodeInput
\And the result is: \CodeOutput

\label{} to refer to it.

Since numbered examples environment define the current label to be the number of the current line, an interesting application is to use \label to refer to it.
\VerbCommand!()

(Everything is normal here!)
\RenewExample[numbered]{example}
{\ttfamily#}{#}{}
\LineNumber{example}{1cm}
\begin{example}
TeX !underline(\TeX)TeX
%Second line !label(myline)
\end{example}
\CodeInput
gives \CodeOutput and the label is on line \ref{myline}.

- $\CodeEscape{(Character)}$
- $\CodeEscape{\langle Character \rangle}$

In normal mode, this command does absolutely nothing. However, in produce mode, \textit{(Character)} becomes an escape character to form control sequences that will be expanded when writing to the file under production. It’s useful mainly to put the values defined by $\texttt{\textbackslash ProduceFile}$ somewhere in your file. For instance, the following code

\CodeEscape!
\ProduceFile{mypack.sty}[mypack][v.2.1][2009/02/24]
\begin{code}
\ProvidesPackage{!FileName}[!FileDate! space !FileVersion! space My super package.]
\end{code}

will write $\texttt{\textbackslash ProvidesPackage(mypack)[2009/02/24 v.2.1 My super package.]}$ to mypack.sty.

2.3 Using \texttt{fancyvrb}

\CodeDoc\ is minimally compatible with \texttt{fancyvrb}, in the sense that verbatim characters defined and undefined with $\texttt{\textbackslash DefineShortVerb}$ and $\texttt{\textbackslash UndefineShortVerb}$ are recognized in produce mode (hopefully). Besides, verbatim environments defined with $\texttt{\textbackslash DefineVerbatimEnvironment}$ are automatically added to the list of dangerous environments. The environments offered by \texttt{fancyvrb} and the $\verb-ex$ companion package already belong to that list.

You can even redefine the \texttt{code} environment with \texttt{fancyvrb} facilities.\footnote{It will indeed add \texttt{code} to the list of dangerous environment, which is already the case when \texttt{code} is redefined with $\texttt{\textbackslash RenewExample}$. But \CodeDoc\ evaluates whether an environment is \texttt{code} before checking the list of dangerous environments.} However:

- $\ShortCode$ will stick to the last style defined for \texttt{code} (if it is set to follow this environment).
- Since everything is gobbled after $\texttt{\begin{code}}$ in produce mode, you can freely put your keyval pairs here, as usual with \texttt{fancyvrb}. However, \textit{you should not input these pairs on the following line(s)}, although it’s ok with \texttt{fancyvrb}. The following code will lead \texttt{xleftmargin=1cm} to be written on the file under production.
• The gobble and commandchars parameters will be obeyed in normal mode (since fancyrb is in charge), but not in produce mode, unless you also specify the \Gobble and \VerbCommand parameters (see above) accordingly.

3 Summary of commands

In this section I explain the behavior of all CodeDoc constructions in normal and produce mode respectively. Commands which have some effect in produce mode are subject to the restrictions given in section 1.4.

3.1 Class options

• autoclose
  Normal Mode: Does nothing.
  Produce Mode: The current file is closed when a new one is opened with \ProduceFile.

• index
  Normal Mode: Loads makeid and calls \makeindex. \StopHere automatically launches \printindex.
  Produce Mode: Does nothing.

• noheader
  Normal Mode: Does nothing.
  Produce Mode: No header is written to the file when it is opened.

• obeystop
  Normal Mode: The document stops at \StopHere{(Code)} and executes (Code). If the index option is on, \printindex is executed after (Code).
  Produce Mode: Does nothing.

• tracing0, tracing1, tracing2
  Normal Mode: Does nothing.
  Produce Mode: CodeDoc normally writes a report to the log file. If tracing0 is on, there’s no report; if tracing1 is on (which is default), CodeDoc reports only about opening files and writing code. With tracing2, it also reports about characters defined as \ShortVerb or \CodeEscape, environments added to the list of dangerous environments, etc.

3.2 Environments

• code
  Normal Mode: The content is displayed verbatim according to the style defined for code.
  Produce Mode: The content is written to the file in production.

• example
  Normal Mode: A minimal example environment that provides \CodeInput (in typewriter font) and \CodeInput.
  Produce Mode: The content is skipped.

• invisible
  Normal Mode: The content is skipped.
  Produce Mode: The content is written to the file in production.

3.3 Commands

• \AddBlankLine
  Normal Mode: Does nothing.
  Produce Mode: Adds a blank line to the file in production.

• \bslash
  Normal Mode: Prints \.
  Produce Mode: Does nothing.
\BoxTolerance{\langle Dimension \rangle}

Normal Mode: Excess size tolerated before a verbatim line is reported as an overfull box.

Produce Mode: Does nothing.

\CloseFile{\langle File \rangle}

Normal Mode: \FileName and others are not available anymore.

Produce Mode: Closes \langle File \rangle. No file is considered in production until the next |ProduceFile, even if there are open files. Useless in autoclose mode.

\CodeEscape{\langle Character \rangle}

Normal Mode: Does nothing.

Produce Mode: \langle Character \rangle turns into an escape character in code contexts.

\CodeFont{\langle Font specifications \rangle}

Normal Mode: The style of the code environment if it has not been redefined with \RenewExample. Default is \ttfamily.

Produce Mode: Does nothing.

\CodeInput

Normal Mode: Displays the code of the last example environment verbatim, according to the style defined for that environment.

Produce Mode: Does nothing.

\CodeOutput

Normal Mode: Executes the code of the last example environment, according to the style defined for that environment.

Produce Mode: Does nothing.

\DangerousEnvironment{\langle List of environments \rangle}

Normal Mode: Does nothing.

Produce Mode: The environments in the list are skipped during processing.

\DefineEnvironment{\langle Environment \rangle}

Normal Mode: Prints \langle Environment \rangle according to \PrintMacro and adds it to the index with `(environment)' and a line number typeset according to \DefineIndexFont.

Produce Mode: Gobbles the first characters of \langle Environment \rangle, just in case.

\DefineIndexFont{\langle Font specifications \rangle}

Normal Mode: Style of the page number in the index for \DefineMacro and \DefineEnvironment entries.

Produce Mode: Does nothing.

\DefineMacro{\langle Macro \rangle}

Normal Mode: Prints \langle Macro \rangle according to \PrintMacro and adds it to the index with a line number typeset according to \DefineIndexFont.

Produce Mode: Gobbles the first characters of \langle Macro \rangle, just in case.

\DescribeEnvironment{\langle Environment \rangle}

Normal Mode: Prints \langle Environment \rangle according to \PrintMacro and adds it to the index with `(environment)' and a line number typeset according to \DescribeIndexFont.

Produce Mode: Gobbles the first characters of \langle Environment \rangle, just in case.

\DescribeIndexFont{\langle Font specifications \rangle}

Normal Mode: Style of the page number in the index for \DescribeMacro and \DescribeEnvironment entries.

Produce Mode: Does nothing.

\DescribeMacro{\langle Macro \rangle}

Normal Mode: Prints \langle Macro \rangle according to \PrintMacro and adds it to the index with a line number typeset according to \DescribeIndexFont.

Produce Mode: Gobbles the first characters of \langle Macro \rangle, just in case.

\DocStripMarginpar

Normal Mode: Sets the adequate values for the proper printing of macros with \DescribeMacro and \DefineMacro (and variants for environments), so that they appear \marginpar ed as with DocStrip. More precisely, it executes \reversemarginpar, and sets \marginparpush to 0pt and \marginparwidth to 8pc.

Produce Mode: Does nothing.

\eTeXOff

Normal Mode: All subsequent example environments are processed with an external file, whose extension is \exp.

Produce Mode: Does nothing.

\eTeXOn

Normal Mode: All subsequent example environments are processed without an external file. This is default. (Requires \e-T\eX, of course.)

Produce Mode: Does nothing.
\begin{itemize}
  \item \texttt{\gobble{\text{name}}}
  \begin{description}
    \item [Normal Mode] The number of characters that will be gobbled at the beginning of each example and code environments. In case of a blank line, nothing is gobbled, but a blank line is added. Tab characters count as one character.
    \item [Produce Mode] Same as in normal mode, but when writing to the file in production.
  \end{description}
  \item \texttt{\marg{\text{name}}}
  \begin{description}
    \item [Normal Mode] Does nothing.
    \item [Produce Mode] Text to be written at the beginning of a file when it is opened with \texttt{\producefile{}}. Comment characters will be automatically added at the beginning of each line. Ends of lines are obeyed. If the \texttt{noheader} option is on, nothing is written.
  \end{description}
  \item \texttt{\ignoreprefix{\text{Macro prefix}}}
  \begin{description}
    \item [Normal Mode] Ignores (Macro prefix) when sorting index entries generated by \texttt{\descreemacro{}} and \texttt{\definemacro{}}. (Macro prefix) will be typeset according to \texttt{\printprefix{}} in the index.
    \item [Produce Mode] Does nothing.
  \end{description}
  \item \texttt{\linenumber{\text{name}}{\text{font specifications}}{\text{width}}{\text{number}}}
  \begin{description}
    \item [Normal Mode] The line number of (Name) will be typeset according to (Font specifications) in a box that will spread from the left margin into the main text width by a length of (Width) (0pt by default). The next (Name) will start at (Number) if specified.
    \item [Produce Mode] Does nothing.
  \end{description}
  \item \texttt{\meta{\text{name}}}
  \begin{description}
    \item [Normal Mode] \texttt{\meta{\text{argument}}} prints \texttt{\text{argument}} (mandatory argument).
    \item [Produce Mode] Does nothing.
  \end{description}
  \item \texttt{\oarg{\text{name}}}
  \begin{description}
    \item [Normal Mode] \texttt{\oarg{\text{argument}}} prints \texttt{\text{argument}} (optional argument).
    \item [Produce Mode] Does nothing.
  \end{description}
  \item \texttt{\parg{\text{name}}}
  \begin{description}
    \item [Normal Mode] \texttt{\parg{\text{argument}}} prints \texttt{\text{argument}} (picture argument).
    \item [Produce Mode] Does nothing.
  \end{description}
\end{itemize}

\texttt{\definemacro{\text{name}}{\text{code input}}{\text{code output}}{\text{immediate execution}}}
\begin{description}
  \item [Normal Mode] Creates (Name) as an example environment to provide \texttt{\codeinput{}} as (Code input) (where the code to be typeset is represented by \#) and \texttt{\codeoutput{}} as (Code output) (where the code to be executed is represented by \#). When encountered, (Name) executes (Immediate execution). (Code input), (Code output) and (Immediate execution) can be empty.
  \item [Options]
    \begin{description}
      \item [numbered:] Each line of (Name) is numbered.
      \item [continuous:] Each line of (Name) is numbered and numbering continues from one (Name) to the other.
      \item [visibleEOL:] If (Name) is processed with e-\LaTeX{}, This prevents ends of lines and commented parts of lines from being removed before anything is executed in \texttt{\codeinput{}}. See page 13 for a discussion.
    \end{description}
  \item [Produce Mode] Adds (Name) to the list of dangerous environments and gobbles the remaining arguments.
  \item \texttt{\producefile{\text{file name}}{\text{file version}}{\text{file date}}}
  \begin{description}
    \item [Normal Mode] Provides (File) as \texttt{\filesource{}}, (File name) as \texttt{\filename{}}, (File version) as \texttt{\fileversion{}} and (File date) as \texttt{\filedate{}}.
    \item [Produce Mode] Opens (File) and writes the header (unless \texttt{noheader} is on), unless (File) is already open and \texttt{\autoclose{}} is not specified, in which case \texttt{CodeDoc} will simply put (File) back in production. Subsequent code will be written to this file. Closes the current file if \texttt{\autoclose{}} is on. Provides (File name) as \texttt{\filename{}}, (File version) as \texttt{\fileversion{}} and (File date) as \texttt{\filedate{}}. To be used with \texttt{\codesign{}{\text{escape}}{}}.
    \item [Produce Mode] Same as \texttt{\newexample{}} to redefine (Name).
    \item [Produce Mode] Same as \texttt{\renameexample{}} to redefine (Name).
  \end{description}
\ShortCode{⟨Character⟩}

Normal Mode: Turns ⟨Character⟩ into a shorthand for \begin{document} and \end{document}.

Produce Mode: Like in normal mode: everything between two ⟨Characters⟩ will be written to the file in production.

\ShortVerb{⟨Character⟩}

Normal Mode: Turns ⟨Character⟩ into a shorthand for \verb.

Produce Mode: Subsequently gobbles everything between two ⟨Characters⟩.

\StartIgnore

Normal Mode: Does nothing.

Produce Mode: Marks the end of \StartIgnore.

\StopIgnore

Normal Mode: Does nothing.

Produce Mode: Marks the end of \StopIgnore.

\TabSize{⟨Number⟩}

Normal Mode: Sets the number of spaces to represent a tab character in verbatim contexts.

Produce Mode: Does nothing.

\UndoCodeEscape

Normal Mode: Does nothing.

Produce Mode: Sets the \CodeEscape character to a normal character.

\UndoShortCode

Normal Mode: Sets the \ShortCode character to a normal character.

Produce Mode: Sets the \ShortCode character to a normal character.

\UndoShortVerb

Normal Mode: Sets the \ShortVerb character to a normal character.

Produce Mode: Sets the \ShortVerb character to a normal character.

\UndoVerbBreak

Normal Mode: Sets the \VerbBreak character to a normal character.

Produce Mode: Sets the \VerbBreak character to a normal character.

\UndoVerbCommand

Normal Mode: Sets the \VerbCommand characters to normal characters.

Produce Mode: Sets the \VerbCommand character to normal characters.

\VerbBreak{⟨Character⟩}

Normal Mode: Turns ⟨Character⟩ into a line breaker in verbatim contexts; more precisely, the line will break where ⟨Character⟩ appears and will be indented with the same amount of space as the original one. ⟨Character⟩ is ignored in \CodeOutput.

Produce Mode: Ignores ⟨Character⟩ when writing to the file in production.

\VerbCommand{⟨Escape⟩}{⟨Left brace⟩}{⟨Right brace⟩}

Normal Mode: Turns ⟨Escape⟩ into an escape character in verbatim contexts, and ⟨Left brace⟩ and ⟨Right brace⟩ into characters of category 1 and 2 respectively. In \CodeOutput, ⟨Escape⟩ gobbles all subsequent letters and everything between ⟨Left brace⟩ and ⟨Right brace⟩ is gobbled too.

Produce Mode: Does the same as normal mode for \CodeOutput. Letters following ⟨Escape⟩ are gobbled, as is everything between ⟨Left brace⟩ and ⟨Right brace⟩.
Part II
Implementation

The usual things ( ; is my \CodeEscape character). Turning ^^? into an active character is less usual but useful to delimit ends of code material.

\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{;FileName}[:FileDate :FileVersion Code and documentation in one file.]
\makeatletter
\catcode`^^?=13

4 Options and basic definitions

\cd@GetClass

Options are mostly conditional switching, \cd@tracingmode will be used in an \ifcase statement. \cd@GetClass will be analyzed to retrieve the class and its options.

\cd@GetOptions\PassOptionsToClass

\cd@LoadClass

We define \cd@LoadClass as a recursive retrieval of options, then passed to the class with \PassOptionsToClass, which we load. This is done only if we’re not in produce mode, in which case no class is loaded.
Still in normal mode, we load `makeidx` if required and define `\StopHere` accordingly.

```latex
\ifcd@makeindex
  \RequirePackage{makeidx}
  \makeindex
\else
  \let\printindex\relax
\fi
\ifcd@obeystop
  \ifcd@makeindex
    \long\def\StopHere#1{#1\relax\par\printindex\end{document}}
  \else
    \long\def\StopHere#1{#1\relax\par\end{document}}
  \fi
\else
  \long\def\StopHere#1{}
\fi
```

5 Normal mode

Although the following code is used in normal mode only, I did not feel like embedding hundreds of lines under a `\ifcd@produce` conditional. Pure superstition, perhaps.

Here's the switch for \TeX and some shorthands.

```latex
\newif\ifcd@eTeX
\@ifundefined{eTeXversion}{\cd@eTeXfalse}{\cd@eTeXtrue}
\def\cd@Warning{\ClassWarningNoLine{codedoc}}
\def\cd@Error#1{\ClassError{codedoc}{#1}{} }
```

5.1 Describing macros

Most of the following macros are imitated from DocStrip, in a simpler but less careful manner.

The first two are straightforward.

```latex
\def\DocStripMarginpar{\reversemarginpar\marginparpush0pt\relax\marginparwidth8pc\relax}
\def\PrintMacro#1{\noindent\marginpar{\raggedleft\strut\ttfamily#1}\ignorespaces}
```

The `\DescribeIndexFont` and its companions first turn @ into a letter, so that a control sequence containing it is recognized as such, sets `\c@Index`, used in the `\ifcase` statement below (a simple conditional could do the job, since there are only two values, but there might be more someday if one wants to distinguish other index entries, like `\use` macros), and pass their arguments to `\PrintMacro` with the first token `\string`ed (even in the case of an environment, because someone might describe its environment with a `\begin{myenv}` command). In case of a macro, the argument is also passed to `\c@MakeEntry` to index it.

The hyperref package does not work properly with indexes if a style is specified with | in the entry. Since we use such styles, and since we want to use hyperref, we circumvent the problem with `\hyperpage` added to the style. By default, it does nothing, but if the user loads hyperref, it will have the adequate meaning.

```latex
\newcount\c@Index
\def\hyperpage#1{#1}
\def\hyperpage@#1{#1}
\def\DescribeIndexFont#1{\gdef\cdatDescribeFont#1{#1\hyperpage{#1}}}
\def\DescribeIndexFont{}
\def\DescribeMacro{\makeatletter\c@DescribeMacro}
\def\c@DescribeMacro#1{\%}
\def\makeatother{}
\def\c@MakeEntry{0}\%}
\def\c@MakeEntry{}\c@EndOfEntry\%}
\def\PrintMacro{\string#1\%}
```

22
This takes two arguments but considers only the first one, so that \DescribeMacro{\texttt{\textbackslash \texttt{foo\marg{Argument}}} \texttt{\cd@MakeEntry}{\cd@EndOfEntry}} will ignore \marg{Argument}. We pass that argument to \cd@AnalyzeEntry with the escape character removed (for a proper indexing), call \cd@IgnorePrefix on the result and finally \cd@MakeEntry.

The aim of this macro is to process \AtChar. Indeed, \AtChar is MakeIndex's operator to signal that an entry should be indexed under another name (as done here). But \AtChar is also a very popular letter in \TeX's world when it comes to macros. DocStrip's solution is to create a special style file for MakeIndex, so that the function of \AtChar is taken over by another character. But then, when a user compiles a DocStrip document, this style file must be indicated to MakeIndex, which many people might not do. So I prefer to leave MakeIndex alone and process the entry beforehand, replacing \AtChar by a character denotation. That's the job of \cd@AnalyzeEntry, which scans the macro name token by token and replace \AtChar by \cd@AnalyzeEntry.

Here comes the mechanism to remove prefixes when sorting entries. \IgnorePrefix simply resets some values and call \cd@IgnorePrefix on its argument along with a terminator.

This analyzes the prefix just like \cd@AnalyzeEntry above and replaces all occurrences of \AtChar by \cd@IgnorePrefix. Since the name of the macro is \string'ed when subjected to \DefineMacro and others, we also \string all letters of the prefix, which have then category code 12.
Then we just scan the prefix to compute the number of characters it is made of. \cd@AnalyzePrefix is defined accordingly to take the right number of characters out of a macro name (fed in \cd@MakeEntry above) and lump them into \cd@TempPrefix, and define the rest of the entry as the remaining characters up to the terminator.

Comparing prefixes is simply a matter of string testing. In case they match, the entry is redefined as the \cd@RestOfEntry, so that macros will be indexed with the prefix removed.

Finally, \cd@MakeEntry indexes the macro under its name with a prefixed escapechar (since it was removed above) and \Prefix in case it was found to match. We also set some default values.
These again are imitated from the DocStrip bundle, with less care.

The only problem is that hyperref defines \textorpdfstring with \newcommand instead of \def. So we obviously can’t define it here, and we wait for the beginning of the document.

Before entering the intricate realm of verbatim text, here are some simpler definitions.

The choice might seem rather conservative, but things are less dangerous this way.
We also define two templates for error messages in case the user wants to \texttt{undo...} something that was never done or define a new character while one is already in use.

Before defining any character, we run some tests: is it a bad character, and is there another character already in use? In the latter case, if \texttt{\cd@BadChar} should be switched to \texttt{true}. If none of the above applies, we switch the conditional to \texttt{true} define \texttt{\cd@ShortVerb} \texttt{error} with \texttt{\cd@DefErr}. We also store the character’s original catcode to restore if undone.

Then we use the \texttt{~} with lowercase trick to define the character.

A \texttt{\ShortVerb} character makes the adequate modifications to display text verbatim. \texttt{\cd@Verbatim} is \texttt{CodeDoc}’s container of all such modifications (mostly catcode changing). \texttt{\catcode`#1=13} is necessary because the character might be one of the specials whose catcode is changed in \texttt{\cd@Verbatim}, e.g. &. We also launch \texttt{\cd@ShortVerb} which works like \texttt{\verb}. \texttt{\leavevmode} is needed in case the \texttt{\ShortVerb} character starts a paragraph, as in the one you’re reading.
Finally we (re)define \undo to restore the original catcode and switch the appropriate conditional. Last but not least, we make the character active.

\undo

This is the default definition for this command, when no \shortverb has been defined.

\shortverb

\shortverb works with the same pattern as \shortverb with important variations. First, we check whether there's an optional argument.

\cdsmake\shortcode

Then we define the real macro. We store the name of the environment and run the same tests as above.

\cdserror

Then we check whether the environment exists, thanks to \example which is defined for \example when created with \newexample.

\cdserror

This is the same as above: we state that a character has been defined as a \shortcode.

\cdsactivate\shortcode

Then we define the character to launch the appropriate environment, but with \ifcd that Code turned to true. What will happen depends on the status of the environment. If it is the default code environment, it will call \cds\shortcode as defined here, which is equivalent to \code itself (see below). On the other hand, if the environment is an example environment, the special example macro will be called and delimit its argument with \cds\shortend, which is the \shortcode character itself. \cdsactivate\shortcode is needed to reactivate the character in case it was one of the specials, as we did for \shortverb.
The rest is equivalent to \ShortVerb above.

\VerbBreak \VerbBreak starts as above.

However, \VerbBreak characters become active only in verbatim contexts. We create \cd@ActivateVerbBreak to that end. When active the character stores the current value of \everypar and then empties it (because the broken line should start with nothing).

Then we set a scratch dimension to \cd@FirstSpaces times the width of a space in the current font. \cd@FirstSpaces is incremented by spaces and tabs at the beginning of each lines. In case the current environment is numbered, we increase our scratch dimension by the width of the box containing the number, stored in \langle Environment \rangle \@cd@boxwidth.

```
\dimen0=\cd@FirstSpaces\fontdimen2\font %
\advance\dimen0 \@cd@boxwidth %
\fi%
```
Finally, we create a paragraph, turn to horizontal mode, restore `everypar` in its initial value and create a space of the desired width, namely the same as the space at the beginning of the original broken line.

The character should be ignored in `\CodeOutput`, and this is what we do here. The `\UndoVerbBreak` variant simply sets these commands to `\relax`.

`\VerbCommand` is similar once again. We define `\cd@ActivateVerbCommand` to change the codes of the characters to 0, 1 and 2 in verbatim contexts and `\cd@IgnoreVerbCommand` to turn the second character into a command that gobbles its argument, delimited by the third character. This is straightforward, but the first character is more complicated: it has to gobble letters and only letters.

```
\def\cd@IgnoreVerbBreak\relax
\def\cd@ActivateVerbCommand\relax
\def\cd@IgnoreVerbCommand\relax
\def\\cd@ActivateVerbCommand\relax
\def\\cd@IgnoreVerbCommand\relax
\def\\cd@ActivateVerbCommand\relax
```

Gobbling letters is not a very delicate process. We take the next token, check whether it is of category 11, and eat it away if it is the case. That’s the reason why \VerbCommand is not very sound. If the next token happens to be a macro (as might be the case since in \CodeOutput, since the escape character is turned back to 0), trying to evaluate its catcode is not a good idea.

Finally, \CodeEscape doesn’t do much in normal mode. We simply check characters.

\verbdefs
Here comes the time to do some verbatim. We start with space. \VerbBreak is the conditional switched to true if we’re in a starred verbatim environment. We define the visible space character to be space of category 12 in typewriter font, as usual.

Since we want spaces at the beginning of a line to count how many they are, so that \VerbBreak can properly break the line, we don’t equate the space character with \@xobeysp (\LaTeX’s verbatim space) or \SpaceChar directly; instead, \SpaceChars will print the space, being called by real spaces in \VerbTab and \VerbSpace. (\^I denotes a tab character).

5.3 Verbatim definitions

\SpaceChars Here comes the time to do some verbatim. We start with space. \Star is the conditional switched to true if we’re in a starred verbatim environment. We define the visible space character to be space of category 12 in typewriter font, as usual. (\^I denotes a tab character).
In verbatim contexts, a space takes the next character as an argument; in case `\ifcd@NewLine` is true, which it is at the beginning of every line (thanks to an `\everypar`), it increments `\cd@FirstSpaces`, which is used by `\VerbBreak`. A tab character does the same except that the `\cd@FirstSpaces` is increased by the value of `\TabSize` (stored in `\cd@TabSize`). In case the next character is not a space or a tab, `\ifcd@NewLine` is set to false.

Spaces leaves a `\cd@ObeySpaces` while tabs create an empty box of width `\TabSize` times the width of a space in the current font.

413 `\edef\cd@VerbSpace#1{%`\cd@ObeySpaces%`\ifcd@NewLine\advance\cd@FirstSpaces1\relax\fi%`\if#1\~\else\if\char26#1\else\cd@NewLinefalse\fi\fi#1}{{}`

418 `\if\char26#1\else\if\char26#1\else\cd@NewLinefalse\fi\fi#1%`\cd@VerbTab#1%`\leavevmode\hbox\to\cd@TabSize\fontdimen2\font{\hss}`\ifcd@NewLine\advance\cd@FirstSpaces\cd@TabSize\fi%`\if\char26#1\else\if\char26#1\else\cd@NewLinefalse\fi\fi#1;

Here comes the verbatimizer. First, we cancel the parindent and sets `\hfuzz` to `\cd@BoxTolerance`, which stores the argument of `\BoxTolerance`.

423 `\def\cd@Verbatim{%`\parindent\z@%`\hfuzz=\cd@BoxTolerance%`Then, if a `\ShortVerb` was defined, we undo it, so that it appears as any other character in this context. If this verbatim was called by the `\ShortVerb` character itself, remember that it restores itself to 13.

426 `\if\cd@ShortVerb%`\UndoShortVerb%;

428 `\fi%`If we're not in a verbatim context called by `\ShortCode`, we undo it, for the same reason.

429 `\if\cd@ShortCode%`\else%`\if\cd@ShortCodeChar%`\UndoShortCode%`\fi%;

434 `\fi%`We change the usual catcodes and reactivate the `\ShortCode` character, just in case it was changed by `\dospecials` or `\@noligs`. We activate the verb break and the verb command, and the rest is straightforward.

435 `\let\do\@makeother\dospecials\@noligs%`\if\cd@ShortCode%`\cd@ActivateShortCode%;

439 `\fi%;`\if\cd@ShortCode%`\cd@ActivateVerbBreak%;

444 `\catcode`\~`=13\relax%`

448 `\cd@MakeSpace}%;

These are pretty straightforward too. I defined a macro instead of a simple dimension or number, because it seems to me that something like `\TabSize{25}` is much more common in the \LaTeX world than `\TabSize25`. Besides, a `\relax` is automatically added, which avoids errors.

444 `\newdimen\cd@BoxTolerance%`\def\BoxTolerance#1{\cd@BoxTolerance=#1\relax};

446 `\def\TabSize#1{\chardef\cd@TabSize=#1\relax};

447 `\TabSize2;`
5.4 The default code environment

\CodeFont The basic code environment is quite simple. First, we define \CodeFont, which simply stores its argument in \cd@CodeFont, to be released later. The following macros are explained more properly in the definition of \NewExample below.

\def\CodeFont#1{\def\cd@CodeFont{#1}}
\CodeFont{\ttfamily}
\newcount\code@cd@LineNumber
\def\code@cd@boxwidth{0pt}
\def\code@cd@BoxStyle{\rmfamily\footnotesize}
\gdef\code@cd@LineNumberBox{%
  \global\advance\code@cd@LineNumber1\relax%
  \def\@currentlabel{\the\code@cd@LineNumber}%
  \hbox to\code@cd@boxwidth{%\hss\code@cd@BoxStyle\relax\the\code@cd@LineNumber\enspace}}%
\let\code@cd@EOL\iffalse%
\textbf{We create a paragraph and stores the name of the environment (used in \VerbBreak to check the width of the line number box).}

\def\code{%
  \endgraf%
  \bgroup%
  \def\cd@ExampleName{code}%
  \everypar{%\code@cd@LineNumberBox\cd@NewLinetrue\cd@FirstSpaces0\relax}%
  \cd@CodeFont%
}\textbf{We launch the verbatim definitions and the complicated \cd@ObeyLines (see below) that makes ends of lines work properly (gobbling characters if needed).}

\cd@VerbBreak\cd@ObeyLines%
\textbf{Every new paragraph, i.e. every line in that context, typeset the line number and switches some values explained above. We also set the font.}
\everypar{%\code@cd@LineNumberBox\cd@NewLinetrue\cd@FirstSpaces0\relax}%
\cd@CodeFont%
\textbf{Finally, we call the proper macro, depending on whether \code was called by \begin{code}, \begin{code*} or the \ShortCode character.}

\ifcd@ShortCode%
  \global\cd@ShortCodetrue%
  \else\ifcd@Star%
    \global\cd@Startrue%
    \else%
      \let\cd@next\cd@Code%
      \fi\fi\cd@next}
\textbf{The starred variant of \code switches to true the conditional used just above. Let’s also define the invisible environment, which takes an argument delimited by \end{invisible} and thus needs to turn some catcodes.}
\expandafter\def\csname code*\endcsname{\cd@Startrue\code}
\def\invisible{%
  \bgroup%
  \catcode`\=12 \catcode`\{=12 \catcode`\}12 \catcode`\^{}=13 \%
487 \cd@Invisible

The ^^? character is used to delimit the end of the verbatim material (this is important because all ends of line scan ahead, see below). Since it is compared in an \ifx conditional, I define it to do nothing but with a distinct definition.

488 \gdef^^?{\cd@UnlikelyCommand}
489 \gdef\cd@UnlikelyCommand{}

\begin{code}
\begin{code*}
\end{code}
\end{code*}

That's the reason why we distinguish \cd@Code and \cd@StarCode. Apart from that, they do the same: they typeset their argument (the first one is the end of the line) and close the environment. \cd@StartGobble is, obviously, the character gobbler for the first line. \cd@Invisible also matches its end but prints nothing.

490 \begingroup
491 \catcode`|=0
492 \catcode`<=1
493 \catcode`>=2
494 \catcode`{=12
495 \catcode`}=12
496 \catcode`\^^M=13 %
497 \catcode`\\=12 %
498 |gdef|cd@Code#1\^^M#2|end\begin{code}<|cd@StartGobble#2\^^?|egroup|end\begin{code}>%
499 |gdef|cd@StarCode#1\^^M#2|end\begin{code*}<|cd@StartGobble#2\^^?|egroup|end\begin{code*}>%
500 |gdef|cd@Invisible#1\^^M#2|end\begin{invisible}<|egroup|end\begin{invisible}>%
501 \endgroup

Here comes a fastidious part. Because we want to gobble characters at the beginning of each line (according to \Gobble), ends of lines do not simply create a new paragraph, they also give a look at the next line and gobble the adequate number of characters. Unfortunately, their definition changes slightly according to the context (default code and examples with or without \TeX). Let’s set the stage.

502 \newcount\cd@GobbleCount%
503 \begingroup
504 \catcode`\^^M13\relax%

\begin{code}
This is the gobbler called at the beginning of the material enclosed in a default code environment. If we meet ^^?, i.e. if the environment is empty, we do nothing.

505 \gdef\cd@StartGobble#1\%
506 \ifx#1\^^?%
507 \cd@GobbleCount=0 \%
508 \let\cd@next\relax%

Else, if we have reached the value set by \Gobble (stored in \cd@GobbleNum), we replace the token we were considering in the stream.

509 \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
510 \cd@GobbleCount=0 \%
511 \def\cd@next{\%}

If we meet an end of line character, that is, if the environment begins with a blank line, we put it back too (it will create a paragraph, among other things).

512 \else\ifx#1\^^M%
513 \cd@GobbleCount=0 \%
514 \def\cd@next{\%}

Finally, if none of the above apply, we keep gobbling.

515 \else%
516 \advance\cd@GobbleCount1 \%
517 \let\cd@next\cd@StartGobble%
518 \fi\fi\fi\cd@next\%
In the code environment, ends of lines act exactly like \cd@StartGobble except that they create a paragraph in the first three cases.

5.5 Example environments

Examples are quite different from the default code environment, since they provide both the input and the output of a code. Besides, if available, they make use of \vTeX.

Here’s the command to switch from \vTeX to external file.

\eTeXOn
\eTeXOff

\NewExample and \RenewExample work similarly but in an inverted way. Both test for options and launch \cd@NewExample on the options and example name if nothing is wrong. Beforehand, they turn # into an active character, which will be \let later to the code material with additional macros.

\cd@GobbleThree
Use \protect\NewExample\space to redefine it\%
\let\cd@next\cd@GobbleThree\%
\else\expandafter\ifx\csname #2\endcsname\code\%
\def\CodeFont\%
\cd@Error\%
You have redefined the ‘code’ environment.\MessageBreak
\string\CodeFont\space is no longer operative\%
\fi\%
\def\cd@next{\catcode`#=13 \cd@@NewExample{#1}{#2}}\%
\fi\cd@next\%
\def\cd@GobbleThree#1#2#3{}

Here is the working mechanism behind both \NewExample and \RenewExample. Since \# will have a special function, we do some \texttt{catcode} changing. The definition is \texttt{\long}, of course.

\begingroup
\catcode`"=6 \%
\catcode`#=13 \%
\long\gdef\cd@@NewExample"1"2"3"4"5{% \%
We define some default values: \texttt{\langle Example\rangle@\cd@EOL} is a switch used when the example is processed with \epsilon-T\!\!\!\!\TeX, indicating whether ends of lines are visible or not. By default, they aren’t, but options may change it. \texttt{\langle Example\rangle@\cd@LineNumberBox} is the command used in examples to typeset the line number. By default, it is set to \texttt{\relax} because examples have no line number.
\texttt{\langle Example\rangle@\cd@ExampleName} \%
We store the name of the example to be retrieved when the environment is processed, but actually it is stored here for the options. Finally, we analyze options with a terminator.
\expandafter\gdef\csname"2@\cd@EOL\endcsname{\iffalse}%
\expandafter\let\csname"2@\cd@LineNumberBox\endcsname\relax%
\def\cd@ExampleName{"2}%
\cd@ExampleOptions"1,\cd@end,%
\gdef\cd@MakeExample{"5}%

Finally, we launch the example maker with the name of the environment (to match its proper end).
\cd@Example{"2}%
\gdef\\codeInput{%
We also define the starred version of \texttt{\langle Example\rangle}, whose only difference is to switch the star conditional. Finally, we restore the category code of \# and close.
\expandafter\def\csname"2\endcsname{%
\global\cd@Startrue%
\gdef\\codeInput{%
\def\\cd@ExampleName{"2}%
\let\#\cd@Input "%3}%
\gdef\\codeOutput{%
\def\\cd@ExampleName{"2}%
\let\#\cd@Output "4}%
\gdef\\cd@MakeExample{"5}]%

Finally, we launch the example maker with the name of the environment (to match its proper end).
\cd@Example{"2}]%
Now we process options. First we define some keywords.

\cd@numbered
\cd@continuous
\cd@visibleEOL
\cd@empty

This is the option processor. It is recursive and stops when it meets the terminator. It simply stores the name of the option and acts accordingly.

If the option is numbered, we create a new count register, set the width of the box containing the number to 0pt by default, and define the style of this number to be \relax by default too. They will be modified by \LineNumber.

We then define the macro executed by the environment for the line number; it increments the count, stores its value as the current label for \label and \ref, create a box of the desired width, flushes everything to the right, executes the style and typeset the value of the counter.

If the option is continuous, we do the same thing, except that the count register is created if and only if it does not already exist (so that a modified \continuous example environment will continue where it stopped; the user may use \LineNumber to start back from 0), and the \advance of the count is \global, so that the last value is always retained from one environment to the other.
The `visibleEOL` option simply sets the relevant conditional to `true`.

The `LineNumber` is straightforward. After some testing, it sets the macro created above to the values specified. If a a square bracket follows, it executes `\cd@SetLineNumber`.

The default example environment is thus easily created.

And here comes the core example environment. First, some catcode changing.

This prepares the conditions for the processing of the material. Let's start with the usual stuff:
Now, if the environment was called by a \ShortCode character, there is no environment to close \cd@EndEnv executes \end{Environment}. We call \cd@MakeExampleEnd, defined below, on the character, and we reactivate this character just in case it was one of the special.

If the environment was called by a regular \begin{Environment} statement, we define the proper end (the argument comes from \(\langle\text{Example}\rangle\), see the definition in \cd@@NewExample above). If there exists a \ShortCode character, we undefine it.

If there's a short verb, we turn it off, we set tabs to 12 so they are written to the file as any other character, we activate ends of lines and in case \(\varepsilon\text-T\!E\!X\) is to process the example, we also activate comment characters (\(\varepsilon\text-T\!E\!X\)'s scanning mechanism is peculiar and commented parts of the code wouldn't be taken into account otherwise).

\cd@MakeExampleEnd \cd@ExampleEnd \cd@ExampleEnd end. It also launches the real processing, depending on the use of \(\varepsilon\text-T\!E\!X\) or not.

The argument has been passed in \cd@Example above, and is either \end{Environment} (with the proper catcodes) or the \ShortCode character.

In case we’re using \(\varepsilon\text-T\!E\!X\), we close some groups and environments, empty \everypar and assign the input. We switch the star conditional after that, because it is needed when the input is assigned and \cd@Verbatim is called.
If we’re not using ε-TEx, we do some testing beforehand. We just want to inform the user that we’re opening an external file. If it already exists, we keep silent.

```
\else
\def\cd@ExampleEnd##1^^M##2#1{% 
  \expandafter\ifx\csname cd@TestRead\endcsname\relax%
  \newread\cd@TestRead%
  \fi%
  \openin\cd@TestRead=\jobname.exp %
  \ifeof\cd@TestRead\relax%
  \cd@Warning{%
    You're not running on eTeX or you've said \string\eTeXOff.\MessageBreak%
    I create the file \jobname.exp to produce\MessageBreak%
    the example environment on line \the\inputlineno.\MessageBreak%
    You can delete it whenever you want, but\MessageBreak%
    keeping it prevents this message from reappearing.}%
  \fi%
  \closein\cd@TestRead %
\fi}
```

If it does not already exist, we create the output stream \cd@expFile, which opens an external scratch file for example processing.

```
\def\cd@noeTeXEOL{\ifx#1\relax\let\relax\cd@next\fi%}
```

Finally, we close everything and assign input once again.

```
\begingroup
  \catcode\^^M\active%
  \gdef\cd@noeTeXEOL#1{%
    \ifx#1\relax\let\relax\cd@next\fi%
    \ifx#1\^^M\let\cd@next\relax%
      \immediate\write\cd@expFile{\cd@noeTeXEOL}%
    \else\ifx#1\^^M\let\cd@next\relax%
      \cd@GobbleCount=0 \let\relax\cd@next%
    \fi\fi\fi%
  \let^^M\relax%
  \let\cd@next\relax%
\endgroup
```

\cd@expFile

5.5.1 Examples without ε-TEx

\cd@noeTeXEOL

Here’s how ends of lines are processed when writing the code material to an external file. If we find ‘^^?’, which marks the end of the material, we stop.

```
\begingroup
  \catcode\^-\active%
  \gdef\cd@noeTeXEOL#1{%
    \ifx#1\relax\let\relax\cd@next\fi%
    \cd@GobbleCount=0 %
    \let\relax\cd@next%
    \let\cd@next\relax%
    \let\cd@next\relax%
  \let^^M\relax%
  \let\cd@next\relax%
\endgroup
```

If we find an end of line, that means there’s a blank line, and we write it to the \jobname.exp.

```
\else\ifx#1\relax\let\relax\cd@next\fi%
\cd@GobbleCount=0 %
\def\cd@next{\immediate\write\cd@expFile{\cd@noeTeXEOL}}%
```
If we have gobbled enough characters, we write the line to the external file. Otherwise, we repeat.

```latex
\ifnum\cd@GobbleCount=\cd@GobbleNum\%
\cd@GobbleCount=0 \%
\def\cd@next{\cd@LineWrite#1}\
\else\%
\advance\cd@GobbleCount1 \%
\let\cd@next\cd@noeTeXEOL\fi\fi\cd@next}\
```

The line written is delimited by its end. This explains the `^` suffix at the end of the material on line 742. In case `\end{Example}` occurs on its own line, we need a terminator, hence the first `^`. If it occurs at the end of the last line, as in ... end of code `\end{code}`, we need `^\par` so that the argument of \cd@LineWrite is properly delimited. The first `^` is then written to the file, but it expands to nothing. Since \cd@LineWrite calls `\cd@noeTeXEOL`, we need another delimitator, hence the second `^`.

```latex
\gdef\cd@LineWrite#1^\par{\immediate\write\cd@expFile{#1}\cd@noeTeXEOL}\
```

Now we define the macro that will be used in \CodeInput (where \# is \let to \cd@Input) and \CodeOutput (where it is \let to \cd@Output).

```latex
\newtoks\cd@Everypar\%
\gdef\cd@AssignInput{%
\gdef\cd@Input{%
\bgroup\cd@Everypar\everypar{%
\leavevmode\csname\cd@ExampleName@cd@LineNumberBox\endcsname\relax%
\cd@NewLinetrue\cd@FirstSpaces0\relax\the\cd@Everypar\relax}%
\cd@Verbatim\def\par{\leavevmode\endgraf}%
\input{\jobname.exp}%
\egroup}%
```

The output also reads from the file and simply ignores verb breaks and commands.

```latex
\gdef\cd@Output{%
\bgroup\cd@IgnoreVerbBreak\cd@IgnoreVerbCommand\input{\jobname.exp}%
\egroup}%
```

Finally, we execute the last argument to \NewExample, i.e. what was dubbed here \textit{Immediate execution}.

```latex
\cd@MakeExample}\
```

### 5.5.2 Examples with $\varepsilon$-\TeX

Examples with $\varepsilon$-\TeX are much more complicated. We use the \scantokens command, whose function is to read its argument as if catcodes were not fixed. For instance, \def\scan#1{{\catcode`\=12\scantokens{#1}}} \scan\foo yields \foo, although the backslash was an escape character when read. The problem is that \scantokens interprets ends of lines and comments characters with their current values. Ends of lines yields a `\par` token as usual; the problem is that this token is scanned anew, and if you have turned the backslash to a category 12 character, it will appear as such. Moreover, commented parts of a line are ignored. For instance, \scan
a\% mycomment

yields \texttt{a\% mycomment\^^M\^^M b\^^M}. So \texttt{\textbackslash\texttt{s\textbackslash\texttt{c\textbackslash\texttt{a\textbackslash\texttt{n\textbackslash\texttt{t\textbackslash\texttt{o\textbackslash\texttt{k\textbackslash\texttt{e\textbackslash\texttt{s}}}}}}} as it stands is not appropriate for verbatim material. 

The solution is to turn ends of lines and comments to other categories beforehand. Thus the previous example yields \texttt{a\% mycomment\^^M\^^M b\^^M}. (The final end of line is added by \texttt{\textbackslash\texttt{s\textbackslash\texttt{c\textbackslash\texttt{a\textbackslash\texttt{n\textbackslash\texttt{t\textbackslash\texttt{o\textbackslash\texttt{k\textbackslash\texttt{e\textbackslash\texttt{s}}}}}}}.) Now we need some hacking to produce the desired result.

\begin{verbatim}
\texttt{\textbackslash \texttt{cd\textbackslash\texttt{Input}}} \begin{verbatim}

700 \long\gdef\cd@AssigneTeXInput#1{%
701 \gdef\cd@Input{%
702 \bgroup%
703 \cd@Everypar\everypar%
704 \everypar%
705 \leavevmode\cname\cd@ExampleName \cd@LineNumberBox\end\cname\relax%
706 \cd@NewLine\true\cd@FirstSpaces\0\relax\the\cd@Everypar\relax}%
707 \cd@Verbatim%

We define ends of lines as yet another gobbling mechanism. We use \texttt{\^^?} once again to delimit material, and define it to make ends of lines ignored in case it is read, so that the additional \texttt{\^^M} at the end of \texttt{\textbackslash\texttt{s\textbackslash\texttt{c\textbackslash\texttt{a\textbackslash\texttt{n\textbackslash\texttt{t\textbackslash\texttt{o\textbackslash\texttt{k\textbackslash\texttt{e\textbackslash\texttt{s}}}}}}} will be ineffective.

708 \catcode\^^M=13 %
709 \let\^^M\cd@eTeXStartGobble%
710 \catcode\^^?=13 %
711 \catcode\%=13 %
712 \def\^^?{\catcode\^^M=9\relax}%
713 \scantokens{^^M#1^^?}%
714 \egroup}%

\texttt{\textbackslash \texttt{cd\textbackslash\texttt{Output}}} \begin{verbatim}

Output is still worse. Even comments are active.

804 \gdef\cd@Output{%
805 \bgroup%
806 \cd@IgnoreVerbBreak%
807 \catcode\^^?=13 %
808 \catcode\%=13 %
809 \catcode\^^M=13 %

The next step depends on the user's choice about ends of lines. If they are visible, we process the material as is, with special definitions of \% and \texttt{\^^M} to mimic \TeX{}'s normal behavior.

810 \cname\cd@ExampleName \cd@EOL\end\cname%
811 \cd@VisibleComment%
812 \let\^^M\cd@TeXOutVisibleEOL%
813 \def\^^?{\let\^^M\relax}%
814 \cd@IgnoreVerbCommand%
815 \scantokens{\%\#1\#2}%

If ends of lines are not visible, we execute the material beforehand with only \% and \texttt{\^^M} effective, to remove unwanted code. Macros are not executed because the backslash is still of category 12. Once ends of lines are thus processed, we scan everything anew, ignoring the last \texttt{\^^M} and \texttt{\^^?}, which has a special function (see below).

816 \else%
817 \cd@ActiveComment%
818 \let\^^M\cd@TeXOutEOL%
819 \def\^^?{\catcode\^^M=9\relax}%
820 \scantokens{\#1\#2}%
821 \cd@IgnoreVerbCommand%
822 \catcode\^^M=9 %
823 \catcode\^^?=9 %
824 \expandafter\scantokens\expandafter{\cd@exinput}%
825 \fi%
826 \egroup}%
827 \cd@MakeExample\egroup}%

\end{verbatim}
\end{verbatim}

41
Once again, macros to gobble the right number of characters at the beginning of each line. These are for the input. It is not possible to put \cde\TeXStartGobble directly at the beginning of \scantokens, because the backslash would not be understood as an escape character. Thus we have to \let \^M to it, and once it has done its job, make it change the meaning of \^M to \cde\TeXEOL. (That's also the reason why we couldn't reuse the gobble macro of the default code environment, although they are quite similar.)

\begin{verbatim}
828 \gdef\cde\TeXStartGobble#1{%  
829   \ifx#1^^M%  
830      \cd@GobbleCount=0 %  
831      \let\cd@next\relax%  
832   \else\ifnum\cd@GobbleCount=\cd@GobbleNum%  
833      \cd@GobbleCount=0 %  
834      \let\^M\cde\TeXEOL%  
835      \def\cd@next{#1}%  
836   \else\ifx#1\^M%  
837      \cd@GobbleCount=0 %  
838      \let\^M\cde\TeXEOL%  
839      \let\cd@next\^M%  
840   \else%  
841      \advance\cd@GobbleCount1 %  
842      \let\cd@next\cde\TeXStartGobble%  
843   \fi\fi\cd@next}  
844  
845 \gdef\cde\TeXEOL#1{%  
846   \ifx#1^^M%  
847      \let\^M\relax\let\cd@next\relax%  
848   \else\ifx#1\^M%  
849      \par%  
850      \let\cd@next\^M%  
851   \else%  
852      \def\cd@next{#1}%  
853   \fi\fi\cd@next}  
854  
855 \gdef\cde\TeXOutVisibleEOL#1{%  
856   \ifx#1^^M%  
857      \cd@GobbleCount=0 %  
858   \else\ifnum\cd@GobbleCount=\cd@GobbleNum%  
859      \cd@GobbleCount=0 %  
860      \def\cd@next{\let\^M\leavemode\endgraf\^M}%  
861   \else\ifx#1\^M%  
862      \par%  
863   \fi\fi\cd@next}  
864  
865 \gdef\cde\TeXOutEOL#1{%  
866   \catcode\^^@=12\relax%  
867   \ifx#1^^M%  
868      \cd@GobbleCount=0 %  
869   \else\ifnum\cd@GobbleCount=\cd@GobbleNum%  
870      \cd@GobbleCount=0 %  
871      \let\cd@next\relax%  
872   \else%  
873      \par%  
874   \fi\fi\cd@next}  
\end{verbatim}

And now, the output. If ends of lines are visible, we set them to create a \par if the next character is another end of line (i.e. if we find a blank line) or to put it back into the stream otherwise, with a space before.

\begin{verbatim}
879 \gdef\cde\TeXOutVisibleEOL#1{%  
880   \ifx#1^^M%  
881      \let\^M\relax%  
882   \else\ifnum\cd@GobbleCount=\cd@GobbleNum%  
883      \cd@GobbleCount=0 %  
884      \let\cd@next\relax%  
885   \else%  
886      \par%  
887   \fi\fi\cd@next}  
888 \gdef\cde\TeXOutEOL#1{%  
889 \catcode\^M=12\relax%  
890 \ifx#1\^M%  
891 \else\ifnum\cd@GobbleCount=\cd@GobbleNum%  
892 \cd@GobbleCount=0 %  
893 \else%  
894 \par%  
895 \fi\fi\cd@next}  
\end{verbatim}

If ends of lines are not visible, i.e. if they are processed before anything else, we do something similar, except that we add a dummy character, which will be ignored when the material is scanned, but will nonetheless prevent the formation of macro names across lines. Tail recursion is forbidden, since this will be used in a \edef, so we \expandafter instead.

\begin{verbatim}
899 \catcode\^M=12\relax%  
900 \gdef\cde\TeXOutEOL#1{%  
901   \ifx#1\^M%  
902 \else%  
903 \par%  
904 \fi\fi\cd@next}  
\end{verbatim}
No we deal with comments. First we do some catcode changing. (We need a comment character since we're currently in a group where ends of lines are active).

If ends of lines are visible we define comments to eat everything until the end of the line and then launch a macro whose sole purpose is to remove spaces at the beginning of the next line.

If ends of line are not visible, we do the same in the \expandafter way.

5.6 File management

Here are some simple macro for the reader's relief.

Closing a file in normal mode simply makes all file identification macros unavailable.
That's why, in normal mode, we close a file right now. We nonetheless create a dummy file name for the sake of \ProduceFile below.

\ifcd@produce%
\def\FileName{}
\def\FileVersion{}
\def\FileDate{}
\else
\CloseFile{}
\def\FileSource{}
\newcount\@cd@LineCount%
\fi

In normal mode, the main job of \ProduceFile is to reset some line number counts. In autoclose mode, there's only one counter, since files are closed when a new one is opened.

\def\ProduceFile#1{% 
\ifcd@autoclose% 
\code@cd@LineNumber0\relax%
\else% 
\expandafter\csname\FileSource @cd@LineCount\endcsname=\code@cd@LineNumber%
\expandafter\ifx\csname #1@cd@LineCount\endcsname\relax%
\expandafter\newcount\csname #1@cd@LineCount\endcsname%
\code@cd@LineNumber0\relax%
\else% 
\expandafter\code@cd@LineNumber\csname #1@cd@LineCount\endcsname%
\fi%
\fi%

We reset \FileName and others, because their definition is optional. \FileSource is mandatory and is the actual argument of \ProduceFile. We launch the appropriate macro if a left bracket follows.

\def\FileName{% 
\cd@Error{No \string\FileName\space has been given to \FileSource}}%
\def\FileVersion{% 
\cd@Error{No \string\FileVersion\space has been given to \FileSource}}%
\def\FileDate{% 
\cd@Error{No \string\FileDate\space has been given to \FileSource}}%
\edef\FileSource{#1}%
\@ifnextchar[\cd@GetFileName\relax}%

These are straightforward and don't need any comment.

\cd@GetFileName
\cd@GetFileVersion
\cd@GetFileDate
\FileSource

What if I want a comment?
Finally, we define those macros that have no effect in normal mode to have, well, no effect. Since comment signs are 'other' characters in produce mode, we change their catcode here too, so that the user may close the argument to \Header after a comment sign.

\Header \cd@HeaderGobble \AddBlankLine \StartIgnore \StopIgnore \DangerousEnvironment

Since comment signs are 'other' characters in produce mode, we change their catcode here to o, so that the user may close the argument to \Header after a comment sign.

\def\Header{\bgroup\catcode\%=12 \cd@HeaderGobble}
\long\def\cd@HeaderGobble#1{\egroup}
\let\AddBlankLine\relax
\let\StartIgnore\relax
\let\StopIgnore\relax
\def\DangerousEnvironment#1{}

6 Produce mode

We now turn to produce mode, where codedoc becomes CodeDoc and strange things happen.

6.1 Messages

\cd@Tracing \cd@TChar \cd@TUChar \cd@TCode
\ifcase\cd@tracingmode
\def\cd@Tracing#1{}
\def\cd@TChar#1#2{}
\def\cd@TUChar#1{}
\def\cd@TCode{}\or
\def\cd@Tracing#1{\immediate\write17{On line \the\cd@ProduceLine: #1.}}
\def\cd@TChar#1#2{\bgroup
\escapechar\m@ne\cd@Tracing{\string#1 defined as \string\#2}
\egroup}
\def\cd@TUChar#1{\bgroup
\escapechar\m@ne\cd@Tracing{\string\#1 undone}
\egroup}
\def\cd@TCode{\immediate\write17{*** Code written from line \the\cd@ProduceLine space to \the\inputlineno space to \cd@CurrentFile. ***}}
\or
\def\cd@Tracing#1{\immediate\write17{\^^J! CodeDoc Error: \^^J#1\^^Jl.\the\cd@ProduceLine\^^J}}
\def\cd@TChar#1#2{\bgroup
\escapechar\m@ne\cd@Tracing{\string#1 defined as \string\#2}
\egroup}
\def\cd@TUChar#1{\bgroup
\escapechar\m@ne\cd@Tracing{\string\#1 undone}
\egroup}
\def\cd@TCode{\immediate\write17{*** Code written from line \the\cd@ProduceLine space to \the\inputlineno space to \cd@CurrentFile. ***}}
\fi

\cd@CDError \cd@CDWarning \cd@NoFileWarning
\def\cd@CDError#1{\immediate\write17{\^^J? CodeDoc Warning: \^^J#1\^^Jl.\the\cd@ProduceLine \^^J}}
\def\cd@CDWarning#1{\immediate\write17{\^^J! CodeDoc Error: \^^J#1\^^Jl.\the\cd@ProduceLine \^^J}}
\def\cd@NoFileWarning{\cd@CDWarning{No file in production. This code will be lost.}}

6.2 Testing strings

In produce mode, CodeDoc is a string tester; more precisely it imitates \TeX's normal mechanism: the escape character is turned into an active character that gathers letters following it and executes the name they form (in a modified fashion, however, to execute only relevant macros).

First, we redefine what happens at the end of the class to alter the behavior of special characters. However, we maintain comments and turn \ into an active character.
By default, `\normalsize` is an error message, so we redefine it. We start the report.

We don't load any font, so there's no need to bother with overfull boxes nor outputs. However, by pure superstition, I prefer some care. Does he know what he's doing?

Most of the following are already 0. However, `\tracingcommands2` would explode the log file, so we take some care once again.

Some characters are special, to say the least. We need to be able to recognize them.

Here comes the definition of the escape character as itself... The backslash can't be allowed to have `\catcode 0`, otherwise control sequences would form and fire. We don't want that, obviously. On the other hand, some control sequences should be executed, so they must be form beforehand. Here's how it works. First, it stores the current line number for messages.
Then it turns ends of lines and comments to other characters, because we don’t want to pass them unnoticed. If the next character is of category code 11, we start forming a control sequence. Otherwise, we gobble it and stop.

| bgroup |
| catcode`| ^\=12 |
| catcode`| ^\=12 |
| gdef | cd@MacroName{} |
| ifnum | catcode`^\#1=11 |
| def | cd@next{|cd@Gather#1} |
| else |
| def | cd@next{|egroup|relax} |
| fi |

For forming macro names is quite simple: if the next character is a letter, we add it to the temporary name. Otherwise, we store it in cd@NextChar and start doing what \TeX{} does when it has formed a control sequence.

| long | gdef | cd@Gather#1{} |
| ifnum | catcode`^\#1=11 |
| xdef | cd@MacroName{ |cd@MacroName#1} |
| let | cd@next|cd@Gather |
| else |
| gdef | cd@NextChar{#1} |
| let | cd@next|cd@GobbleSpace |
| fi | cd@next |

endgroup

\cd@Gather
Forming macro names is quite simple: if the next character is a letter, we add it to the temporary name. Otherwise, we store it in \cd@NextChar and start doing what \TeX{} does when it has formed a control sequence.

\cd@MacroName
\cd@NextChar

| long | gdef | cd@Gather#1{} |
| ifnum | catcode`^\#1=11 |
| xdef | cd@MacroName{ |cd@MacroName#1} |
| let | cd@next|cd@Gather |
| else |
| gdef | cd@NextChar{#1} |
| let | cd@next|cd@GobbleSpace |
| fi | cd@next |

endgroup

\cd@GobbleSpace
That is, we skip spaces and ends of lines, so that the real next character will be put next to the formed control sequence, in case it is an argument.

In case the next argument is none of the above, we call \cd@Evaluate, which will expand the macro, on the next character.

| long | def | cd@GobbleSpace{} |
| let | cd@next|cd@TakeNextChar |
| ifx | cd@NextChar|cd@Space |
| else | ifx | cd@NextChar|cd@Tab |
| else | ifx | cd@NextChar|cd@EndOfLine |
| else | ifx | cd@NextChar|cd@Comment |
| let | cd@next|cd@GobbleEndOfLine |
| else |
| egroup |
| def | cd@next|\{expandafter|cd@Evaluate|cd@NextChar} |
| fi | fi | fi | fi | cd@next |

endgroup

\cd@TakeNextChar
These do what they say.

| long | def | cd@TakeNextChar#1{|gdef | cd@NextChar#1{ |cd@GobbleSpace} |
| begingroup |
| catcode`^\#1=12 |
| gdef | cd@GobbleEndOfLine#1{ |cd@GobbleSpace} |
| gdef | cd@NextChar#2{ |
| cd@GobbleSpace} |
| endgroup |

\cd@Evaluate
Finally, we take the name thus formed, and execute \langle Name\rangle@Produce. As you might imagine, the only macros containing the @Produce suffix are defined by CodeDoc. So, most of the time, this execution will be no more than a \relax. Which is exactly what we want.

| def | cd@Evaluate{ |csname | cd@MacroName@Produce |endcsname} |
6.3 Macros executed in produce mode

To understand what follows, simply remember that \( \langle \text{Macro} \rangle \@ \text{Produce} \) is executed when CodeDoc encounters \( \langle \text{Macro} \rangle \). So, for instance, \( \langle \text{ShortVerb} \rangle \@ \text{Produce} \) is \( \langle \text{ShortVerb} \rangle \) in produce mode.

Macro names will become quite long, so we add some left margin.

\( \text{\textbackslash cd@Gobble} \) First, some gobbler.

\begin{verbatim}
1079 \def\cd@Gobble##1{}
\end{verbatim}

Macros like \( \langle \text{ShortVerb} \rangle \) can take four kinds of argument. If you want \( + \) to be a \( \langle \text{ShortVerb} \rangle \), you can say \( \langle \text{ShortVerb} \rangle \), \( \langle \text{ShortVerb} \rangle \), \( \langle \text{ShortVerb} \rangle \) and \( \langle \text{ShortVerb} \rangle \). Since CodeDoc has already considered the next character when executing \( \langle \text{ShortVerb} \rangle \@ \text{Produce} \), its catcode can’t be changed, and a left brace is of category 12 and a backslash of category 13. So we have to gobble the next character if it is one of them.

\( \text{\textbackslash cd@PrepareChar} \) takes a macro as an argument and replaces it in the stream with the next character gobbled or not. The backslash is turned into an escape character to handle the \( \langle \text{ShortVerb} \rangle \) case, where the left brace is gobbled; the backslash hasn’t been read yet, so we can use it.

\begin{verbatim}
1080 \def\cd@PrepareChar##1{%
1081 \catcode'\\=0 %
1082 \def\cd@next{\expandafter##1\cd@Gobble}
1083 \ifx\cd@NextChar\cd@LeftBrace%
1084 \else\ifx\cd@NextChar\cd@Escape%
1085 \else%
1086 \def\cd@next{##1}
1087 \fi\fi}
\end{verbatim}

Thus, \( \langle \text{ShortVerb} \rangle \@ \text{Produce} \) calls \( \text{\textbackslash cd@PrepareChar} \) with \( \cd@MakeShortVerb@Produce \), which will do the real job to the character. We define fancyvrb’s \( \text{\textbackslash DefineShortVerb} \) to do the same thing. \( \text{\textbackslash cd@VerbList} \) contains all such characters, since \( \text{\textbackslash DefineShortVerb} \) can define several of them. It will be used in writing environments to neutralize them.

\begin{verbatim}
1088 \def\ShortVerb@Produce{\cd@PrepareChar\cd@MakeShortVerb@Produce\cd@next}
1089 \let\DefineShortVerb@Produce\ShortVerb@Produce
1090 \def\cd@VerbList{}
\end{verbatim}

Now we inform the user that the character was \( \langle \text{ShortVerb} \rangle \)’ed.

\begin{verbatim}
1091 \def\cd@MakeShortVerb@Produce##1{%
1092 \cd@TChar{##1}{ShortVerb}
\end{verbatim}

We add it to \( \cd@VerbList \).

\begin{verbatim}
1093 \expandafter\def\expandafter{\cd@VerbList}\expandafter{\cd@VerbList##1.}
\end{verbatim}

And we simply define the character to gobble everything until its next occurrence.

\begin{verbatim}
1094 \let\cd@ShortVerb@Produce=\relax
1095 \def\cd@ShortVerb@Produce{\cd@TUChar{ShortVerb (from fancyvrb)}
1096 \catcode\#1=12 \catcode\\=13}
1097 \let\UndoShortVerb@Produce\relax
1098 \def\UndoShortVerb@Produce{\cd@TUChar{ShortVerb}}
1099 \catcode\#1=13 \catcode\\=12\relax
1100 \let\cd@ShortVerb@Produce=\relax
\end{verbatim}

We define a variant for fancyvrb, because it takes an argument.

\begin{verbatim}
1101 \let\cd@UndefinedShortVerb@Produce=\relax
1102 \def\cd@UndefinedShortVerb@Produce{\cd@PrepareChar\cd@UndefinedShortVerb@Produce\cd@next}\%
1103 \def\cd@UndefinedShortVerb@Produce##1{
1104 \cd@TUChar{ShortVerb (from fancyvrb)}
1105 \catcode\#1=12 \catcode\\=13\relax
\end{verbatim}

48
In produce mode, the \verb@Break@ character is simply ignored.

This is useful for \verb@CodeOutput@ and also \verb@NewExample@.

The \verb@CodeOutput@ character in produce mode is similar to its counterpart in normal mode, except that it follows what code does in this mode. So give a look at the definition of the code environment to understand what is going on here.

\verb@Command@ characters do what they do in \verb@CodeOutput@ in normal mode. The escape gobble letters and the braces gobble what they contain.

First, we store the escape character for the message.

Then we turn it into a letter gobbler.

\verb@TempEsc@
This is not what you think it is. We're not considering whether the character to come is a left brace, but whether \VerbCommand, i.e. the character following \VerbCommand, was a left brace; this means that a right brace is to come, and we want to gobble it before processing what follows.

\ifx\cd@NextChar\cd@LeftBrace\def\cd@next{\expandafter\cd@VerbBraces@Produce\cd@Gobble}\else\let\cd@next\cd@VerbBraces@Produce\fi\cd@next\end
\VerbCommand

The rest is pretty straightforward and similar to what we did in normal mode.

\verb!CodeEscape! is easy: we simply define a macro to turn the character into an escape in code contexts.

\def\CodeEscape@Produce{\cd@PrepareChar\cd@CodeEscape@Produce\cd@next}
\def\cd@CodeEscape@Produce#1{\cd@TChar{#1}{CodeEscape}}\def\cd@ActivateCodeEscape{\catcode`#1=0\relax}\catcode`\=13\relax}
\let\cd@ActivateCodeEscape\relax
\def\UndoCodeEscape@Produce{\cd@TUChar{CodeEscape}\let\cd@ActivateCodeEscape\relax}

These two macros launch the option gobbler if there are any. \VerbCommand is defined later because it takes its argument between braces of category 12, like other macros.

Ignoring the input boils down to modifying the definition of \VerbCommand until it founds \StopIgnore. Meanwhile, it does nothing.
The produce version of LaTeX’s \verb\gobbles its argument after it has checked for a star.

\beginverbatim
\def\verb@Produce{\count@=0 \cd@VerbEater}
\def\cd@VerbEater#1{\
  \ifcase\count@ \%
  \ifx#1* \%
    \count@=1 \%
    \let\cd@@VerbEater\cd@VerbEater
  \else \%
    \def\cd@@VerbEater##1#1{}
  \fi \%
  \else \%
    \def\cd@@VerbEater##1#1{}
  \fi\cd@@VerbEater}
\endverbatim

The normal counterparts of these might take dangerous arguments, so we need to neutralize them. The first four gobble two tokens, i.e. a left brace and/or an escape character, so the following macro won’t form. The last three just gobble the escape character.

\beginverbatim
\def\begin@Produce{\ifx\cd@NextChar\cd@LeftBrace\expandafter\begin@@Produce\fi}
\def\end@Produce{\ifx\cd@NextChar\cd@LeftBrace\expandafter\end@@Produce\fi}
\endverbatim

The produce version of \Gobble is similar to the normal version, except that it takes cares of braces. \Gobble@@Produce is defined below.

\beginverbatim
\def\Gobble@Produce#1{\ifx\cd@NextChar\cd@LeftBrace%\def\cd@next{\expandafter\Gobble@@Produce\cd@NextChar}%\else\def\cd@next{\chardef\cd@GobbleNum=#1\relax}%\fi\cd@next}
\endverbatim

The header is an easy matter. The only thing not to forget is to change the catcode of \back to 0.

\beginverbatim
\newif\ifcd@HeaderFirstLine
\begingroup
\catcode`\^^M=13 %
\catcode`\%=12 /
\gdef\Header@Produce{\%}
\bgroup\%}
\catcode`\%=12 /\%}
\catcode`\*=0 /51
\%}
\endverbatim
We'll need these presently.

Here comes the macros that take their arguments between braces of category 12.

The \if... will be needed in \input@Produce.

This defines \cd@Header, which is executed in \ProduceFile, to write the text input by the user to the newly opened file. The group we close was opened in \Header@Produce.

\begin statements simply check their argument: if it is code, code* or invisible, it turns to writing mode. Otherwise, the name of the argument is checked against the list of dangerous environments. See below where normal braces are restored.
There's only one thing that can wake an \end statement: document. If it finds
\end{document}, CodeDoc stops. Otherwise, \end statements are ignored.

We define these right now, to be used later.

We need a terribly boring definition of \input for the default header, so that files
are properly tracked back to their source. Besides, \input in \TeX's way, i.e. without
braces, is not allowed anymore, if it is to be read by CodeDoc in produce mode. I feel
like removing the whole thing altogether.

Lines 1289 to 1298 were added in version 0.2. I had overlooked the fact that if an
\input file ended with a control sequence, then the rather complicated mechanism of
\cd@Gather and its friends would run into the end of the file and produce an error
message. With good old \TeX, I don’t know how to overcome this; hence the warning.
With \TeX, however, I use \everyeof to add a pair of braces just for the sake of
some harmless tokens. Anyway, who's using \TeX anymore?

If we find a dangerous environment, we launch this on its name, which eats everything
until \end{\langle Name\rangle}.
Back to normal braces. This is a default value needed in \input@Produce. The extension is just a guess, of course.

This is the checking mechanism used in \begin statement to detect dangerous environments. Note that we check all environments in their starred version too.

6.4 Writing environments

CodeDoc looks for code, code* and invisible environments and process them line by line.

First, we need a recursive code changer.

This is the writing macro, called by \begin when the appropriate argument is found, or by the \ShortCode character. \dospecials is probably useless since all specials are already done, but at least it changes the category of the escape and the comment.

We turn all verb characters (defined by fancyvrb’s \DefineShortVerb) into other characters, ignore the verb break, neutralize the short code if we’re not in a short code environment (the redefinition of \cduChar just prevents an unwanted message sent to the user if tracing is 2) an reactive it otherwise, ignore \VerbCommand and activate \CodeEscape. We turn ends of lines into proper gobbler once again.
Finally we launch the adequate macro. They all do the same thing, but they look for different \end statements.

This is similar to the version for examples without \texttt{-T E X} in normal mode, i.e. it writes to an external file, specified in \texttt{\cd@produceFile}.

And here is the end. It is the first \texttt{\cd@produceEOL}, which launches everything. The conditional switches between an error message (no file in production) and a report (code written).
6.5 File management

This the final step: handling files in produce mode.

\def\cd@Closed{closed}
\def\cd@Open{open}
\def\cd@Wait{wait}

Some basic definitions. \@unused is \LaTeX{}'s unattributed stream for messages. We let it write to the log file. \cd@ProduceFile is the writing macro (used in writing environments above); as long as no file is open, it does nothing.

\newcount\cd@ProduceCount
\def\cd@ProduceFile#1{}
\def\AddBlankLine@Produce{\cd@ProduceFile{}}

This is called by \ProduceFile, via \ProduceFile@Produce above. If the file is closed or already in production, we signal it to the user:

\begin{verbatim}
\let\cd@next\relax
\expandafter\ifx\csname #1@Status\endcsname\cd@Closed
  \cd@CDError{}
  File `#1' has already been closed. --J%
  If I open it again, it will be erased. --J%
  I can't do that. I quit. Sorry.}
\let\cd@next\@@end
\else\expandafter\ifx\csname #1@Status\endcsname\cd@Open
  \cd@CDWarning{}
  File `#1' is currently in production. --J%
  Why do you try to open it again?}
\else\expandafter\ifx\csname #1@Status\endcsname\cd@Wait
\expandafter\let\csname #1@Status\endcsname\cd@Open
\fi\fi
\end{verbatim}

The file is waiting if it has been opened previously and another one has been opened too afterward, provided autoclose is off. In which case, we set it to open:

\begin{verbatim}
\let\cd@next\@End
\expandafter\ifx\csname #1@Status\endcsname\cd@Closed
  \cd@CDError{}
  File `#1' has already been closed. --J%
\else\expandafter\ifx\csname #1@Status\endcsname\cd@Open
  \cd@CDWarning{}
  File `#1' is currently in production. --J%
\fi\fi
\end{verbatim}

\begin{verbatim}
\let\cd@next\relax
\def\cd@ProduceFile{\immediate\write\csname #1@Stream\endcsname}
\let\cd@NoFileWarning\relax
\def\cd@ProduceFile{}
\let\cd@CurrentFile{#1}
\end{verbatim}

We set the current file to wait and define the one we're dealing with to be the current file:

\begin{verbatim}
\expandafter\let\csname \cd@CurrentFile @Status\endcsname\cd@Wait
\def\cd@CurrentFile{#1}
\end{verbatim}

This sounds strange

\begin{verbatim}
\}\\
\end{verbatim}
Now, if the file has never been opened, we need an output stream. If they were all allocated, we look whether some were made available thanks to a `\CloseFile`.

If no stream is found, CodeDoc feels so bad that it quits.

Else, we’re very happy, and if there is already a file in production, we close it or let it wait.

Then we define our file as the current one, let the world know that it is open, allocate the stream to its name, open it, etc., and launch a macro to retrieve some information if any.

If there was an available stream in the first place, we do exactly the same.
This is designed to retrieve optional information following `ProduceFile`. We undo the `ShortVerb` and `ShortCode` because they might appear there. (My `ShortCode` is a slash, which is used in date too.) We also set the backslash as an escape character, because control sequences might appear here.

In all cases, if nothing follows, and if the `noheader` option is off, we write the header to the file.

```
def\cdGetFilename@Produce[#1]{
  \xdef\FileName{#1}
  \catcode`\active
  \@ifnextchar[{
    \catcode`\z@ \cdGetFileVersion@Produce}
  \ifcdNoheader\else\cdHeader\fi\egroup}}
def\cdGetFileVersion@Produce[#1]{
  \xdef FileVersion{#1}
  \catcode`\active
  \@ifnextchar[{
    \catcode`\z@ \cdGetFileDate@Produce}
  \ifcdNoheader\else\cdHeader\fi\egroup}}
def\cdGetFileDate@Produce[#1]{
  \xdef\FileDate{#1}
  \ifcdNoheader\else\cdHeader\fi\egroup}
def CloseFile@@Produce#1{
  \ifcdAutoclose
    \expandafter\ifx\csname #1@Status\endcsname\relax
      \cdCDWarning{\string CloseFile\space is redundant.}
    \else\expandafter\ifx\csname #1@Status\endcsname\cdClosed
      \cdCDWarning{\string CloseFile\space is redundant.}
    \fi
  \else
    \cdCDWarning{\string CloseFile\space is redundant.}
  \fi
```

Closing a file is a lot of uninteresting testing...
If everything is okay, beside closing the file, we also define the no-file warning and neutralize the writing macro. We also add the stream allocated to that file to \cd@StreamList, so that it may be retrieved if all other streams are unavailable.

```
1537 \else
1538 \cd@Tracing{I close file #1}
1539 \expandafter\let\csname #1@Status\endcsname\cd@Closed
1540 \def\cd@TempFile{#1}
1541 \ifx\cd@TempFile\cd@CurrentFile
1542 \def\cd@NoFileWarning{\cd@CDWarning{No file in production. This code will be lost.}}
1543 \def\cd@ProduceFile##1{\%
1544 \fi
1545 \edef\cd@StreamList{%
1546 \cd@StreamList\expandafter\the\csname #1@Stream\endcsname,}
1547 \fi\fi\fi}
```

The last thing to do is to build that list of streams made available by the closing of a file.

```
1548 \def\cd@StreamList{}
1549 \def\cd@BuildList#1cd@end,{\def\cd@StreamList{#1}}
```

When we look for a stream, we simply check the content of \cd@BuildList, and if we find the terminator, this means that no stream has been made available. Otherwise, we define \cd@ProduceStream, which will be allocated to the file we're trying to open, as the first stream we find in the list, and we rebuild the latter with the remaining numbers.

```
1550 \newif\ifcd@stream
1551 \def\cd@FindStream#1,{\%
1552 \def\cd@TempArg{#1}
1553 \ifx\cd@TempArg\cd@end
1554 \cd@streamfalse
1555 \let\cd@@next\relax
1556 \else
1557 \cd@streamtrue
1558 \chardef\cd@ProduceStream=#1 %
1559 \let\cd@@next\cd@BuildList
1560 \fi\cd@@next}
```

Finally, here's the default header.

```
1561 \catcode`%=12\relax
1562 \edef\cd@Header{
1563 \noexpand\cd@ProduceFile{% This is \noexpand\FileName, produced by the CodeDoc class
1564 \~% with the `produce' option on.
1565 \~% To create the documentation, compile \cd@CurrentSource
1566 \~% without the `produce' option.
1567 \~% SOURCE: \noexpand\cd@CurrentSource
1568 \~% DATE: \noexpand\FileDate
1569 \~% VERSION: \noexpand\FileVersion
1570 }}
1571 \catcode`%=14\relax
1572 \makeatother
```

... and we say goodbye. The end.

59
Index

This index was generated by the \DescribeMacro-like commands. It only reports where macros are described (page numbers in normal font) and defined (page numbers in italics). In the current version, CodeDoc does not index macros when used in the code.

Entries are sorted ignoring the \@cd and \@cd@@ prefixes.

\@cd@LineCount, 44
\@documentclasshook, 45
\@cdActivateCodeEscape, 50
\@cdActivateShortCode, 27
\ActivateShortCode@Produce, 49
\@cdActivateVerbBreak, 28
\@cdActivateVerbCommand, 29
\@cdActivateComment, 43
\AddBlankLine, 6, 17, 45
\AddBlankLine@Produce, 56
\@cdActivateVerbCommand, 22
\AddEnvironmentWarning, 9, 18, 45
\AddEnvironment@Produce, 52
\@cdDangerousExample@Produce, 52
\@DefErr, 26
\DefineEnvironment, 7, 18, 22
\DefineEnvironment@Produce, 51
\DefineEnvironment@Produce, 48
\DefineEnvironment@Produce, 51
\DefineIndexFont, 7, 18, 22
\DefineMacro, 7, 18, 22
\DefineMacro@Produce, 51
\DefineShortVerb@Produce, 48
\DefineShortVerb@Produce, 51
\DefineLongVerb@Produce, 51
\DefineMacro@Produce, 51
\DefineEnvironment@Produce, 51
\DefineEnvironment@Produce, 52
\DocStripMarginpar, 7, 18, 22
\@cdDocumentString, 52
\@cdEatBOL, 43
\@cdEmpty, 36
\@cdEnd, 21
\@end@Produce, 51, 53
\@EndOfLine, 46
\@cdError, 45
\@cdEscape, 46
\@cd@TeXEOL, 42
\@cd@TeXUff, 14, 18, 34
\@cd@TeXUn, 14, 18, 34
\@cd@TeXOutEOL, 42
\@cd@TeXOutVisibleEOL, 42
\@cd@TeXStartGobble, 42
\@cd@Evaluate, 50
\@cd@Evaluate, 47
\@cd@Example, 37
\@cd@ExampleEnd, 38
\begin@Produce, 51
\begin@@Produce, 52
\BoxTolerance, 6, 17, 31
\bslash, 8, 17, 25
\@cd@BuildList, 59
\@cd@CheckEnvironment, 54
\@cd@CheckEnvironment, 54
\@cd@CheckEnvironment, 54
\@cd@CloseFile, 5, 18, 43
\@cd@CloseFile@Produce, 58
\@cd@CloseFile@Produce, 53
\@cd@Code, 33
\@cd@code, 32
\@cd@code (environment), 5, 17
\@cd@CodeEscape, 16, 18, 30
\@cd@CodeEscape@Produce, 50
\@cd@CodeEscape@Produce, 50
\@cd@CodeFont, 5, 18, 32
\@cd@CodeInput, 10, 18, 35, 37
\@cd@CodeOutput, 10, 18, 35, 37
\@cd@CodeString, 52
\@cd@CodeWrite, 54
\@cd@Comment, 46
\@cd@ComparePrefix, 24

60
What am I doing here?