Version 0.2

PDFToolbox

ari.feiglin@gmail.com

PDFToolbox offers a variety of tools for creating documents in plain TEX. These include packages for structuring documents, coloring documents, etc. PDFToolbox is a collection of packages intended to be used with plain TEX. It is intended to be self-contained and does not promise compatibility with other packages.

PDFToolbox is still experimental and may be subject to breaking changes. If you have an important document relying on it, the author advises keeping

PDFToolbox is known to not interact with the `color`, `xcolor`, `tikz` and all related packages. This may or may not be changed in the future.

This documentation is split into sections corresponding to the different collections in PDFToolbox. These are:

(1) Data manipulation: counters, dictionaries, etc.

(2) Document structure: layouts, table of contents, indices, etc.

(3) Graphics: colors, diagrams, colored boxes, etc.

PDFToolbox depends only on the apnum package.

PDFToolbox is provided as opensource free software under the MIT license.

Contents

I. pdfToolbox in brief

1 pdfData

The pdfData section of the PDFToolbox toolbox is meant for creating instances of and manipulating datatypes.

1.1 Arrays

In the `pdfData/arrays` file, PDFToolbox defines various macros for creating and manipulating arrays. There are two types of arrays, which are different in the macros used for them and the way they are stored internally.

- (Normal) arrays: these arrays are stored in the traditional way: an array $[1; 2; 3]$ is stored in a macro whose meaning is equivalent to `\X{1}\X{2}\X{3}`. Manipulation of the array is done by defining `\X`, and then executing the array macro.
- Macro arrays: these arrays are stored in a collection of macros: each element is stored in its own indexed macro. So an array $[1; 2; 3]$ will be stored in three macros, whose values are 1, 2, 3 respectively.

All arrays are zero-indexed.

1.1.1 Normal Arrays

`\createarray` {⟨*name*⟩}: creates an (normal) array whose name is *name*.

`\ensurearray` {⟨*name*⟩}: ensures that an array by the name of *name* exists.

`\localizearray` {⟨*name*⟩}: localizes (see localization) the array named by *name*.

`\appendarray` {⟨*name*⟩}{⟨*value*⟩}: appends *value* to the end of the array array named by *name*. *value* is inserted according to `\currdef`.

`\prependarray` {⟨*name*⟩}{⟨*value*⟩}: prepends *value* to the end of the array array named by *name*. *value* is inserted according to `\currdef`.

`\appendarraymany` {⟨*name*⟩}{{⟨*value1*⟩}{⟨*value2*⟩}...{⟨*valueN*⟩}}: appends *value1* through *valueN* to the end of the array array named by *name*. Each *value* is inserted according to `\currdef`.

`\arraylen` {⟨*name*⟩}: expands to the length of the array specified by *name*.

`\getarraylen` {⟨*name*⟩}⟨*macro*⟩: inserts the length of the array specified by *name* into the macro *macro*.

`\arraymap` {⟨*name*⟩}⟨*macro*⟩: if the array specified by *name* is equivalent to [`x0`; ...; `xN`] then doing `\arraymap{name}\X` will execute `\X{x1}{0}...\X{xN}{N}`.

`\indexarray` {⟨*name*⟩}{⟨*i*⟩}⟨*macro*⟩: Puts the ith element in the array specified by *name* into the macro *macro*.

`\removearray` {⟨*name*⟩}{⟨*i*⟩}⟨*macro*⟩: Removes the ith element in the array specified by *name* and places it into the macro *macro*.

`\removeitemarray` {⟨*name*⟩}{⟨*value*⟩}: Removes all instances of *value* from the array specifed by *name* (comparison is done using `\ifx` on macros containing *value* and the current index).

`\printarray` {⟨*name*⟩}: Prints the array specified by *name*.

`\copyarray` {⟨*src*⟩}{⟨*dest*⟩}: Copies the array *src* into *dest*.

`\concatenatearrays` {⟨*arr1*⟩}{⟨*arr2*⟩}{⟨*dest*⟩}: Concatenates the arrays *arr1* and *arr2* and places the result into a new array *dest*.

`\initarray` {⟨*name*⟩}{⟨*x1*⟩,...,⟨*xN*⟩}: Creates a new array by the name of *name* equivalent to [`x1`; ...; `xN`].

`\findarray` {⟨*name*⟩}{⟨*value*⟩}: Checks if the value *value* exists in the array *name* (checking is done via `\ifx`). If the value exists, the value `\True` is placed into `\@return@value`, otherwise it is equal to `\False`.

`\uniqueappendarray` {⟨*name*⟩}{⟨*value*⟩}: Appends *value* to the array *name* only if it does not already exist in *name* (`\@return@value` is set accordingly).

`\convertarray` {⟨*src*⟩}{⟨*dest*⟩}: Converts a normal array *src* to a macro array *dest*.

`\mergesort` {⟨*src*⟩}{⟨*dest*⟩}: Sorts the array *src* and places the result in *dest*.

1.1.2 Macro Arrays

`\createmarray` {⟨*name*⟩}: Creates a macro array by the name of *name*.

`\localizemarray` {⟨*name*⟩}: Localizes (see localization) the macro array specified by *name*.

`\appendmarray` {⟨*name*⟩}{⟨*value*⟩}: Appends *value* to the macro array specified by *name*.

`\printmarray` {⟨*name*⟩}: Prints the macro array specifed by *name*.

`\convertmarray` {⟨*src*⟩}{⟨*dest*⟩}: Converts the macro array *src* into a normal array *dest*.

`\copymarray` {⟨*src*⟩}{⟨*dest*⟩}: Copies the macro array *src* into *dest*.

`\initmarray` {⟨*name*⟩}{⟨*x1*⟩`,...,`⟨*xN*⟩}: Creates a macro array *name* whose value is equivalent to [`x1`, ..., `xN`].

`\findmarray` {⟨*name*⟩}{⟨*value*⟩}⟨*macro*⟩: Searches for *value* in the macro array *name*. If found, sets `\@return@value` to `\True` and *macro* to the index where *value* was found. Otherwise `\@return@value` is set to `\False`.

1.2 Stacks

In the `pdfData/stacks.tex` file, PDFToolbox offers macros for creating and manipulating stack data structures. There are two types of stacks, which differ in how they store their data. They are generally used for different purposes:

- Normal stacks: these are normal stacks which store just the values given.
- Macro stacks: these stacks are meant to store only macros: they store both the definition and name of the macro.

1.2.1 Normal Stacks

`\createstack` {⟨*name*⟩}: Creates a normal stack by the name of *name*.

`\stackpush` {⟨*name*⟩}{⟨*value*⟩}: Pushes the value *value* onto the stack specified by *name*.

`\stackdecrement` {⟨*name*⟩}: Pops from the top of the stack specified by *name* (deleting the value).

`\stackpop` {⟨*name*⟩}⟨*macro*⟩: Pops from the top of the stack specified by *name* into *macro*.

`\stacktop` {⟨*name*⟩}⟨*macro*⟩: Places the top of the stack specified by *name* into the macro *macro* without popping.

1.2.2 Macro Stacks

Macro stacks store macros, as opposed to values. When pushing a macro `\X` onto the stack, not only is the meaning of `\X` stored, but so is its name.

`\createmacrostack` {⟨*name*⟩}: Creates a macro stack by the name of *name*.

`\macrostackpush` {⟨*name*⟩}⟨*macro*⟩: Pushes the macro *macro* onto the macro stack specified by *name*.

`\macrostackdecrement` {⟨*name*⟩}: Pops from the top of the macro stack specified by *name* (deleting the value).

`\macrostackset` {⟨*name*⟩}: If the top of the macro stack specified by *name* has name `\X` and value `value`, sets `\X` to `value`.

`\macrostackpop` {⟨*name*⟩}: Pops from the top of the macro stack specified by *name* (same as `\macrostackset`, but also pops the value off of the stack).

`\macrostackpeek` {⟨*name*⟩}⟨*macro1*⟩⟨*macro2*⟩: If the top of the macro stack specified by *name* is (`\X`, `value`), then `\X` is placed into *macro1*, and `value` into *macro2*.

1.3 Localization

Using macro stacks, PDFToolbox allows for *localization*. This gives the user the ability to create block scopes (as opposed to just plain-ol' TEX groups). The usage is simple and as follows:

(1) The user enters a scope using `\beginscope`.

(2) The user *localizes* a macro `\X` by doing `\localize\X`.

(3) The user exits the scope using `\endscope`. Once the scope is exited, the previous definition of localized macros is restored.

So for example,

```
\def\X{0}
\beginscope
    \localize\X
    \def\X{1}
    \X
    \beginscope
        \def\X{2}
        \X
    \endscope
    \X
\endscope
\X
```

Will output 1 2 2 0. As opposed to

```
\def\X{0}
\bgroup
    \def\X{1}
    \X
    \bgroup
        \def\X{2}
        \X
    \egroup
    \X
\egroup
\X
```

Which will output 1 2 1 0.

1.4 Counters

In the `pdfData/counters.tex`, PDFToolbox implements counters. Counters are simple wrappers over plain-TEX counters. They hold integer values, are mutable, and can be made dependent on one another so that when one is altered another is set to zero.

`\createcounter {`⟨*name*⟩`}[`⟨*c1*⟩`,...,`⟨*cN*⟩`]`: Creates a counter by the name *name* dependent on counters *c1*,...,*cN*.

`\adddependentcounter {`⟨*secondary*⟩`}{`⟨*primary*⟩`}`: Makes the *secondary* counter dependent on the *primary* one; whenever *primary* is (non-independently; see e.g. `\seticounter`) altered, *secondary* is set to zero.

`\zerodependents {`⟨*primary*⟩`}`: Sets to zero all counters dependent on *primary*.

`\setcounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Sets *counter* to *amount* (zeroing all counters dependent on *counter*).

`\advancecounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Advances *counter* by *amount* (zeroing all counters dependent on *counter*).

`\seticounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Sets *counter* to *amount* (*without* zeroing all counters dependent on *counter*).

`\advanceicounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Advances *counter* by *amount* (*without* zeroing all counters dependent on *counter*).

`\counter {`⟨*name*⟩`}`: Returns the TEX counter corresponding to the PDFToolbox counter *name*. Useful for example when printing the value of a counter: simply do `\the\counter{name}`.

1.5 Dictionaries

In the `pdfData/dictionaries.tex` file, PDFToolbox implements dictionaries (also colloquially known as "hashmaps" or "maps"). These are simple maps between keys and values.

`\createdict` {⟨*name*⟩}: Creates a dictionary by the name *name*.

`\adddict` {⟨*name*⟩}{⟨*key*⟩}{⟨*value*⟩}: Adds the (*key* : *value*) key-value pair to the dictionary specified by *name*.

`\indexdict` {⟨*name*⟩}{⟨*key*⟩}: Expands to the value of *key* in the dictionary *name*.

`\keyindict` {⟨*name*⟩}{⟨*key*⟩}: Sets `\@return@value` according to if *key* is found in the dictionary *name*.

1.6 Mappings

In `pdfData/key-value.tex`, PDFToolbox implements the ability to pass key-value parameters to macros.

`\mapkeys` {⟨*options*⟩}{⟨*input*⟩}: Maps the key-value pairs given in *input* according to *options*. *options* is itself a set of key-value pairs, where the value of each key is an array which may contain:

- `name` (required): the name of the macro to give the value of the key;
- `required`: added if the key is required;
- `definition`: what definition macro to use for defining the value (e.g. `\def`, `\edef`);
- `mapping`: how to map the input to the value: the input is defined relative to `definition` into a macro wrapped with `mapping`;
- `default`: the default value of the key.

Or the value may be empty (no array), which means it is *valueless* and acts as a boolean flag.

So for example, you may have a macro defined like so:

```
\def\puthi#1{Hello (#1)}

\def\getinput#1{%
    \mapkeys{
        first={
            name=fst,
            required,
            definition=\edef,
            mapping=\puthi%
        },
        second={
            name=snd,
            default=A. Feiglin%
        }%
    }{#1}%
}

\getinput{first=pdftoolbox}
(\fst) (\snd)
```

This will output `(Hello (pdftoolbox)) (A. Feiglin)`.

`\keyexists` {⟨*key*⟩}⟨*macro*⟩`\lastkeys`: This is an internal command, added to this documentation only due to its usefulness. Given a key name *key*, this macro checks if it exists in the map corresponding to the last call to `\mapkeys` (the macro itself is more versatile, but we restrict it to this case). If the key does not exist, then *macro* is set to `\_nul`. This is useful with valueless keys.

`\mapkeys` is a bit finnicky when it comes to spaces and commas, but the rule is simple: place a comment at the end of each list. That means that within each key's array, you must place a comment at the end (otherwise an extraneous space is added to the value), and after the last key's array you must place a comment.

2 pdfDstruct

The pdfDstruct section of the PDFToolbox toolbox is for managing the structure of your documents.

2.1 Layout

In `pdfDstruct/layout.tex`, PDFToolbox provides a macro `\setlayout` for setting up the layout of the document. The use is

`\setlayout {[page width=`⟨*wd*⟩`,] [page height=`⟨*ht*⟩`,] [horizontal margin=`⟨*mwd*⟩`,] [vertical margin=`⟨*vwd*⟩`]}`

2.2 Hyperlinks

In `pdfDstruct/hyperlinks.tex`, PDFToolbox provides macros for creating and managing hyperlinks.

`\anchor [`⟨*type*⟩`]{`⟨*name*⟩`}`: Creates an anchor (a reference, if you will) to the current point in the document.

`\gotoanchor [`⟨*type*⟩`]{`⟨*name*⟩`}{`⟨*material*⟩`}`: Creates a clickable field containing *material* which, when clicked, will go to the anchor labeled with the type *type* and name *name*.

`\url {`⟨*url*⟩`}{`⟨*material*⟩`}`: Creates a clickable field containing *material* which, when clicked, will redirect to the url *url*.

`\createbordertype {`⟨*type*⟩`}{`⟨*color*⟩`}{`⟨*wd*⟩`}`: Sets the border type of anchor type *type* to be of color *color* and width *wd*. Urls have border type `url`. If a type doesn't have a specified border type, the `default` one is used.

2.3 Fonts

In `pdfDstruct/fonts.tex`, PDFToolbox provides macros for accessing and controlling fonts.

`\addfont {`⟨*name*⟩`}{`⟨*sizes*⟩`}`: This will add a font by the name *name* so that it is accessible by PDFToolbox. *sizes* is a key-value dictionary which specifies the font codes for different sizes of the font. For example, in `pdfDstruct/fonts.tex` is the usage:

```
\addfont{rm}{%
    default=cmr10,
    5pt=cmr5,
    6pt=cmr6,
    7pt=cmr7,
    8pt=cmr8,
    9pt=cmr9,
    10pt=cmr10,
    12pt=cmr12,
    17pt=cmr17
}
```

So now PDFToolbox has access to the computer modern roman font (`cmr`) at the sizes specified. The purpose of the default size is for when a size is not available. For example, requesting the `rm` font at size 13 will give you `cmr10 at 13pt`. The default size is required.

PDFToolbox provides the following fonts:

rm: `cmr`	it: `cmti`	bf: `cmbx`	sc: `cmcsc`	mi: `cmmi`	sy: `cmsy`	ex: `cmex`	sl: `cmsl`
ss: `cmss`	tt: `cmtt`	msam: `msam`	msbm: `msbm`	eufm: `eufm`	rsfs: `rsfs`		

`\applyfontcode `⟨*font code*⟩: Applies the font specified by *font code*. For example, `\applyfontcode cmr10` will set the font to `cmr10`.

`\setfontfamily {`⟨*font*⟩`}{`⟨*family*⟩`}`: Sets math font family *family* to the font *font* (which is specified by `\addfont`). For example, `\setfontfamily{rm}{0}` sets the alpha-numeric font family to `rm`.

`\setfont {`⟨*font*⟩`}`: Sets the current font to *font*. The current font is stored in the macro `\currfont`.

`\setscale {`⟨*scale*⟩`}`: Sets the current font scale to *scale*. The current font scale is stored in the macro `\currscale`.

`\setfontandscale {`⟨*font*⟩`}{`⟨*scale*⟩`}`: Sets the current font to *font* and scale to *scale*.

PDFToolbox also provides the following font switches (which are simple wrappers around `\setfont` which also set `\fam`):

`\bf, \it, \bb, \sf, \sl, \frak, \scr`

`\mathfonttable {`⟨*family*⟩`}[`⟨*offset*⟩`]{`⟨*table*⟩`}`: The `\mathfonttable` macro's purpose is to define multiple mathematical characters for the same family. *table* consists of a sequence of macros followed by numbers

(e.g. `\square0`) which correspond to the name of the macro and the math type (in this case 0: ordinary/`\mathord`). `\mathfonttable` will iterate over *table* and `\mathchardef` the macro to be equal to the character at the current position in family *family* of the type specified. If *offset* is specified, it will start iterating over the family starting from the offset.

More explicitly, if *family* is `X` and the `i`th index in the table is `\X N`, then the macro does essentially

```
\mathchardef\X = XNi
```

To skip over an index, simply write `\__`.

Using `\mathfonttable`, PDFToolbox defines the following:

`\boxdot`: $\boxdot$	`\boxplus`: $\boxplus$	`\boxtimes`: $\boxtimes$	`\square`: $\square$
`\blacksquare`: $\blacksquare$	`\diamond`: $\diamond$	`\blackdiamond`: $\blacklozenge$	`\rotateclockwise`: $\circlearrowright$
`\rotatecounterclockwise`: $\circlearrowleft$	`\rightleftharpoons`: $\rightleftharpoons$	`\leftrightharpoons`: $\leftrightharpoons$	`\boxminus`: $\boxminus$
`\Vdash`: $\Vdash$	`\Vvdash`: $\Vvdash$	`\vDash`: $\vDash$	`\twoheadrightarrow`: $\twoheadrightarrow$
`\twoheadleftarrow`: $\twoheadleftarrow$	`\leftleftarrows`: $\leftleftarrows$	`\rightrightarrows`: $\rightrightarrows$	`\upuparrows`: $\upuparrows$
`\downdownarrows`: $\downdownarrows$	`\upright harpoon`: $\upharpoonright$	`\downrightharpoon`: $\downharpoonright$	`\upleftharppon`: $\upharpoonleft$
`\downleftharpoon`: $\downharpoonleft$	`\rightarrowtail`: $\rightarrowtail$	`\leftarrowtail`: $\leftarrowtail$	`\leftrightarrows`: $\leftrightarrows$
`\rightleftarrows`: $\rightleftarrows$	`\Lsh`: $\Lsh$	`\Rsh`: $\Rsh$	`\rightsquigarrow`: $\rightsquigarrow$
`\leftrightsquigarrow`: $\leftrightsquigarrow$	`\looparrowleft`: $\looparrowleft$	`\looparrowright`: $\looparrowright$	`\circeq`: $\circeq$
`\succsim`: $\succsim$	`\gtrsim`: $\gtrsim$	`\gtrapprox`: $\gtrapprox$	`\multimap`: $\multimap$
`\therefore`: $\therefore$	`\because`: $\because$	`\Doteq`: $\Doteq$	`\triangleq`: $\triangleq$
`\precsim`: $\precsim$	`\lesssim`: $\lesssim$	`\lessapprox`: $\lessapprox$	

2.4 Hooks

PDFToolbox provides a tool, inspired by LaTeX, called *hooks* (source in `pdfDstruct/hooks.tex`). Hooks are simply snippets of code that can be inserted into macros and then altered later. An example is given at the end of this section.

`\createhook {`⟨*name*⟩`}`: Creates a hook by the name of *name*.

`\appendtohook {`⟨*name*⟩`}{`⟨*code*⟩`}`: Appends *code* to the hook specified by *name*.

`\prependtohook {`⟨*name*⟩`}{`⟨*code*⟩`}`: Prepends *code* to the hook specified by *name*.

`\callhook {`⟨*name*⟩`}`: Calls the hook specified by *name*.

PDFToolbox provides a builtin hook called `end` which is executed by `\bye`. Throughout the document, you can add macros to an array called `document data`, then all these definitions are written to the file `\jobname.data` by the `end` hook.

Specifically, you can use the `\docdata` macro to add a macro to the document's data, e.g. if you have a macro `\name` which has the author's name (say, S. Lurp), you can do `\docdata\name`, and this will write the line `\gdef\name{S. Lurp}` to the data file. Then at the beginning of the document next compilation, you can load all definitions in the data file.

2.5 Indices

In `pdfDstruct/index.tex`, PDFToolbox provides macros for creating an index. The index is organized into *categories* and *items* within each category, and an associated *value*. A category may be something like "manifolds" and an item within this category may be "topological" which has a value corresponding to the page number where topological manifolds are defined.

`\indexize {`⟨*options*⟩`}`: Adds an item to the index, specified by options, which has fields:

(1) `category` (required): the category of the item;

(2) `item`: the item of the item;

(3) `value` (required): the value of the item;

(4) `expand value` (valueless): added if `value` should be expanded (e.g. if `value` is a macro corresponding to the page number, it needs to be expanded);

(5) `add hyperlink` (valueless): whether or not the item's values should be hyperlinked.

`\seealso {`⟨*options*⟩`}`: Adds a "see also" item to the index: one which redirects to another index item. *options* is a map which has fields:

(1) `category` (required): the category of the item;

(2) `item`: the item of the item;

(3) `dest` (required): the destination of the "see also" (e.g. if the item is "wedge product", you may want to also see "exterior product", and so the destination may be "exterior product");

(4) `hyperlink`: an anchor to link to;

(5) `index link` (valueless): a flag of whether or not the anchor is within the index.

To link to an item within the index, suppose of category `C` and item `I`, set `hyperlink` to `C:I` (or just `C:` if `I` is empty), and set `index link`.

`\index`: Prints the index.

`\addtoindex {`⟨*category*⟩`}[`⟨*item*⟩`]`: Adds an item to the index of category *category* and item *item*. Its value is `\@defaultindexval` (by default `\the\pageno`), and `expand value` and `add hyperlink` are set.

2.6 Lists

In `pdfDstruct/lists.tex`, PDFToolbox provides macros for creating lists of text.

There are two types of lists: unenumerated and enumerated. Unenumerated lists start with `\blist` and end with `\elist`. Each item begins with `\item`. The symbol used for each bullet point is determined by the nested depth of the list. For a depth of N, the symbol used is stored in the macro `\listsyleN`.

Similarly enumerated lists start with `\benum` and end with `\elist`. Each item begins with `\item`, and the style for the enumeration is determined by the depth of the list. For a depth of N, the nth element is styled with `\enumstyleN{n}`. It is put in a box of width `\enumstyleN@wd`.

To add text in between items (not as part of the list), you can use `\mtext`.

2.7 Table of Contents

In `pdfDstruct/tableofcontents.tex`, PDFToolbox provides macros for creating and displaying tables of content.

`\addtoccontent {`⟨*marker*⟩`}{`⟨*title*⟩`}{`⟨*value*⟩`}{`⟨*depth*⟩`}{`⟨*anchor*⟩`}`: Adds content to the table of contents. The marker is *marker* (e.g. 1.1; this is printed to the left of the title), title is *title* (e.g. chapter name), value is *value* (e.g. page number), depth is *depth*, and is linked to the anchor *anchor*. The depth *depth* determines the style used in the table (see `\settocdepthformat`).

`\tableofcontents`: Prints the table of contents.

`\settocdepthformat {`⟨*depth*⟩`}{`⟨*options*⟩`}`: Sets the format of the table of contents at the depth *depth*. *options* is a map with the following fields:

- `marker`: the style for the marker (default is `\setfont{rm}`; the marker is passed as a parameter to `marker`);
- `marker buffer`: the buffer between the title and marker (default is `.25cm`);
- `title`: the style for the title (default is `\setfont{rm}`; the title is passed as a parameter to `title`);
- `value`: the style for the value (default is `\setfont{rm}`; the value is passed as a parameter to `value`);
- `leader`: the leader to add between the title and value (default is nothing);
- `indent`: the amount to indent the line (default is 0pt);
- `buffer`: the amount of buffer to add around the line (default is 0pt).

PDFToolbox provides four types of sectioning: parts, sections, subsections, and subsubsections. Each has a counter in its name (e.g. `section`), and a macro with the current section name (e.g. `\currsection`).

`\section (*){`⟨*title*⟩`}`: Adds a section to the document. If the asterisk is added, the section is a "pseudosection": the section counter is not incremented and not displayed, and the section is not added to the table

of contents. Otherwise the section counter is incremented and displayed, and the section is added to the table of contents.

`\subsection (*){⟨title⟩}`: Adds a subsection to the document. If the asterisk is added, the subsection is a "pseudosubsection": the subsection counter is not incremented and not displayed, and the subsection is not added to the table of contents. Otherwise the subsection counter is incremented and displayed, and the subsection is added to the table of contents.

`\subsubsection (*){⟨title⟩}`: Adds a subsubsection to the document. If the asterisk is added, the subsubsection is a "pseudosubsubsection": the subsubsection counter is not incremented and not displayed, and the subsubsection is not added to the table of contents. Otherwise the subsubsection counter is incremented and displayed, but the subsubsection is still not added to the table of contents.

3 pdfGraphics

The pdfGraphics section of the PDFToolbox toolbox is for pdf-specific graphics macros. You can use it to create colorful documents with illustrations, etc.

3.1 Colors

In `pdfGraphics/colors.tex`, PDFToolbox provides macros for coloring text and areas of your document.

`\color` ⟨*color space*⟩`{`⟨*code*⟩`}`
`\color {`⟨*name*⟩`}` : Switches the color of the document. In its first form, *color space* corresponds to either `rgb` or `cmyk`, and *code* is either an rgb or cmyk code. In its second form, if *name* is a predefined color name (see `\definecolor`), the color is switched to it.

`\localcolor` ⟨*color space*⟩`{`⟨*code*⟩`}{`⟨*text*⟩`}`
`\localcolor {`⟨*name*⟩`}{`⟨*text*⟩`}` : Switches the color of *text*, according to the options provided (see `\color`).

`\definecolor {`⟨*name*⟩`}{`⟨*color space*⟩`}{`⟨*code*⟩`}`: Defines a color of name *name* whose space is *color space* (either `rgb` or `cmyk`) of code *code* (either an rgb or cmyk code).

`\letcolor {`⟨*new name*⟩`}{`⟨*name*⟩`}`: Defines a color of name *new name* to be equal to the existing color of name *name*.

`\definecolormacro {`⟨*name*⟩`}{`⟨*color space*⟩`}{`⟨*code*⟩`}`: Calls `\definecolor`, and also defines a macro of name *name* which is equivalent to `\localcolor <color space>{<code>}{#1}`.

The following colors are defined:

red blue green yellow orange purple white black darkgreen grey

`\highlightbox` ⟨*color space*⟩`{`⟨*code*⟩`}{`⟨*material*⟩`}`
`\highlightbox {`⟨*name*⟩`}{`⟨*material*⟩`}` : Colors the background of the material *material* according to the color provided. For example `\highlightbox {red}{pdfToolbox}` will yield pdfToolbox.

`\coloredbox` ⟨*color space*⟩`{`⟨*code*⟩`}{`⟨*material*⟩`}`
`\coloredbox {`⟨*name*⟩`}{`⟨*material*⟩`}` : Like `\highlightbox` but adds a buffer of space around *material* in accordance with `\bufferwidth` and `\bufferheight`. For example the following code: `\coloredbox {red}{pdfToolbox}`; will yield pdfToolbox.

`\framecoloredbox` ⟨*color space*⟩`{`⟨*code*⟩`}{`⟨*material*⟩`}`
`\framecoloredbox {`⟨*name*⟩`}{`⟨*material*⟩`}` : Like `\coloredbox` but adds a frame around *material* of width `\framewidth`. For example `\framecoloredbox {red}{pdfToolbox}` will yield pdfToolbox.

`\framebox {`⟨*material*⟩`}`: Adds a frame around *material* with a buffer of `\bufferwidth` and `\bufferheight` of width `\framewidth`.

`\curvedcolorbox {`⟨*stroke color*⟩`}{`⟨*bg color*⟩`}{`⟨*material*⟩`}{`⟨*curve control*⟩`}`: Creates a curved color framed box around *material* with frame color *stroke color* and background color *bg color* (which may be names or of the form `<color space>{<code>}`. The curve's stroke width is determined by `\curvewidth`, and the buffer around the material is determined by `\curvebuffer`.

control is a sequence of 8 symbols of the form ⟨*bline*⟩⟨*bldot*⟩⟨*lline*⟩⟨*tldot*⟩⟨*tline*⟩⟨*trdot*⟩⟨*rline*⟩⟨*brdot*⟩ where each ⟨`X`*dot*⟩ corresponds to whether or not a corner is curved or not (`bl` for bottom left, `tl` for top left,

etc.), and each ⟨X*line*⟩ corresponds to whether or not a border is drawn or not (`b` for bottom, `l` for left, etc.). For a corner, `.` corresponds to a curve and `X` for a right corner. For a border, `-` corresponds to drawing the line and `_` to not.

A shadow of color `\boxshadowcolor` is added to to the box, at an x and y offset of `\shadowxoff` and `\shadowyoff` respectively.

So for example:

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{-.-.-.-.}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{-X-.-.-.}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{-.-X-.-.}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{-.-.-X-.}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{-.-.-.-X}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{_X-X_X-X}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{_._._._.}':` pdfToolbox

`\curvedcolorbox {blue}{red}{\color {white}pdfToolbox}{-X_X-X_X}':` pdfToolbox

`\fakebold {`⟨*material*⟩`}`: Bolds the material *material* (essentially just thickening the stroke width according to `\fakeboldwidth`).

`\flip {`⟨*material*⟩`}`: ƨqifl *material* about its vertical axis.

3.2 Colorboxes

In `pdfGraphics/colorboxes.tex`, PDFTOOLBOX provides macros for pretty printing textboxes (ppboxes). These are simply colored textboxes which can split across pages. There are two kinds of pretty textboxes: `ppboxes` and `linedppboxes`.

`\bppbox {`⟨*bg color*⟩`}{`⟨*stroke color*⟩`}{`⟨*fg color*⟩`}[`⟨*curve control*⟩`] ... \eppbox`: This creates a `ppbox`, which is just a wrapper around `\curvedcolorbox`.

`\blinedppbox {`⟨*bg color*⟩`}{`⟨*stroke color*⟩`}{`⟨*fg color*⟩`} ... \elinedppbox`: This creates a colored textbox with a rule down the left side. For example:

This is a `linedppbox` with a red background, black stroke, and white text.

The width of the rule is determined by `\pprulewd`, the vertical buffer within the box (around the text) is determined by `\pprulevbuf`, and the horizontal buffer on the left is `\pprulehbuf`.

3.3 Illustrating

In `pdfGraphics/pdfdraw.tex`, PDFTOOLBOX provides macros for creating illustrations.

This feature scares me. Its implementation is a mess and I am scared to change it; but I will need to at some point.

`\bdrawing ...\edrawing`: Begin a drawing environment. The drawing environment is a plane as large as the drawings within it. $(0, 0)$ corresponds to the bottom left corner.

`\addnode {`⟨*text*⟩`}{`⟨*x*⟩`}{`⟨*y*⟩`}{`⟨*name*⟩`}`: Creates a node by the name of *name* with text *text* at coordinate (x, y). You can access the following values (called node-relative coordinates): `<name>.left`, `<name>.top`, `<name>.right`, `<name>.bottom`, `<name>.xcenter`, `<name>.ycenter`.

`\drawpath {`⟨*start x*⟩`}{`⟨*start y*⟩`}{`⟨*end x*⟩`}{`⟨*end y*⟩`}{`⟨*x off*⟩`}{`⟨*y off*⟩`}{`⟨*start cap*⟩`}{`⟨*end cap*⟩`}{`⟨*color*⟩`}`: Draws a line from (*start x*, *start y*) to (*end x*, *end y*). This is offset by *off x* on the x-axis and *off y* on the y-axis (these are dimensions). *start cap* is the linecap used at the starting point, and *end cap* is the linecap used at the end point (see `\definelinecap`). The line is drawn in the color *color*.

The coordinates may be numeric values or node-relative coordinates (see `\addnode`).

`\drawbezier {`⟨*start x*⟩`}{`⟨*start y*⟩`}{`⟨*end x*⟩`}{`⟨*end y*⟩`}{`⟨*off*⟩`}{`⟨*curvature*⟩`}{`⟨*start cap*⟩`}{`⟨*end cap*⟩`}{`⟨*color*⟩`}`: Draws a curve from (*start x*, *start y*) to (*end x*, *end y*) with curvature *curvature*. This is offset by *off*,

which must be a pair of the form `{<x off>}{<y off>}` corresponding to the x-axis offset and y-axis offset respectively (dimensions). *start cap* is the linecap used at the starting point, and *end cap* is the linecap used at the end point (see `\definelinecap`). The line is drawn in the color *color*.

The coordinates may be numeric values or node-relative coordinates (see `\addnode`).

`\definelinecap {`⟨*name*⟩`}{`⟨*code*⟩`}{`⟨*width*⟩`}`: Defines a linecap by the name of *name*. *code* is the code which draws the linecap (see Internals of pdfDraw), and *width* is the width of the linecap.

The provided linecaps are:

`>`: → `<`: ← `|-`: ⊢ `-|`: ⊣ `>>`: ≫ `<<`: ≪ `o`: ∘

There is also an empty linecap `-`.

Outside of drawing environments, PDFToolbox provides a macro to make diagrams, `\drawdiagram`. Its usage is `\drawdiagram {`⟨*table*⟩`}{`⟨*arrows*⟩`}`. *table* is a normal TEX alignment table (similar format as `\halign`, without the preamble). *arrows* is a collection of `\diagarrow` macro calls.

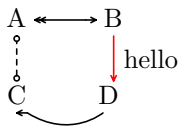
`\diagarrow {`⟨*options*⟩`}`: Draws an arrow in a `\drawdiagram` diagram. *options* contains the following keys:

- `from` (required): the cell from which to start the arrow. Cells start indexing at `{1,1}` for the top left cell where the first number is the row and the second the column;
- `to` (required): the cell to end the arrow;
- `left cap` (default `-`): the start linecap;
- `right cap` (default `>`): the end linecap;
- `color` (default `black`): the color to draw the arrow in;
- `x off` (default `0pt`): the x-axis offset;
- `y off` (default `0pt`): the y-axis offset;
- `text`: the text to add on the arrow;
- `x distance` (default `0pt`): the amount to move the text on the x-axis;
- `y distance` (default `0pt`): the amount to move the text on the y-axis;
- `slide` (default `.5`): where to place the text relative to the arrow;
- `curve`: the amount to curve the arrow;
- `dashed` (valueless): add to make the arrow line dashed;
- `dotted` (valueless): add to make the arrow line dotted;
- `origin orient`: the placement of the start of the arrow relative to the origin (a pair like `{left,bottom}`);
- `dest orient`: the placement of the end of the arrow relative to the destination (a pair like `{left,bottom}`).

So for example,

```
\drawdiagram{
    A&B\cr
    C&D
}{
    \diagarrow{from={1,1}, to={1,2}, left cap=<<}
    \diagarrow{from={1,2}, to={2,2}, color=rgb{1 0 0}, text={hello}, x distance=.5cm}
    \diagarrow{from={2,2}, to={2,1}, curve=10pt, origin orient={xcenter,bottom}, dest orient={xcenter,bottom}}
    \diagarrow{from={2,1}, to={1,1}, dashed, left cap=o, right cap=o}
}
```

Will yield



Between each row of the diagram, space of width `\diagrowbuf` is added. Between each column, `\diagcolbuf`. The height of each row is at least `\diagrowheight` and the width of each column is at least `\diagcolwidth`.

3.4 Listings

In `pdfGraphics/ptb-listings.tex`, PDFTOOLBOX provides macros for writing code listings. The mechanism for how PDFTOOLBOX's listing works is greatly inspired by Petr Olšák's OpTEX. The mechanism is largely the same, though the implementation may differ.

`\setupverb`: This will set up a verbatim environment, essentially changing all special category codes to 12.

`\blisting` ⟨*first line*⟩ `...` `\elisting`: Writes `...` in a verbatim environment, with syntax highlighting if set (see `\loadsyntax` and `\setsyntax`). *first line* (the rest of the line after `\blisting`) will be executed as normal (so you can set syntax here; see `\setsyntax`). The line number of each line in the listing is stored in `\lstlinenum`, which is not reset after each listing.

`\listfile {`⟨*file*⟩`}[`⟨*start*⟩`-`⟨*end*⟩`]`: Creates a listing from file *file*. Reads between lines *start* and *end* (inclusive). If *start* isn't provided, starts from 1. If *end* isn't provided reads until the end (actually a large number).

`\loadsyntax {`⟨*language*⟩`}`: Loads in the necessary information for *language* syntax highlighting. The information is input from the ptb-syntax file `ptb-syntax-`*language*. See the internals of this section for more information on how to write such a file.

Currently PDFTOOLBOX provides support for syntax highlighting of TEX (*language* is `TeX`) and of C (*language* is `C`).

`\setsyntax {`⟨*language*⟩`}`: Sets the syntax to be used for syntax highlighting. This must be used after `\loadsyntax` for *language*.

Some useful macros for customizing syntax highlighting are the following:

- `\lstlineskip`: the space added between each line in the listing;
- `\lstvbuf`: the space added before and after the listing;
- `\lstlinenumbuf`: the kerning added between the number and code on each line in the listing;
- `\lstnumfontset`: sets the font (and whatever else, e.g. color) of the numbers of each line in the listing;
- `\lstfontset`: sets the font (and whatever else, e.g. color) of the content of each line;
- `\lststrut`: the strut added to each line in the listing (for uniform spacing).

Some useful colors to be aware of:

- `lst-fg`: the default foreground color of the listing;
- `lst-bg`: the background color of the listing;
- `lst-comment`: the colors of comments (must be activated in the ptb-syntax file);
- `lst-number`: the colors of numbers (must be activated in the ptb-syntax file).

PDFTOOLBOX also provides a token list `\everylisting` which is inserted before every listing. So for example doing `\everylisting={\lstlinenum=0}` will reset the line numbering before each listing.

II. pdfToolbox internals

1 Utilities

In `pdfToolbox-utils.tex`, PDFToolbox provides various useful utilities for a variety of (relatively) simple tasks.

1.1 Simple Macros

`\_checkloaded {`⟨*name*⟩`}`: Place this at the beginning of a package or a file in a package to ensure you don't include the same file multiple times. It will check if *name* has already been loaded: if it has been, it stops input; otherwise it remembers that *name* has been loaded and continues inputting it.

A few useful short macros:

- `\_xp`: shorthand for `\expandafter`;
- `\_nul`: defined to be `\_nul`; useful as a marker (used, for example, to mark the end of something);
- `\_id`: defined as `\def\_id#1{#1}`;
- `\_gobble`: gobbles the next parameter;
- `\_gobbletilnul`: gobbles until it sees `\_nul` (definition is `\def\_gobbletilnul#1\_nul{}`);
- `\_mstrip`: given a control sequence, returns its name without the escape character;
- `\True`: defined to be `\True`; used when returning a value;
- `\False`: defined to be `\False`; used when returning a value;
- `\glet`: `\global\let`;
- `\_xplet`: takes two inputs `A` and `B`, suppose they expand to `X` and `Y` respectively. Then `\_xplet{A}{B}` is equivalent to `\let XY`;
- `\_afterfi`: within an `\if...\fi` construct, placing code inside `\_afterfi` will execute it (if the condition matches) after the `\fi`;
- `\say`: prints the input on the terminal (on its own line).

`\_ifnextchar` ⟨*char*⟩`{`⟨*first*⟩`}{`⟨*second*⟩`}` `\@ifnextchar` ⟨*char*⟩`{`⟨*first*⟩`}{`⟨*second*⟩`}`: Inspired by LaTeX. Looks at the following character, if it is equal to *char*, executes *first* and otherwise executes *second*. The following character is left in the input stream.

`\_ifstar {`⟨*first*⟩`}{`⟨*second*⟩`}` `\@ifstar {`⟨*first*⟩`}{`⟨*second*⟩`}`: Inspired by LaTeX. Looks at the following character, if it is an asterisk, executes *first* and otherwise executes *second*. The asterisk is removed from the stream.

`\_nopt {`⟨*dim expression*⟩`}`: Expands to the computation of *dim expression* (a dimension expression) without the trailing `pt`.

`\_noptfloor {`⟨*dim expression*⟩`}`: Expands to the whole part of the computation of *dim expression* (a dimension expression) without the trailing `pt`.

`\literal` ⟨*macro definition*⟩: Equivalent to `\def\X<macro definition>\X`.

`\_getline` ⟨*macro*⟩: Reads until a linebreak and then passes that to *macro* as its parameter.

`\reverse` ⟨*macro*⟩`{`⟨*list*⟩`}`: Reverses *list* and puts the result in *macro*.

1.2 Setters

PDFToolbox has a concept of *setters*: these are the macros used for defining things. There are four three: `\currlet, \currdef, \curredef, \currset`. These generally alternate between `\let, \def, \edef, \empty` and `\glet, \gdef, \xdef, \global`. You can change the definitions via the two macros `\localsetters` and `\globalsetters`.

So for example, if you'd like to use an array and make the changes global, you'd first execute `\globalsetters`.

1.3 Repeating Macros

`\commap` ⟨*macro*⟩`{`⟨*list*⟩`}`: If *list* is a comma-separated list of the form `x1,...,xN` and *macro* is `\X`, this will execute `\X{x1}...\X{xN}`.

`\map` ⟨*macro*⟩`{`⟨*list*⟩`}`: If *list* is a list of the form `x1\dots xN` where each `xI` is a group or a single token, and *macro* is `\X`, this will execute `\X{x1}...\X{xN}`.

`\_repeat {`⟨*times*⟩`}{`⟨*code*⟩`}`: Executes *code times* times.

`\_prepeat {`⟨*times*⟩`}`⟨*macro*⟩: If *times* is `N` and *macro* `\X`, executes `\X{1}...\X{N}`.

`\_varrepeat {`⟨*start*⟩`}{`⟨*stop*⟩`}{`⟨*step*⟩`}`⟨*comparison*⟩⟨*macro*⟩: If *macro* is `\X`, *start* is `i`, step is `d`, and stop is `f`: executes `\X{i}\X{i+d}\X{i+2d}...\X{i+Nd}` until the condition (`i+Nd /it comparison /tt f`) is satisfied.

2 pdfData Internals

Due to the nature of its use, most of the macros defined in the pdfData section have already been explained. The only part of pdfData which requires explanation regarding its internals is mappings, which offers richer features than already explained.

2.1 Mappings

Mappings are stored in two places: a *key list*, which is simply a macro consisting of pairs of the form `{key}{value}`, and macros `\key@k` (the second `k` is variable in the name) whose definition is `v`.

Essentially, the major macro in this part is `\_mapkeys_with_setter`. Its usage is

`\_mapkeys_with_setter` ⟨*mapkey macro*⟩⟨*key macro*⟩`{`⟨*map*⟩`}`

where *mapkey macro* is the macro which manages the creation of a key-value pair (explained below), *key macro* is a macro to store the list of keys, and *map* is a map of key-value pairs.

What happens is `\_mapkeys_with_setter` will iterate over *map* and for every key-value pair (`k`, `v`) if the setter *mapkey macro* is `\M` and *key macro* is `\K`, it calls `\M \K{k}{v}`. This should (if `\M` is defined properly) update `\K` to include the pair (`k`, `v`). Furthermore, it should store the value `v` in the macro `\key@k` (the second `k` is variable in the name).

The macro `\_update_lastkeys` is provided for the former: to update `\K`. Simply pass `\_update_lastkeys \K{k}{v}`. The simplest setter (*mapkey macro*) is `\_vanilla_mapkey`, which does exactly what was described and nothing more. Its definition is simply:

```
\def\_vanilla_mapkey#1#2#3{%
    \_xp\def\csname key@\_id#2\endcsname{#3}%
    \_update_lastkeys{#1}{#2}{#3}%
}
```

You can use the macro `\getvalue` to get the value of a key: its definition is simply

```
\def\getvalue#1{%
    \csname key@#1\endcsname%
}
```

Another macro is `\keyexists` whose use is

`\keyexists {`⟨*key*⟩`}`⟨*macro*⟩⟨*key list*⟩

It checks if the key *key* is in *key list*, and if it is, defines *macro* to be equal to the key. Otherwise *macro* is defined to be `\_nul`. For this reason, if you'd like a key to have no value, it is advised to use the `\novalue` macro (whose definition is just `\novalue`).

Another setter is `\_vardef_mapkey`, whose only difference from `\_vanilla_mapkey` is that instead of `\def`ing `\key@k` to be equal to `v`, `\_vardef_mapkey` uses `\_vardef` instead of `\def` (which can be set before calling `\_vardef_mapkey`), and `\_vardefs` `\key@k` to be the (once) expansion of `\_varmap{v}` (where `\_varmap`) can also be set before calling (`\_vardef_mapkey`).

`\mapkeys` is defined as follows:

```
\def\mapkeys#1#2{%
    \_mapkeys_with_setter\_vanilla_mapkey\_keymappings{#1}%
    \_xp\_setdefaults\_xp{\_keymappings}%
    \_mapkeys_with_setter\_protected_mapkey\lastkeys{#2}%
    \_check_required_supplied%
}
```

So first it gets the key-value pairs in *options* (`#1`) using `\_vanilla_mapkey`; it places the results in `\_keymappings`. Then it sets the default values (this is what `\_setdefaults` does; as well as figuring out which keys are required). Then `\mapkeys` calls `\_mapkeys_with_setter` using the setter `\_protected_mapkey` on *input* (`#2`). It stores the results in `\lastkeys`. Then it checks that the required keys have been supplied (`\_check_required_supplied`).

The setter `\_protected_mapkey` is more complicated than the previously-discussed setters. Its use, like all setters, is

`\_protected_mapkey` ⟨*key list*⟩`{`⟨*key*⟩`}{`⟨*value*⟩`}`

But in this case, *key* has a value also in `\_keymappings` as well; this value corresponds to another map containing the settings of *key* (`name`, `default`, `required`, etc.). So now `\_protected_mapkey` will find the settings of *key*, and get the values of each field (via `\_mapkeys_with_setter`). Then it calls `\_vardef_mapkey` with *key* and *value*, using the definitions of `\_vardef` and `\_varmap` according to the settings. Finally it sets the macro `name` (if provided in the settings) to be equal to the value.

3 pdfGraphics Internals

3.1 Colors

There are some useful macros in the `pdfGraphics/colors.tex`, here we describe them.

These macros and file require a clean-up. Unfortunately many other macros are dependent on them, and I am scared to significantly alter anything. One day, though.

`\_rgb_encode` `{`⟨*rgb code*⟩`}`
`\_rgb_encodebg` `{`⟨*rgb code*⟩`}`
`\_rgb_encodefg` `{`⟨*rgb code*⟩`}`
`\_cmyk_encode` `{`⟨*cmyk code*⟩`}`
`\_cmyk_encodebg` `{`⟨*cmyk code*⟩`}`
`\_cmyk_encodefg` `{`⟨*cmyk code*⟩`}`: Gets the code for the specified color for the foreground or background or both.

`\_setcolor_code` `{`⟨*pdf code*⟩`}`: Sets the current color using *pdf code* (which can be obtained using one of the above macros). Essentially just pushing *pdf code* onto the color stack. After the current group, calls `\_pdfcolor_restore`.

`\_pdfcolor_restore`: Restores the color (pops from the color stack).

`\_color_set` `{`⟨*color space*⟩`}{`⟨*color code*⟩`}`
`\_colorbg_set` `{`⟨*color space*⟩`}{`⟨*color code*⟩`}`
`\_colorfg_set` `{`⟨*color space*⟩`}{`⟨*color code*⟩`}`: Sets the current color using *color code* according to *color space* (either `rgb` or `cmyk`).

`\_color_defined` `{`⟨*name*⟩`}`
`\_colorbg_defined` `{`⟨*name*⟩`}`
`\_colorfg_defined` `{`⟨*name*⟩`}`: Sets the current color according to the color *name* (see `\definecolor`).

`\_getcolorparam` ⟨*macro*⟩`{`⟨*place*⟩`}`⟨*color*⟩: Gets the pdf code for *color* (which may be of the form `rgb{...}`, `cmyk{...}`, or `{name}`), and calls *macro* with it as a parameter. *place* is either `fg`, `bg`, or left empty.

`\_setcolor` `{`⟨*place*⟩`}{`⟨*color*⟩`}`: Sets the current color according to *place* and *color*. *place* is either `fg`, `bg`, or left empty.

`\_getcolor` `{`⟨*place*⟩`}{`⟨*color*⟩`}`: Expands to the pdf code for *color* (*place* is either `fg`, `bg`, or left empty).

3.2 Colorboxes

PDFTOOLBOX provides a relatively simple interface for creating colorboxes like `\bppbox`. The main macro is `\_splitcontentbox`, whose usage is

`\_splitcontentbox` {⟨*buffer*⟩}⟨*macroT*⟩⟨*macroS*⟩⟨*macroM*⟩⟨*macroE*⟩

Which repetitively splits the box `\_contentbox` into `\_splitbox` to fill the remaining material on a page or in the box itself. Then the split box is passed to *macroX* for pretty formatting. *macroT* is if the material fits entirely on a single page, otherwise the first box uses *macroS*, the last box uses *macroE*, and all intermediate boxes use *macroM*. *buffer* is the total amount of vertical buffering that *macro* adds to the box it prints.

So to create your own prettyprint-box (ppbox), you create two macros, say `\beginpp` and `\endpp`. In `\beginpp` you add the code which should go before the ppbox and starts getting content for `\_contentbox`. For example, it could be as simple as:

```
\def\beginpp#1#2{%
    \def\_colorcontentbox{%
        \hbox{\coloredbox{#1}{\_setcolor{}{#2}\box\_splitbox}}%
    }%
    \par\kern.5cm\null\par%
    \setbox\_contentbox=\vbox\bgroup
        \hsize=\dimexpr\hsize-\bufferwidth * 2\relax%
}

\def\endpp{%
    \egroup%
    \_splitcontentbox{\bufferwidth * 2}%
        \_colorcontentbox\_colorcontentbo\_colorcontentbo\_colorcontentboxxx%
    \kern.5cm\relax%
}
```

This creates a ppbox which is simply a wrapper around `\coloredbox`. It colors the background in `#1` and the foreground in `#2`.

In depth, here's how it works:

(1) First, `\beginpp` defines `\_colorcontentbox` to simply place `\_splitbox` into a `\coloredbox` of color `#1`, and sets the foreground color to `#2`.

(2) Then it adds some space before the start of the first ppbox. The reason for the `\null\par` is to move the kern from the list of recent contributions to the main vertical list (see, e.g. the TEXbook for more information on TEX's output routines).

(3) Then `\beginpp` begins reading content for `\_contentbox`. It alters `\hsize` to compensate for the buffer added by `\coloredbox`.

(4) When `\endpp` is called, it first stops the capture of `\_contentbox` with `\egroup`.

(5) Then it calls `\_splitcontentbox{\bufferwidth * 2}\_colorcontentbox`, which splits the captured material (in `\_contentbox`) and places each `\_splitbox` in `\_colorcontentbox`, which was defined in `\beginpp`. `\bufferwidth * 2` corresponds to the amount of vertical buffering `\_colorcontentbox` adds to `\_splitbox`.

(6) `\endpp` adds buffering after the final ppbox.

3.2.1 The Mechanism

In this section we describe the mechanism through which PDFTOOLBOX creates colorboxes. A good reference for this section, which discusses the mechanism through which TEX creates pages (the output routine) is "The Advanced TEXbook" by David Salomon.

We begin with a simple macro which expands to the amount of space left in the page.

ptb-colorboxes

```
\def\_spaceleft{%
    \ifdim\pagetotal=\z@%
        \dimexpr\vsize - \topskip\relax%
    \else%
        \dimexpr \pagegoal - \pagetotal - \topskip\relax%
```

ptb-colorboxes

```
    \fi%
}
```

\pagetotal is the height of the main vertical list (MVL). \pagegoal is the goal height of the MVL. It is generally equal to \vsize, but when footnotes or similar are generated, their heights are subtracted from \pagegoal. At the beginning of each page, \pagegoal is set to \maxdimen (hence the use of the \ifdim).

PDFToolbox stores the content to be split across pages in \box_contentbox.

The following macro _getsplitdim gives the dimension that _contentbox should be split into. Its parameter is the size of the extra vertical glue that is added to each box.

ptb-colorboxes

```
\newif\if_recheck
\newif\if_lastbox
\newif\if_firstbox
\def\_getsplitdim#1{%
    \ifdim\dimexpr\_spaceleft-#1\relax>\ht\_contentbox%
        \edef\_splitdim{\the\ht\_contentbox}%
        \_recheckfalse%
        \_lastboxtrue%
    \else%
        \_lastboxfalse%
        \ifdim\dimexpr\_spaceleft-#1\relax>\z@%
            \edef\_splitdim{\the\dimexpr\_spaceleft-#1\relax}%
            \_recheckfalse%
        \else
            \if_recheck%
                \vfil\break\vfilneg%
                \_recheckfalse%
            \else%
                \null\par%
                \_rechecktrue%
            \fi%
            \_getsplitdim{#1}%
        \fi%
    \fi%
}
```

(1) First we check if there is enough space to place the entirety of the box into the page. If so set _splitdim to the height of the box, and set _lastbox to true since we wil be placing the entirity of the box.

(2) Otherwise, check if there is any space left on the current page (recall that #1 is the amount of vertical space added.) If there is, then set _splitdim to the amount of space left.

(3) Otherwise, we perform the following checks:

- **(i)** If _recheck is false, then we add \null\par to the page. This just moves all recent contributions to the MVL. The reason for this is that we need to get rid of all the material that came before the colorbox, and we then recheck the dimension, and set _recheck to true.
- **(ii)** Otherwise, the MVL is up-to-date, and there is still not enough room. So we try to fill in the rest of the vertical space and \break. If for whatever reason this doesn't work, we add \vfilneg to remove the glue added.

In either case we get the dimension again.

Now, the main macro is _splitcontentbox:

ptb-colorboxes

```
\def\__splitcontentbox#1#2#3#4#5{
```

ptb-colorboxes

```
    \unless\ifvoid\_contentbox%
        \_getsplitdim{#1}%
        \setbox\_splitbox = \vsplit\_contentbox to\_splitdim\relax%
        \setbox0=\vbox{%
            \if_firstbox%
                \if_lastbox%
                    #2%
                \else%
                    #3%
                \fi%
            \else%
                \if_lastbox%
                    #5%
                \else%
                    #4%
                \fi%
            \fi%
        }%
        \vbox to\z@{\copy0\vss}%
        \kern\ht0\relax\penalty\z@%
        \_firstboxfalse%
        \__splitcontentbox{#1}{#2}{#3}{#4}{#5}%
    \fi%
}
\def\_splitcontentbox#1#2#3#4#5{{%
    \splittopskip=\z@%
    \boxmaxdepth=\z@%
    \offinterlineskip%
    \_firstboxtrue%
    \__splitcontentbox{#1}{#2}{#3}{#4}{#5}%
}}
```

It first sets `\splittopskip` to `0pt` so that no extra glue is added to the top of `\vsplit`. Then `\boxmaxdepth` is also set to `0pt` so that the depth of the split boxes will be `0pt` and we can deal only with height. We also turn off interlineskip so there is no extra glue added around the split boxes (these do not affect the contents of `\_contentbox` since it has already been boxed).

Now we repeat until `\_contentbox` is empty:

(1) we get the amount of space to split the box into via `\_getsplitdim` (explained above);

(2) we split `\_contentbox` into `\_splitbox` of this dimension;

(3) `#2` is the output routine of the colorbox, it places the contents of `\_splitbox` into whatever format the user specifies. We set `\box0` to this;

(4) we add the box to the page, and set a penalty of 0 so that the page can be broken at that point if necessary.

The definition of `\bppbox` and `\eppbox` are a little enlightening:

ptb-colorboxes

```
\def\_ppbuf{0pt}
\def\_bppbox#1#2#3[#4]{%
    \_getdotsnlines#4\_nul%
    \def\_colorcontentboxT{%
        \hbox{\curvedcolorbox{#2}{#1}{\_setcolor{}{#3}\box\_splitbox}{#4}}%
    }%
    \def\_colorcontentboxS{%
        \hbox{\curvedcolorbox{#2}{#1}{\_setcolor{}{#3}\box\_splitbox}%
```

ptb-colorboxes

```
            {_X\_linel\_dottl\_linet\_dottr\_liner X}%
        }%
    }%
    \def\_colorcontentboxM{%
        \hbox{\curvedcolorbox{#2}{#1}{\_setcolor{}{#3}\box\_splitbox}%
            {_X\_linel X_X\_liner X}%
        }%
    }%
    \def\_colorcontentboxE{%
        \hbox{\curvedcolorbox{#2}{#1}{\_setcolor{}{#3}\box\_splitbox}%
            {\_lineb\_dotbl\_linel X_X\_liner\_dotbr}%
        }%
    }%
    \par%
    \kern\_ppbuf\relax%
    \null\par% Move the kern from recent contributions to MVL
    \setbox\_contentbox=\vbox\bgroup%
        \hsize=\dimexpr\hsize-\_actual_curve_buf * 2\relax%
}

\def\bppbox#1#2#3{%
    \_ifnextchar[ {\_bppbox{#1}{#2}{#3}}{\_bppbox{#1}{#2}{#3}[-.-.-.-.]}%
}

\def\eppbox{%
    \egroup%
    \_splitcontentbox{2\_actual_curve_buf}%
        \_colorcontentboxT\_colorcontentboxS\_colorcontentboxM\_colorcontentboxE%
    \vskip\_ppbuf\relax%
}
```

What's of interest here is how `\_bppbox` defines the different `\_colorcontentbox`s. Firstly, `\_getdotsnlines` is a macro which defines `\_line`*side* and `\_dot`*corner* according to the stream of 8 characters which follow it (until `\_nul`). Now, `\_colorcontentboxT` is defined as you'd expect. And `\_colorcontentboxS` is defined so that it preserves the top corners that are input as well as all but the bottom side. `\_colorcontentboxM` and `\_colorcontentboxE` are defined similarly.

3.3 Illustrating

This is a complicated and messy part of PDFToolbox. Documentation will be added once it is cleaned up.

3.4 Listings

3.4.1 The Mechanism

We first begin by discussing the mechanism for how listings work in PDFToolbox. Credit where credit is due; the mechanism is greatly inspired and copied from Petr Olšák's OpTEX, though the implementation may differ. The mechanism is relatively simple: all PDFToolbox does is the following:

(1) call `\the\everylisting` and whatever is given in the remaining line after `\blisting`;

(2) call `\setupverb` which changes the catcode of special characters and `^^M` to 12;

(3) capture the entirety of the listing from `\blisting` to `\elisting`;

(4) set line spacing (via `\_setuplstlines`);

(5) call `\the\_commandcapture` which simply sets up macros which are called in `\the\_listingcommands`, which is called immediately afterward (see below);

(6) call `\the\_commandexecute` which executes the commands added in `\the\_listingcommands` on the listing, and then `\the\_macrocallmanager` which alters the definitions of the macros in `\_listingcommands` to their proper definitions (explanation later);

(7) at this point, the listing is set up so that everything is ready for printing;

(8) the line manager is set up (which handles printing each line in the listing);

(9) the additional vertical buffering added by `\syntaxoutbox` is computed into `\syntaxoutboxbuf` by `\syntaxoutboxsetbuf`;

(10) the listing is processed through a ppbox dictated by `\syntaxoutbox`.

This is a deceptive description of how this process works. But what's important is `\_commandexecute`, which has all the commands for setting up syntax highlighting and the verbatim environment. This is not a token list to messed with by the user directly, it should be done indirectly through `\_listingcommands`, which in turn should be altered indirectly through ptb-syntax files.

But if no syntax is set, the definition of `\_commandexecute` is essentially:

```
\_commandexecute={
    \_execute{\_r_replace{ }{\w \w}}
    \_execute{\_r_replace{^^M}{\w\n\n\w}}
}
```

`\_execute` is defined simply to `\def\_execute#1{#1\_code}` right before execution, where `\_code` is the captured listing. And `\_r_replace` is defined to simply be `\def\_r_replace#1#2#3{\replace{#3}{#1}{#2}}`. So when `\_commandexecute` is called, the result is simply two lines: `\replace{\_code}{ }{\w \w}` and `\replace{\_code}{^^M}{\w\n\n\w}`.

`\replace` ⟨*macro*⟩`{`⟨*pattern*⟩`}{`⟨*replacement*⟩`}`: Replaces (the expansion of) *pattern* with (the expansion of) *replacement* in the definition of *macro*.

So the result is that now `\_code` contains the listing, but where each space is now swapped with `\w \w` and each line ending with `\w\n\n\w`. `\_code` is actually defined to be the listing wrapped in `\n\w...\w\n`, so the result is that every line in the listing is wrapped in `\n...\n` and every word is wrapped in `\w...\w`.

Now suppose you wanted to replace all occurrences of `hi` with `hello`. You'd need to add the line `\_execute{\_r_replace{\w hi\w}{\w hello\w}}` to `\_commandexecute` (note that the pattern and replacement are wrapped in `\w`s). You can do so with the following command:

`\_add_command_replace {`⟨*pattern*⟩`}{`⟨*replacement*⟩`}`: Adds

`\_execute {\_r_replace {`⟨*pattern*⟩`}{`⟨*replacement*⟩`}}`

to `\_commandexecute`.

But this is unwieldy, so the actual mechanism used is as follows:

(1) `\_commandcapture` sets the definitions of `\replace` and `\replacefromto` (see below) to `\_add_command_replace` and `\_add_command_replacefromto`.

(2) `\_listingcommands` contains uses of `\replace` and `\replacefromto`, which are executed. This adds the required lines to `\_commandexecute`.

(3) `\_commandexecute` is executed.

Now to explain `\loadsyntax` and `\setsyntax`: `\loadsyntax {`⟨*language*⟩`}` reads from the ptb-syntaxptb-syntax file `ptb-syntax-`*language*`.tex`, which should define a token list `\_`*language*`_listingcommands`. Then `\setsyntax` simply sets `\_listingcommands` to `\_`*language*`_listingcommands`.

3.5 Usage

Now notice an issue: `\replace` (and `\replacefromto`) both expand their arguments. What if the arguments expand to invalid code? This is the purpose of `\_macrocallmanager` and two basic macros: `\call` and `\mlcall`. They are set to `\relax` so they aren't expanded in `\replace` and friends, and `\_macrocallmanager` sets them to their proper definitions:

`\call` ⟨*macro*⟩`{`⟨*parameters*⟩`}{`⟨*last*⟩`}`: Simply calls `\macro` ⟨*parameters*⟩`{`⟨*last*⟩`}`.

`\mlcall` ⟨*macro*⟩`{`⟨*parameters*⟩`}{`⟨*last*⟩`}`: If *last* is equal to `x1\n\n x2\n\n...\n\n XN`, expands to `\call <macro>{<parameters>}{x1}\n\n...\n\n\call <macro>{<parameters>}{xN}`.

Since `\call` and friends are redefined only after `\_commandexecute` is executed, to call a macro without it expanding you can use them. For example, to replace `int` with `int` colored red, you can use `\replace{\w int\w}{\w\call localcolor{{red}}{int}\w}`.

Two shortcuts are provided: `\c` and `\mc`. `\c {`⟨*color*⟩`}{`⟨*text*⟩`}` will set the color of *text* (in a `\call`), and `\mc {`⟨*color*⟩`}{`⟨*text*⟩`}` will also set the color but in a `\mlcall`.

Now what is `\replacefromto`?

`\replacefromto` ⟨*macro*⟩`{`⟨*start*⟩`}{`⟨*end*⟩`}{`⟨*replacement*⟩`}`: Replacement is a macro definition with a single pattern (e.g. `{#1}`). `\replacefrom` matches

⟨*start*⟩`##1`⟨*end*⟩

in the expansion of *macro* and replaces it with *replacement*, this redefines *macro*.

So for example if `\X` expands to `(.)(.)`, `\replacefrom\X(){[#1]}` will redefine `\X` to be `[.][.]`.

3.5.1 An Example

PDFToolbox provides syntax highlight for the C language in `ptb-syntax-C.tex`, whose content is:

ptb-syntax-C

```
\global\newtoks\lstCcolors
\global\newtoks\_C_listingcommands

\global\lstCcolors={
    \definecolor{preprocessor}{rgb}{0 0 1}
    \definecolor{special char}{rgb}{.7 0 .7}
    \definecolor{keyword}{rgb}{1 0 0}
    \definecolor{quote}{rgb}{.6 .6 0}
}

\global\_C_listingcommands={
    \the\lstCcolors
    \replace {\string\"} {{\string\"}}
    \replacefromto "" {\mc{quote}{"#1"}}
    \replacefromto {//}\n {\c{lst-comment}{//#1}\n}
    \replacefromto {/*}{*/} {\mc{lst-comment}{/*#1*/}}
    \replacefromto {\string#}\n {\c{preprocessor}{\string##1}\n}
    \bgroup\lccode`?=`\{\lccode`!=`\}\lccode`.=`\%\lowercase{\egroup
        \replace ?{\w\c{special char}{?}\w}%
        \replace !{\w\c{special char}{!}\w}%
        \replace .{\w\c{special char}{.}\w}%
    }
    \edef\_regA{{!@\string$\string^\string&*()-+=[];:,.<>/}}%$
    \def\_regB#1{\replace{#1}{\w\c{special char}{#1}\w}}
    \_xp\map\_xp\_regB\_regA
    \def\_regB#1{\replace{\w#1\w}{\w\c{keyword}{#1}\w}}
    \map\_regB{%
        {auto}{bool}{break}{case}{char}{const}{continue}{default}{do}{double}{else}{enum}%
        {extern}{false}{float}{for}{goto}{if}{inline}{int}{long}{NULL}{register}{restrict}%
        {return}{short}{signed}{sizeof}{static}{struct}{switch}{true}{typedef}{union}%
        {unsigned}{void}{volatile}{while}%
    }
    \def\_regB#1{\replace{\w#1}{\w\c{lst-number}{#1}\w}}
    \map\_regB{0123456789}
}

```

Let us now explain each part of the file:

ptb-syntax-C

```
\global\newtoks\lstCcolors
\global\newtoks\_C_listingcommands

\global\lstCcolors={
    \definecolor{preprocessor}{rgb}{0 0 1}
    \definecolor{special char}{rgb}{.7 0 .7}
    \definecolor{keyword}{rgb}{1 0 0}
    \definecolor{quote}{rgb}{.6 .6 0}
}
```

This defines two new token lists, `\lstCcolor` (for use within the file), and `\_C_listingcommands`, which as explained previously is what sets the listing commands for the `C` syntax. `\lstCcolor` has C-specific colors, changing it allows you to change the colors of C-specific highlighting. Note that all changes in this file must be global.

ptb-syntax-C

```
\global\_C_listingcommands={
    \the\lstCcolors
```

Begins the definition of `\_C_listing_commands` by first defining C-specific colors.

ptb-syntax-C

```
    \replace {\string\"} {{\string\"}}
    \replacefromto "" {\mc{quote}{"#1"}}
    \replacefromto {//}\n {\c{lst-comment}{//#1}\n}
    \replacefromto {/*}{*/} {\mc{lst-comment}{/*#1*/}}
    \replacefromto {\string#}\n {\c{preprocessor}{\string##1}\n}
```

(1) Swaps `\"` with `{\"}`, so that the quotation in `\"` (backslash-quote) is not replaced by the following lines.

(2) Colors between `"` and `"` with a multiline coloring of color `quote`.

(3) Colors between `//` and `\n` (the end of the line) with the color of a comment. Notice that it adds back in the `\n`; otherwise this will mess up the line-reading.

(4) Replaces between `/*` and `*/` with a multline coloring of color `lst-comment`.

(5) Replaces from `#` to the end of line with a coloring of color `preprocessor`. (Again adding back in `\n`.)

ptb-syntax-C

```
    \bgroup\lccode`?=`\{\lccode`!=`\}\lccode`.=`\%\lowercase{\egroup
        \replace ?{\w\c{special char}{?}\w}%
        \replace !{\w\c{special char}{!}\w}%
        \replace .{\w\c{special char}{.}\w}%
    }
```

Sets replacement for TEX-reserved characters (open and close curly braces, percent).

ptb-syntax-C

```
    \edef\_regA{{!@\string$\string^\string&*()-+=[];:,.<>/}}%$
    \def\_regB#1{\replace{#1}{\w\c{special char}{#1}\w}}
    \_xp\map\_xp\_regB\_regA
```

Replaces non-character letters with a coloring and word break (since, e.g. `x.y` is not a single word). Note the use of `\map`.

ptb-syntax-C

```
    \def\_regB#1{\replace{\w#1\w}{\w\c{keyword}{#1}\w}}
```

ptb-syntax-C

```
    \map\_regB{%
        {auto}{bool}{break}{case}{char}{const}{continue}{default}{do}{double}{else}{enum}%
        {extern}{false}{float}{for}{goto}{if}{inline}{int}{long}{NULL}{register}{restrict}%
        {return}{short}{signed}{sizeof}{static}{struct}{switch}{true}{typedef}{union}%
        {unsigned}{void}{volatile}{while}%
    }
```

Replaces keywords with color `keyword`.

ptb-syntax-C

```
    \def\_regB#1{\replace{\w#1}{\w\c{lst-number}{#1}\w}}
    \map\_regB{0123456789}
```

Colors numbers with `lst-number`.

3.5.2 Changing the Output

PDFToolbox outputs the listing in colorboxes according to `\syntaxoutbox`. The default is just as follows:

ptb-listings

```
\def\syntaxoutbox#1{%
    \vbox{\offinterlineskip%
        \hbox{\the\lstheader}%
        \hbox{\coloredbox{lst-bg}{\color{lst-fg}#1}}%
        \hbox{\the\lstfooter}%
    }
}
```

This just places the content in a colored box of color `lst-bg` with a text color of `lst-fg`, along with placing a header and footer. We must also set `\syntaxoutboxbuf`, which is the total amount of vertical buffering added by `\syntaxoutbox` to its contents. This is done by `\syntaxoutboxsetbuf` which must define a macro `\syntaxoutboxbuf` to be the total amount of extra vertical space `\syntaxoutbox` adds.

But we can also do, for example:

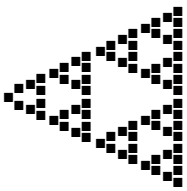
```
\letcolor{lst-stroke}{lst-fg}
\def\syntaxoutbox#1{%
    \hbox{\curvedcolorbox{lst-stroke}{lst-bg}{\_setcolor{}{lst-fg}#1}{-.-.-.-.}}%
}
\def\syntaxoutboxsetbuf{%
    \edef\syntaxoutboxbuf{\the\dimexpr\_actual_curve_buf * 2\relax}%
}
```

Now listings will have a curve colored box:

```
\def\X#1#2{%
    \ifnum#1>0 %
        {\X{\numexpr#1-1\relax}{#2}}^{\X{\numexpr#1-1\relax}{#2}}_{\X{\numexpr#1-1\relax}{#2}}%
    \else%
        \vcenter{\hbox{$\m@th\scriptscriptstyle#2$}}%
    \fi%
}

$$\X{4}\blacksquare$$
```

(The code outputs the following by the way:)



III. Acknowledgments

Many thanks to my family: my two brothers, my mother and father, and my sister. Thank you for your eternal and unwavering support throughout my life, both in the good and the bad.

Thank you to plante (github) for the guidance and mentoring in the way of TEX. Many of the macros in this project are due to, or inspired by, him.

Thank you to the Mathematics Discord server (invite) for fostering a welcoming community where anyone can learn math, and for first introducing me to the world of TEX.

Thank you to all my friends for their continued support and interest.

Thank you to my dogs, past and present. I adore you both, and will forever.