`colordoc`
TEXnicolor documentation

Federico Garcia
`federook@gmail.com`

2010/04/16

Abstract

The `colordoc` package builds on the `doc` package to provide color highlighting of TEX syntactical conventions, especially curly braces, `\if`-dots-`\fi` pairs, and definitions.

Contents

1 Use and samples

To use the package, simply load it with `\usepackage`. (A few user commands for customization are mentioned below.) The package is designed for use with `doc`, mainly in `dtx` files. You can use it in your own, or simply add the package to any `dtx` in order to get the coloring of the code.

This `dtx` doesn't itself load the package (since odds are it's not installed), but it contains the package's code in order to make the samples. I've been careful and tested the package separately, but there's always the risk that some subtle thing is different—please let me know if you find something. In any case, the official supported version is `colordoc.sty`, not `colordoc.dtx`.

The package has three options: `braces` (default) colors the braces and pairs of matching macros (`\if`...`\fi`); `contents` colors what's inside these delimiters; `numbers` (also `numbered`) puts subscripts after the delimiters; and `nothing` just does nothing. The first three options also highlight newly defined things.

1.1 Option `nothing`

This doesn't change anything and the result is exactly as if not loading the package:

```
⟨*sample⟩
\newcount\@sample
\newif\if@sample
\newbox\samplebox\newcounter{samplecount}
\newcommand\test{\ifhmode\gdef\@test{%
        \loop\ifnum\@sample\z@
            \advance\@sample\@ne
            \repeat}
    \else\ifcat\noexpand#1a%
            \begin{minipage}
                Test sample. \PackageWarning{sample}{Don't do this!}%
            \end{minipage}
         \else\ifodd\@sample
                \@namedef{@@sample}{\textit{It}\space\textsf{(\textbf{Note})}}%
```

Sample text.

```
        \fi  \fi
        \xdef\@test{\relax}%
    \fi
    }
⟨/sample⟩
```

1.2 Option `braces` (default)

Here the delimiters are colored, as well as new commands:

```
⟨*sample⟩
\newcount\@sample
\newif\if@sample
\newbox\samplebox\newcounter{samplecount}
\newcommand\test{\ifhmode\gdef\@test{%
        \loop\ifnum\@sample\z@
```

```
            \advance\@sample\@ne
            \repeat}
    \else\ifcat\noexpand#1a%
            \begin{minipage}
                Test sample. \PackageWarning{sample}{Don't do this!}%
            \end{minipage}
         \else\ifodd\@sample
                \@namedef{@@sample}{\textit{It}\space\textsf{(\textbf{Note})}}%
```

Sample text.

```
        \fi  \fi
        \xdef\@test{\relax}%
    \fi
    }
⟨/sample⟩
```

1.3 Option `contents`

Here what is colored is the stuff within delimiters, not the delimiters themselves:

```
⟨*sample⟩
\newcount\@sample
\newif\if@sample
\newbox\samplebox\newcounter{samplecount}
\newcommand\test{\ifhmode\gdef\@test{%
        \loop\ifnum\@sample\z@
            \advance\@sample\@ne
            \repeat}
    \else\ifcat\noexpand#1a%
            \begin{minipage}
                Test sample. \PackageWarning{sample}{Don't do this!}%
            \end{minipage}
         \else\ifodd\@sample
                \@namedef{@@sample}{\textit{It}\space\textsf{(\textbf{Note})}}%
```

Sample text.

```
        \fi  \fi
        \xdef\@test{\relax}%
    \fi
    }
⟨/sample⟩
```

1.4 Option `numbers` (black and white)

This replaces coloring with numbers in subscript and underlining for the new commands:

```
⟨*sample⟩
\newcount\@sample
```

```
\newif\if@sample
\newbox\samplebox\newcounter{_1samplecount_1}
\newcommand\test{_1\ifhmode_2\gdef\@test{_3%
    \loop_4\ifnum\@sample\z@
      \advance\@sample\@ne
      \_4repeat_3}
  \_2else_2\ifcat_3\noexpand#1a%
      \begin{_4minipage_4}
        Test sample. \PackageWarning{_4sample_4}{_4Don't do this!_4}%
      \end{_4minipage_4}
     \_3else_3\ifodd_4\@sample
        \@namedef{_5@@sample_5}{_5\textit{_6It_6}\space\textsf{_6(\textbf{_7Note_7})_6}_5}%
```

Sample text.

```
    \_4fi  \_3fi
    \xdef\@test{_3\relax_3}%
  \_2fi
  _1}
⟨/sample⟩
```

1.5 Further variations

`\textnew`
`color-1`
`color-2`
⋮
`\ColorLevels`

The user can easily customize the given options. `\textnew` holds the formatting of the macro names after `\def`, `\new`..., etc., and can be easily modified. The actual colors are held in `color-1`, `color-2`, etc., and can be changed through the `color` package's `\definecolor`. There are four colors (in addition to the default black) in the `braces` option, and five in `contents`. Should the user want to change this, `\ColorLevels`⟨*n*⟩ sets things up for *n* colors; their actual hue still has to be defined through `\definecolor`.

2 Warnings, decisions, wishes

1. *It would be nice if highlighting is suppressed for braces that open and close within the same code line.* I tried this several times. In the end I don't think it's worthwhile the amount of hacking it requires. Using some TeX trickery, you can make the opening brace find out whether there is a closing brace before the end of the line. That much is possible and fairly robust; but the problem is that there might be several other braces apart from the first one; the full task involves counting opening and closing braces before the end of the line—a case of multiple macro reversion with unknown numbers of arguments, in the tricky conditions of the `macrocode` environment.

2. *Smart* `\repeat`*.* I personally use and have seen '`\loop\if`... `\repeat`' more frequently than '`\loop`... `\repeat\if`.' So `colordoc` works for the former construction: `\loop` suppresses the behavior of the following `\if`, and itself behaves as an opening delimiter—closed by `\repeat`. The program *will* get confused if the latter construction is used. There is really no way around this.

3. *LaTeX conditionals* (`\@ifundefined`, etc.), that do *not* behave as delimiters, will be treated correctly by `colordoc`. But this is because of the lucky coincidence that those conditionals have the `@` at the beginning. `colordoc` treats anything starting `\if` as a delimiter. If you are defining fancy conditionals, it might be a good idea to use `\@if`...

4. *In an emergency,* when something in the code does confuse `colordoc`, you can manually set `\BraceLevel` to a number, and restore order. Just like that: `\BraceLevel1` (or even `\BraceLevel=1`).

5. *The main text may appear colored* sometimes. This is due, who knows how, to the several-pass cross-reference mechanisms. One more run will usually make it good.

6. *I wish the color of the backslash could be easily controlled.* But the backslash is typeset by `doc` before `\macro@name` starts assembling the letters of the command. `\macro@name` is what I hacked, and what was easiest and best to hack, but then I had no control over the backslash.

7. *Does the program know when a macro is being defined?* While working on the package I realized that by coloring any macro that is being defined, I didn't need to religiously append `\begin{macro}` to generate the marginpars. With the coloring of new commands, the visual reference was there already. At a wild point I thought that maybe the package would add the `\begin{macro}`'s automatically! Of course this is actually impossible, and in any case the user would have no control over when and where to put it. But it is true that the need for the marginpars with macro names is eased. This `dtx` doesn't have many (none in the code) and thus I'm using a little less paper.

8. *METAFONT support.* I thought it would be simply changing `{` for `(`. This much is easy, but apart from that catching `if-endif` is impossible for a language with no "escape character" that would allow `doc` scan for a macro name. The solution is to create a wholly new package adapting `doc` to METAFONT.

3 The package

3.1 Introduction

This program modifies Frank Mittelbach's `doc` package to use colors in the code listing of a program. With a natural emphasis on TeX, color is used mainly to highlight matching curly braces ('`{`' and '`}`', henceforth simply 'braces'), `\if`...-`\fi` constructions, and other TeX programming stuff.

Originally I imagined coloring pairs of matching braces with the same color (different for each level of nesting). In 2002–3 I prepared the 'interactive version' in the CD accompanying my uncle's book *El Universo LaTeX* [1], and developed some macros to color matching braces in the examples. That `pdf` was prepared in a larger font, suitable for on-screen reading, and this made the colored braces very easy to spot. It was fun work, but not vary robust (it was intended for my own use), and it certainly required some retouching of the code that was being listed—in part, because other stuff was being done to it, like highlighting the particular commands that each example illustrated, and making them links to their description in the text. So it wasn't really a literate programming tool (where the point is to use *the same* source as both the working program and its typeset-able listing). But that was where I saw that coloring the code makes a difference in code readability.

So when I did the first tries for the present package I started by coloring the braces. But such small characters (made even `\small`'er by `macrocode`) did not really stand out, and the colored brace matching was nice, but less than obvious to the eye. Then I think my wife Jen proposed coloring not the braces but the stuff inside them! I initially liked this a lot. (An architect and graphic designer, Jen would eventually be in charge of defining the default color schemes of the package). The result was more what I had in mind: the matching is really obvious, and the colored code is not as obnoxious as I first feared.

However, that is true when we're only considering braces. After adding the rest of the features (`\if`'s, highlighting of newly defined macros, etc.), it became clear that, sadly, there are really a lot of things colored all over the place. So I am back to the coloring of the braces, not the contents, as a default. Coloring the contents is offered as an option—another option was thought of anyway, for black-and-white needs (i.e., when printing): instead of colors we can give braces a numeric subscript.

Just to give it a chance, I am using the "contents" option in the code of the present package below.

3.2 General strategy

We will have to hack the `macrocode` environment, where the code listing actually happens, and which is analogous to the usual `verbatim`, except that other things happen—addition of line numbers being the most obvious. `{` and `}` will have to be made active characters—not too much of a problem because they are innocuous in `macrocode`, not having to perform any special tasks.

Catching `\if`'s, and analogously catching `\def`'s, `\edef`'s, `\newcommand`'s, etc. (since it would be nice to highlight the commands that are being defined), seems harder. It will mean making `\` active, and making it read ahead for the name of the macro it introduces: take letter by letter, find the end of a command name (is it true, as intuitively felt, that we can simply look for a character with a category code other than 'letter'?), decide what to do with it, and hope that this doesn't break anything else, and that you can prove it.

Two good news: within `macrocode` (or `verbatim`), not much does much—all the special characters' special meanings are suppressed, so, after all, there isn't much that can be 'broken'. It actually turns out to be a pretty danger-free low-level hacking setting.

And the second good news: Frank Mittelbach, in `doc` itself, already did most of the job about finding a macro's name! This is because of his bold dream, and execution thereof, of making `macrocode` compile index entries from the macros used in the code. Since the `doc` package not only implements literate programming, but also illustrates it very well (the explanations[1] are quite sufficient to understand a code that is itself very readable, and the functions are broken into logical sections), it is very easy to achieve the results of `colordoc` with only relatively small, targeted modifications.

So let's start by setting up `colordoc` as a package, and then the main modification from `doc`:

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{colordoc}[2010/04/18 Documentation in TeXnicolor (Federico Garcia)]
\let\doc@macrocode\macro@code
```

The new meaning of `macrocode` depends on `colordoc`'s options, and will be established there (section 3.4). It will always do what it does in `doc` (now `\doc@macrocode`) plus more stuff; among this additional stuff is the need to make the braces active, through `\@makebracesactive`:

```
\def\@makebracesactive{\catcode`\{\active\catcode`\}\active}
```

3.3 Highlighting of special commands

When `doc` finds a `\` in the user's code, it starts gathering the command name that follows, in order to make an index entry (except when the command is in the `\DoNotIndex` list). We are going to hack two of the commands involved: `\macro@name`, that takes each letter and adds it to the name that is being collected—we need this to find partial command names like `\if`... or `\new`...—and `\macro@finish`, that operates on a completed macro name—which we will use to catch complete things like `\fi` without finding `\finish`. We

---

[1] I mean the explanations on how the macros work and are implemented. The explanations to the user, on how to *use* the macros, are paradoxically less friendly!

won't really re-use the original definitions (as we did with `\macro@code`, because we are changing, not just adding to, the behavior of the original `\macro@namepart` and `\macro@finish`.

```
\def\macro@name#1{%
    \edef\macro@namepart{\macro@namepart#1}%    Like in doc
    \ifx\macro@namepart\colordoc@new\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@newif\@ifcheckallowedfalse\fi
    \ifx\macro@namepart\colordoc@if\if@ifcheckallowed
        \gdef\macro@style{\@openingbrace}%
        \else\global\@ifcheckallowedtrue\fi\fi
    \futurelet\next\more@macroname%             Like in doc
    }
\def\macro@finish{%
    \ifx\macro@namepart\colordoc@newenvironment\let\futuremacro@style\relax\fi
    \ifx\macro@namepart\colordoc@newcounter\let\futuremacro@style\relax\fi
    \ifx\macro@namepart\colordoc@def\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@edef\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@gdef\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@xdef\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@renewcommand\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@providecommand\let\futuremacro@style\textnew\fi
    \ifx\macro@namepart\colordoc@loop\let\macro@style\@openingbrace\@ifcheckallowedfalse\fi
    \ifx\macro@namepart\colordoc@repeat\let\macro@style\@closingbrace\fi
    \ifx\macro@namepart\colordoc@fi\let\macro@style\@closingbrace\fi
    \ifx\macro@namepart\colordoc@else\def\macro@style{\@closingbrace\relax\@openingbrace}\fi
    \ifx\macro@namepart\colordoc@csname\let\macro@style\@openingbrace\fi
    \ifx\macro@namepart\colordoc@endcsname\let\macro@style\@closingbrace\fi
```

The original definition of `\macro@finish` simply typesets the macro name (held in `\macro@namepart`), and then proceeds to process it (or not, according to `\ifnot@excluded`) as an index entry. We will add the formatting, both of the present macro with `\macro@style`, and of the very next one with `\futuremacro@style`.

```
    \macro@style\macro@namepart\global\let\macro@style\relax
    \global\let\macro@style\futuremacro@style\global\let\futuremacro@style\relax
    \ifnot@excluded%                            From here on like in doc
        \edef\@tempa{\noexpand\SpecialIndex{\bslash\macro@namepart}}%
        \@tempa \fi}
```

With these redefinitions we have made the program able to find the macro names that trigger special formatting. Some of them trigger it for themselves, some trigger it for the coming macro. This all happens through `\macro@style` and `\futuremacro@style`. In particular, `\if` and other behave like an opening brace, making `\macro@style` equal to `\@openingbrace` (discussed later on). `\fi` and others are like a closing brace, while `\else` is *both* a closing and an opening brace.

Some macros, however, should suppress this special behavior of `\if`, because after them `\if` doesn't work as a delimiter: after a `\newif`, and after `\loop` (rather, `\loop` itself is a delimiter, paired with `\repeat`). This is the point of the test `\if@ifcheckallowed`. (See some warnings about this in section 2).

```
\newif\if@ifcheckallowed\@ifcheckallowedtrue
```

The `\new`... commands (`\newcount`, `\newif`, etc., including `\newcommand`), as well as the different `\def`'s, will highlight whatever the next macro name is. The default highlighting depends on the package options, which define the default `\textnew`. The command can freely be redefined by the user—when used by the program, if you need to consider argument handling, it is always followed by `\macro@namepart` (a single token).

The special macro names are found by comparing the macros found in the code, either complete (at `\finishmacro@name`) or not (at `\macro@name`), with pre-defined contents. The comparison is done by `\ifx`, that compares only two tokens, so that we actually have to define commands to contain the comparands we're looking for. The actual comparisons happened above, but here are the comparands:

```
\def\colordoc@csname{csname}          \def\colordoc@endcsname{endcsname}
\def\colordoc@def{def}                \def\colordoc@edef{edef}
\def\colordoc@gdef{gdef}              \def\colordoc@xdef{xdef}
\def\colordoc@if{if}    \def\colordoc@else{else}    \def\colordoc@fi{fi}
\def\colordoc@loop{loop}              \def\colordoc@repeat{repeat}
\def\colordoc@new{new}                \def\colordoc@newif{newif}
\def\colordoc@newcounter{newcounter}\def\colordoc@newenvironment{newenvironment}
\def\colordoc@providecommand{providecommand}
\def\colordoc@renewcommand{renewcommand}
```

`\newenvironment` and `\newcommand` are needed, even though the search for a starting `\new` will find them, because in their cases we actually need to suppress the highlighting: their argument is not a macro name, but a simple argument. If we didn't suppress highlighting, then whatever is the next actual macro name will be (wrongly) highlighted. (The argument to `newenvironment` will then not be highlighted in any special way, but it doesn't seem worthwhile to implement a whole additional infrastructure for this one case.) `\newif`, on the other hand, also needs to suppress special treatment for the following `\if`...

I decided not to include `\let`, because `\let` is used for auxiliary definitions, seldom legitimate ones that one would want to have quick reference to. You might also miss `\bgroup`, `\begingroup`, `\egroup` and `\endgroup`, whose equivalence to braces would make them obvious candidates for the changing-color macro list. I even included them at first, until I realized that one of their main use is to trick TeX because one needs *not* to match delimiters. Thus they should also not change colors. The other use of these commands, namely the creation of groups for local definitions, will sadly go uncolored; in case of need the user should consider simply using `{` and `}`.

3.4 The braces

The handling of braces is less straightforward. We'll have `{` and `}` defined generically in terms of `\@openingbrace` and `\@closingbrace`, commands that act on either `\char'173` or `\char'175` (`\{` and `\}` would produce the braces in the default font, not the one used for code). Their definitions depend on the options of the package.

3.4.1 `numbers`

Here we will put a subscript number after the brace through `\textsubscript` (made available by the `fixltx2e` package, loaded later). This option is duplicated to provide both `numbered` and `numbers`.

```
\newif\ifcolorcode@
\DeclareOption{numbers}{%
    \def\macro@code{\doc@macrocode\@makebracesactive}
    \colorcode@false
    \let\colordoc@scheme\relax
    \def\textnew#1{\textit{\underline#1\/}}
    \def\@openingbrace#1{\global\advance\BraceLevel\@ne#1\textsubscript{\the\BraceLevel}}
    \def\@closingbrace{\textsubscript{\the\BraceLevel}\global\advance\BraceLevel\m@ne}
    }
\DeclareOption{numbered}{%
    \def\macro@code{\doc@macrocode\@makebracesactive}
    \colorcode@false
    \let\colordoc@scheme\relax
    \def\textnew#1{\textit{\underline#1\/}}
    \def\@openingbrace#1{\advance\BraceLevel\@ne#1\textsubscript{\the\BraceLevel}}
    \def\@closingbrace{\textsubscript{\the\BraceLevel}\advance\BraceLevel\m@ne}
    }
```

3.4.2 `nothing`

```
\DeclareOption{nothing}{%
    \let\@openingbrace\relax
    \let\@closingbrace\relax
    \colorcode@false
    \let\colordoc@scheme\relax
    \let\textnew\relax
    }
```

3.4.3 `braces`

This option will color only the brackets. With the `color` package loaded (later), it can define its color scheme (see 3.6).

On the other hand, we need to take care of the potentially infinite nesting. The levels will cycle through a finite number n of colors (established by the color scheme). `\@modinc` and `\@moddecr` (defined below 3.5) step the level counter up and down, modulo n. `\@bracecolor` always means ‘the color specified by the color scheme for the current nesting level’.

```
\DeclareOption{braces}{%
    \def\macro@code{\doc@macrocode\@makebracesactive}
    \colorcode@true
    \let\@colorcode\relax
    \def\colordoc@scheme{\braces@colorscheme}
    \def\@openingbrace#1{\@modinc\BraceLevel{\@bracecolor#1}}
    \def\@closingbrace#1{{\@bracecolor#1}\@moddecr\BraceLevel}
    }
```

3.4.4 `contents`

This is similar, but different enough. The color scheme has to be set to `\contents@colorscheme`. In addition, since the code (not only isolated characters or macros) is being colored, we need to keep track of the appropriate color at the start of each `macrocode`. So `\macro@code` is redefined differently from the other options, to include `\@bracecolor`. Similarly, the end of the environment should bring back the default color.

```
\DeclareOption{contents}{%
    \colorcode@true
    \def\colordoc@scheme{\contents@colorscheme}
    \def\macro@code{\doc@macrocode\@makebracesactive\@bracecolor}
    \let\docendmacro@code\endmacro@code
    \def\endmacro@code{\docendmacro@code\normalcolor}
```

Code line numbers should also be guarded against coloring:

```
    \let\doctheCodelineNo\theCodelineNo
    \def\theCodelineNo{\normalcolor\doctheCodelineNo}
```

The last difference is that the actual braces are not affected by the new color.

```
    \def\@openingbrace#1{#1\@modinc\BraceLevel
        \global\let\@codecolor\@bracecolor\@codecolor}
    \def\@closingbrace{\@moddecr\BraceLevel
        \global\let\@codecolor\@bracecolor\@codecolor}
    }
```

The options will indicate which package to load (`color`'s driver should be specified in `\documentclass`, since there is no user control here).

```
\DeclareOption*{\typeout{Unknown option (`\CurrentOption')}}
\ExecuteOptions{braces}
\ProcessOptions
\ifcolorcode@\RequirePackage{color}\else\RequirePackage{fixltx2e}\fi
```

3.4.5 Main brace engine

We need a counter to keep track of the nesting level of the brackets. It should be available to the user for him to take control in emergencies.

```
\newcount\BraceLevel    \BraceLevel\z@
```

Now we define `{` and `}`. This has to be done within a group where they are active characters:

```
\bgroup
\catcode`\[\@ne\catcode`\]\tw@
\@makebracesactive
\gdef{[\@openingbrace[\char'173]]
\gdef}[\@closingbrace[\char'175]]
```

Now that we are in this group we set up the other stuff that takes special category codes. The definition of `\xmacro@code` (the auxiliary macro that catches the end of a `macrocode` environment) looks exactly the same as in the original `doc` package, but it has to be re-done because in `colordoc` the braces are active.

Now, how to tell the present `dtx` that the ‘`%    \end{macrocode}`’ string below is part of the code, not its end?! Well, we'll use `macrocode*`.[2]

```
\catcode`\|\z@\catcode`\%12
\catcode`\␣\active\catcode`\\\active
|gdef|xmacro@code#1%␣␣␣␣\end{macrocode}[#1|end[macrocode]]
```

Let's take the hint to provide support for the starred versions right here. Again, thanks to the modular design of `doc`, this is really painless. The only point is that in a starred environment the space is not active, so we have to deactivate it. It becomes an ‘other’ kind of character, that would try to print something (`! Missing \begin{document}`) if we let it happen, even as an end-of-line, so the whole thing has to go in one line (the comment character is sensitive as well).

```
|catcode`| 12|gdef|sxmacro@code#1%    \end{macrocode*}[#1|end[macrocode*]]|egroup
```

---

[2] Just as in `doc.dtx`. In that file there is also a new comment character, whose full point I can't see and might be related to some other issue. In any case, that trickery is not necessary here.

3.5 Auxiliary functions

For cycling through the color scheme we need a pair of arithmetic functions modulo `\@colorlevels`.

```
\def\@modinc#1{%
    \global\advance#1\@ne
    \ifnum#1>\@colorlevels
        \global#1\@ne\fi}
\def\@moddecr#1{%
    \global\advance#1\m@ne
    \ifnum#1>\z@\else
        \global#1\@colorlevels\fi}
```

The actual color changes are performed by `\@bracecolor`. This is a function of `\BraceLevel`, through auxiliary commands `\@color`⟨*\BraceLevel*⟩.

```
\def\@bracecolor{\@nameuse{@color\the\BraceLevel}}
```

The auxiliary commands have to be defined in advance. Their number depends on `\@colorlevels`, but since the user can change this at any point, the program needs to be ready to apply new definitions. So, the user will have `\ColorLevels`⟨*new number of colors*⟩: this sets `\@colorlevels`, and calls the definition routine. (Apart from this, the user is responsible for defining any colors beyond the original 5.)

```
\def\ColorLevels#1{%
    \@colorlevels#1%
    \@tempcnta\@colorlevels
    \loop\ifnum\@tempcnta>\z@
        \expandafter\edef\csname @color\the\@tempcnta\endcsname{%
            \noexpand\color{color-\the\@tempcnta}}
        \advance\@tempcnta\m@ne
        \repeat
    }
```

3.6 Color schemes

The colors themselves are `\definecolor`'ed as `color-1`, `color-2`, etc., up to `\@colorlevels`. The two options that involve colors define the defaults, but the user can change them at any time.

The `braces` color scheme needs eye-catching colors, since a lot of the coloring happens on single characters. The outer level will be color 0, since in principle it should never happen. However, it can happen when for some reason the code confuses `colordoc` (for example, when the braces are used in some special way and are unbalanced in the code; see section 2 for other cases). The default scheme is blue-purple-green-yellow.

```
\newcount\@colorlevels
\def\braces@colorscheme{%
    \ColorLevels4
    \BraceLevel\z@
    \definecolor{color-0}{rgb}{0,0,0}
    \definecolor{color-1}{cmyk}{1,.5,0,.3}
    \definecolor{color-2}{cmyk}{0,1,0,.2}
    \definecolor{color-3}{rgb}{0,.6,0}
    \definecolor{color-4}{cmyk}{0,0,1,.6}
    \def\textnew##1{\textit{\color{color-def}##1\/}}
    \definecolor{color-def}{rgb}{0.8,0,0}
    }
```

For the `contents` option, where whole chunks of code will be colored (instead of only braces and a few macros), we want milder colors. In fact, assuming that most of the code listing in a `dtx` actually happens at brace level 2 (since most of it is the contents of `\def`'s, `\newcommand`'s, etc.), we have set level 2 to black. The outer level is a darkish purple, and then comes a green, a brown, and a blue.

```
\def\contents@colorscheme{%
    \ColorLevels5
    \BraceLevel\@ne
    \definecolor{color-0}{rgb}{0,0,0}
    \definecolor{color-1}{cmyk}{0,.5,0,0.6}
    \definecolor{color-2}{rgb}{0,0,0}
    \definecolor{color-3}{rgb}{.15,0.15,.75}
    \definecolor{color-4}{rgb}{0,.47,0}
    \definecolor{color-5}{cmyk}{0,0.72,1,.4}
    \def\textnew##1{\textit{\color{color-def}##1\/}}
    \definecolor{color-def}{rgb}{0.8,0,0}
    }
```

4 Initialization

The macro name scanning mechanism of `doc` happens only if it is producing an index. That in turn is true only if the user has asked to `\EnableCrossrefs`. The reason why this is not the default is that it slows down the processing of the file. `colordoc` requires the macro name scanning, but not the index, and I thought of modifying the effect of these switches to always do the scanning but be selective on the index. I am afraid this is too risky, as the switch `\ifscan@allowed` is used variously in `doc`. Since speed is not so much of a concern nowadays, I prefer to leave this untouched. We simply need to make sure that the mechanism is allowed even if the user doesn't request it.

```
\EnableCrossrefs
\let\macro@style\relax
\let\futuremacro@style\relax
\colordoc@scheme
⟨/package⟩
```

References

[1] Rodrigo De Castro, *El Universo LaTeX*, 2nd. ed.. Bogotá: Universidad Nacional de Colombia, Facultad de Ciencias, Departamento de Matemáticas, 2003.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.