Prototype reimplementation of LaTeX 2ε's block environments using templates

LaTeX Project*

v0.9m 2026-01-26

Abstract

Contents

*Initial reimplementation of lists done by Bruno Le Floch, generalized second version with tagging support by Frank Mittelbach.

1 Introduction

The list implementation in LaTeX 2_ε serves a dual purpose: it implements real lists such as `itemize` or `enumerate`, but it is also used as the basis for vertical blocks, i.e., to specify the vertical spacing and paragraph handling after such block, e.g., in environments like `center`, `quote`, `verbatim`, or in the theorem environments. They are all implemented as "trivial" lists with a single (hidden) item.

While this was convenient to get a consistent layout using a single implementation it is not adequate if it comes to interpreting the structure of a document, because environments based on `trivlist` should not advertise themselves as being a "list" — after all, from a semantic point of view they aren't lists.

The approach taking here is therefore to offer separate template types: `block` (horizontally or vertically oriented data that needs some handling at the start and the end), `para` (that deals with different paragraph layouts), `list` (that handles list related parameters, and `item` (for item layouts and handling).

To address the independent aspects we have the template type `blockenv` that ties them together as necessary when we build document level environments.

For example, a `quote` environment would make use of a (display) `block` and some `para` instance while a standard `enumerate` would make use of a display `block`, a `list`, and an `item` and a `para` instance. An inline list (like `enumerate*` from the `enumitem` package) would be using the same `list` instance but a different (horizontally oriented) `block` instance build from a different template.

Instead of a `list` instance to handle the inner structure of the environment one can use an instance of the type `captionedtext` to produce a display environment with an associated heading/caption, such as a theorem-like environment or a proof environment. Further possibilities (not yet implemented) are templates for producing boxed text or formal quotes like those produced by the `csquotes` package.

2 Template types and templates for blocks and lists

2.1 Template types

2.1.1 The template type 'blockenv'

Arg: 1 key/value list to alter the default parameters of the template instances used by the particular blockenv environment

Arg: 2 Boolean to suppress a number in case this environment normally produces a numbered caption

Arg: 3 Caption/heading text in case this environment supports a caption (most don't), otherwise `\NoValue`

Arg: 4 Sub-caption/heading text in case this environment supports a caption (most don't), otherwise `\NoValue`

Semantics:

This template type is used to implement document-level environments. It defines a `block` instance to handle the layout at the "edge" of the environment data, possibly some paragraph setup through a `para` instance, potentially an "inner" instance for more

complicated environments (such as lists), and possibly some additional setup code for certain environments.

Arguments 2–4 are passed to the instance handling the inner structure, e.g., `list` or `captionedtext` which may or may not make use of it.

It also defines how the `blockenv` behaves with respect to nesting, e.g., does it change when nested and if so how many levels of nesting are supported, etc.

Finally, the template type defines how it appears in a tagged PDF document, what tag names are used, how they are role-mapped and whether it adds additional attributes, etc.

2.1.2 The template type 'block'

Arg: 1 key/value list to alter the default block parameters

Semantics:

Handle the layout aspects of a block of data. In case of a "display" block (i.e., vertically oriented) the spacing and page breaking as well as the handling if the block starts a paragraph or ends one, that is, if text is immediately following the block without being separated by an empty line, then this text is considered to be in the same paragraph as the block.

In case of a horizontally oriented block it covers any special handling at the start and end of the block, e.g, extra spacing, prohibiting or encouraging line breaks, and so forth.

2.1.3 The template type 'para'

Arg: 1 key/value list to alter the default item parameters

Semantics:

Sets up paragraph-specific parameters for H&J, e.g., to implement justification variations, the behavior of `\\` etc. The instances are used in higher-level templates, e.g., in a `block`.

2.1.4 The template type 'list'

Arg: 1 key/value list to alter the default item parameters

Arg: 2 Boolean to suppress a number in case this list environment also produces a numbered heading/caption

Arg: 3 Caption/heading text in case this environment supports a caption (lists normally don't), otherwise `\NoValue`

Arg: 4 Sub-caption/heading text in case this environment supports a caption, otherwise `\NoValue`

Semantics:

Handle the aspects related to list design, e.g., the use and formatting of counters, etc.

Standard LaTeX 2_ε lists have no heading/caption, so arguments 2–4 are ignored in the standard `list` template. But special lists, such as a list of ingredients for a cookbook, might so there might be other templates that make use of them in the future.

Note that this template type does not cover block-related aspects, i.e., a list instance could be used both for a display list or for an inline list.

2.1.5 The template type 'captionedtext'

Arg: 1 key/value list to alter the default item parameters

Arg: 2 Boolean to suppress a number in case this environment also produces a numbered heading/caption

Arg: 3 Caption/heading text for this text block; if not given then `\NoValue`

Arg: 4 Sub-caption/heading text in case this environment supports a caption, otherwise `\NoValue`

Semantics:

Produces a text block with an associated caption/heading, e.g., a theorem-like environment. There may not be a user-supplied caption text—the caption may consist of a fixed text only like "Lemma".

Handles the aspects related to the caption design and typically supports keys for adjusting the layout of the body text, e.g., its font, etc.

Note that this template type does not cover block-related aspects, e.g., the dimensions of the display block are handled there.

2.1.6 The template type 'item'

Arg: 1 key/value list to alter the default item parameters

Semantics:

A sub-type used as part of `list` to easily cover alternative layout for list items.

2.1.7 The template type 'thmstyle'

Arg: 1 key/value list to alter the default item parameters

Arg: 2 Boolean to suppress a number in case this environment also produces a numbered heading/caption

Arg: 3 Caption/heading text for this text block; if not given then `\NoValue`

Arg: 4 Sub-caption/heading text in case this environment supports a caption, otherwise `\NoValue`

Semantics:

A sub-type used as part of `captionedtext` when producing theorem-like environments. It does the bulk of the work and sets up most of the formatting. It has been separated out because many theorem-like environments use the same theorem layout and only differ in the fixed caption text they generate.

Not all templates of type `captionedtext` use `thmstyle` as an inner instance, e.g., proofs are implemented with a template that does everything necessary directly.

2.2 Templates

2.2.1 The `blockenv` template 'std'

Attributes:

`name` (*tokenlist*) Name of the environment used in tracing and error messages.

`tag-name` (*tokenlist*) Name of the tag used for the block inside the PDF. If not explicitly given the name is defined by the `tagging-recipe`. Note that in case of `tagging-recipe=basic` no tag for the block is produced, so any key settings are ignored. Default: ⟨*empty*⟩

`tag-attr-class` (*tokenlist*) An explicit tag class attribute. Default: ⟨*empty*⟩

`tagging-recipe` (*tokenlist*) Defines the way tagging is done. Currently the values `basic`, `standard`, and `list` are supported. Default: `standard`

`transparent-level` (*boolean*) Is this `blockenv` transparent for any blocks nested inside? Default: `false`

`legacy-code` (*tokenlist*) Legacy setup code. This is executed after legacy defaults (from `\@listi`, `\@listii`, etc.) are used but before the block instance is called. Default: ⟨*empty*⟩

`block-instance` (*tokenlist*) Part of the name of the `block` instance that is called. The full name has a `-`⟨*level*⟩ appended. Default: `std-display`

`para-instance` (*tokenlist*) Paragraph settings to use within the environment. If ⟨*empty*⟩ then the current (outer) values are retained. However, the `inner-instance` template might reset/overwrite some of the `para` values, e.g., `list` makes used of `\listparindent` to explicitly set the paragraph indentation for compatibility. Default: ⟨*empty*⟩

`inner-level-counter` (*tokenlist*) Name of an existing (!) counter that is incremented and used to determine final name of the `inner-instance` or empty if always the same inner instance should be used.

`max-inner-levels` (*tokenlist*) Maximum number of nested environments of this kind. Only relevant if there is a `inner-level-counter` specified. Default: `4`

`inner-instance-type` (*tokenlist*) Template type of the inner instance. Currently supported types are `list` and `captionedtext`. Default: ⟨*empty*⟩

`inner-instance` (*tokenlist*) Name of the inner instance (if any). If there is an `inner-level-counter` then the instance name gets `-`⟨*counter value*⟩ appended.
Default: ⟨*empty*⟩

`tagging-suppress-paras` (*boolean*) *describe* Default: `false`

`final-code` (*tokenlist*) Final setup code Default: `\ignorespaces`

Semantics & Comments: The `blockenv` type handles the overall setup for the document-level environments.

This `blockenv` template supports the legacy list setting that are found in many document classes in the macros `\@listi`, `\@listii`, up to `\@listvi`. It also uses the counter `\@listdepth` to track nesting of block, again mainly to support legacy setups (internally it gives it a more appropriate name but it remains accessible through the LaTeX 2_ε name).

The internal block nesting level is stored (for historical reasons) in the `\@listdepth` counter and incremented by each block by one. The starting value at top-level (outside any block) is zero. A block environment with `transparent-level=true` also increments the level before it evaluates and sets its parameters but then decrements it again, just before it starts processing its body.

The template first checks that the block is not too deeply nested.

After the level was increased then corresponding `\@list...` macro to update the legacy defaults is called.

It then sets up the tagging via the `tagging-recipe` setting and executes any code in `legacy-code`.

Afterwards it calls the appropriate `block` instance based on `block-instance` and current level, e.g., `std-display-1`.

Then it sets up paragraph parameters if a `para-instance` was specified (otherwise they stay as they are).

If a `inner-instance` was specified this is called next, or more precisely: if no `inner-level-counter` was specified the instance `inner-instance` is called.

Otherwise, the `inner-level-counter` is incremented and the instance with the name `inner-instance-inner-level-counter` is called.

Finally, the `final-code` is executed (by default `\ignorespaces`).

The maximum number of `blockenv`s that can be nested into each other is restricted by the LaTeX counter `maxblocklevels` with a default value of `6`. If this value is increased then it is necessary to provide additional instances, e.g., `std-display-7`, etc. Decreasing is, of course, always possible, then some of the instances defined are not used and instead the user gets an error that there is too much nesting going on.

If the key `transparent-level` is set to `true` then such an environment alters the nesting level only temporarily (while processing the `blockenv` template) and you can therefore nest those environments as often as you like (a typical example would be `flushleft` anywhere in the nesting hierarchy) as long as the level isn't already at `maxblocklevels`).

2.2.2 The `block` template 'std'

Attributes:

`begin-vspace` (*skip*) Vertical space before the block. Default: `\topsep`

`begin-extra-vspace` (*skip*) Extra vertical space before the block if the block forms its own paragraph. Default: `\partopsep`

`begin-unchained-vspace` (*skip*) Vertical space before the block to use if this is a nested block, both blocks have items or captions, and these should not be chained; see description below. Default: `.5\topsep`

`para-vspace` (*skip*) The default for ordinary blocks is to use the `\parskip` from the outer galley. In lists and some other special blocks this is then changed. Default: `\parskip`

`end-vspace` (*skip*) Vertical space after the block. Default: value from `begin-vspace`

`end-extra-vspace` (*skip*) Extra vertical space after the block if the block forms its own paragraph. Default: value from `begin-extra-vspace`

`item-vspace` (*skip*) The space in front of an item if the block is a list; if not, the setting has no effect. Default: `\itemsep`

`begin-penalty` (*integer*) Penalty for breaking before the block. Default: `\@beginparpenalty`

`end-penalty` (*integer*) Penalty for breaking after the block. Default: `\@endparpenalty`

`item-penalty` (*integer*) Penalty for breaking before an item in the list (except the first). Default: `\@itempenalty`

`left-margin` (*length*) Space on the left of the block. Default: `\leftmargin`

`right-margin` (*length*) Space on the right of the block. Default: `\rightmargin`

`para-indent` (*length*) Paragraph indention for paragraphs within the block. Default: `0pt`

Semantics & Comments: Sets up the main block parameters, e.g. its spacing before and after and the indentation on either side.

It also sets up some parameter defaults for the inner level, e.g., `item-penalty`, `item-vspace` and `para-indent`, which may get overwritten by inner instances that are called.

The vertical spacing before the block covers four different use cases: If there is a caption or an item waiting to be placed, and this item allows for "chaining", and the new block also wants to place an item then no space is added (spacing was already added by the outer block). Instead, the items are chained and placed that the start of the block, i.e., producing a layout like the two nested `itemize` environments here:

- – A second-level item
  – Another …

  More text for the first-level item
- Another first-level item

In that case there is also no vertical space after the block. If the items should not be chained (as specified by the setup of the outer block), then one gets a result like this one (using `itemize` environments inside `description` with different treatment of individual description `\items`):

An normal label
- A second-level item
- Another …

More text for the first-level item

An unchained label
- A second-level item
- Another …

More text for the first-level item

A normal label Another first-level item

If "unchaining" happens, as in the second item, then vertical spacing with the value of `begin-unchained-vspace` is used and at the end you get `end-vertical-space`.

Otherwise, if there is no item or caption waiting to be placed you get a vertical space of `begin-vspace` before the block and if the block is its own paragraph you additionally get `begin-extra-vspace` added to this.

Note that LaTeX 2_ε always chained the list items, so the ability to prohibit this is new functionality.

2.2.3 The para template 'std'

Attributes:

`para-indent` (*length*) Default: `\parindent`

`begin-hspace` (*skip*) Horizontal skip added just in front of the indentation box if non-zero Default: `0pt`

`left-hspace` (*skip*) Default: `0pt`

`right-hspace` (*skip*) Default: `0pt`

`end-hspace` (*skip*) Default: `\@flushglue`

`fixed-word-spaces` (*boolean*) Default: `false`

`final-hyphen-demerits` (*integer*) Default: `5000`

`newline-cmd` (*function(0)*) This defines the meaning of `\\` Default: `\@normalcr`

`para-attr-class` (*tokenlist*) Default: `justify`

Semantics & Comments: The `begin-hspace` (normally `0pt`) is the counterpart of `end-hspace` (which is normally `0pt plus 1fil`). It can be useful in special paragraph shapes. The skip is only inserted into the paragraph if it is non-zero. If it is made non-zero then paragraphs are always at least one line including a construct like `\noindent\par`!

TODO: to be further documented

2.2.4 The `list` template 'std'

Attributes:

`counter` (*tokenlist*) Counter name to be used in a numbered list or empty, if the list is unnumbered. Default: ⟨*empty*⟩

`item-label` (*tokenlist*) Label "string" for a fixed label or as generated from the current `counter` value. Default: ⟨*empty*⟩

`start` (*integer*) Start value for the counter if the list is numbered, otherwise irrelevant. Default: `1`

`resume` (*boolean*) Should a numbered list be resumed from the last instance?. Default: `false`

`item-instance` (*instance*) Instance of type `item` to be used to format the label string. Default: `basic`

`item-vspace` (*skip*) The space in front of an item in the list. If not specified the value specified in the block template instance is used.

`item-penalty` (*integer*) Penalty for breaking before an item (except the first). If not specified the value specified in the block template instance is used.

`item-indent` (*length*) Horizontal displacement of the item. Default: `0pt`

`label-width` (*length*) Width reserved for the formatted item label. Default: `\labelwidth`

`label-sep` (*length*) Horizontal separation between label and following text. Default: `\labelsep`

`legacy-support` (*boolean*) Is formatting the label via `\makelabel` supported? Default: `false`

Semantics & Comments: Sets up handling of list material, e.g., numbering (if any), layout of items and list elements, and tagging, if requested.

2.2.5 The `item` template 'std'

Attributes:

`counter-label` (*function1*) *unused.* Default: `\arabic{#1}`

`counter-ref` (*function1*) *unused.* Default: value from `counter-label`

`label-ref` (*function1*) *unused.* Default: `#1`

`label-autoref` (*function1*) *unused.* Default: `item #1`

`label-format` (*function1*) Formatting of the label, questionable the way it is used. Default: `#1`

`label-strut` (*boolean*) Add a `\strut` to the label? Default: `false`

`label-align` (*choice*) Supported values `left`,`center`, `right`, and `parleft`. *Only partly implemented.* Default: `right`

`label-placement` (*choice*) Placement of the label in relation to a directly following label (of a following inner list). Supported values are `chained`, `unchained`, and `standalone`. Default: `chained`

`label-boxed` (*boolean*) Should the label be boxed? Default: `true`

`next-line` (*boolean*) Default: `false`

`text-font` (*tokenlist*) *unused.*

`compatibility` (*boolean*) Default: `true`

Semantics & Comments: This template is only rudimentary implemented at the moment. It probably needs other keys and the existing ones need a proper implementation!

fix

2.2.6 The `captionedtext` template 'thmlike'

Attributes:

`counter` (*tokenlist*) Counter name to be used if the caption is numbered, otherwise empty. Default: ⟨*empty*⟩

`title` (*tokenlist*) Fixed part of the caption, e.g., a theorem-like environment may want to specify "Lemma" here. Default: ⟨*empty*⟩

`style` (*instance*) Instance of type `thmstyle` that actually implements the theorem-like environment. Default: `plain`

Semantics & Comments: The template combines the fixed `title` and a number (if present) with the caption text as specified on the document element, if one is given, e.g., "Theorem 1. (Fermat)". See also the `proof` template, which handles this differently.

The bulk of the work is then outsourced to an instance of type `thmstyle`. As many such theorem-like environments share the same layout and only differ in the first caption string they use, there is this split for convenience.

2.2.7 The `captionedtext` template 'proof'

Attributes:

`title` (*tokenlist*) Heading for the environment unless overwritten on document level. Default: `Proof`

`punct` (*tokenlist*) Punctuation following the heading. Default: `.`

`caption-placement` (*choice*) Supported values `chained`,`unchained`, and `standalone` Default: `unchained`

`before-hspace` (*skip*) Horizontal displacement of the heading. Default: `0pt`

`after-hspace` (*skip*) Space following the heading, only relevant if text follows on the same line. Default: `5pt`

`caption-decls` (*tokenlist*) Declarations that are applied to the whole caption, e.g., some font settings. Default: ⟨*empty*⟩

`title-format` (*function1*) Formatting applied to the `title` value. Default: `#1`

`punct-format` (*function1*) Formatting applied to the `punct` value. Default: `#1`

`body-decls` (*tokenlist*) Declarations that are applied to body of the environment, e.g., font settings. Default: ⟨*empty*⟩

Semantics & Comments: The "unnumbered?" argument (`#2`) is ignored, as proofs aren't numbered. The template makes use of the caption argument (`#3`) but in contrast to theorem-like environments this template replaces the `title` key value with the content of this argument (if not `\NoValue`).

Typically there is only one layout for proofs so that there is no need to split the formatting over two templates as done for theorem-like environment. That's the reason why the template has several layout customization parameters.

2.2.8 The `thmstyle` template 'std'

Attributes:

`numbered` (*boolean*) Is this kind of environment numbered? Default: `true`

`space` (*tokenlist*) Space to be applied between elements of the heading Default: `\␣`

`punct` (*tokenlist*) Punctuation following the heading. Default: `.`

`caption-placement` (*choice*) Supported values `chained`,`unchained`, and `standalone` Default: `unchained`

`before-hspace` (*skip*) Horizontal displacement of the heading. Default: `0pt`

`after-hspace` (*skip*) Space following the heading, only relevant if text follows on the same line. Default: `5pt`

`order` (*commalist*) Order of elements in the environment caption/heading. Supported values are `title`, `number`, `punct`, `space`, and `note`. Default: `title,space,number,space,note`

`caption-decls` (*tokenlist*) Declarations that are applied to the whole caption, e.g., some font settings. Default: ⟨*empty*⟩

`title-format` (*function1*) Formatting applied to the `title` value. Default: `#1`

`number-format` (*function1*) Formatting applied to the `number` value. Default: `#1`

`punct-format` (*tokenlist*) Formatting applied to the `punct` value. Default: `#1`

`note-format` (*function1*) Formatting applied to the `note` value. Default: `(#1)`

`body-decls` (*tokenlist*) Declarations that are applied to body of the environment, e.g., font settings. Default: ⟨*empty*⟩

Semantics & Comments: Numbering of the environment is suppressed unconditionally if the `numbered` is set to `false`. Otherwise the environment is numbered except when `#2` is `\BooleanTrue`, i.e., if the star form of the environment was used.

The caption of the environment can consist of a title, a number, a punctuation, some spaces and a note. Their order is defined by the key `order`. If a component is specified but has no value, e.g., no note or the numbering suppressed on an individual environment, then the component and any preceding spaces are ignored.

Spaces between elements are uniform (as one can only specify `space` in the `order` key, but it is possible to use this several times in a row and adjust the `space` key accordingly.

Alternatively, one can omit using `space` in the `order` key and instead put all necessary spacing into the individual `...-format` keys. This approach is used, for example, if a theorem style is set up with `\newtheoremstyle` and its ninth argument contains a declaration such as

```
\thmname{#1}\thmnumber{ #2}\thmnote{ (#3)}
```

This is then translated to

```
order         = {title,number,punct,note} ,
title-format  = {#1} ,
number-format = { #2} ,
note-format   = { (#3)} ,
```

when `\newtheoremstyle` sets up a new instance. The downside of this approach is that `\swapnumbers` would not work with such styles (because it would be necessary to transfer the space inside value for the `number-format` key to the value of `title-format`).

If you look closely you also see that in the `order` key a `punct` was added in the list even though it was not present originally. This is they way `\newtheoremstyle` worked and so we mimic that.

3 Declaring standard display block environments and their instances

Historically the LaTeX kernel has defined a number of block environments directly, e.g., `center` or lists like `itemize`, but left others to be set up by document classes. For now we declare all of them here, but in the future, some (or even all) might get moved to new class files.

`\SimpleBlockEnv` Most of the standard block environments have no need for a caption, so to simplify the setup we have added the command `\SimpleBlockEnv` that hides the arguments 2–4 required by a `blockenv` instance and gives them suitable values, i.e., `\BooleanFalse\NoValue\Novalue`. This way, a document level definition for the `center` environment will look like this:

```
\NewDocumentEnvironment{center} { !O{} }
   { \SimpleBlockEnv{center}{#1} } { \BlockEnvEnd }
```

instead of the more verbose

```
\NewDocumentEnvironment{center} { !O{} }
  { \UseInstance{blockenv}{center}{#1} \BooleanFalse \NoValue \NoValue }
  { \BlockEnvEnd }
```

We use `!O{}` for the optional argument so that it is only recognized if it immediately follows `\begin{center}` without any spaces to avoid that a `[` at the start of the body text is misinterpreted as the opening bracket of the optional argument. This is only done for environments where this could be a problem.

This will then call the `center` instance of type `blockenv` that handles the rest.

`\BlockEnv` `\BlockEnvEnd` For the environments that make use of the other arguments, we offer `\BlockEnv` as syntactic sugar so that most environment declarations look similar. And we use `\BlockEnvEnd` in both cases to finish off.

```
⟨*class-code⟩
```

In the following sections we provide for all block environments the top-level definition and all instances that are used by it. Instances of type `block` are often reused across the environments, in which case we just provide cross-references. Note that this is a design decision, different classes my want to have adjusted settings for individual environments, in which case they would provide special `block` instances instead of reusing, say, the `std-display-`⟨*level*⟩ instances.

3.1 The `display` and `displayflattened` environments

`displayblock` (*env.*) `displayblockflattened` (*env.*) There are two basic block environments (`displayblock` and `displayblockflattened`) which are similar to LaTeX 2ε's `trivlist` except that they aren't degenerated lists and thus have no hidden `\item` inside.

```
\NewDocumentEnvironment{displayblock}{ !O{} }
  { \SimpleBlockEnv{displayblock} {#1} } { \BlockEnvEnd }
\NewDocumentEnvironment{displayblockflattened}{ !O{} }
  { \SimpleBlockEnv{displayblockflattened} {#1} } { \BlockEnvEnd }
```

3.1.1 Their `blockenv` instances

`blockenv displayblock` (*inst.*) This is like LaTeX 2ε's `trivlist`, i.e., it produces a vertical block with default setting, but doesn't put a list inside but uses a `<Div>` structure.
We list all keys, those with default values, commented out.

```
\DeclareInstance{blockenv}{displayblock}{std}
{
   name                = displayblock
%  ,tagging-recipe      = standard
%  ,tag-name            =
%  ,tag-attr-class      =
  ,transparent-level    = true
%  ,legacy-code         =
%  ,block-instance      = std-display
%  ,para-instance       =
%  ,tagging-suppress-paras = false
%  ,inner-instance      =
%  ,inner-instance-type =         % not relevant as there is no inner instance
%  ,inner-level-counter =         % not relevant as there is no inner instance
%  ,max-inner-levels    = 4        % not relevant as there is no inner instance
%  ,final-code          = \ignorespaces
}
```

The `block` uses the instance `std-display` which is shown below.

nv displayblockflattened (*inst.*) This flattens inner paragraphs without any surrounding tag structure by using the `basic` tagging recipe.

```
\DeclareInstance{blockenv}{displayblockflattened}{std}
{
   name                   = displayblockflattened
  ,tagging-recipe         = basic
  ,tagging-suppress-paras = true
  ,transparent-level      = true
}
```

3.1.2 Their `block` instances

We provide 6 nesting levels (as in LaTeX 2_ε). If you want to provide more you need to change the `maxblocklevels` counter, offer further `std-display-`⟨*level*⟩ instances but also define further (legacy) `\list`⟨*romannumeral*⟩ commands for the defaults. If not, then the settings from the previous level are reused automatically—which may or may not be good enough).

```
\setcounter{maxblocklevels}{6}
```

block std-display-1 (*inst.*)
block std-display-2 (*inst.*)
block std-display-3 (*inst.*)
block std-display-4 (*inst.*)
block std-display-5 (*inst.*)
block std-display-6 (*inst.*)

We show all keys here for reference, with those using their default values commented out:

```
\DeclareInstance{block}{std-display-1}{std}
  {
%    ,begin-vspace       = \topsep
%    ,begin-extra-vspace = \partopsep
%    ,para-vspace        = \parskip
%    ,end-vspace         = \KeyValue{begin-vspace}
%    ,end-extra-vspace   = \KeyValue{begin-extra-vspace}
%    ,item-vspace        = \itemsep
%    ,begin-penalty      = \UseName{@beginparpenalty}
%    ,end-penalty        = \UseName{@endparpenalty}
     ,left-margin        = 0pt
%    ,right-margin       = \rightmargin
%    ,para-indent        = 0pt
  }
\DeclareInstanceCopy{block}{std-display-2}{std-display-1}
\DeclareInstanceCopy{block}{std-display-3}{std-display-1}
\DeclareInstanceCopy{block}{std-display-4}{std-display-1}
\DeclareInstanceCopy{block}{std-display-5}{std-display-1}
\DeclareInstanceCopy{block}{std-display-6}{std-display-1}
```

3.2 The `center`, `flushleft`, and `flushright` environments

All three environments use the `std-display` instance as block instance. They only differ in the choice of para instance.

center (*env.*)
flushleft (*env.*)
flushright (*env.*)

For now we redeclare various document environments as late as possible in order to make tagging work, even if classes have changed the definitions. Of course, this means that such changes get lost.

```
\AddToHook{begindocument/before}[./legacy-core]{
  \RenewDocumentEnvironment{center} { !O{} }
  { \SimpleBlockEnv{center}{#1} } { \BlockEnvEnd }
```

```
  \RenewDocumentEnvironment{flushright} { !O{} }
  { \SimpleBlockEnv{flushright}{#1} } { \BlockEnvEnd }
  \RenewDocumentEnvironment{flushleft} { !O{} }
  { \SimpleBlockEnv{flushleft}{#1} } { \BlockEnvEnd }
}
```

3.2.1 Their `blockenv` instances

`blockenv center` (*inst.*) The `center` environment is defined through the `blockenv` instance `center` which makes use of the `block` instance `std-display-`⟨*level*⟩ and the `para` instance `center`. The block nesting level is not incremented. With respect to tagging, text separated by `\par` commands (or empty lines) inside the environment is not tagged as separate paragraphs, i.e., the whole environment is considered to be part of an outer paragraph.

```
\DeclareInstance{blockenv}{center}{std}
{
   name                   = center
  ,tag-name               =
  ,tag-attr-class         =
  ,tagging-recipe         = basic
  ,tagging-suppress-paras = true
  ,inner-level-counter    =
  ,transparent-level      = true
  ,legacy-code            =
  ,block-instance         = std-display
  ,para-instance          = center
  ,inner-instance         =
}
```

`blockenv flushleft` (*inst.*) Same as `center` except that we use the `para` instance `raggedright`.

```
%\DeclareInstance{blockenv}{flushleft}{std}
%{
%   name                   = flushleft
%  ,tag-name               =
%  ,tag-attr-class         =
%  ,tagging-recipe         = basic
%  ,tagging-suppress-paras = true
%  ,inner-level-counter    =
%  ,transparent-level      = true
%  ,legacy-code            =
%  ,block-instance         = std-display
%  ,para-instance          = raggedright
%  ,inner-instance         =
%}
```

Or more concise in the source and perhaps even faster in processing if only few keys are changed:

```
 \DeclareInstanceCopy{blockenv}{flushleft}{center}
 \EditInstance{blockenv}{flushleft}{
     name          = flushleft
    ,para-instance = raggedright }
```

`blockenv flushright` (*inst.*) Same game for `flushright`.

```
\DeclareInstanceCopy{blockenv}{flushright}{center}
```

```
\EditInstance{blockenv}{flushright}{
   name          = flushright
  ,para-instance = raggedleft }
```

3.2.2 Their `block` instances

They all use the `block` instances `std` which have already been set up in section 3.1.2.

3.2.3 Their `para` instances

Formatting of paragraphs is handled through the `para-instance` key which either refers to a instance of type `para` or is empty, in which case the handling of paragraphs is inherited. The predefined instances are discussed in section 4.

3.3 The `quote` and `quotation` environments

LaTeX 2_ε has two environments for quoting: `quote` and `quotation`. By default they differ only in indentation of inner paragraphs. This is handled by using separate block instances. The paragraph setup is inherited. The block nesting level is incremented.

The tag names are both role-mapped to `<BlockQuote>`.

`quote` (*env.*)
`quotation` (*env.*)

We can't use `\RenewDocumentEnvironment` for `quote` and other environments that are class defined, because some classes aren't implementing them at all. So we use `\DeclareDocumentEnvironment` instead. This problem will vanish if all such definitions move in new versions of the classes instead.

```
\AddToHook{begindocument/before}[./legacy-quotes]{
  \DeclareDocumentEnvironment{quote}{ !O{} }
    { \SimpleBlockEnv{quote} {#1} } { \BlockEnvEnd }
  \DeclareDocumentEnvironment{quotation}{ !O{} }
    { \SimpleBlockEnv{quotation} {#1} } { \BlockEnvEnd }
}
```

3.3.1 Their `blockenv` instances

`blockenv quotation` (*inst.*)

For the `quotation` environment:

```
\DeclareInstance{blockenv}{quotation}{std}
{
   name                = quotation
  ,tag-name            = \UseStructureName{block/quotation}
  ,tag-attr-class      =
  ,tagging-recipe      = standard
  ,inner-level-counter =
  ,transparent-level   = false
  ,legacy-code         =
  ,block-instance      = quotation
  ,inner-instance      =
}
```

`blockenv quote` (*inst.*)

For the `quote` environment:

```
\DeclareInstance{blockenv}{quote}{std}
{
   name                = quote
```

```
  ,tag-name            = \UseStructureName{block/quote}
  ,tag-attr-class      =
  ,tagging-recipe      = standard
  ,inner-level-counter =
  ,transparent-level   = false
  ,legacy-code         =
  ,block-instance      = quote
  ,inner-instance      =
}
```

3.3.2 Their `block` instances

`block quote-1` (*inst.*) Default layout is to indent equally from both sides.

```
\DeclareInstance{block}{quote-1}{std}
  { right-margin = \KeyValue{left-margin} }
```

`block quote-2` (*inst.*)
`block quote-3` (*inst.*)
`block quote-4` (*inst.*)
`block quote-5` (*inst.*)
`block quote-6` (*inst.*)

```
\DeclareInstanceCopy{block}{quote-2}{quote-1}
\DeclareInstanceCopy{block}{quote-3}{quote-1}
\DeclareInstanceCopy{block}{quote-4}{quote-1}
\DeclareInstanceCopy{block}{quote-5}{quote-1}
\DeclareInstanceCopy{block}{quote-6}{quote-1}
```

`block quotation-1` (*inst.*) Quotation additionally changes the `para-indent`.

```
\DeclareInstance{block}{quotation-1}{std}
   { para-indent = 1.5em , right-margin = \KeyValue{left-margin} }
```

`block quotation-2` (*inst.*)
`block quotation-3` (*inst.*)
`block quotation-4` (*inst.*)
`block quotation-5` (*inst.*)
`block quotation-6` (*inst.*)

```
\DeclareInstanceCopy{block}{quotation-2}{quotation-1}
\DeclareInstanceCopy{block}{quotation-3}{quotation-1}
\DeclareInstanceCopy{block}{quotation-4}{quotation-1}
\DeclareInstanceCopy{block}{quotation-5}{quotation-1}
\DeclareInstanceCopy{block}{quotation-6}{quotation-1}
```

3.4 The `verse` environment

The `verse` environment of LaTeX is intended for poetry. Not sure what that should mean with respect to tagging.

`verse` (*env.*) Implementation is like `quote` etc.

```
\AddToHook{begindocument/before}[./legacy]{
  \DeclareDocumentEnvironment{verse}{!O{}}
    { \SimpleBlockEnv{verse} {#1} } { \BlockEnvEnd }
}
```

3.4.1 Their `blockenv` instances

`blockenv verse` (*inst.*)

```
\DeclareInstance{blockenv}{verse}{std}
{
   name                = verse
  ,tag-name            = \UseStructureName{block/verse}
  ,tag-attr-class      =
  ,tagging-recipe      = standard
  ,inner-level-counter =
  ,transparent-level   = false
```

```
  ,legacy-code          =
  ,block-instance       = quote       % reuse?
  ,para-instance        = verse
  ,inner-instance       =
}
```

The special indentation on continuation lines (the way LaTeX handled poetry is done in the `para` instance `verse`, defined later on.

3.5 The `verbatim`, `verbatim*` and `alltt` environments

`verbatim` (*env.*)
`verbatim*` (*env.*)

Here are the definitions for the verbatim environments They look somewhat different than others (but this isn't the final definition). At the moment we use 2 optional arguments, the second is only there so that there is yet another scan even if one optional argument got detected. That then scans away the newline so that afterwards we can reinsert one via `\obeyedline`. A better solution will be to use a `c` specifier for grabbing the body, but that is for another day not Christmas Eve.

fix

```
\AddToHook{begindocument/before}[./legacy-verbatims]{
  \RenewDocumentEnvironment{verbatim}{ ={legacy-code} !o !o }
    { \SimpleBlockEnv{verbatim} {#1} \obeyedline } { \BlockEnvEnd }

  \RenewDocumentEnvironment{verbatim*}{ ={legacy-code} !o !o }
    { \SimpleBlockEnv{verbatim*} {#1}  \obeyedline } { \BlockEnvEnd }
```

`alltt` (*env.*)
`alltt*` (*env.*)

The alltt package implements a variation on verbatim handling where backslash and braces retain their normal meanings. We also reimplement it using the template approach The `alltt*` variant didn't exist in the package, but it is trivial to set it up as well.

The parsing here should be adjusted as well, eventually.

```
  \NewDocumentEnvironment{alltt}{ ={legacy-code} !o }
    { \SimpleBlockEnv{alltt} {#1} } { \BlockEnvEnd }
  \NewDocumentEnvironment{alltt*}{ ={legacy-code} !o }
    { \SimpleBlockEnv{alltt*} {#1} } { \BlockEnvEnd }
}
```

3.5.1 Their `blockenv` instances

`blockenv verbatim` (*inst.*)

The `verbatim` environment is defined through `blockenv` instance `verbatim` that makes use of the `block` instance `verbatim-`⟨*level*⟩ and the `para` instance `justify`. The block nesting level is not incremented. Verbatim processing requires various catcode changes, etc. and as a consequence a special parsing routine that grabs the whole environment while these catcodes are in force. This setup is done in the `final-code` key and its last action is to initiate the special parsing.

```
\DeclareInstance{blockenv}{verbatim}{std}
{
   name                  = verbatim
  ,tag-name              = \UseStructureName{block/verbatim}
  ,tag-attr-class        =
  ,tagging-recipe        = standard
  ,tagging-suppress-paras = true
  ,inner-level-counter   =
  ,transparent-level     = true
  ,legacy-code           =
  ,block-instance        = verbatim
  ,inner-instance        =
```

```
  ,para-instance          = justify
```

Here is where `verbatim` and `verbatim*` technically differ: in the former we set up spaces to become nonbreakable spaces (if necessary followed by a `\pdffakespace` in the pdfTEX engine) and in `verbatim*` we set it up to generate visible space chars.

```
  ,final-code             = \legacyverbatimsetup{invisible}
```

Then we start the special scanning process to look for `\end{verbatim}` with special catcodes and grab everything in between. For `verbatim*` we use `\@sxverbatim` to look for `\end{verbatim*}` instead.[1]

```
                            \@xverbatim
}
```

The role-mapping is `<verbatim>` to `<Code>` and `<codeline>` to `<Sub>` (which is role mapped to `<Span>` in pdf 1.7). Sub inside Code is allowed according the errata of ISO 32005. The paragraphs inside verbatim are flattened. Line numbers should be inside the `<codeline>` structure and be tagged either as `<Lbl>` or `<Artifact><Lbl>`.

`blockenv verbatim*` (*inst.*) The implementation of `verbatim*` is similar using the `blockenv` instance `verbatim*`. Its `final-code` sets up visible spaces and a slightly different parsing that grabs everything up to `\end{verbatim*}`. Otherwise the setup is identical.

```
\DeclareInstance{blockenv}{verbatim*}{std}
{
   name                   = verbatim
  ,tag-name               = \UseStructureName{block/verbatim}
  ,tag-attr-class         =
  ,tagging-recipe         = standard
  ,tagging-suppress-paras = true
  ,inner-level-counter    =
  ,transparent-level      = true
  ,legacy-code            =
  ,block-instance         = verbatim
  ,inner-instance         =
  ,para-instance          = justify
  ,final-code             = \legacyverbatimsetup{visible}
                            \@sxverbatim
}
```

`blockenv alltt` (*inst.*) The implementation of the `alltt` environment from the alltt is more or less identical as well. We just need a slightly different final code to keep backslash and braces functional.

```
\DeclareInstance{blockenv}{alltt}{std}
{
   name                   = alltt
  ,tag-name               = \UseStructureName{block/verbatim}   % private tag instead?
  ,tag-attr-class         =
  ,tagging-recipe         = standard
  ,tagging-suppress-paras = true
  ,inner-level-counter    =
  ,transparent-level      = true
  ,legacy-code            =
  ,block-instance         = verbatim
  ,inner-instance         =
  ,para-instance          = justify
```

[1] Perhaps there should be some other command names for this?

Now set up the special environment settings with most characters verbatim. We don't even have to scan ahead for the `\end{alltt}` because backslash and braces still have their normal meaning.

```
  ,final-code             = \legacyallttsetup {invisible}
}
```

`blockenv alltt*` (*inst.*)

The `alltt*` variant didn't exist in the alltt package, but it is trivial to set it up as well.

```
\DeclareInstance{blockenv}{alltt*}{std}
{
   name                   = alltt*
  ,tag-name               = \UseStructureName{block/verbatim}   % private tag instead?
  ,tag-attr-class         =
  ,tagging-recipe         = standard
  ,tagging-suppress-paras = true
  ,inner-level-counter    =
  ,transparent-level      = true
  ,legacy-code            =
  ,block-instance         = verbatim
  ,inner-instance         =
  ,para-instance          = justify
  ,final-code             = \legacyallttsetup {visible}
}
```

3.5.2 Their `block` instances

`block verbatim-1` (*inst.*)
`block verbatim-2` (*inst.*)
`block verbatim-3` (*inst.*)
`block verbatim-4` (*inst.*)
`block verbatim-5` (*inst.*)
`block verbatim-6` (*inst.*)

Verbatim instances have there own levels so that one can specify specific indentations or vertical separations between lines.

```
\DeclareInstance{block}{verbatim-1}{std}
  {
    ,left-margin      = 0pt
    ,para-vspace      = 0pt
  }

\DeclareInstanceCopy{block}{verbatim-2}{verbatim-1}
\DeclareInstanceCopy{block}{verbatim-3}{verbatim-1}
\DeclareInstanceCopy{block}{verbatim-4}{verbatim-1}
\DeclareInstanceCopy{block}{verbatim-5}{verbatim-1}
\DeclareInstanceCopy{block}{verbatim-6}{verbatim-1}
```

3.6 The `trivlist` environment

In LaTeX 2ε `trivlist` was used to define various display environments that aren't really lists at all. To support such legacy definitions (even though they should be updated to achieve proper tagging) we continue to support and implement it as a `list` environment with a few hardwired settings mimicking the original behavior.

maybe we should simply implement it as a `displayblock` instance (at least when doing tagging)

3.7 The standard lists: `itemize`, `enumerate`, and `description`

`itemize` (*env.*)
`enumerate` (*env.*)
`description` (*env.*)

For the standard lists everything is managed by the blockenv instances.

```
\AddToHook{begindocument/before}[./legacy-lists]{
  \RenewDocumentEnvironment{itemize}{!O{}}
    { \SimpleBlockEnv{itemize} {#1} } { \BlockEnvEnd }
```

```
  \RenewDocumentEnvironment{enumerate}{!O{}}
    { \SimpleBlockEnv{enumerate} {#1} } { \BlockEnvEnd }
  \DeclareDocumentEnvironment{description}{!O{}}
    { \SimpleBlockEnv{description} {#1} } { \BlockEnvEnd }
}
```

3.7.1 Their `blockenv` instances

`blockenv itemize` (*inst.*) The `itemize` environment is defined through the `blockenv` instance `itemize` which makes use of the `block` instance `list-`⟨*level*⟩, and an inner instance `itemize-`⟨*inner-level*⟩ of type `list`. The paragraph setup is inherited.[2] The ⟨*inner-level*⟩ is determined through `\@itemdepth`. The block nesting level and the inner list nesting level are incremented.

```
\DeclareInstance{blockenv}{itemize}{std}
{
   name                = itemize
  ,tag-name            = \UseStructureName{block/itemize}
  ,tag-attr-class      = itemize
  ,tagging-recipe      = list
  ,inner-level-counter = \@itemdepth
  ,transparent-level   = false
  ,max-inner-levels    = 4
  ,legacy-code         =
  ,block-instance      = std-list
  ,inner-instance-type = list
  ,inner-instance      = itemize
  ,para-instance       =
}
```

`blockenv enumerate` (*inst.*) The `enumerate` environment is similar to `itemize` but uses the `blockenv` instance `enumerate`, the `block` instance `list-`⟨*level*⟩, and the inner instance `enumerate-`⟨*inner-level*⟩. The ⟨*inner-level*⟩ is determined through `\@enumdepth`.

```
\DeclareInstance{blockenv}{enumerate}{std}
{
   name                = enumerate
  ,tag-name            = \UseStructureName{block/enumerate}
  ,tag-attr-class      = enumerate
  ,tagging-recipe      = list
  ,transparent-level   = false
  ,max-inner-levels    = 4
  ,legacy-code         =
  ,block-instance      = std-list
  ,inner-level-counter = \@enumdepth
  ,inner-instance-type = list
  ,inner-instance      = enumerate
}
```

[2] In the LaTeX 2_ε implementation justified paragraphs where forced, even if the whole document was set in ragged text. If this slightly strange behavior is desired then one has to set the `para-instance` key to `justify`.

`blockenv description` (*inst.*) The `description` environment uses the `blockenv` instance `description`, the `block` instance `list-`⟨*level*⟩, and the inner instance `description` (no dependency on the nesting level), i.e., the environment has the same appearance on all nesting levels.

```
\DeclareInstance{blockenv}{description}{std}
{
   name                = description
  ,tag-name            = \UseStructureName{block/description}
  ,tag-attr-class      = description
  ,tagging-recipe      = list
  ,inner-level-counter =
  ,transparent-level   = false
  ,legacy-code         =
  ,block-instance      = std-list
  ,inner-instance-type = list
  ,inner-instance      = description
}
```

3.7.2 Their `block` instances

`block std-list-1` (*inst.*)
`block std-list-2` (*inst.*)
`block std-list-3` (*inst.*)
`block std-list-4` (*inst.*)
`block std-list-5` (*inst.*)
`block std-list-6` (*inst.*)

The block instances for the various list environments use the same underlying instance (well, by default) and nothing needs to be set up specifically (because that is already done in the legacy `\list`⟨*romannumeral*⟩ unless a different layout is wanted.

```
\DeclareInstance{block}{std-list-1}{std}{
%    begin-vspace       = \topsep
%   ,begin-extra-vspace = \partopsep
```

This is the only one we have to explicitly set for lists if the default setup is wanted.

```
   ,para-vspace        = \parsep
%   ,end-vspace         = \KeyValue{begin-vspace}
%   ,end-extra-vspace   = \KeyValue{begin-extra-vspace}
%   ,item-vspace        = \itemsep
%   ,begin-penalty      = \UseName{@beginparpenalty}
%   ,end-penalty        = \UseName{@endparpenalty}
%   ,left-margin        = \leftmargin
%   ,right-margin       = \rightmargin
%   ,para-indent        = 0pt
}

\DeclareInstanceCopy{block}{std-list-2}{std-list-1}
\DeclareInstanceCopy{block}{std-list-3}{std-list-2}
\DeclareInstanceCopy{block}{std-list-4}{std-list-3}
\DeclareInstanceCopy{block}{std-list-5}{std-list-4}
\DeclareInstanceCopy{block}{std-list-6}{std-list-5}
```

If the legacy `\list`⟨*romannumeral*⟩ is not used in a modern class then, of course, these instances all need to set up the different parameters explicitly. The new implementation of the standard classes (will) show that approach.

3.7.3 Their `list` instances

For all list instances we have to say what kind of label we want (`item-label`) and how it should be formatted.

`list itemize-1` (*inst.*)
`list itemize-2` (*inst.*)
`list itemize-3` (*inst.*)
`list itemize-4` (*inst.*)

For `itemize` environments this is all we need to do and we refer back to the external definitions rather than defining the `item-label` code in the instance to ensure that old documents still work.

```
\DeclareInstance{list}{itemize-1}{std}{ item-label = \labelitemi   }
\DeclareInstance{list}{itemize-2}{std}{ item-label = \labelitemii  }
\DeclareInstance{list}{itemize-3}{std}{ item-label = \labelitemiii }
\DeclareInstance{list}{itemize-4}{std}{ item-label = \labelitemiv  }
```

`list enumerate-1` (*inst.*)
`list enumerate-2` (*inst.*)
`list enumerate-3` (*inst.*)
`list enumerate-4` (*inst.*)

`enumerate` environments are similar, except that we also have to say which counter to use on each level.

```
\DeclareInstance{list}{enumerate-1}{std}
  { item-label = \labelenumi ,   counter = enumi   }
\DeclareInstance{list}{enumerate-2}{std}
  { item-label = \labelenumii ,  counter = enumii  }
\DeclareInstance{list}{enumerate-3}{std}
  { item-label = \labelenumiii , counter = enumiii }
\DeclareInstance{list}{enumerate-4}{std}
  { item-label = \labelenumiv ,  counter = enumiv  }
```

`list description` (*inst.*)

The `description` lists also use only a single list instance with only one key not using the default:

```
\DeclareInstance{list}{description}{std} { item-instance = description }
```

Of course, if handling of description lists should differ in nested lists all one has to do is to provide an `inner-level-counter` and then define `description-1`, `description-2`, etc.

3.7.4 Their `item` instances

`item basic` (*inst.*)
`item description` (*inst.*)

There are two item instances to set up: `description` for use with the `description` environment and `basic` for use with all other lists (up to now).

```
\DeclareInstance{item}{basic}{std}
                 { label-align = right }
\DeclareInstance{item}{description}{std}
  {
    ,label-format = \normalfont\bfseries #1
    ,label-align  = left
  }
```

3.8 The legacy `list` and `trivlist` environments

`list` (*env.*)

The legacy 2e list environment is more complicated as we have to get the extra arguments accounted for.

```
\AddToHook{begindocument/before}[./legacy]{
  \RenewDocumentEnvironment{list}{O{} m m }
    {
```

We do this by storing them away and then call the list instance. Inside this instance the `legacy-code` key contains `\legacylistsetup` which makes use of the stored values.

```
      \tl_set:Nn \l_@@_legacy_env_params_tl
         {
           \tl_set:Nn \@itemlabel {#2}
```

```
            #3
          }
```

The LaTeX 2_ε lists don't support captions so we use `\SimpleBlockEnv`.

```
      \SimpleBlockEnv{list} {#1}
    }
    { \BlockEnvEnd }
}
```

`trivlist` (*env.*) LaTeX 2_ε defined `trivlist` as an implementation of `list` (or rather the other way around).

Replace with code not using `\list`

```
\AddToHook{begindocument/before}[./legacy]{
  \RenewDocumentEnvironment{trivlist}{ !O{} }
                            { \list[#1]{}
                              {
                                \dim_zero:N \leftmargin
                                \dim_zero:N \labelwidth
                                \cs_set_eq:NN \makelabel \use:n
                              }
                            }
    { \BlockEnvEnd }
}
```

3.8.1 Its `blockenv` instance

`blockenv list` (*inst.*) The generic `list` environment of LaTeX 2_ε is modeled with a `blockenv` instance named `list`, a `block` instance named `std-list-`⟨*level*⟩, and an inner instance named `legacy` (with no dependency on the nesting level). This environment has two arguments and customization of the layout is expected to be directly set in the second argument. For this reason this `legacy` instance is something that shouldn't be changed (all that is attempted to provide a way to support legacy setups).
To set up the default settings (as they were used in LaTeX 2_ε) the `legacy-code` key gets `\legacylistsetup` assigned that contains the necessary code to set up these defaults. Changing the `blockenv` is therefore not recommended for the legacy `list` environment.

```
\DeclareInstance{blockenv}{list}{std}
{
   name                = list
  ,tag-name            = \UseStructureName{block/list}
  ,tag-attr-class      =
  ,tagging-recipe      = list
  ,transparent-level   = false
  ,legacy-code         = \legacylistsetup
  ,block-instance      = std-list
  ,inner-level-counter =
  ,inner-instance-type = list
  ,inner-instance      = legacy
}
```

3.8.2 Its `list` instance

`list legacy` (*inst.*) For the legacy `list` environment there is only one instance which is reused on all levels. This is done this way because the legacy `list` environment sets all its parameters through

its arguments. So this instances shouldn't really be touched. It sets the `legacy-support` key to true, which means that the list code uses `\makelabel` for formatting the label.

```
\DeclareInstance{list}{legacy}{std} {
  ,item-instance = basic
  ,legacy-support = true
}
```

3.9 Theorem-like environments declared through `\newtheorem`

In standard LaTeX theorem-like environments are not defined directly, but with the help of a `\newtheorem` declaration. That allows specifying the typeset environment title, e.g., "Lemma", and the counter to use to number the environments, e.g., they could be all numbered individually or one could number them using the same counter as some other theorem-like environment.

This was first augmented by the theorem package which implemented the idea of a `\theoremstyle`; this is now considered obsolete. Michael Downes from the AMS improved on these early ideas and wrote the amsthm package, which offered more functionality including a `\newtheoremstyle` declaration and for the document level a `\swapnumbers` and an `proof` environment. It also provided star-forms for `\newtheorem` (to define an unnumbered environment) and allowed to use star-forms of the theorem-like environments to suppress numbering on an individual instance in the document.

This new implementation based on templates, is supposed to cover the functionality of amsthm including it declarations so that documents that use amsthm explicitly or implicitly via their class should continue to work seamlessly.

For other packages that provide theorem-like environments we have to see if they could be easily remodeled using the new implementation or if there is a need for extended templates.

Assuming declarations such as

```
% \swapnumbers               % <- commented out
\theoremstyle{definition}
\newtheorem{axiom}[def]{Axiom}
```

in a document, then the following instances of type `blockenv` and `captionedtext` are declared by `\newtheorem`.

3.9.1 The `blockenv` instances they use

Given the above input `\newtheorem` defines the following `blockenv` instance:

```
\DeclareInstance{blockenv}{axiom}{std}
{
   name                = theorem-like
  ,tag-name            = \UseStructureName{block/theorem-like}
  ,tagging-recipe      = standalone
  ,transparent-level   = true
  ,block-instance:e    = thm-
                         \IfInstanceExistsTF{block}
                            { thm-definition-1 }
                            { definition } { plain }
  ,inner-instance-type = captionedtext
```

```
  ,inner-instance      = axiom
  ,para-instance       = justify
}
```

The setting for `block-instance` means that it checks if a `block` instance with the name `thm-definition-1` exists. If so then the value `thm-definition` is used, otherwise `thm-plain` is used which is always defined, i.e., if the theoremstyle does not specify any special vertical spacing the `block` instance from the `plain` style is reused.

What varies from `blockenv` instance to instance are the values for `block-instance` and `inner-instance`.

We use `<theorem-like>` as the structure name and role-map it to a `<Sect>` because that can hold a `<Caption>`.

3.9.2 The `captionedtext` instances they use

The instance of type `captionedtext` is also defined by `\newtheorem` and in this case it looks like this:

```
\DeclareInstance{captionedtext}{axiom}{thmlike}
{
  ,counter = def
  ,title   = Axiom       % <-- that the title provided to \newtheorem
  ,style   = definition  % <-- that's the used \theoremstyle
}
```

If we uncomment the `\swapnumbers` line in the example above then we get

```
  ,style   = definition-swap
```

in the `captionedtext` instance instead.

3.9.3 The `thmstyle` instances they use

New theorem styles can be declared with `\newtheoremstyle` which then generates an instance of type `thmstyle`. Alternatively, it is, of course, possible to declare the instances directly (which gives you a bit more flexibility). A few such styles are predeclared, matching what is offered by amsthm. These are shown below.

`thmstyle plain` (*inst.*) The main style used for many theorem-like environments, i.e., the one you get if no special `\theoremstyle` has been specified.

```
\DeclareInstance{thmstyle}{plain}{std}
  {
    ,caption-placement = unchained
    ,numbered          = true
    ,space             = \
    ,punct             = .
    ,before-hspace     = 0pt
    ,after-hspace      = 5pt plus 1pt minus 1pt
    ,order             = {title, space, number, punct, space, note}
    ,caption-decls     = \bfseries
    ,title-format      = #1
    ,number-format     = #1
    ,punct-format      = #1
```

```
    ,note-format       = (#1)
    ,body-decls        = \itshape
  }
```

`thmstyle remark` (*inst.*) The `remark` is like `plain` with two changes:

```
\DeclareInstanceCopy{thmstyle}{remark}{plain}
\EditInstance{thmstyle}{remark}
{
  ,caption-decls = \itshape
  ,body-decls    = \normalfont
}
```

`thmstyle definition` (*inst.*) The `definition` is like `plain` with only a difference in the font used for the body:

```
\DeclareInstanceCopy{thmstyle}{definition}{plain}
\EditInstance{thmstyle}{definition}
{
  ,body-decls = \normalfont
}
```

`thmstyle legacy2e` (*inst.*) Vanilla LaTeX 2$_\varepsilon$ (without amsthm loaded) had a slightly different default. We provide this under the name `legacy2e`. It doesn't use a punctuation after the number and it has slightly different vertical spacing (defined by `thm-legacy2e-1` below).
Thus, to reprocess an old document for tagging that uses `\newtheorem` without loading amsthm one has to set `\theoremstyle{legacy2e}` to avoid layout changes. How such a compatibility setting is automated is not yet decided.

```
\DeclareInstanceCopy{thmstyle}{legacy2e}{plain}
\EditInstance{thmstyle}{legacy2e}{ punct = }
```

3.9.4 The `block` instances they use

`block thm-plain-1` (*inst.*)
`block thm-plain-2` (*inst.*) Theorems do not support nesting, so in theory we have only one to set up. There are, however, documents that put theorem-like environments inside of lists or other block environments. While that is in most case somewhat dubious, it can make sense, for example, in `description` lists. So we support it by providing `thm-plain` instances for levels 1 and 2. If somebody really nests them further down, then more such instances need to be declared.
The LaTeX default reused the general value of `\parindent` and `\parskip` and, of course, they start at the outer margin.

```
\DeclareInstance{block}{thm-plain-1}{std}
{
  ,begin-extra-vspace = 0pt
  ,left-margin        = 0pt
  ,para-indent        = \parindent
  ,para-vspace        = \parskip
}
\DeclareInstanceCopy{block}{thm-plain-2}{thm-plain-1}
```

`block thm-remark-1` (*inst.*)
`block thm-remark-2` (*inst.*) The `\thmstyle` for "remarks" is defined by amsthm to use less vertical spacing. It therefore needs its own `block` instance.

```
\DeclareInstance{block}{thm-remark-1}{std}
{
  ,begin-vspace       = 0.5\topsep
```

```
  ,begin-extra-vspace = 0pt
  ,left-margin        = 0pt
  ,para-indent        = \parindent
  ,para-vspace        = \parskip
}
\DeclareInstanceCopy{block}{thm-remark-2}{thm-remark-1}
```

`block thm-legacy2e-1` (*inst.*)
`block thm-legacy2e-2` (*inst.*)

These are like the `plain` ones but without resetting `begin-extra-vspace` to zero.

```
\DeclareInstance{block}{thm-legacy2e-1}{std}
{
  ,left-margin        = 0pt
  ,para-indent        = \parindent
  ,para-vspace        = \parskip
}
\DeclareInstanceCopy{block}{thm-legacy2e-2}{thm-legacy2e-1}
```

3.10 The `proof` environment (from amsthm)

`proof` (*env.*)

The `proof` environment expects one optional argument holding an alternative title for the proof. We parse this optional argument as an implicit key/value argument, so that it is possible to interpret it either as the value for the key `note` or as a key/value list that holds special key settings for this particular environment instance. The result is analyzed by `\ParseLaTeXeTheoremlike` which then calls a `blockenv` instance with the name `proof`.

In addition we have to set up handling of QED symbols using `\pushQED` and `\popQED` using the logic already defined in amsthm. Details on all this is given in the code section of this module but normally this top-level declaration doesn't require any changes.

```
\NewDocumentEnvironment{proof}{ ={note}o }
   { \pushQED{\qed}%
     \ParseLaTeXeTheoremlike {proof} \BooleanTrue {#1} }
   { \popQED \BlockEnvEnd }
```

`blockenv proof` (*inst.*)

A proof uses its own `proofblock` instance of type `block` for vertical spacing. As the proof has a heading we use a `captionedtext` instance with name `proof` as the inner instance and the paragraphs of the proof are justified.

```
 \DeclareInstance{blockenv}{proof}{std}
{
   name                = proof
  ,tag-name            = \UseStructureName{block/proof}
  ,tag-attr-class      =
  ,tagging-recipe      = standalone
  ,inner-level-counter =
  ,transparent-level   = true
  ,legacy-code         =
  ,block-instance      = proof
  ,inner-instance-type = captionedtext
  ,inner-instance      = proof
  ,para-instance       = justify
}
```

`captionedtext proof` (*inst.*) We use a special `captionedtext` template to set up the proof because proofs are not numbered and the argument to a proof environment has a somewhat different semantic meaning than that of theorem-like environments.

```
\DeclareInstance{captionedtext}{proof}{proof}
  {
    ,title         = Proof
    ,punct         = .
    ,before-hspace = 0pt
    ,after-hspace  = 5pt plus 1pt minus 1pt
    ,caption-decls = \itshape
    ,title-format  = #1
    ,punct-format  = #1
    ,body-decls    = \normalfont
  }
```

3.10.1 Block instances for the proofs

`block proof-1` (*inst.*)
`block proof-2` (*inst.*) Blocks for proofs are pretty normal (the values are taken from the amsthm implementation):

```
\DeclareInstance{block}{proof-1}{std}
{
  ,begin-vspace       = 6pt plus 6pt
  ,left-margin        = 0pt
  ,para-indent        = \parindent
  ,para-vspace        = \parskip
}
\DeclareInstanceCopy{block}{proof-2}{proof-1}
```

4 Declaring `para` instances

Display block environments often require special paragraph settings and therefore have a `para-instance` key to specify and appropriate instance. Here are the standard instances that are predefined for this purpose.

`para justify` (*inst.*) Justifying is exactly what the default values do, so the instance hasn't any special setup.

```
\DeclareInstance{para}{justify}{std}
{
%  ,para-attr-class       = justify
%  ,para-indent           = \parindent
%  ,begin-hspace          = 0pt
%  ,left-hspace           = \z@skip
%  ,right-hspace          = \z@skip
%  ,end-hspace            = \@flushglue
%  ,final-hyphen-demerits =  5000
%  ,newline-cmd           = \@normalcr
}
```

`para center` (*inst.*) Centering a paragraph means putting stretchable glue on both sides.

```
\DeclareInstance{para}{center}{std}
{
  ,para-attr-class        = center
  ,para-indent           = 0pt
```

```
%  ,begin-hspace          = 0pt
  ,left-hspace           = \@flushglue
  ,right-hspace          = \@flushglue
  ,end-hspace            = \z@skip
  ,final-hyphen-demerits = 0
  ,newline-cmd           = \@centercr
}
```

`para raggedright` (*inst.*) This is the plain TEX version of ragged right, which basically means no hyphenation unless a word is truly longer than a line. This implements `flushleft`.

```
\DeclareInstance{para}{raggedright}{std}
{
  ,para-attr-class       = raggedright
  ,para-indent           = 0pt
%  ,begin-hspace          = 0pt
  ,left-hspace           = \z@skip
  ,right-hspace          = \@flushglue
  ,end-hspace            = \z@skip
  ,final-hyphen-demerits = 0
  ,newline-cmd           = \@centercr
}
```

`para raggedleft` (*inst.*) This here is for `flushright`.

```
\DeclareInstance{para}{raggedleft}{std}
{
  ,para-attr-class       = raggedleft
  ,para-indent           = 0pt
%  ,begin-hspace          = 0pt
  ,left-hspace           = \@flushglue
  ,right-hspace          = \z@skip
  ,end-hspace            = \z@skip
  ,final-hyphen-demerits = 0
  ,newline-cmd           = \@centercr
}
```

this should be moved elsewhere

Here are the attribute definitions used in the `para-attr-class` in the above instances:

```
\tagpdfsetup
  {
    ,role/new-attribute = {justify}    {/O /Layout /TextAlign/Justify}
    ,role/new-attribute = {center}     {/O /Layout /TextAlign/Center}
    ,role/new-attribute = {raggedright}{/O /Layout /TextAlign/Start}
    ,role/new-attribute = {raggedleft} {/O /Layout /TextAlign/End}
  }
```

`\centering` `\raggedleft` `\raggedright` `\justifying` These instances are also used to implement declarations for direct use in documents or in user definitions.

```
\DeclareRobustCommand\centering  {\UseInstance{para}{center}{}}
\DeclareRobustCommand\raggedleft {\UseInstance{para}{raggedleft}{}}
\DeclareRobustCommand\raggedright{\UseInstance{para}{raggedright}{}}
\DeclareRobustCommand\justifying {\UseInstance{para}{justify}{}}
```

LATEX's default is to typeset paragraphs justified.

```
\justifying
```

(*End of definition for* `\centering` *and others.*)

`para verse` (*inst.*) For the `verse` environment we use a special para instance. If the right hand side should be ragged then a different `right-hspace` is needed.

```
\DeclareInstance{para}{verse}{std}
{
  para-attr-class       = justify ,
  para-indent           = 0pt ,
  begin-hspace          = -1.5em ,
  left-hspace           =  1.5em ,
  right-hspace          = 0pt ,
  end-hspace            = \@flushglue ,
  final-hyphen-demerits = 0 ,
  newline-cmd           = \@centercr ,
}
```

⟨/class-code⟩

5 Advice on adjusting the layout of standard block environments

to document

6 Tagging support

6.1 Paragraph tags

Paragraphs in LaTeX can be nested, e.g., you can have a paragraph containing a display quote, which in turn consists of more than one (sub)paragraph, followed by some more text which all belongs to the same outer paragraph.

In the PDF model and in the HTML model that is not supported — a limitation that conflicts with real life, given that such constructs are quite normal in spoken and written language.

The approach we take to resolve this is to model such "big" paragraphs with a structure named `<text-unit>` and use `<text>` (role-mapped to `<P>`) only for (portions of) the actual paragraph text in a way that the `<text>`s are not nested. As a result we have for a simple paragraph the structures

```
<text>
  <text>
    The paragraph text …
  </text>
</text>
```

The `<text-unit>` structure is role-mapped to `<Part>` or possibly to `<Div>` so we get a valid PDF, but processors who care can identify the complete paragraphs by looking for `<text-unit>` tags.

In the case of an element, such as a display quote or a display list inside the paragraph, we then have

```
<text-unit>
  <text>
    The paragraph text before the display element …
  </text>
  <display element structure>
    Content of the display structure possibly involving inner <text-unit> tags
  </display element structure>
  <text>
    … continuing the outer paragraph text
  </text>
</text-unit>
```

In other words such a display block is always embedded in a `<text-unit>` structure, possibly preceded by a `<text>…</text>` block and possibly followed by one, though both such blocks are optional.

Thus an `itemize` environment that has some introductory text but no text immediately following the list would be tagged as follows:

```
<text-unit>
  <text>
    The intro text for the itemize environment …
  </text>
  <itemize>
    <LI>
      <itemlabel> label </itemlabel>
      <itembody>
        The text of the first item involving <text-unit> as necessary …
      </itembody>
    </LI>
    <LI>
      The second item …
    </LI>
    … further items …
  </itemize>
</text-unit>
```

The `<itemize>` is roll-mapped to `<L>`.

For some display blocks, such as centered text, we use a simpler strategy. Such blocks still ensure that they are inside a `<text-unit>` structure but their body uses simple `<text>` blocks and not `<text-unit><text>` inside, e.g., the input

```
This is a paragraph with some
\begin{center}
   centered lines

   with a paragraph break between them
\end{center}
followed by some more text.
```

will be tagged as follows:

```
<text-unit>
  <text>
    This is a paragraph with some
  </text>
  <text /O /Layout /TextAlign/Center>
    centered lines
  </text>
  <text /O /Layout /TextAlign/Center>
    with a paragraph break between them
  </text>
  <text>
    followed by some more text.
</text-unit>
```

The text-unit structures are added by using the tagging sockets `para/semantic/begin` and `para/semantic/end` declared in `lttagging.dtx`. They can be disabled by assigning these sockets the plug `noop`.

6.1.1 Tagging recipes

There are a number of different tagging recipes that implement different tagging approaches. They are selected through the `tagging-recipe` of the `blockenv` template. Currently the following values are implemented:

`standalone` This recipe does the following:

- Ensure that the `blockenv` is not inside a `<text-unit>` structure. If necessary, close the open one (and any open `<text>` structure).
- Text inside the body of the environment start with `<text-unit><text>` unless the key `tagging-suppress-paras` is set to `true` (which is most likely the wrong thing to do because we then get just `<text>` as the structure).
- At the end of the environment close `</text>` and possibly an inner `</text-unit>` if open.
- Finally, ensure that after the environment a new `<text-unit>` is started, if appropriate, e.g., if text is following.

`basic` This recipe does the following:

- Ensure that the `blockenv` is inside a `<text-unit>` structure, if necessary, start one.
- If inside a `<text-unit><text>`, then close the `</text>` but leave the `<text-unit>` open.
- Text inside the body of the environment start with `<text-unit><text>` if `tagging-suppress-paras` is set to `false`, otherwise just with `<text>`.
- At the end of the environment close `</text>` and possibly an inner `</text-unit>` if open.

- Then look if the environment is followed by an empty line (`\par`). If so, close the outer `</text-unit>` and start any following text with `<text-unit><text>`. Otherwise, don't and following text restarts with a just a `<text>` (and no paragraph indentation)

standard This recipe is like the `basic` one as far as handling `<text-unit>` and `<text>` is concerned. In addition

- it starts an inner tagging structure (i.e., which is therefore a child of the outer `<text-unit>`).
- By default this structure is a `<Div>` unless overwritten by the key `tag-name`. If that key is used, a suitable role-map needs to be provided for the name given.
- At the end of the environment that inner structure is closed again so that we are back on the `<text-unit>` level from the outside.
- Then the lookahead for an empty line is done as described previously.

list This recipe is like the `standard` one except that

- the inner structure is a list (`<L>`).
- Furthermore everything is set up so that we have list items (`<LI>`) with suitable substructures (`<itemlabel>` for the item labels and `<itembody>` for the item bodies).
- If the key `tag-name` is specified, this is used as the tag name for the whole list instead of `<L>`. Of course, it should then have a suitable rolemap.
- If the key `tag-attr-class` is specified then this is used as the class attribute. Again, this requires a suitable setup on the outside.
- At the end of the environment the `</itembody>`, `</LI>`, and `</L>` (or the tag name used) are closed.
- Then the lookahead for an empty line is done as described previously.

7 Tracing and debugging

`\DebugBlocksOn`
`\DebugBlocksOff`
`\block_debug_on:`
`\block_debug_off:`

These commands enable/disable debugging messages for blocks. They also enable/disable debugging of templates (e.g., call `\DebugTemplatesOn` or `\DebugTemplatesOff`).

cleanup

The data that is produced is rather verbose and largely guided (so far) by what seemed helpful while developing the code. This needs some cleanup at a later stage. At the moment, if you have the following simple document

```
\DocumentMetadata{tagging=on, lang=en}

\documentclass{article}

\DebugBlocksOn

\begin{document}
```

```
8           \begin{itemize}[item-vspace=3pt]
9             \item             A normal item
10            \item[\textbf{+}] A special item
11          \end{itemize}
12        \end{document}
```

then you will get the following information on the screen and in the `.log` file:

```
[Template] ==> Use 'blockenv' instance: itemize on input line 8
[Template] ==>   template: 'std'; arguments: |item-vspace=3pt|\BooleanFalse |\NoValue |\NoValue |
[Template] ==> Use 'block' instance: std-list-1 on input line 8
[Template] ==>   template: 'std'; argument: |item-vspace={3pt}|
[Blocks] ==> @endpe=false on input line 8
[Template] ==> Use 'list' instance: itemize-1 on input line 8
[Template] ==>   template: 'std'; arguments: ||\BooleanFalse |\NoValue |\NoValue |
[Blocks] ==> Set first block everypar on input line 8
[Blocks] ==> template:list:std end

[Template] ==> Use 'item' instance: basic on input line 9
[Template] ==>   template: 'std'; argument: ||
[Blocks] ==> Set item block everypar on input line 9
[Blocks] ==> ... in item block everypar on input line 9
[Blocks] ==> increment P on input line 9
[Blocks] ==> Set noop block everypar on input line 9

[Template] ==> Use 'item' instance: basic on input line 10
[Template] ==>   template: 'std'; argument: |label={\textbf {+}}|
[Blocks] ==> item with optional
[Blocks] ==> Set item block everypar on input line 10
[Blocks] ==> ... in item block everypar on input line 10
[Blocks] ==> increment P on input line 10
[Blocks] ==> Set noop block everypar on input line 10

[Blocks] ==> blockenv common ending on input line 11

[Blocks] ==> flattened=false on input line 12
[Blocks] ==> Structure-end text-unit after displayblock on input line 12
```

8 New and redefined kernel command

`\SimpleBlockEnv`
`\BlockEnv`
`\BlockEnvEnd`
`\g_block_nesting_depth_int`

to be documented

`\legacyverbatimsetup`
`\legacyallttsetup`
`\legacylistsetup`

to be documented

`\@setupverbinvisiblespace`

A counterpart definition to the kernel command `\@setupverbinvisiblespace`, needed as we need to handle real space chars in verbatim.

`\newtheorem`
`\newtheoremstyle`

Reimplemented to fit the template approach. `\newtheoremstyle` was defined by amsthm.

`\@nthm`
`\@xnthm`
`\@ynthm`
`\@thm`
`\@xthm`
`\@ythm`
`\@othm`
`\@begintheorem`
`\@opargbegintheorem`
`\@endtheorem`

These are no longer used (to be removed).

`\item`
`\@itemlabel`

The `\item` is redefined.

`\c@maxblocklevels`

A counter to increase or decrease the number of supported level. If increased, one needs to supply additional level instances.

`\begin`

The `\begin` is slightly redefined to handle `\@doendpe` better. TODO: move to kernel

`\@doendpe`

The original LaTeX 2_ε command is augmented to allow for tagging.

`\para_end:`

TODO: consider name, document

`para/begin`

The para/begin hook is enhanced to support list ends

9 The Implementation

523 ⟨*package-start⟩

524 ⟨@@=block⟩

```
525 \ProvidesPackage {latex-lab-testphase-block}
526                   [\ltlabblockdate\space v\ltlabblockversion\space
527                             blockenv implementation]
```

9.1 Candidates for kernel changes

General kernel changes, also loaded by the sec and toc code.

```
\RequirePackage{latex-lab-kernel-changes}
```

For testing we temporarily load it here (it has to come before the definition of `\DebugBlocksOff` below:

```
\RequirePackage{latex-lab-testphase-context}
\ExplSyntaxOn
```

9.1.1 Augmented `\SetKnownTemplateKeys`

`\SetKnownTemplateKeys` A key/val list passed to `\SetKnownTemplateKeys` can either be empty (in which we do not want to start up the parsing machinery) or it could be `\NoValue` in which we do not want to do that either. The latter can happen, for example, with `verbatim` where we define the optional argument with `={legacy-code} !o` so that people can write `\begin{verbatim}[\small]` a syntax promoted by the TUGboat class.

```
\cs_set_protected:Npn \SetKnownTemplateKeys #1#2#3
  {
```

An "empty" argument (or rather one that is empty after one expansion) is most likely the case that happens most often so we test for this first.

```
    \tl_if_empty:oTF {#3}
      {
        \tl_set_eq:NN  \UnusedTemplateKeys \c_empty_tl
      }
      {
        \tl_if_novalue:nTF {#3}
          {
            \tl_set_eq:NN  \UnusedTemplateKeys \c_empty_tl
          }
          {
            \keys_set_known:noN { template / #1 / #2 } {#3}
                                \UnusedTemplateKeys
          }
      }
  }
```

(*End of definition for* `\SetKnownTemplateKeys`*. This function is documented on page* ??.)

`\SetTemplateKeys` Same kind of extension for `\SetTemplateKeys`:

```
\cs_set_protected:Npn \SetTemplateKeys #1#2#3
  {
    \tl_if_empty:oF {#3}
      {
        \tl_if_novalue:nF {#3}
          {
            \keys_set:no { template / #1 / #2 } {#3}
          }
      }
  }
```

(*End of definition for* `\SetTemplateKeys`*. This function is documented on page* ??.)

9.1.2 Tracing templates and instances

\template_debug_typeout:n

I guess that tracing macro is needed in several modules, so should become public (or at least kernel).

```
\cs_new_protected:Npn \template_debug_typeout:n { \__template_debug_typeout:n }
```

(*End of definition for* `\template_debug_typeout:n`. *This function is documented on page* ??.)

9.1.3 Handling \par after the end of the list

An empty line (or a `\par`) after a list has semantic meaning as it defines whether then following text is logically within the same paragraph as the list (no empty line) or whether it starts a new paragraph and the paragraph containing the list ends at the end of the list (empty line after the list). This is handled by LaTeX using a legacy flag called `@endpe` and set of commands inside the generic `\end` (calling `\@doendpe`) and as part of the list environments identifying themselves as "paragraph ending environments" (by setting this flag).

For the reimplementation of the list environments including support of tagging we need to augment that mechanism slightly and add some kernel hook(s) to add the tagging code if needed.

\@doendpe

The original LaTeX 2_ε command is augmented to allow for tagging. TODO: use sockets for this and move to the kernel eventually.

```
\def\@doendpe{\@endpetrue
  \def\par
    {
```

If we are processing a `$$` math display and we encounter a real `\par` after it, we need to add a `\parskip` when tagging is done, because the one added by TeX is always canceled by the processing in `\__math_tag_dollardollar_display_end:` in that case. This is signaled by the global legacy switch `@domathendpe` which is set to true in that case. Once the skip is applied we set it to false. If there is no `\par` at all, it will be reset in `\everypar` when the next paragraph starts.

```
      \if@domathendpe
        \skip_vertical:n { \tex_parskip:D }
        \@domathendpefalse
      \fi
      \@restorepar
      \clubpenalty\@clubpenalty
```

At this point we add the tagging code that closes an open `<text-unit>`, `<text>` tag combination, if necessary:

```
      \tag_socket_use:n {@doendpe}
```

The standard `\par` command (`\par_end:`) acts on `@endpe` and attempts to close a still open `<text-unit>`s and this would be wrong if it was already closed above. So we have to reset the switch to false first.

```
      \@endpefalse
      \everypar{}
      \par
    }
  \everypar{{\setbox\z@\lastbox}
            \everypar{}
            \@endpefalse
```

Not sure what is faster: testing for the status of the switch or setting it unconditionally to false (globally), probably roughly the same, so we set it always:

```
%            \if@domathendpe
             \@domathendpefalse
%            \fi
  }
}
```

(*End of definition for* `\@doendpe`. *This function is documented on page 38.*)

`tagsupport/@doendpe` (*socket*) The socket used in the `\@doendpe` TODO: if this goes into the kernel, the name should probably be different.

```
\socket_if_exist:nF{ tagsupport/@doendpe }
  {
    \NewTaggingSocket {@doendpe}{0}
  }
```

`default` (*plug*) If a display block ends and is followed by a blank line we have to end the enclosing paragraph tagging structure.

```
\NewTaggingSocketPlug {@doendpe}{default}
  {
   \bool_if:NT \l__tag_para_bool
     {
```

Given that restoring `\par` through the legacy LaTeX 2_ε method can take a few iterations (for example, in case of nested lists, e.g., `...\end{itemize} \item ...\par` it can happen that the socket code is called while `@endpe` is already handled and then we should not attempt to close a `<text-unit>` structure). So we need to check for this.

```
       \legacy_if:nT { @endpe }
         {
```

If the display block currently ending was "flattened" (i.e., uses simplified paragraphs that are not tagged by a combination of `<text-unit>` followed by `<text>`, but simply with a `<text>`), then we don't have to do anything, because the `<text>` is already closed.

```
           \__block_debug_typeout:n
             { flattened= \bool_if:NTF
                            \l__tag_para_flattened_bool
                            {true}{false}
                  \on@line }
           \bool_if:NF \l__tag_para_flattened_bool
             {

               \UseTaggingSocket{para/semantic/end}
                {
                  \__block_debug_typeout:n{Structure-end~
                      \l__tag_para_main_tag_tl\space
                      after~ displayblock \on@line     }
                }
             }
         }
     }
  }
\AssignTaggingSocketPlug{@doendpe}{default}
```

\if@domathendpe
\@domathendpefalse
\@domathendpetrue

Signal that special paragraph handling after a math display is required.

```
\newif\if@domathendpe
\def\@domathendpefalse{\global\let\if@domathendpe\iffalse}
\def\@domathendpetrue {\global\let\if@domathendpe\iftrue}
```

(*End of definition for* `\if@domathendpe`, `\@domathendpefalse`, *and* `\@domathendpetrue`.)

There is a general bug in the para handling: when the output routine is triggered the current setting of `@endpe` affects what happens in the OR. but it shouldn't so we need to reset its value (which is global) and set it back after the OR has finished. This is what the following code does (final implementation should probably not involve a normal hook):

fix in kernel

```
\AddToHook{build/column/before}{%
  \if@endpe \@endpefalse \aftergroup\@endpetrue \fi
}
```

9.1.4 Other useful expl3 commands

This section collects expl3 commands that will be useful in the code here and possibly generally.

__block_skip_set_to_last:N
__block_skip_remove_last:

Set a skip register to the value of an immediately preceding skip or zero if there was none.

```
\cs_new_protected:Npn \__block_skip_set_to_last:N #1 {
  \skip_set:Nn #1 { \tex_lastskip:D }
}
```

Remove a skip previous skip if it is directly in front (not allowed in unrestricted vertical mode).

```
\cs_new_eq:NN \__block_skip_remove_last: \tex_unskip:D
```

(*End of definition for* `\__block_skip_set_to_last:N` *and* `\__block_skip_remove_last:`.)

check

Not sure this is still necessary (or even correct) after the move to `\NoValue`.

```
\cs_generate_variant:Nn \tl_if_novalue:nTF { o }
```

(*End of definition for* `\tl_if_novalue:oTF`. *This function is documented on page* **??**.)

9.2 Tracing and debugging interfaces

This follows the same convention as in other modules, but eventually that should be given a better implementation.

refactor at some stage

\g__block_debug_bool

Boolean to indicate if we want to get debugging info from commands and templates handling block displays.

```
\bool_new:N \g__block_debug_bool
```

(*End of definition for* `\g__block_debug_bool`.)

__block_debug:n
__block_debug_typeout:n

Put debugging info in the code, displayed or not displayed depending on the value in `\g__block_debug_bool`.

```
\cs_new_eq:NN \__block_debug:n \use_none:n
\cs_new_eq:NN \__block_debug_typeout:n \use_none:n
```

(*End of definition for* `\__block_debug:n` *and* `\__block_debug_typeout:n`.)

\block_debug_on:
\block_debug_off:
__block_debug_gset:

Changing the debugging status.

```
\cs_new_protected:Npn \block_debug_on:
  {
    \bool_gset_true:N \g__block_debug_bool
    \__block_debug_gset:
  }
\cs_new_protected:Npn \block_debug_off:
  {
    \bool_gset_false:N \g__block_debug_bool
    \__block_debug_gset:
  }
\cs_new_protected:Npn \__block_debug_gset:
  {
    \cs_gset_protected:Npx \__block_debug:n ##1
      { \bool_if:NT \g__block_debug_bool {##1} }
    \cs_gset_protected:Npx \__block_debug_typeout:n ##1
       { \bool_if:NT \g__block_debug_bool
         { \iow_term:x { ^^J [Blocks]~ ==>~ ##1} } }
  }
```

(*End of definition for* `\block_debug_on:`, `\block_debug_off:`, *and* `\__block_debug_gset:`. *These functions are documented on page 36.*)

\DebugBlocksOn
\DebugBlocksOff

If we are debugging blocks we also want to know about template instances, so we turn the debugging for templates as well (for now).

```
\cs_new_protected:Npn \DebugBlocksOn  { \block_debug_on: \template_debug_on: }
\cs_new_protected:Npn \DebugBlocksOff { \block_debug_off: \template_debug_off: }
\DebugBlocksOff
```

(*End of definition for* `\DebugBlocksOn` *and* `\DebugBlocksOff`. *These functions are documented on page 36.*)

\DebugSwitchesOn
\DebugSwitchesOff

This debugs the use of legacy switches (so perhaps better called \DebugLegacySwitchesOn) but so far it was just a quick debugging aid while I was trying to understand. It needs some further thoughts and is probably not necessary at all in the end.

```
\cs_new_protected:Npn \DebugSwitchesOn {
  \AddToHookWithArguments{cmd/legacy_if_gset_false:n/before}[debug]
                         {\typeout{[Switch]~==>~ ##1~=~false~(global)}}
  \AddToHookWithArguments{cmd/legacy_if_gset_true:n/before}[debug]
                         {\typeout{[Switch]~==>~ ##1~=~true~(global)}}
  \AddToHookWithArguments{cmd/legacy_if_set_false:n/before}[debug]
                         {\typeout{[Switch]~==>~ ##1~=~false}}
  \AddToHookWithArguments{cmd/legacy_if_set_true:n/before}[debug]
                         {\typeout{[Switch]~==>~ ##1~=~true}}
}
\cs_new_protected:Npn \DebugSwitchesOff {
  \RemoveFromHook{cmd/legacy_if_gset_false:n/before}[debug]
  \RemoveFromHook{cmd/legacy_if_gset_true:n/before}[debug]
  \RemoveFromHook{cmd/legacy_if_set_false:n/before}[debug]
  \RemoveFromHook{cmd/legacy_if_set_true:n/before}[debug]
}
%\DebugSwitchesOn
%\DebugSwitchesOff
```

(*End of definition for* `\DebugSwitchesOn` *and* `\DebugSwitchesOff`. *These functions are documented on page* ??.)

9.3 Template types and template interfaces

This section is devoted to the template interfaces, and the template code is covered later.

`blockenv` (*type*)
`list` (*type*)
`captionedtext` (*type*)
`thmstyle` (*type*)
`block` (*type*)
`item` (*type*)
`para` (*type*)

All template types expect a first key–value argument used to tweak template parameters at a specific point in the document for a single environment or command. The template types `blockenv`, `list`, `captionedtext`, and `thmstyle` take three more arguments which are a boolean for suppressing numbering, a possible caption, and a possible sub-caption.

```
\NewTemplateType{blockenv}{4}
\NewTemplateType{list}{4}
\NewTemplateType{captionedtext}{4}
\NewTemplateType{thmstyle}{4}

\NewTemplateType{block}{1}
\NewTemplateType{item}{1}
\NewTemplateType{para}{1}
```

`blockenv std` (*templ.*)

```
\DeclareTemplateInterface{blockenv}{std}{4}
{
  name                    : tokenlist ,
```

If not explicitly set then `tag-name` and `tag-attr-class` are set by the `tagging-recipe`. However, we have to default both to ⟨*empty*⟩ so that nested blocks do not inherit from the outer level.

```
  ,tag-name               : tokenlist =
  ,tag-attr-class         : tokenlist =
  ,tagging-recipe         : tokenlist = standard
  ,transparent-level      : boolean   = false
  ,legacy-code            : tokenlist =
  ,block-instance         : tokenlist = std-display
```

Paragraph instance is normally inherited so no default.

```
  ,para-instance          : tokenlist
  ,inner-level-counter    : tokenlist
  ,max-inner-levels       : tokenlist = 4
  ,inner-instance-type    : tokenlist =
  ,inner-instance         : tokenlist =
  ,tagging-suppress-paras : boolean = false
  ,final-code             : tokenlist = \ignorespaces
}
```

`block std` (*templ.*)

```
\DeclareTemplateInterface{block}{std}{1}
{
  ,begin-vspace           : skip = \topsep
  ,begin-extra-vspace     : skip = \partopsep
  ,begin-unchained-vspace : skip = .5\topsep
  ,para-vspace            : skip = \parskip
  ,end-vspace             : skip = \KeyValue{begin-vspace}
  ,end-extra-vspace       : skip = \KeyValue{begin-extra-vspace}
  ,item-vspace            : skip = \itemsep
  ,begin-penalty          : integer = \UseName{@beginparpenalty}
  ,end-penalty            : integer = \UseName{@endparpenalty}
  ,item-penalty           : integer = \UseName{@itempenalty}
```

```
  ,left-margin            : length = \leftmargin
  ,right-margin           : length = \rightmargin
  ,para-indent            : length = 0pt
}
```

`para std` (*templ.*)

```
\DeclareTemplateInterface{para}{std}{1}
{
  ,para-attr-class       : tokenlist = justify
  ,para-indent           : length = \parindent
  ,begin-hspace          : skip = 0pt
  ,left-hspace           : skip = 0pt
  ,right-hspace          : skip = 0pt
  ,end-hspace            : skip = \@flushglue
  ,fixed-word-spaces     : boolean = false
  ,final-hyphen-demerits : integer = 5000
  ,newline-cmd           : function(0) = \@normalcr
}
```

`list std` (*templ.*)

```
\DeclareTemplateInterface{list}{std}{4}
{
  ,counter         : tokenlist =
  ,item-label      : tokenlist =
  ,start           : integer = 1
  ,resume          : boolean = false
  ,item-instance   : instance{item} = basic
  ,item-vspace     : skip = \itemsep
  ,item-penalty    : integer = \UseName{@itempenalty}
  ,item-indent     : length = \itemindent
  ,label-width     : length = \labelwidth
  ,label-sep       : length = \labelsep
  ,legacy-support  : boolean = false
}
```

`item std` (*templ.*)

```
\DeclareTemplateInterface{item}{std}{1}
  {
    ,counter-label   : function{1} = \arabic{#1}
    ,counter-ref     : function{1} = \KeyValue{counter-label}
    ,label-ref       : function{1} = #1
    ,label-autoref   : function{1} = item~#1
    ,label-format    : function{1} = #1
    ,label-strut     : boolean = false
    ,label-align     : choice {left,center,right,parleft} = right
    ,label-boxed     : boolean = true
    ,next-line       : boolean = false     % <- review viz standalone below
    ,text-font       : tokenlist
    ,compatibility   : boolean = true
    ,label-placement : choice {chained,unchained,standalone} = chained ,
  }
```

`captionedtext thmlike` (*templ.*) The `captionedtext thmlike` template for theorem-like environments has only three keys because it delegates most of the work to the `thmstyle` template specified in the key `style`.

```
\DeclareTemplateInterface{captionedtext}{thmlike}{4}
{
  ,counter         : tokenlist =
  ,title           : tokenlist =      % <- bad name?
  ,style           : instance{thmstyle} = plain
}
```

`captionedtext proof` (*templ.*) In contrast, the `captionedtext proof` template implements all of the `proof` environment without any delegation and therefore shows several keys for customizing the layout (similar to those seen with `thmstyle std`).

```
\DeclareTemplateInterface{captionedtext}{proof}{4}
{
  ,title             : tokenlist = Proof
  ,punct             : tokenlist = .
  ,caption-placement : choice {chained,unchained,standalone} = unchained
  ,before-hspace     : skip = 0pt
  ,after-hspace      : skip = 5pt
  ,caption-decls     : tokenlist =
  ,title-format      : function{1} = #1
  ,punct-format      : function{1} = #1
  ,body-decls        : tokenlist =
}
```

`thmstyle std` (*templ.*)

```
\DeclareTemplateInterface{thmstyle}{std}{4}
{
  ,numbered          : boolean = true
  ,space             : tokenlist = \    % <- bad name?
  ,punct             : tokenlist = .
  ,caption-placement : choice {chained,unchained,standalone} = unchained
  ,before-hspace     : skip = 0pt
  ,after-hspace      : skip = 5pt
  ,order             : commalist = { title, space, number, punct, space, note }
  ,caption-decls     : tokenlist =
  ,title-format      : function{1} = #1
  ,number-format     : function{1} = #1
  ,punct-format      : function{1} = #1
  ,note-format       : function{1} = (#1)
  ,body-decls        : tokenlist   =
}
```

9.4 Implementation of templates

9.4.1 Some notes on the LaTeX 2ε legacy switches

LaTeX 2ε used a number of switches to manage its list environments and everything that was based on them.

For the reimplementation I made some notes about the original usage and how this got changed (while keeping the names for now).

Some of these switches really need to keep their names, e.g., `@nobreak` or `minipage`, because they are used all over the place. Others can probably replaced with L3 booleans which makes things faster and cleaner, but for now I kept them too.

9.4.1.1 Original usage:

```
%
% @newlist (global):  signal that we are at the start of a list
%
% -> true   at the start of a list before the first item when control
%           is returned to document
% -> false  in everypar setting the first item
% -> false  at end of list if still true (after generating an error)
%
% -> tests: at list start setting @noparitemtrue and @noparlisttrue
%
%
% @inlabel (global): signaling that some item label waits to be typeset
%
% -> true   in \@item
% -> false  at list end
% -> false  in everypar after label has been typeset
% -> false  in \newpage after \leavemode to typeset item label
%           (probably not needed)
%
% -> tests: at list start setting @noparitemtrue and @noparlisttrue
% -> tests: at list end to ensure that dangling label is typeset
% -> tests: in \@item to output a dangling item label by switching to hmode
% -> tests: in \everypar to output a dangling item label
% -> tests: in \newpage to output a dangling item label before the page is ended
% -> tests: in tagging hook {para/begin}{kernel}
%
%
% @noparlist (local):
%
% -> true   at start of list if already @inlabel=true
% -> false  at start of list otherwise
%
% -> tests: in \endtrivlist. If true suppress vertical spacing after the list
%
%
% @noparitem (local):
%
% -> true   at start of list if already @inlabel=true
% -> false
%
```

9.4.1.2 Repurpose:

```
%
% Interpret legacy switches as follows (keeping the names for now)
%
% @newlist   ->  signals that we are at the start of a new block with a caption or
%                at the start of a list block expecting an item next
%
```

```
% In other words this is now really start of a block
% with inner structure.
%
% @noparlist ->  signals that we are on a new block with @inlabel already true, i.e.,
% and this placement should happen horizontally
%
% @inlabel   ->  Signals that we have at least one item or caption waiting to be typeset
% inside the label box
%
% @noparitem ->  dropped (handled directly)
%
```

9.4.2 Implementation of `blockenv` templates

So far there is only one, but who knows … — however, the majority will be vertically oriented blocks, so we make this the `std`.

`blockenv std` (*templ.*)

```
\DeclareTemplateCode{blockenv}{std}{4}
{
   name                   = \l__block_env_name_tl
  ,tag-name               = \l__block_tag_name_tl
  ,tag-attr-class         = \l__block_tag_class_tl
  ,tagging-recipe         = \l__block_tagging_recipe_tl
  ,transparent-level      = \l__block_transparent_level_bool
  ,legacy-code            = \l__block_legacy_code_tl
  ,block-instance         = \l__block_block_instance_tl
  ,para-instance          = \l__block_para_instance_tl
  ,tagging-suppress-paras = \l__tag_para_flattened_bool
  ,inner-level-counter    = \l__block_inner_level_counter_tl
  ,max-inner-levels       = \l__block_max_inner_levels_tl
  ,inner-instance-type    = \l__block_inner_instance_type_tl
  ,inner-instance         = \l__block_inner_instance_tl
  ,final-code             = \l__block_final_code_tl
}
{
  \template_debug_typeout:n{~\space template:~ 'std;~
                            arguments:~ \exp_not:n{|#1|#2|#3|#4|}}
  \UseHook{blockenv}
```

We first evaluate the key list passed from the document (if any). All known keys are used, the remainder is stored in `\UnusedTemplateKeys` to be passed to any inner instances below.

```
  \SetKnownTemplateKeys{blockenv}{std}{#1}
```

We need to know later if we have nested `blockenv`s inside a flattened environment. Whenever we start a new blockenv we increment `\l__tag_block_flattened_level_int` if it is already different from zero. If it is zero we increment it if flattening is requested. Thus a value of `0` means no flattening requested so far and `1` means this is the first blockenv requesting flattening. In either case we have to make sure that the blockenv is surrounded by a `<text-unit>` tag, while for any value above `1` we have to omit the `<text-unit>`.

```
  \int_compare:nNnTF \l__tag_block_flattened_level_int > 0
```

```
      {
        \int_incr:N \l__tag_block_flattened_level_int
      }
      {
        \bool_if:NT \l__tag_para_flattened_bool
             {
               \int_incr:N \l__tag_block_flattened_level_int
             }
      }
  \tl_if_empty:NF \l__block_inner_level_counter_tl
     {
       \int_compare:nNnTF  \l__block_inner_level_counter_tl >
                           { \l__block_max_inner_levels_tl - 1 }
           { \@toodeep }
           { \int_incr:N \l__block_inner_level_counter_tl }  % not clean "o"?
     }
```

Legacy defaults are only roped in if the list level changes. For display blocks that remain on the same level the current values are kept.

```
  \int_compare:nNnTF  \g_block_nesting_depth_int >
                      { \c@maxblocklevels - 1 }
     { \@toodeep }
     {
       \int_gincr:N \g_block_nesting_depth_int
```

If there are no legacy defaults for that level then the next line does nothing, i.e., the current values (from the last level) become the defaults for the next.
If have a transparent level (e.g., something like a `center` environment) we omit setting the legacy defaults, because that is the way LaTeX 2_ε lists worked as well.

```
       \bool_if:NF \l__block_transparent_level_bool
         {
           \use:c { @list
             \int_to_roman:n { \g_block_nesting_depth_int } }
         }
     }
```

If we are doing tagging we load one of the available recipes for tagging, which alters various kernel hooks to add appropriate tagging structures.

```
     \UseTaggingSocket{block/recipe}{\l__block_tagging_recipe_tl}
```

The default for `list` environments is that they have an empty label and are not numbered (something that is then overwritten by the setup of a specific list). We ensure this here even for non-lists, because we need a defined state that then can be overwritten by the legacy setup code for the `list` environment in `\l__block_legacy_code_tl`. This is needed in case lists are nested as they otherwise would inherit outer values (and suddenly an `itemize` would start incrementing an outer `enumerate` counter, etc.

```
  \tl_clear:N \@itemlabel
  \tl_clear:N \@listctr
  \legacy_if_set_false:n { @nmbrlist }
```

Then run the legacy setup code if any is given in the instance.

```
  \l__block_legacy_code_tl
```

Next call a block instance at the appropriate level passing it any remaining key/value from the optional document-level argument (i.e., those now stored in `\UnusedTemplateKeys`).

```
\exp_args:Nee \UseInstance{block}
              { \l__block_block_instance_tl - \int_use:N
                \g_block_nesting_depth_int }
              \UnusedTemplateKeys
```

After this instance has been processed, any remaining unused keys are stored in `\UnusedTemplateKeys` and we can make use of this data later as long as we do not call another instance that also does unused key processing and overwrites it. But this is what happens below, so we better save its current value for now.

```
\tl_set_eq:NN \l__block_unused_blockenv_keys_tl \UnusedTemplateKeys
```

After the block instance call the para and then inner (list) instance if either or both are specified (which may not be the case).

```
\tl_if_empty:NF \l__block_para_instance_tl
  {
```

For now we don't offer to alter instance parameters here so we pass an empty argument.

```
    \exp_args:Ne \UseInstance{para}{ \l__block_para_instance_tl } {}
  }
```

The inner instance may have its own levels or none depending on which the instance name differs. Again we pass it the optional key/value list.

```
\tl_if_empty:NF \l__block_inner_instance_tl
  {
```

We expand the first two arguments so that we get proper names for template type and instance, because `\UseInstance` is not doing that for us in the right way.

```
    \exp_args:Nee
    \UseInstance{ \l__block_inner_instance_type_tl }
                { \l__block_inner_instance_tl
                  \tl_if_empty:NF \l__block_inner_level_counter_tl
                     %   not clean use "o"?
                     { - \int_use:N \l__block_inner_level_counter_tl }
                }
                \l__block_unused_blockenv_keys_tl
                  #2      % <-- \BooleanTrue or False
                { #3 }    % <-- \NoValue or content
                { #4 }    % <-- \NoValue or content
```

Again the instance may have processed a few keys from the so far unused keys, so we update `\l__block_unused_blockenv_keys_tl` to match the new reality.

```
    \tl_set_eq:NN \l__block_unused_blockenv_keys_tl \UnusedTemplateKeys
  }
```

At this point, the `\l__block_unused_blockenv_keys_tl` token list should either be empty or it should contain only keys that are suitable for the item template, but right now there is no code to test that can test the latter; it would help probably if we have an interface for this.

fix

For now we handle that when the first item is encountered, but that isn't really clean.

```
%  \tl_if_empty:NF \l__block_unused_blockenv_keys_tl
%    {
%       % check if only item template keys remain
%    }
```

If this is supposed to be a transparent block environment then we have to decrement the nesting level again so that nested environments think nothing is there.

```
\bool_if:NT \l__block_transparent_level_bool
   {  \int_gdecr:N \g_block_nesting_depth_int }
```

We finish off with `\l__block_final_code_tl` which defaults to `\ignorespaces` so that spaces between `\begin{...}` and the start of the text are ignored.

```
  \l__block_final_code_tl
}
```

decide

Might want a hook or a socket for legacy support.

```
\NewHook{blockenv}
```

`\BlockEnv` `\SimpleBlockEnv`

To simplify the environment declarations later we provide two simple commands that invoke a `blockenv` instance. The matching counterpart to these commands is `\BockEnvEnd` (defined below) that carries out all necessary action when a block environment ends.

```
\cs_new_protected:Npn \BlockEnv            % #1#2#3#4 implicit
   { \UseInstance{blockenv} }
```

This here is the most common one that hides arguments 2–4 when they aren't needed, e.g., in a `center` environment.

```
\cs_new_protected:Npn \SimpleBlockEnv #1#2
   { \UseInstance{blockenv}{#1}{#2} \BooleanFalse \NoValue \NoValue }
```

(*End of definition for* `\BlockEnv` *and* `\SimpleBlockEnv`*. These functions are documented on page 37.*)

`\g_block_nesting_depth_int`

LaTeX 2_ε already has a counter to record the nesting depth of blocks, but we want our own name because it isn't really tied to "lists" any more. However, `\@listdepth` is really part of the legacy interface (for example `minipage` alters it to point to a different counter) so that we are stuck with using at least indirectly for now and the following line makes this look like an L3 integer variable but internally expands to `\@listdepth`:

```
\cs_new_protected:Npn \g_block_nesting_depth_int { \@listdepth }  % a fake int
                                                                  % for now
```

(*End of definition for* `\g_block_nesting_depth_int`*. This function is documented on page 37.*)

`\l__block_unused_blockenv_keys_tl`

The token list that holds key values we haven't yet used while we are processing the instances in a block environment.

```
\tl_new:N \l__block_unused_blockenv_keys_tl
```

(*End of definition for* `\l__block_unused_blockenv_keys_tl`.)

`\l__tag_block_flattened_level_int`

Count the levels of nested `blockenv`s starting with the first that is "flattened". The counter is defined in lttagging.dtx, but until the next release 11/24 we set it up here too

```
\int_if_exist:NF \l__tag_block_flattened_level_int
  {
    \int_new:N \l__tag_block_flattened_level_int
  }
```

(*End of definition for* `\l__tag_block_flattened_level_int`.)

`\c@maxblocklevels`

A counter to increase or decrease the number of supported level. If increased, one needs to supply additional level instances.

```
\newcounter{maxblocklevels}
\setcounter{maxblocklevels}{6}
```

(*End of definition for* `\c@maxblocklevels`*. This function is documented on page 38.*)

\BlockEnvEnd

The code executed when a `blockenv` ends is 99% the same for all `blockenv`s (at least up to now). Small differences exist, though. They are accounted for first in the conditionals. We make this a public command so that new block environments can be set up without the need to resort to L3 layer programming.

```
\cs_new_protected:Npn \BlockEnvEnd {
  \__block_debug_typeout:n{blockenv~ common~ ending \on@line}
```

If this block is not a transparent one we have to decrement the level now again, otherwise that had happened earlier:

```
  \bool_if:NF \l__block_transparent_level_bool
    { \int_gdecr:N \g_block_nesting_depth_int }
```

If the `@inlabel` switch is true, i.e., if there is a caption or an item waiting to be placed we move to horizontal mode to get them typeset.

```
  \legacy_if:nT { @inlabel }
    {
      \mode_leave_vertical:
      \legacy_if_gset_false:n { @inlabel }
    }
```

If we are ending a list environment and we have not seen any `\item`, i.e., `@newlist` is still true, we raise an error. In basic a "displayblock" scenario `@newlist` will always be false, but if such an environment appears inside an outer list then `\noitemerr` could still be triggered and that is undesirable (as the missing item will be detected at the wrong point and again later, during the outer list processing). We therefore run it only if the current environment is a list.

```
  \__block_if_list:T { \legacy_if:nT { @newlist } { \@noitemerr } }
  \mode_if_horizontal:TF
       { \__block_skip_remove_last: \__block_skip_remove_last: \par }
       { \@inmatherr{\end{\@currenvir}} }
```

Once we are back in vertical mode we can add the appropriate closing tagging structure(s), if we are doing tagging.

```
  \__kernel_displayblock_end:
```

Resetting the `@newlist` switch is also only done if the current environment is a list.

```
  \__block_if_list:T { \legacy_if_gset_false:n { @newlist } }
```

There is a possibility that the `@nobreak` switch is still true so we set it back just in case.

```
  \legacy_if_gset_false:n { @nobreak }
```

What to do in terms of vertical spacing in different situations is still somewhat open to debate, right now this is more or less implementing what LaTeX 2_ε list environments have been doing.

some redesign/extensions here?

```
%   \__block_debug_typeout:n{@noparlist =
%                        \legacy_if:nTF { @noparlist }{true}{false}}
  \legacy_if:nF { @noparlist }
    {
      \__block_skip_set_to_last:N \l_tmpa_skip
      \dim_compare:nNnT \l_tmpa_skip > \c_zero_dim
        {
          \skip_vertical:n { - \l_tmpa_skip }
          \skip_vertical:n { \l_tmpa_skip + \parskip - \@outerparskip }
        }
      \addpenalty \@endparpenalty
      \addvspace \l__block_topsepadd_skip
```

LaTeX 2ε triggered the paragraph handling after a list at this point here, i.e., only if the list didn't start a paragraph. One can make a case for that, but it can be somewhat surprising to the user and there is a good argument that even such a list could be followed explanatory text that is part of the same paragraph and doesn't start a new one.

decide which logic we want to use! If the old logic is used we need to close the text-unit ourselves in the true branch

```
%       \legacy_if_gset_true:n { @endpe }
    }
```

So this is for now always done. Probably `\l__block_topsepadd_skip` above should be added only if the paragraph ends here and not if it continues, so this need some further cleanup.

decide

Finally, we have a socket that handles the `\par` handling after the block. Normally, we use it with the `on` plug (check for a following `\par`) but in the case of standalone environments we assign it the `off` plug.

```
  \socket_use:n {block/endpe}
}
```

(*End of definition for* `\BlockEnvEnd`. *This function is documented on page 37.*)

`\__block_if_list:T`

revisit and correct

The following code may need some redesigning, as there is no good test for "is this environment a 'list' that has `\items`". For now this here does the trick well enough.

```
\cs_new:Npn \__block_if_list:T
   { \tl_if_eq:NnT \l__block_block_instance_tl {std-list} }
```

(*End of definition for* `\__block_if_list:T`.)

`\__kernel_displayblock_end:` The kernel hook for tagging at the end of the block.

```
\cs_new_protected:Npn \__kernel_displayblock_end: {
  \__block_debug_typeout:n{\detokenize{__kernel_displayblock_end:}}
}
```

(*End of definition for* `\__kernel_displayblock_end:`.)

`block/endpe` (*socket*) This socket is responsible for the end environment `\par` handling. We define two plugs for it (`on` and `off`).

```
\socket_new:nn {block/endpe} {0}
```

`on` (*plug*)
`off` (*plug*) The plugs set the legacy `@endpe` switch. This must always happen because block environments with different settings can be nested and should not inherit the setting from the outer environment.

We can't use `\legacy_if_gset_true:n` because this is now doing more than setting the legacy switch:

```
\socket_new_plug:nnn{block/endpe} {on}  { \@endpetrue  }
\socket_new_plug:nnn{block/endpe} {off} { \@endpefalse }
\socket_assign_plug:nn{block/endpe}{on}
```

9.4.3 Implementation of para templates

`para std` (*templ.*)

```
\DeclareTemplateCode{para}{std}{1}
{
  ,para-indent           = \parindent
```

integrate/fix

The next parameter needs integrating in the basic paragraph handling (not done yet) and it should therefore probably a public name like the rest.

```
  ,begin-hspace          = \l_para_begin_skip
  ,left-hspace           = \leftskip
  ,right-hspace          = \rightskip
  ,end-hspace            = \parfillskip
```

fix

Next isn't yet implemented (and the variable name is wrong).

```
  ,fixed-word-spaces     = \l__par_fixed_word_spaces_bool  % name??
  ,final-hyphen-demerits = \finalhyphendemerits
  ,newline-cmd           = \\
  ,para-attr-class       = \l__tag_para_attr_class_tl
}
{
  \template_debug_typeout:n{~\space template:~ 'std';~
                            argument:~ \exp_not:n{|#1|}}
  \SetTemplateKeys{para}{std}{#1}

  \skip_set:Nn \@rightskip \rightskip
}
```

`\__para_handle_indent:`

We insert `\l_para_begin_skip` directly in front of the indentation box. This way it is hidden from any special setting of `\everypar` (whether that is used to remove the indentation box or whether it attempts to do something with the first token(s) of the paragraph). However, we only insert it if it differs from 0.0pt to avoid adding `\penalty 10000 \glue 0.0` all over the place.

```
\tl_const:Ne \c__zero_skip_tl { \skip_use:N \z@skip }
\tl_new:N  \l__para_begin_skip_tl

\cs_set:Npn \__para_handle_indent: {
  \tl_set:Ne \l__para_begin_skip_tl { \skip_use:N \l_para_begin_skip }
  \if_meaning:w \l__para_begin_skip_tl
                \c__zero_skip_tl
  \else:
    \nobreak
    \tex_hskip:D \l_para_begin_skip
  \fi:
  \box_use_drop:N \g_para_indent_box
}
```

(*End of definition for* `\__para_handle_indent:`.)

`\para_raw_noindent:`

`\para_raw_noindent:` doesn't call `\__para_handle_indent:` so we have to manually do the `\l_para_begin_skip` handling.

```
\cs_set:Npn \para_raw_noindent: {
  \mode_if_vertical:TF
       {
         \tex_everypar:D {
           \tex_everypar:D { \g__para_standard_everypar_tl }
           \tl_set:Ne \l__para_begin_skip_tl { \skip_use:N \l_para_begin_skip }
           \if_meaning:w \l__para_begin_skip_tl
                         \c__zero_skip_tl
           \else:
             \nobreak
             \tex_hskip:D \l_para_begin_skip
```

```
            \fi:
            \the\everypar }
        }
        { \msg_error:nn { latex2e }{ raw-para } }
    \tex_noindent:D
}
```

(*End of definition for* `\para_raw_noindent:`. *This function is documented on page* **??**.)

9.4.4 Implementation of `block` templates

`block std` (*templ.*) In contrast to the LaTeX 2$_\varepsilon$ implementation we do not directly use `\listparindent` here but a private register of the template. The reason is that block template instances are also used outside of lists.

```
\DeclareTemplateCode{block}{std}{1}
{
  ,begin-vspace           = \topsep
  ,begin-extra-vspace     = \partopsep
  ,begin-unchained-vspace = \l__block_unchained_skip
  ,para-vspace            = \parsep
```

fix

The bottom skips aren't used yet, even if set instead as before `\topsep` is applied there.

```
  ,end-vspace             = \l__block_botsep_skip
  ,end-extra-vspace       = \l__block_parbotsep_skip
  ,item-vspace            = \itemsep
  ,begin-penalty          = \@beginparpenalty
  ,end-penalty            = \@endparpenalty
  ,item-penalty           = \@itempenalty
  ,right-margin           = \rightmargin
  ,left-margin            = \leftmargin
  ,para-indent            = \l__block_parindent_dim
}
{
    \template_debug_typeout:n{~\space template:~ 'std';~
                              argument:~ \exp_not:o{\exp_after:wN |#1|}}
    \SetKnownTemplateKeys{block}{std}{#1}
```

The code largely follows the logic of LaTeX 2$_\varepsilon$'s `trivlist` implementation as far as it applicable for the "display block" but coded using the L3 programming layer. However, we keep most of the legacy variables (e.g., `@noskipsec`) if there is some chance that they are set/used in classes or packages.

```
    \legacy_if:nTF { @noskipsec }
```

A `@noskipsec` heading is a heading that is placed in the same line as the following text (using `\everypar`) but not if that text starts with a display block, so we ensure that the heading gets typeset now.

This is similar to the standalone case for block captions so perhaps that can be combined, check

```
        { \mode_leave_vertical: }
```

If no such heading is waiting we might have a block caption waiting to be typeset and this might be requested to be set "unchained". In that case we also have to ensure that this gets typeset now.

The situation is slightly different though, because we want to end in vertical mode in that case also add some special vertical space and have to properly deal with avoiding page breaks.

```
        {
```

```
      \bool_if:NT \g__block_label_unchained_bool
        {
          \__block_debug_typeout:n{Set~ captioned~ block~ everypar \on@line }
          \cs_set_eq:NN \__block_everypar: \__block_captioned_everypar_std:
          \legacy_if:nT { @inlabel }
                 {
                   \hbox_unpack_drop:N \g__block_labels_box
                   \legacy_if_gset_false:n { @inlabel }
                   \par
                   \nobreak
                   \skip_vertical:n { \l__block_unchained_skip }
                   \legacy_if_gset_true:n { @nobreak }
                 }
        }
    \skip_set:Nn \l__block_topsepadd_skip { \topsep }
    \mode_if_vertical:TF
      {
        \skip_add:Nn \l__block_topsepadd_skip { \partopsep }
```

At this point it is safe to add tagging structure(s) so we have a kernel-owned hook here for tagging. This is used to possibly start a paragraph structure (to surround the block, for example, in case of lists) and possibly do some other preparation for tagging the block.

```
        \__kernel_displayblock_beginpar_vmode:
      }
      {
```

If we are in horizontal mode then the displayblock has to return to vertical mode now (after removing any immediately preceding skip or kern. But before we actually issue the `\par` we execute a kernel hook in which we can add tagging code. This hook is "weird" because by default it does nothing, but if tagging is wanted it takes an argument and grabs the following `\par` in order to put tagging code before and after the `\par`.

```
        \__block_skip_remove_last: \__block_skip_remove_last:
        \__kernel_displayblock_beginpar_hmode:w \par
      }
```

Next lines set some paragraph defaults, any of them may get overwritten if there is a `para-instance` specified on the `blockenv` instance.

```
    \skip_zero:N \leftskip
    \skip_set_eq:NN \rightskip   \@rightskip
    \skip_set_eq:NN \parfillskip \@flushglue
```

The next lines establish a parshape which is retained across paragraphs be executing `\para_end:` within a group and thus reestablishing the parshape for the next paragraph again. In case a list got started `\par` is ignored until we have seen an `\item` (or we have executed `\par` one thousand times.

```
    \int_zero:N \par@deathcycles
    \@setpar
      {
        \legacy_if:nTF { @newlist }
          {
            \int_incr:N \par@deathcycles
            \int_compare:nNnTF \par@deathcycles > { 1000 }
                { \@noitemerr
                  { \para_end: }
```

```
                }
            }
            {
              { \para_end: }
            }
        }

      \dim_set_eq:NN \parindent \l__block_parindent_dim
      \dim_add:Nn \linewidth { - \rightmargin - \leftmargin }
      \dim_add:Nn \@totalleftmargin { \leftmargin }
      \tex_parshape:D 1 ~ \@totalleftmargin \linewidth
```

This is the point where we are ready to add the tagging structure for the block, e.g., an `<L>`, a `<Figure>` or some other structure.

```
      \__kernel_displayblock_begin:
```

Finally, we have to output the vertical separation and penalty at the start of the block and make corrections for a change in `\parskip` and some other housekeeping, unless this block is inside a list and the list `\item` has not yet placed. In that case the vertical space and penalty is suppressed. This is controlled through the legacy switches `@inlabel`, `minipage`, and `@nobreak`.
Now we are back to legacy list implementation …

```
      \skip_set_eq:NN \@outerparskip \parskip
      \skip_set_eq:NN \parskip \parsep
%
      \legacy_if:nTF { @inlabel }
        {
          \legacy_if_set_true:n { @noparlist }
          \hbox_gset:Nn \g__block_labels_box
            {
              \skip_horizontal:n { - \leftmargin }
              \hbox_unpack_drop:N \g__block_labels_box
              \skip_horizontal:n { \leftmargin }
            }
```

document 2e logic used here

```
          \legacy_if:nF { @minipage } % Why this chunk of code?
            {
              \__block_skip_set_to_last:N \l__block_tmpa_skip
              \skip_vertical:n { - \l__block_tmpa_skip }
              \skip_vertical:n { \l__block_tmpa_skip +
                                 \@outerparskip - \parsep }
            }
        }
        {
          \legacy_if_set_false:n { @noparlist }
          \legacy_if:nT { @newlist } { \@noitemerr }
          \legacy_if:nTF { @nobreak }
            {
```

We are not resetting `@nobreak` here as it should also apply to the upcoming item.

```
              \addpenalty{ 10000 }
              \addvspace{ \skip_eval:n{\@outerparskip-\parsep} }
            }
            {
              \addpenalty \@beginparpenalty
```

```
            \addvspace { \skip_eval:n { \l__block_topsepadd_skip +
                                        \@outerparskip } }
            \addvspace { - \parsep }
          }
    }
}
```

__block_captioned_everypar_std:

The captioned text is typeset at the start of a paragraph using code triggered in \everypar (by setting __block_everypar to this code here).

```
\cs_new_protected:Npn \__block_captioned_everypar_std: {
    \__block_debug_typeout:n{...~ in~ captioned~ block~ everypar \on@line }
```

First set some control switches to false:

```
    \legacy_if_set_false:n { @minipage }
    \legacy_if_gset_false:n { @newlist }
```

The @inlabel is normally true at this point, but if we also have @nobreak then the same routine is called again at the next paragraph to reset \clubpenalty and at that point the \g__block_labels_box has been typeset and @inlabel is false.

```
    \legacy_if:nT { @inlabel }
       {
```

Typeset the saved label (aka captioned text):

```
         \legacy_if_gset_false:n { @inlabel }
         \para_omit_indent:
         \box_use_drop:N \g__block_labels_box
         \__kernel_list_label_after:n { \PARALABEL }     % <- change
                                                         % this name
         \penalty \c_zero_int
       }
```

If @nobreak is true we prevent a break after the first line by setting \clubpenalty.

```
   \legacy_if:nTF { @nobreak }
       {
         \legacy_if_gset_false:n { @nobreak }
         \int_set:Nn \clubpenalty { 10000 }
       }
       {
```

Otherwise we reset \clubpenalty and disable __block_everypar.

```
         \int_set_eq:NN \clubpenalty \@clubpenalty
         \__block_debug_typeout:n{Set~ noop~ block~ everypar \on@line }
         \cs_set_eq:NN \__block_everypar: \prg_do_nothing:
       }
}
```

(*End of definition for* __block_captioned_everypar_std:.)

__kernel_displayblock_begin:
__kernel_displayblock_beginpar_hmode:w
__kernel_displayblock_beginpar_vmode:

The internal kernel hooks for tagging.

```
\cs_new_protected:Npn \__kernel_displayblock_begin: {
  \__block_debug_typeout:n
     {\detokenize{__kernel_displayblock_begin:}}
}
\cs_new_protected:Npn \__kernel_displayblock_beginpar_hmode:w {
  \__block_debug_typeout:n
     {\detokenize{__kernel_displayblock_beginpar_hmode:w}}
}
```

```
\cs_new_protected:Npn \__kernel_displayblock_beginpar_vmode: {
  \__block_debug_typeout:n
     {\detokenize{__kernel_displayblock_beginpar_vmode:}}
}
```

(*End of definition for* `\__kernel_displayblock_begin:`, `\__kernel_displayblock_beginpar_hmode:w`, *and* `\__kernel_displayblock_beginpar_vmode:`.)

9.4.5 Implementation of `list` templates

This `list` is one of the template types that can be used as an `inner-type` in a `blockenv`; the other one currently implemented is `captionedtext`.

`\@itemlabel` `\@listctr`

Both `\@itemlabel` and `\@listctr` from the LaTeX 2_ε list implementation are used (or set) by various packages. We therefore use them too, so that these packages have a fighting chance to work with the new tagging-aware implementation for `list`.

```
\tl_new:N \@itemlabel          % should have a top-level definition
\tl_new:N \@listctr            % should have a top-level definition
```

(*End of definition for* `\@itemlabel` *and* `\@listctr`. *These functions are documented on page 38.*)

`\__block_evaluate_saved_user_keys:nn`

Keys set on individual list environments may be intended to alter the behavior of the template instance that defines the `\item` command. If meant to alter only a single `\item` command one would specify them in the optional argument of the `\item`, but if they should alter all items the right place would be the list environment. For this reason we need to store the values and then set them inside the `\item` template code using `\SetKnownTemplateKeys` in the appropriate context (template type and template name). This is done in `\__block_evaluate_saved_user_keys:nn`. The context is provided in the two arguments (because different list environments may use different `\item` instances based on different templates. By default the command does nothing because most environments do not have user key settings.

```
\cs_new_eq:NN \__block_evaluate_saved_user_keys:nn \use_none:nn
```

Maybe something like this should become a public function, but for now this is a one-off for the `\item` command and therefore coded inline and internal to the block code.

```
%\cs_new:Npn \__block_save_user_keys:n #1 {
%  \tl_if_empty:nTF {#1}
%     { \cs_set_eq:NN \__block_evaluate_saved_user_keys:nn \use_none:nn }
%     { \cs_set:Npe \__block_evaluate_saved_user_keys:nn ##1##2
%          { \SetKnownTemplateKeys{##1}{##2}{ \exp_not:n{#1} }    }
%}
```

(*End of definition for* `\__block_evaluate_saved_user_keys:nn`.)

`list std` (*templ.*)

This template implements numbered and unnumbered lists and can be combined with display blocks or with inline blocks.

```
\DeclareTemplateCode{list}{std}{4}
{
  ,counter         = \l__block_counter_tl
  ,item-label      = \l__block_item_label_tl
  ,start           = \l__block_counter_start_int
  ,resume          = \l__block_resume_bool
  ,item-instance   = \__block_item_instance:n
  ,item-vspace     = \itemsep
```

```
%  ,item-para-vspace   = \parsep
  ,item-penalty    = \@itempenalty
  ,item-indent     = \itemindent
  ,label-width     = \labelwidth
  ,label-sep       = \labelsep
  ,legacy-support  = \l__block_legacy_support_bool % FMi questionable
}
{
  \template_debug_typeout:n{~\space template:~ 'std';~
                            arguments:~ \exp_not:o{\exp_after:wN |#1|#2|#3|#4|}}
```

We start by looking at the user supplied keys in `#1`. If there aren't any we reset `\__block_evaluate_saved_user_keys:nn` to do nothing. Otherwise we evaluate and set the keys in the context of the current list template. In addition we prepare `\__block_evaluate_saved_user_keys:nn` for execution in the template for `\item`.

```
  \tl_if_empty:oTF {#1}
     { \cs_set_eq:NN \__block_evaluate_saved_user_keys:nn \use_none:nn }
     {
       \SetKnownTemplateKeys{list}{std}{#1}
```

The setup for `\__block_evaluate_saved_user_keys:nn` is a bit tricky and has to be done with `\cs_set:Npe` even though we don't want to expand anything and therefore use `\exp_not:n` inside. All this does is that any `#` passed in via `#1` is doubled (e.g., from `label-format=\fbox{#1}` which is represented as `...\fbox{##1}`). Otherwise, we would end up with a replacement text like

```
\SetTemplateKeys {#1}{#2}{label-format=\fbox {#1}}
```

instead of

```
\SetTemplateKeys {#1}{#2}{label-format=\fbox {##1}}
```

resulting in very odd and puzzling behavior later on.
The definition of `\__block_evaluate_saved_user_keys:nn` made here is later used when an `\item` is processed and passes remaining keys to the item instance. After that nothing should remain, so we test that and issue an error if not.

```
       \cs_set:Npe \__block_evaluate_saved_user_keys:nn ##1##2
          { \SetKnownTemplateKeys{##1}{##2}{
              \exp_not:o { \UnusedTemplateKeys }
            }
            \exp_not:n {
              \tl_if_empty:NF \UnusedTemplateKeys
                 {
                   \msg_error:nnee { block } { unknown-keys }
                       { \l__block_env_name_tl \space environment}
                       \UnusedTemplateKeys
                 }
            }
          }
```

Has this list a counter name defined in the instance?

```
  \tl_if_empty:NTF \l__block_counter_tl
    {
```

If no counter name has been specified as part of the instance setup the list might still be numbered if it is a legacy list that uses `\usecounter` in the second argument of the legacy `list` environment. However, in that case we don't have to do much because `\usecounter` sets up `\@listctr` and sets it to zero so that the first item is numbered 1.
So all we do is to check if there was a `start` value given that differs from 1 and if so we change the counter value to match that. This makes it possible to define a legacy `list` in which the counter doesn't start with 1 by explicitly setting the counter value in the second argument of the `list` environment but also overwriting that through a `start` key setting on invocation.

```
1213      \int_compare:nNnF \l__block_counter_start_int = 1
1214        {
1215          \int_gset:cn{ c@ \@listctr }
1216              { \l__block_counter_start_int - 1 }
1217        }
1218      }
```

In that case we only check if we should resume a previous list (`\@listctr` should be set in that case through the legacy method as well so we should be able to use it).

If a counter is set in the list instance we use that one. This should be the name of a LaTeX counter that is already allocated externally—no runtime check is made for this: if it is not declared one will get "no such counter" error when the list is used.

```
1219      {
1220       \@nmbrlisttrue
1221       \tl_set_eq:NN \@listctr \l__block_counter_tl
1222       \bool_if:NF \l__block_resume_bool
1223         {
1224           \int_gset:cn{ c@ \@listctr }
1225               { \l__block_counter_start_int - 1 }
1226         }
1227      }
```

Does the current instance have an item label representation? This would be possible whether or not we have a numbered list. If yes, then we use this for `\@itemlabel`, otherwise we expect that `\@itemlabel` is provided from the outside, e.g., as part of the `list` environment argument.

```
1228    \tl_if_empty:NF \l__block_item_label_tl
1229      {
1230        \tl_set_eq:NN \@itemlabel \l__block_item_label_tl
1231      }
```

Finally, we signal that we are at the start of a new list (which affects how the first `\item` is handled and how `\par` commands are interpreted.

```
1232      \legacy_if_gset_true:n { @newlist }
```

If we encounter horizontal material before the first `\item` we do want a `\@noitemerr` straight away, because afterwards we end up with tagging structure faults whose cause is the missing `\item`. So we set up `\__block_everypar:` to test for this; when the first `\item` is encountered this will get reset. This is only relevant for vertical lists, when dealing with inline lists one would need to test for something else to identify that there is horizontal material between the start of the list and the first `\item` (maybe some `\spacefactor` trick could be used then, or the material is boxed first and the width is inspected as suggested by Joseph).

Think about a better implementation at some point.

```
1233      \__block_debug_typeout:n{Set~ first~ block~ everypar \on@line }
1234      \cs_set_eq:NN \__block_everypar: \__block_item_everypar_first:
```

```
  \__block_debug_typeout:n{template:list:std~end}
}
```

The message that is used above when we are left with keys that are unknown:

```
\msg_new:nnnn { block } { unknown-keys }
  { Some~ keys~ specified~ on~ the~ #1~ are~ unknown. }
  {
    The~ following~ keys~ are~ unknown~ and~ their~
    values~ are~ ignored:\\
    \space\space #2\\
    Perhaps~ a~ misspelling~ or~ the~ current~ template~
    instance~ uses~ special~ keys.
  }
```

9.4.6 Implementation of `item` templates

`item std` (*templ.*) The item template has one hidden key `label` which is not available on the template for setting because it is only used to receive any optional data passed to the `\item` command. We therefore declare it with `\keys_define:nn` and ensure that the optional argument data to `\item` (if it is not a key/value list already) is passed to this `label` key.

```
\keys_define:nn { template/item/std }
                { label .tl_set:N = \l__block_label_given_tl }
```

alignment is mostly wrong (test short medium and multiline labels)

```
\DeclareTemplateCode{item}{std}{1}
  {
    ,counter-label   = \__block_counter_label:n
    ,counter-ref     = \__block_counter_ref:n
```

next set of key not yet used

```
    ,label-ref       = \__block_label_ref:n
    ,label-autoref   = \__block_label_autoref:n
    ,label-format    = \__block_label_format:n
    ,label-strut     = \l__block_label_strut_bool
    ,label-boxed     = \l__block_label_boxed_bool
    ,next-line       = \l__block_next_line_bool
    ,text-font       = \l__block_text_font_tl
    ,compatibility   = \l__block_item_compatibility_bool
```

complete

This probably needs a different implementation (and needs completing)

```
    ,label-align     = {
      left    = \tl_set:Nn \l__block_item_align_tl { \relax \hss } ,
      center  = \tl_set:Nn \l__block_item_align_tl { \hss \hss } ,
      right   = \tl_set:Nn \l__block_item_align_tl { \hss \relax } ,
      parleft = \NOT_IMPLEMENTED ,
     }
```

The keylabel-placement is implemented using two booleans (at the moment).

```
    ,label-placement = {
      chained    = \bool_gset_false:N \g__block_label_standalone_bool
                   \bool_gset_false:N \g__block_label_unchained_bool ,
      unchained  = \bool_gset_false:N \g__block_label_standalone_bool
                   \bool_gset_true:N  \g__block_label_unchained_bool ,
      standalone = \bool_gset_true:N  \g__block_label_standalone_bool
                   \bool_gset_false:N \g__block_label_unchained_bool ,
     }
  }
```

Then typeset the label at its natural width by applying `\__block_make_label_box:n` to the label given or to a label constructed from the counter. If it is boxed and reasonably short, add padding to make it at least of size `\labelwidth`, then add another layer of box. This way, when we unpack it in `\g__block_labels_box` it correctly remains boxed in those cases. Afterwards, in the `nextline` case add `\newline` if the label did not fit in the allotted space.

```
{
  \template_debug_typeout:n{~\space template:~ 'std';~
                            argument:~ \exp_not:n{|#1|}}
```

First deal with the key–value input, which in particular may provide a value for the label (the usual optional argument of `\item`). For this we set `\l__block_label_given_tl` to `\c_novalue_tl` so that we can identify if an optional argument was given.

```
  \tl_set_eq:NN \l__block_label_given_tl \c_novalue_tl
```

First we evaluate and set any keys specified on the list environment by calling `\__block_evaluate_saved_user_keys:nn`. Then we do the same with all keys specified on this `\item` command (which may overwrite one or the other setting just made).

```
  \__block_evaluate_saved_user_keys:nn {item}{std}
```

We don't care whether all of the user keys from the list level have been applied, but those explicitly set on the `\item` command should be applicable, so we generate an error if that isn't the case:

```
  \SetKnownTemplateKeys{item}{std}{#1}
  \tl_if_empty:NF \UnusedTemplateKeys
     {
       \msg_error:nnee { block } { unknown-keys }
           { \noexpand\item command }
           \UnusedTemplateKeys
     }
```

next line needs checking after novalue implementation was changed

If no optional argument was given then `\l__block_label_given_tl` is still equal to `\c_novalue_tl` and so we can distinguish that from `\item[]`.

```
  \tl_if_novalue:oTF \l__block_label_given_tl
    {
```

fix

The rest of the code for this template needs work and is both incomplete and partly wrong.

```
      \tl_if_blank:oF \@listctr { \@kernel@refstepcounter \@listctr }
      \bool_if:NTF \l__block_item_compatibility_bool   % not sure that
                                                       %  conditional
                                                       %  makes sense
        { \__block_make_label_box:n { \MakeLinkTarget[\@listctr]{}%
                                      \@itemlabel } } % TODO ?
        { \__block_make_label_box:n { \MakeLinkTarget[\@listctr]{}%
                                      \__block_counter_label:n { \@listctr } } }
    }

    {
      \__block_debug_typeout:n{item~ with~ optional}
      \__block_make_label_box:n {
        \MakeLinkTarget [\l__block_env_name_tl]{}
        \l__block_label_given_tl
      }
```

```
      }
    \bool_if:nT
      {
        \l__block_label_boxed_bool
%   TODO: is \linewidth correct?
        && \dim_compare_p:n
               { \box_wd:N \l__block_one_label_box <= \linewidth }
      }
      {
        \dim_compare:nNnT
          { \box_wd:N \l__block_one_label_box } < \labelwidth
          {
            \hbox_set_to_wd:Nnn \l__block_one_label_box { \labelwidth }
              {
                \exp_after:wN \use_i:nn \l__block_item_align_tl
```

FMi: LaTeX 2_ε keeps the label boxed inside (not unboxed). This means that the content stays rigid and does not vary based on glue setting in the line with the label. There are cases where we do want the unboxed version (I think enumitem offers that in some cases too) but it should probably not the default.

```
%                TODO: customize?
%                \hbox_unpack_drop:N \l__block_one_label_box
                 \box_use_drop:N \l__block_one_label_box
                 \exp_after:wN \use_ii:nn \l__block_item_align_tl
               }
          }
```

Add another box level to the label box:

```
       \hbox_set:Nn \l__block_one_label_box
                    { \box_use_drop:N \l__block_one_label_box }
      }
    \dim_compare:nNnTF { \box_wd:N \l__block_one_label_box } > \labelwidth
      { \bool_set_true:N \l__block_long_label_bool }
      { \bool_set_false:N \l__block_long_label_bool }
    \hbox_gset:Nn \g__block_labels_box
      {
        \hbox_unpack_drop:N \g__block_labels_box
        \skip_horizontal:n { \itemindent - \labelsep - \labelwidth }
        \hbox_unpack_drop:N \l__block_one_label_box
        \skip_horizontal:n { \labelsep }
        \bool_if:NT \l__block_next_line_bool
          { \bool_if:NT \l__block_long_label_bool { \nobreak \hfil \break } }
        % version of \newline inside an hbox that will be unpacked
      }
    % TODO??? FMi what's that?
    % \skip_set_eq:NN \parsep \l__block_item_parsep_skip
```

The next setting is for compatibility: The list template sets `\listparindent` to zero and otherwise doesn't use it any more. However, in the second argument of a legacy `list` environment the user may have set it explicitly to some other value and whatever value it had was then used for `\parindent` within the list. Now we use its value only if it differs from zero but otherwise use whatever the template instances specify. This gives 99.9% compatibility for legacy documents. 100% for definitions using the `list` environment and a setting inside, but if the user used `\listparindent` within the document, e.g.,

inside a `verse` environment there there is one case in which the setting is ignored, i.e., when it was set back to zero. That's a rather unlikely scenario, but it is not impossible. However, I couldn't think of an approach that circumvents such boundary cases.

```
    \dim_compare:nNnF \listparindent = {0pt}
      { \dim_set_eq:NN \parindent \listparindent }
```

Placing the list label(s) is done when the paragraph for the `\item` is started, which executes `\__block_everypar:` inside `para/begin`. By default this command does nothing, now we change it to attach the pending label or labels.

```
    \__block_debug_typeout:n{Set~ item~ block~ everypar \on@line }
    \cs_set_eq:NN \__block_everypar: \__block_item_everypar_std:
  }
```

g__block_label_standalone_bool
g__block_label_unchained_bool

The two booleans for implementing `label-placement` and below `caption-placement`.

```
\bool_new:N \g__block_label_standalone_bool       % tmp until replaced
\bool_new:N \g__block_label_unchained_bool        % tmp until replaced
```

(*End of definition for* `g__block_label_standalone_bool` *and* `g__block_label_unchained_bool`.)

\l__block_item_align_tl

```
\tl_new:N \l__block_item_align_tl
```

(*End of definition for* `\l__block_item_align_tl`.)

\l__block_one_label_box
\g__block_labels_box

Each label is typeset in `\l__block_one_label_box` to be measured. Once this is ready, it is put (boxed or unboxed) in `\g__block_labels_box`, together with any pending labels (for the case where a list begins just after `\item`). This is an analogue of LaTeX 2_ε's `\@labels`, but it is always unboxed before use, to support both boxed and unboxed labels.

```
\box_new:N \l__block_one_label_box
\box_new:N \g__block_labels_box
```

(*End of definition for* `\l__block_one_label_box` *and* `\g__block_labels_box`.)

\l__block_long_label_bool

Track whether the `\l__block_one_label_box` is larger than `\labelwidth`.

```
\bool_new:N \l__block_long_label_bool
```

(*End of definition for* `\l__block_long_label_bool`.)

__block_make_label_box:n
__block_label_format:e

Make one label, wrapped in `\__block_label_format:n`, with an appropriate `\strut` and possibly `\makelabel` in compatibility mode (used for the `list` environment).

```
\cs_new_protected:Npn \__block_make_label_box:n #1
  {
    \hbox_set:Nn \l__block_one_label_box
      {
```

If we do tagging then the contents of this box may need to be wrapped into a structure, e.g., `<Lbl>`.

```
        \tag_socket_use:nnn {block/list/label}{}
          {
```

```
            \__block_label_format:n
              {
                \bool_if:NT \l__block_label_strut_bool { \strut }
                \bool_if:NTF \l__block_legacy_support_bool
                             \makelabel
                             \use:n
                     {#1}
              }
```

And what gets opened also needs closing:

```
          }
      }
  }
```

(*End of definition for* `\__block_make_label_box:n` *and* `\__block_label_format:e`.)

`block/list/label` (*socket*)

I think this socket should just be called list/label

`default` (*plug*)

A tagging socket to tag the label. It takes two arguments so that it can transparently pass the label content. Declaration is in `lttagging.dtx`.

```
\NewTaggingSocketPlug{block/list/label}{default}
 {
   %
   % FMi: this needs a different logic to decide when to make the label
   %     an artifact (after cleaning up the \item code  ), therefore
   %    disabled for now
   %  \tl_if_empty:oTF \@itemlabel
   %     {
   %       \tag_mc_begin:n {artifact}
   %     }
   %     {
   \tagstructbegin{tag=\UseStructureName{block/list/label}}
   \tagmcbegin{tag=\UseStructureName{block/list/label}}
   %     }
   #2
   \tagmcend               % end mc-\UseStructureName{block/list/label} or artifact
   % FMi: unconditionally for now
   %  \tl_if_empty:oF \@itemlabel
   \tagstructend   % end   label
   \tagstructbegin{tag=\UseStructureName{block/list/body}}
 }
\AssignTaggingSocketPlug{block/list/label}{default}
```

`\__block_everypar:`
`\__block_item_everypar_std:`
`\__block_item_everypar_first:`

The `\__block_everypar:` command is executed as part of `para/begin` but most of the time does nothing, i.e., it has the following default definition outside of lists (and most of the time within lists).

```
\cs_new_eq:NN \__block_everypar: \prg_do_nothing:
\AddToHook{para/begin}[items]{\__block_everypar:}
```

Note that we have to make sure that the above code is executed after the hook chunk from `tagpdf` because the latter uses `@inlabel` to make a decision.
By the end of the day both should probably move into the kernel hook instead.

```
\DeclareHookRule{para/begin}{items}{after}{tagpdf}
```

What follows is the version that resets various legacy booleans and puts the label box in the right place and finally resets itself to do nothing next time. `\__block_everypar:` is set to this by the item template so that the next paragraph start runs the code below.

```
\cs_new_protected:Npn \__block_item_everypar_std: {
    \__block_debug_typeout:n{...~ in~ item~ block~ everypar \on@line }
    \legacy_if_set_false:n { @minipage }
    \legacy_if_gset_false:n { @newlist }
    \legacy_if:nT { @inlabel }
       {
         \legacy_if_gset_false:n { @inlabel }
         \box_if_empty:NT \g_para_indent_box { \kern - \itemindent }
         \para_omit_indent:
         \box_use_drop:N \g__block_labels_box
```

After the labels are placed we start a paragraph structure (if appropriate). This is handled in the following kernel hook:

```
         \__kernel_list_label_after:n {LI-}
         \penalty \c_zero_int
       }
   \legacy_if:nTF { @nobreak }
       {
         \legacy_if_gset_false:n { @nobreak }
         \int_set:Nn \clubpenalty { 10000 }
       }
       {
         \int_set_eq:NN \clubpenalty \@clubpenalty
```

Once the label(s) are typeset and we are past any special `@nobreak` handling we reset `\__block_everypar:` to do nothing.

```
         \__block_debug_typeout:n{Set~ noop~ block~ everypar \on@line }
         \cs_set_eq:NN \__block_everypar: \prg_do_nothing:
       }
}
```

This is the definition of `\__block_everypar:` before the first `\item` is encountered.

```
\cs_new_protected:Npn \__block_item_everypar_first: {
  \__block_debug_typeout:n{...~ in~ first~ block~ everypar \on@line }
  \legacy_if:nT { @newlist }  { \@noitemerr }
}
```

(*End of definition for* `\__block_everypar:`*,* `\__block_item_everypar_std:`*, and* `\__block_item_everypar_first:`*.*)

`\l__block_tmpa_skip`

```
\skip_new:N \l__block_tmpa_skip
```

(*End of definition for* `\l__block_tmpa_skip`*.*)

`\l__block_topsepadd_skip` `\l__block_effective_top_skip`

Variables equivalent to LaTeX 2_ε's `\@topsepadd` and `\@topsep`. Roughly equal to a mixture of `topsep`, `partopsep`, and various `parskip` at different nesting levels in lists. The code is really elaborate when `@inlabel` is true.

```
\skip_new:N \l__block_topsepadd_skip
\skip_new:N \l__block_effective_top_skip
```

(*End of definition for* `\l__block_topsepadd_skip` *and* `\l__block_effective_top_skip`*.*)

\item Here we already have all the building blocks. Complain in math mode. Distinguish between first item (do necessary tagging) and later items `\__block_inter_item:` to cleanly close what's before, then call `\__block_item_instance:n` (which calls `\UseInstance{item}{`⟨*instance*⟩`}`) to prepare the upcoming item: it will be actually inserted only once some later material triggers `\everypar`.

```
\AddToHook{begindocument/before}[./legacy-lists]{
  \RenewDocumentCommand{\item}{ ={label}o }
    {
      \@inmatherr \item
```

TODO: Check if test for being outside of a list is sensible

```
      \cs_if_free:NTF \__block_item_instance:n
        {
          \@latex@error{Lonely~\string\item--perhaps~a~missing~
          list~environment}\@ehc
        }
        {
          \legacy_if:nTF { @newlist }
            {
              \__kernel_list_item_begin:
```

The first item of a list also has to change the `@newlist` switch.

```
              \legacy_if_gset_false:n { @newlist }
            }
            { \__block_inter_item: }
```

To avoid unnecessary key/val processing we make a quick check if there was an optional argument.

```
          \tl_if_novalue:nTF {#1}          % avoids reparsing label={}
            { \__block_item_instance:n { } }
            { \__block_item_instance:n {#1} }
```

The `item` instance puts the item label into `\g__block_label_standalone_bool` ready to be placed later. To make that happen we need to signal that by setting the legacy switch `@inlabel` to true. However, if this is a label that should be always placed "standalone" we instead typeset it immediately and ensure that there is no page break after it.

support extra vertical space as well?

```
          \bool_if:NTF \g__block_label_standalone_bool
             {
               \bool_gset_false:N \g__block_label_standalone_bool
               \leavevmode
               \box_use_drop:N \g__block_labels_box
               \par
               \legacy_if_gset_true:n { @nobreak }  % do not break after
                                                    % a standalone item
             }
             {
               \legacy_if_gset_true:n { @inlabel }
             }
          \ignorespaces
        }
    }
}
```

(*End of definition for* `\item`*. This function is documented on page 38.*)

__block_inter_item: Between items. If the previous item had no content then we need to trigger \everypar. Otherwise we simply close the previous item with \par after removing some horizontal space. Between items, there is a penalty and some space.

```
\cs_new_protected:Npn \__block_inter_item: {
  \legacy_if:nT { @inlabel }
                { \indent \par } % case of \item\item
```

\par may have a strange definition and may not get us back to vertical mode in one go, so we better not treat the next line as an else case to the above conditional (for now).

```
   \mode_if_horizontal:T { \__block_skip_remove_last:
                           \__block_skip_remove_last: \par }
```

End any LI-tag, then start the next LI-tag (if doing tagging):

```
  \__kernel_list_item_end:
  \__kernel_list_item_begin:

  \addpenalty \@itempenalty
  \addvspace \itemsep
}
```

(*End of definition for* __block_inter_item:.)

__kernel_list_item_begin:
__kernel_list_item_end:

```
\cs_new_eq:NN \__kernel_list_item_begin: \prg_do_nothing:
\cs_new_eq:NN \__kernel_list_item_end:   \prg_do_nothing:
```

(*End of definition for* __kernel_list_item_begin: *and* __kernel_list_item_end:.)

9.4.7 Implementation of captionedtext and thmstyle templates

captionedtext thmlike (*templ.*) The template for typical theorem-like environments is rather trivial, just setting keys and then passing used keys and the arguments to a thmstyle instance to do the real work.

```
\DeclareTemplateCode{captionedtext}{thmlike}{4}
{
  ,counter         = \l__block_counter_tl
  ,title           = \l__block_title_tl
  ,style           = \l__block_style:nnnn
}
{
```

Some debugging info as usual (showing the arguments that are passed):

```
  \template_debug_typeout:n{~\space template:~ 'thmlike';~
                            arguments:~ \exp_not:o{\exp_after:wN |#1|#2|#3|#4|}}
```

Then we check if there are any keys passed to the instance from the outside.

```
  \SetKnownTemplateKeys{captionedtext}{thmlike}{#1}
```

Finally, we apply the style which is just an instance of type thmstyle:

```
  \l__block_style:nnnn \UnusedTemplateKeys {#2} {#3} {#4}
}
```

\theoremstyle
\l__block_thmstyle_tl All that the \theoremstyle declaration does is saving its argument so that it can be used in \newtheorem.

```
\cs_new_protected:Npn \theoremstyle #1{ \tl_set:Nn \l__block_thmstyle_tl {#1} }
\tl_new:N \l__block_thmstyle_tl
```

And the default is `plain`:

```
\theoremstyle{plain}
```

(*End of definition for* `\theoremstyle` *and* `\l__block_thmstyle_tl`*. This function is documented on page* ??.)

`thmstyle std` (*templ.*) The `thmstyle` implements the theorem-like environment and assumes that the fixed part of the caption is already stored in `\l__block_title_tl`. The reason for this separation into two templates is that typically the same design is used for different theorem-like environments only differing in this fixed string.

```
\DeclareTemplateCode{thmstyle}{std}{4}
{
  ,numbered        = \l__block_numbered_bool
  ,space           = \l__block_space_tl
  ,punct           = \l__block_punct_tl
  ,caption-placement = {
     chained    = \bool_gset_false:N \g__block_label_standalone_bool
                  \bool_gset_false:N \g__block_label_unchained_bool
    ,unchained  = \bool_gset_false:N \g__block_label_standalone_bool
                  \bool_gset_true:N  \g__block_label_unchained_bool
    ,standalone = \bool_gset_true:N  \g__block_label_standalone_bool
                  \bool_gset_false:N \g__block_label_unchained_bool
   }
  ,before-hspace   = \l__block_caption_before_skip
  ,after-hspace    = \l__block_caption_after_skip
  ,order           = \l__block_order_clist
  ,caption-decls   = \l__block_caption_decls_tl
  ,title-format    = \__block_title_format:n
  ,number-format   = \__block_number_format:n
  ,punct-format    = \__block_punct_format:n
  ,note-format     = \__block_note_format:n
  ,body-decls      = \l__block_body_decls_tl
}
{
```

Some tracing:

```
  \template_debug_typeout:n{~\space template:~ 'std';~
                            arguments:~ \exp_not:o{\exp_after:wN |#1|#2|#3|#4|}}
```

Applying any user keys:

```
  \SetKnownTemplateKeys{thmstyle}{std}{#1}
```

Since this is the last template that gets applied for theorem-like environments, all keys should make sense, so if something is left over we better generate an error:

```
  \tl_if_empty:NF \UnusedTemplateKeys
     {
       \msg_error:nnee { block } { unknown-keys }
           { \l__block_env_name_tl \space environment}
           \UnusedTemplateKeys
     }
```

In case there is a dangling `\item` we can either join that with the caption or we can output the item first. For now we provide no customization for this, but it could be made customizable.

```
%  \legacy_if:nT { @inlabel } { \indent \par }
```

Determine if we do numbering:

```
\bool_lazy_or:nnT
  { \tl_if_empty_p:N \l__block_counter_tl }
  { #2 }
  { \bool_set_false:N \l__block_numbered_bool }
```

Save any note for later use (`\#3` might contain `\NoValue`):

```
\tl_set:Nn \l__block_note_tl {#3}
```

If we use numbering then we need a link target and increment the counter.

```
\bool_if:NTF \l__block_numbered_bool
   {
     \@kernel@refstepcounter{ \l__block_counter_tl }
     \MakeLinkTarget{ \l__block_counter_tl }
   }
   {
     \MakeLinkTarget[theorem-like]{}
   }
```

Add the caption into `\g__block_labels_box`:

```
\hbox_gset:Nn \g__block_labels_box
  {
    \box_use_drop:N \g__block_labels_box % <- does nothing if
                                         % there is no dangling label
```

Now apply the declarations that are for the whole caption.

```
    \l__block_caption_decls_tl
```

Then we apply the tagging socket for the caption to the complete content:

```
    \tag_socket_use:nnn {captionedtext/caption} {}
        {
          \skip_horizontal:n { \l__block_caption_before_skip }
```

For flexibility, the inner structure is given as a clist stored in `\l__block_order_clist`. We loop through it and call a processing function for each item in this clist. Everything happens in a group

```
          \clist_map_inline:Nn \l__block_order_clist
                { \group_begin:
                    \use:c { __block_do_##1: }
                  \group_end:
                }
          \skip_horizontal:n { \l__block_caption_after_skip }
        }
  }
```

If the title should be standalone we immediately push it out:

```
\bool_if:NTF \g__block_label_standalone_bool
   {
     \bool_gset_true:N \g__block_label_standalone_bool
     \para_omit_indent:
     \box_use_drop:N \g__block_labels_box
     \par
   }
```

Do we need a `\nobreak` here or is this covered? check

check if @newlist could be replaced by something more general

Otherwise we signal that we are at the start and have a label dangling. The name `@newlist` is a bit unfortunate, but for now we keep this name.

```
      {
        \legacy_if_gset_true:n { @newlist }
        \legacy_if_gset_true:n { @inlabel }
      }
```

Do not break after the first line:

```
    \legacy_if_gset_true:n { @nobreak }
```

Then set up a special everypar to handle the dangling caption:

```
    \__block_debug_typeout:n{Set~ captioned~ block~ everypar \on@line }
    \cs_set_eq:NN \__block_everypar: \__block_captioned_everypar_std:
```

Finally, set up any declarations for the body of the environment:

```
    \l__block_body_decls_tl
  \__block_debug_typeout:n{template:thmstyle:std~end}
}
```

t/captionedtext/caption (*socket*)

```
\NewTaggingSocket{captionedtext/caption}{2}
```

why kernel?

```
\NewTaggingSocketPlug{captionedtext/caption}{kernel}
 {
   \tag_struct_begin:n{tag=\UseStructureName{block/theorem-like/caption}}
   #2
   \tag_struct_end:
 }
\AssignTaggingSocketPlug{captionedtext/caption}{kernel}
```

Here are the functions that are called when the corresponding name appears in the caption clist.

__block_do_title: Handle the title:

```
\cs_new_protected:Npn \__block_do_title: {
```

Check if there is a title.

```
  \tl_if_empty:NTF \l__block_title_tl
```

If the title is empty we drop accumulated but not yet typeset spaces:

```
    { \__block_drop_spaces: }
```

Otherwise we typeset the title, first inserting space (or spaces) that have been waiting.

```
    { \tag_socket_use:nnn {mc} {}{
        \__block_insert_spaces:
```

The title may have its own formatting:

```
        \__block_title_format:n \l__block_title_tl }
    }
}
```

(*End of definition for* `\__block_do_title:`.)

__block_do_note: Formatting of a note (if present) uses the same structure.

```
\cs_new_protected:Npn \__block_do_note: {
  \tl_if_novalue:oTF \l__block_note_tl
     { \__block_drop_spaces: }
     { \tag_socket_use:nnn {mc} {} {
         \__block_insert_spaces:
         \__block_note_format:n \l__block_note_tl }
     }
}
```

(*End of definition for* `\__block_do_note:`.)

__block_do_number: The number (if present) has a similar formatting but it uses an Lbl structure:

```
\cs_new_protected:Npn \__block_do_number: {
  \bool_if:NTF \l__block_numbered_bool
     { \tag_socket_use:nnn {struct-mc} {tag=\UseStructureName{block/theorem-like/label}}
         { \__block_insert_spaces:
           \__block_number_format:n {
             \use:c{ the \l__block_counter_tl } }
         }
     }
     { \__block_drop_spaces: }
}
```

(*End of definition for* `\__block_do_number:`.)

__block_do_punct: The punctuation is handled slightly differently. It unconditionally drops any dangling spaces whether or not it is empty:

```
\cs_new_protected:Npn \__block_do_punct: {
  \__block_drop_spaces:
  \tl_if_empty:NF \l__block_punct_tl
     { \tag_socket_use:nnn {mc} {}{
         \__block_punct_format:n \l__block_punct_tl }
     }
}
```

(*End of definition for* `\__block_do_punct:`.)

__block_do_space:
__block_insert_spaces:
__block_drop_spaces:
\g__block_collected_spaces_tl

What's still missing is what `space` should do. It simply adds a `\l__block_space_tl` to the `\g__block_collected_spaces_tl` tokenlist. This way the clist can contain `space,space,...` to indicate multiple spaces. The storage tokenlist is global as the functions are executed inside their own group, but the collected space is used outside of that group.

```
\cs_new_protected:Npn \__block_do_space: {
  \tl_gput_right:Nn \g__block_collected_spaces_tl \l__block_space_tl
}
```

So `\__block_insert_spaces:` is trivial, all we have to do is to insert the collected space and then clear the tokenlist

```
\cs_new_protected:Npn \__block_insert_spaces: {
  \g__block_collected_spaces_tl
  \tl_gclear:N \g__block_collected_spaces_tl
}
```

Even simpler is `\__block_drop_spaces:`

```
\cs_new_protected:Npn \__block_drop_spaces: {
  \tl_gclear:N \g__block_collected_spaces_tl
}
```

What remains is to declare the tokenlist.

```
\tl_new:N \g__block_collected_spaces_tl
```

(*End of definition for* `\__block_do_space:` *and others.*)

`captionedtext proof` (*templ.*) In case of the templates for proofs we do everything in a single template.

```
\DeclareTemplateCode{captionedtext}{proof}{4}
{
  ,title           = \l__block_title_tl
  ,punct           = \l__block_punct_tl
  ,caption-placement = {
     chained    = \bool_gset_false:N \g__block_label_standalone_bool
                  \bool_gset_false:N \g__block_label_unchained_bool
    ,unchained  = \bool_gset_false:N \g__block_label_standalone_bool
                  \bool_gset_true:N  \g__block_label_unchained_bool
    ,standalone = \bool_gset_true:N  \g__block_label_standalone_bool
                  \bool_gset_false:N \g__block_label_unchained_bool
   }
  ,before-hspace   = \l__block_caption_before_skip
  ,after-hspace    = \l__block_caption_after_skip
  ,caption-decls   = \l__block_caption_decls_tl
  ,title-format    = \__block_title_format:n
  ,punct-format    = \__block_punct_format:n
  ,body-decls      = \l__block_body_decls_tl
}
{
```

Display the template's arguments when tracing:

```
  \template_debug_typeout:n{~\space template:~ 'proof';~
                            arguments:~ \exp_not:o{\exp_after:wN |#1|#2|#3|#4|}}
```

Evaluate document-level key settings. As all given keys should be handled we use `\SetTemplateKeys` to raise an error if one or more are not recognized:

```
  \SetTemplateKeys{captionedtext}{proof}{#1}
```

By default the title is defined by the proof instance, but if the user provides an optional argument that optional argument overwrites the title (in contrast to theorem-like environments that use the optional argument to provide an additional note):

```
  \IfNoValueF {#3} { \tl_set:Nn \l__block_title_tl {#3} }
```

Now we prepare typesetting the title by placing it in the `\g__block_labels_box`:

```
  \hbox_gset:Nn \g__block_labels_box
    {
      \box_use_drop:N \g__block_labels_box % <- does nothing if there
                                           %    is no dangling label
      \l__block_caption_decls_tl
      \tag_socket_use:nnn {captionedtext/caption} {}
          {
            \skip_horizontal:n { \l__block_caption_before_skip }
```

`\__block_do_title:` and `\__block_do_punct:` unnecessarily call `\__block_drop_spaces:` but otherwise they do well, so …

```
            \group_begin: \__block_do_title: \group_end:
            \group_begin: \__block_do_punct: \group_end:
            \skip_horizontal:n { \l__block_caption_after_skip }
          }
    }
```

The remaining code is identical to the one in `thmstyle std`; for documentation see there:

```
  \bool_if:NTF \g__block_label_standalone_bool
     {
       \bool_gset_true:N \g__block_label_standalone_bool
       \para_omit_indent:
       \box_use_drop:N \g__block_labels_box
       \par
     }
     {
       \legacy_if_gset_true:n { @newlist }
       \legacy_if_gset_true:n { @inlabel }
     }
  \legacy_if_gset_true:n { @nobreak }
  \__block_debug_typeout:n{Set~ captioned~ block~ everypar \on@line }
  \cs_set_eq:NN \__block_everypar: \__block_captioned_everypar_std:
  \l__block_body_decls_tl
  \__block_debug_typeout:n{template:captionedtext:proof~end}
}
```

9.5 Tagging support commands

In this section we provide code to the various kernel hooks to support the tagging of different displayblock environments.

`\__block_beginpar_vmode:`

When a block starts out in vertical mode, i.e., is not yet part of a paragraph, we have to start a paragraph structure. However, this is not the case if we are already flattening paragraphs, thus in this case we do nothing. We also do nothing if `@endpe` is currently true, because that means we are right now just after the end of a `blockenv` and in the process of looking if we have to end the current `<text-unit>`, i.e., it is already open. The command is mapped to `\__kernel_displayblock_beginpar_vmode:` in various tagging recipes. It is also used in the math code!

```
\cs_set:Npn \__block_beginpar_vmode: {
  \__block_debug_typeout:n
    { @endpe = \legacy_if:nTF { @endpe }{true}{false} \on@line }
  \legacy_if:nTF { @endpe }
    {
      \legacy_if_gset_false:n { @endpe }
    }
```

We test for `<2` because the first flattened environment has to surround itself with a `<text-unit>`. Only any inner ones then have to avoid adding another `<text-unit>`.

```
    {
      \int_compare:nNnT \l__tag_block_flattened_level_int < 2
          {
```

```
            \UseTaggingSocket{para/semantic/begin}
              { \__tag_para_main_store_struct: }
          }
      }
  }
```

(*End of definition for* `\__block_beginpar_vmode:`.)

`\__block_beginpar_hmode:N` If the block is already part of a part of a paragraph, i.e., when it has some text directly in front, then the first thing to do is to return to vertical mode. However, that should be done without inserting a paragraph end tag, so before calling `\par` to do its normal work, we disable paragraph tagging and restarting afterwards again. The argument to this config point simply gobbles the `\par` following it in the code above (which is used when there is no tagging going on.
The command is mapped to `\__kernel_displayblock_beginpar_hmode:w` in various tagging recipes.

```
\cs_set:Npn \__block_beginpar_hmode:N #1
  {
    \tag_mc_end:
    \__tag_gincr_para_end_int:
    \__block_debug_typeout:n{increment~ /P \on@line }
    \bool_if:NT \l__tag_para_show_bool
      { \tag_mc_begin:n{artifact}
        \rlap{\color_select:n{red}\tiny\ \int_use:N\g__tag_para_end_int}
        \tag_mc_end:
      }
    \tag_struct_end:
    \tagpdfparaOff \par \tagpdfparaOn
  }
```

(*End of definition for* `\__block_beginpar_hmode:N`.)

Paragraph tagging is mainly done using the paragraph hooks. The code is in `lttagging.dtx`.

`/block/startpara/direct` (*socket*) A tagging socket to start a paragraph structure. It takes an argument (which is only used in debugging) that should be gobbled if tagging is not active. Not yet in lttagging (name and function should be reviewed).
This is a similar code to the one used in the para/begin hook but without testing `@endpe`. This is not needed in the standalone case and wrong inside lists.
This code is used in various places and should be a dummy if tagging is not active.

```
\socket_if_exist:nF {tagsupport/block/startpara/direct}
 {
   \NewTaggingSocket {block/startpara/direct}{1}
 }
```

`default` (*plug*)

```
\NewTaggingSocketPlug{block/startpara/direct}{default}
 {
   \bool_if:NF \l__tag_para_flattened_bool
      {
```

```
        \UseTaggingSocket{para/semantic/begin}
          { \__tag_para_main_store_struct: }
      }
  \__tag_gincr_para_begin_int:
  \__block_debug_typeout:n{increment~ P \on@line }
  \tag_struct_begin:n
      {
         tag=\l__tag_para_tag_tl
        ,attribute-class=\l__tag_para_attr_class_tl
      }
  \__tag_check_para_begin_show:nn {green}{#1}
  \tag_mc_begin:n {}
}
\AssignTaggingSocketPlug{block/startpara/direct}{default}
```

The para/end hook code is in lttagging. Currently we still need to remove the tagpdf chunk to avoid that the socket is added twice. We add empty chunks to avoid warning messages from code parts trying to remove the chunks.

```
\AddToHook{para/end}[tagpdf]{}
\RemoveFromHook{para/end}[tagpdf]
\AddToHook{para/end}{}

\def\PARALABEL{NP-}
```

`port/kernel/endpe/vmode` (*socket*) A tagging socket which ends a structure. Used in `\begin` and `\para_end:`. Not yet in lttagging (name and function should be reviewed).

```
\socket_if_exist:nF {tagsupport/kernel/endpe/vmode}
{
  \NewTaggingSocket {kernel/endpe/vmode}{0}
}
```

`default` (*plug*)

```
\NewTaggingSocketPlug{kernel/endpe/vmode}{default}
{
  \if@endpe \ifvmode
    \bool_if:NT \l__tag_para_bool
     {
       \bool_if:NF \l__tag_para_flattened_bool
         {

             \UseTaggingSocket{para/semantic/end}{}
         }
```

`\@endpefalse` is needed by `\para_end:`, see test tagging-0097.

```
      \@endpefalse
    }
  \fi \fi
}
\AssignTaggingSocketPlug{kernel/endpe/vmode}{default}
```

`\para_end:` If we see a `\par` in vmode and a `<text-unit>` is still open we need to close that. For this we check if a request for @endpe was made (but the `\par` redefinition got lost due to (bad?) coding).

```
\cs_set_protected:Npn \para_end: {
```

```
  \scan_stop:
  \mode_if_horizontal:TF {
    \mode_if_inner:F {
         \tex_unskip:D
         \hook_use:n{para/end}
         \@kernel@after@para@end
         \mode_if_horizontal:TF {
           \if_int_compare:w 11 = \tex_lastnodetype:D
             \tex_hskip:D \c_zero_dim
           \fi:
           \tex_par:D
           \hook_use:n{para/after}
           \@kernel@after@para@after
         }
         { \msg_error:nnnn { hooks }{ para-mode }{end}{horizontal} }
    }
  }
  {
```

TODO 2025-07-01. This is not exactly as before, this doesn't insert an `\endpefalse` when tagging is active. Check if this a problem.

```
    \UseTaggingSocket{kernel/endpe/vmode}%
    \tex_par:D
  }
}
```

Now reset LaTeX 2_ε functions to use the changed `\para_end:` [**TODO:** Need to check if `\@@par` is ever used in a way that the vmodetagging hook is needed.]

```
\cs_set_eq:NN \par     \para_end:
\cs_set_eq:NN \@@par \para_end:
\cs_set_eq:NN \endgraf \para_end:
```

(*End of definition for* `\para_end:`*. This function is documented on page 38.*)

`\begin` We need to do a little more than canceling `@endpe` now.

```
\protected\def\begin#1{%
  \UseHook{env/#1/before}%
  \@ifundefined{#1}%
    {\def\reserved@a{\@latex@error{Environment~#1~undefined}\@eha}}%
    {\def\reserved@a{\def\@currenvir{#1}%
        \edef\@currenvline{\on@line}%
        \@execute@begin@hook{#1}%
        \csname #1\endcsname}}%
  \@ignorefalse
  \begingroup
    \UseTaggingSocket{kernel/endpe/vmode}%
    \reserved@a}
```

(*End of definition for* `\begin`*. This function is documented on page 38.*)

`\__kernel_list_label_after:n` If starting the text-unit/text tags got delayed because of a pending label we have to do it after the label got typeset. TODO: it should do nothing without tagging that's why there is a test, this should be better hidden in a tagging socket, but it is not quite clear how to do this.

internally this is now a tagging socket, why the outer test then?

```
\cs_new_protected:Npn \__kernel_list_label_after:n #1 {
   \bool_lazy_and:nnT { \tag_if_active_p: } {\l__tag_para_bool }
```

```
      {
        \tag_socket_use:nn {block/startpara/direct} { #1 }
      }
}
```

(*End of definition for* `\__kernel_list_label_after:n`.)

`\__block_inner_begin:` Start a block that has an inner structure if it isn't also a list. This command is tagging specific, it is mapped to `\__kernel_displayblock_begin:` in some tagging recipes.

```
\cs_new_protected:Npn \__block_inner_begin: {
  \tagstructbegin{tag=\l__block_tag_inner_tag_tl}
}
```

(*End of definition for* `\__block_inner_begin:`.)

`\__block_inner_end:` End a block (which isn't also a list). This command is tagging specific, it is mapped to `\__kernel_displayblock_end:` in some tagging recipes.

```
\cs_new_protected:Npn \__block_inner_end: {
  \__block_debug_typeout:n{block-end \on@line}
  \legacy_if:nT { @endpe }
    {
      \UseTaggingSocket{para/semantic/end}
        { \__block_debug_typeout:n{close~ /text-unit \on@line}}
    }
  \tagstructend        % end inner structure
}
```

(*End of definition for* `\__block_inner_end:`.)

9.5.1 List tags

```
\tl_new:N  \l__tag_L_tag_tl
\tl_set:Nn \l__tag_L_tag_tl {L}

\tl_new:N\l__tag_L_attr_class_tl
\tl_set:Nn \l__tag_L_attr_class_tl {list}

\tagpdfsetup
  {
    ,role/new-attribute = {itemize}
                          {/O /List /ListNumbering/Unordered}
    ,role/new-attribute = {enumerate}
                          {/O /List /ListNumbering/Ordered}
    ,role/new-attribute = {description}
                          {/O /List /ListNumbering/Description}
```

Initially, we had `/None` for the basic `list` environment, but that is not allowed in PDF/UA-2 if the list contains any Lbl tags. So now we default to `Unordered`.

```
    ,role/new-attribute = {list}{/O /List /ListNumbering/Unordered}
  }
```

`\__block_list_begin:` Start a list ...This command is tagging specific, it is mapped to `\__kernel_displayblock_begin:` in a tagging recipe.

```
\cs_set:Npn \__block_list_begin: {
  \tagstructbegin
      {
         tag=\l__tag_L_tag_tl
```

```
          ,attribute-class=\l__tag_L_attr_class_tl
        }
}
```
(*End of definition for* `\__block_list_begin:`.)

`\__block_list_item_begin:` Start tagging a list item. This command is tagging specific, it is mapped to `\__kernel_list_item_begin:` in a tagging recipe.

```
\cs_set:Npn \__block_list_item_begin: {
  \tagstructbegin{tag=\UseStructureName{block/list/item}}
}
```
(*End of definition for* `\__block_list_item_begin:`.)

`\__block_list_item_end:` When a list item ends we have to close `<itembody>` and `<LI>` but also a `<text>` in the special case that the item material ends in a list (identifiable via `@endpe`). This command is tagging specific. This command is copied to `\__kernel_list_item_end:` in the list recipe.

```
\cs_set:Npn \__block_list_item_end: {
  \legacy_if:nT { @endpe }
    {
      \UseTaggingSocket{para/semantic/end}{}
%      \__block_debug_typeout:n{Structure-end~ P~ at~ item-end \on@line }
    }
  \tagstructend \tagstructend   % end \UseStructureName{block/list/body}, LI
}
```
(*End of definition for* `\__block_list_item_end:`.)

`\__block_list_end:` Finally, at the list end we have to close the open `<itembody>`, `<LI>`, `<L>`, and possibly a `<text>` if the last item ends with a list. However, if the user forgot to add an `\item` then there will be no `<LI>` and `<itembody>` open, so we check for the status of `@newlist`. The corresponding no-item error was generated earlier outside the tagging code.
One could argue that it doesn't matter if the tagging is wrong after a `\@noitemerr` was issued. However, there is one case where it isn't an error: In the `thebibliography` environment (which is internally a list) it is often the case that documents start out with an empty environment, not containing any `\bibitem`s. For that reason `\@noitemerr` is redefined inside that environment to only produce a warning; hence we have to produce correct tag structures in that case. This command is tagging specific. This command is copied to `\__kernel_displayblock_end:` in the list recipe.

```
\cs_new_protected:Npn \__block_list_end: {
```
If `@newlist` is true (i.e., when we have an error or warning situation) there is not much to close.

```
  \legacy_if:nF { @newlist }
    {
      \legacy_if:nT { @endpe }
        {
          \UseTaggingSocket{para/semantic/end}
           {\__block_debug_typeout:n{Structure-end~ text-unit~ at~ list-end \on@line }}
        }
      \tagstructend\tagstructend  % end \UseStructureName{block/list/body}, LI
    }
  \tagstructend               % end L
}
```

(*End of definition for* `\__block_list_end:`.)

End of tagging related declarations.

9.5.2 Tagging recipes

`tagsupport/block/recipe` (*socket*) A tagging socket to call the tagging recipe. Declared in lttagging.

```
\socket_if_exist:nF {tagsupport/block/recipe}
 {
   \NewTaggingSocket{block/recipe}{1}
 }
```

`default` (*plug*)

```
\NewTaggingSocketPlug{block/recipe}{default}
 {
   \use:c { __block_recipe_#1: }
 }
\AssignTaggingSocketPlug{block/recipe}{default}
```

`\__block_recipe_basic:` The `basic` recipe simply ensures that the block is inside a `<text-unit>` structure and if necessary starts one. When the block ends and is followed by a blank line the `<text-unit>` structure is closed too, otherwise it remains open and further text starts with just a `<text>` structure.
There is otherwise no inner structure so `\__kernel_displayblock_begin:` and `\__kernel_displayblock_end:` do nothing—`blockenvs` with inner structure use the `standard` or `list` recipe instead.

```
\cs_new_protected:Npn \__block_recipe_basic: {
  \cs_set_eq:NN \__kernel_displayblock_beginpar_hmode:w
                                          \__block_beginpar_hmode:N
  \cs_set_eq:NN \__kernel_displayblock_beginpar_vmode:
                                          \__block_beginpar_vmode:
  \let \__kernel_displayblock_begin:      \prg_do_nothing:
  \let \__kernel_displayblock_end:        \prg_do_nothing:
```

End environment `\par` handling:

```
  \socket_assign_plug:nn{block/endpe}{on}
}
```

(*End of definition for* `\__block_recipe_basic:`.)

`\__block_recipe_standalone:`

The `standalone` recipe produces a block that ensures that a previous `<text-unit>` ends and that after the block a new `<text-unit>` starts.

```
\cs_new_protected:Npn \__block_recipe_standalone: {
  \cs_set_eq:NN \__kernel_displayblock_beginpar_hmode:w
                                          \prg_do_nothing:
  \cs_set_eq:NN \__kernel_displayblock_beginpar_vmode:
                                          \prg_do_nothing:
  \cs_set_eq:NN \__kernel_displayblock_begin: \__block_inner_begin:
  \cs_set_eq:NN \__kernel_displayblock_end:   \__block_inner_end:
```

End environment `\par` handling:

```
  \socket_assign_plug:nn{block/endpe}{off}
```

```
  \tl_if_empty:NTF \l__block_tag_name_tl
     { \tl_set:Nn    \l__block_tag_inner_tag_tl {Sect}            }
     { \tl_set_eq:NN \l__block_tag_inner_tag_tl \l__block_tag_name_tl }
}
```

(*End of definition for* `\__block_recipe_standalone:`.)

`\__block_recipe_standard:` The `standard` recipe does the following:

- surround the block with a `<text-unit>` structure if not already in a `<text-unit>`. In the latter case end the MC and the `<text>` but leave the `<text-unit>` open.

  If we are producing flattened paragraphs, just close any `<text>` but do not open a `<text-unit>`.

- Then open an new (inner) structure (by default `<Div>` but typically the one specified on the instance).

- At the end of the block close the inner structure (`<Div>` or explicit one) but leave the `<text-unit>` open to be either continued or closed due to a following `\par`.

```
\cs_new_protected:Npn \__block_recipe_standard:
{
  \cs_set_eq:NN \__kernel_displayblock_beginpar_hmode:w
                                          \__block_beginpar_hmode:N
  \cs_set_eq:NN \__kernel_displayblock_beginpar_vmode:
                                          \__block_beginpar_vmode:
  \cs_set_eq:NN \__kernel_displayblock_begin: \__block_inner_begin:
  \cs_set_eq:NN \__kernel_displayblock_end:   \__block_inner_end:
```

End environment `\par` handling:

```
  \socket_assign_plug:nn{block/endpe}{on}

  \tl_if_empty:NTF \l__block_tag_name_tl
     { \tl_set:Nn    \l__block_tag_inner_tag_tl {Div}             }
     { \tl_set_eq:NN \l__block_tag_inner_tag_tl \l__block_tag_name_tl }
}
```

(*End of definition for* `\__block_recipe_standard:`.)

`\l__block_tag_inner_tag_tl` The tag name that is used if the block has an inner structure.

```
\tl_new:N \l__block_tag_inner_tag_tl
```

(*End of definition for* `\l__block_tag_inner_tag_tl`.)

`\__block_recipe_list:` The `list` recipe does the following.

- It opens a `<text-unit>`-structure or keeps the current one open (only closing the MC).

- It then starts a new structure role-mapped to L-structure and arranges for handling list items, e.g., Li, itemlabel and itembody structures.

- At the end it closes open list structures as needed but keeps the `<text-unit>`-structure open to continue the paragraph after the list, if necessary.

```
\cs_new_protected:Npn \__block_recipe_list:
{
  \cs_set_eq:NN \__kernel_displayblock_beginpar_hmode:w
                                              \__block_beginpar_hmode:N
  \cs_set_eq:NN \__kernel_displayblock_beginpar_vmode:
                                              \__block_beginpar_vmode:
  \cs_set_eq:NN \__kernel_displayblock_begin: \__block_list_begin:
  \cs_set_eq:NN \__kernel_displayblock_end:   \__block_list_end:
```

The next two lines could be done globally, because they are only called if we do have `\item`s, i.e., if we are in a list. It is therefore also not necessary to reset them in other recipes (right now—this may change if we get more templates (like inline lists)).

```
  \cs_set_eq:NN \__kernel_list_item_begin:    \__block_list_item_begin:
  \cs_set_eq:NN \__kernel_list_item_end:      \__block_list_item_end:
```

End environment `\par` handling:

```
  \socket_assign_plug:nn{block/endpe}{on}
```

Handle the tag name and attribute classes using the key values from the current list instance.

```
  \tl_if_empty:NTF \l__block_tag_name_tl
     { \tl_set:Nn    \l__tag_L_tag_tl {L}               }
     { \tl_set_eq:NN \l__tag_L_tag_tl \l__block_tag_name_tl }
  \tl_if_empty:NTF \l__block_tag_class_tl
     { \tl_set:Nn    \l__tag_L_attr_class_tl {}                 }
     { \tl_set_eq:NN \l__tag_L_attr_class_tl \l__block_tag_class_tl }
}
```

(*End of definition for* `\__block_recipe_list:`.)

```
⟨/package-start⟩
```

10 Support code for document-level block environments

10.1 Verbatim-like environments

10.1.1 Helper commands for `verbatim` and `verbatim*`

`\legacyverbatimsetup`

This code is called as part of the `final-code` of the `blockenv` instance and sets up the special conventions needed for `verbatim` environments. We pass one argument to differentiate between visible and invisible spaces.

This code resembles the LaTeX 2_ε verbatim implementation with a slight twist: in LaTeX 2_ε each code line was a paragraph using `\leftskip=\@totalleftmargin`. This was possible because the whole environment was implemented as a trivlist. As this is no longer the case setting `\leftskip` would alter the layout of a surrounding list. So instead we need to make sure that the paragraph end is executed in a group so that any parshape setup is preserved.

```
⟨*package-finish⟩
⟨@@=⟩
\def\legacyverbatimsetup #1 {%
  \language\l@nohyphenation
  \@tempswafalse
```

```
\def\par{%
  \if@tempswa
    \leavevmode \null {\@@par}\penalty\interlinepenalty
  \else
    \@tempswatrue
    \ifhmode{\@@par}\penalty\interlinepenalty\fi
    \fi
```

might also need a hook not just a socket

Do something at the very beginning of each verbatim line:

```
    \UseSocket{verbatim/startline}%
  }%
  \let\do\@makeother \dospecials
  \obeylines \verbatim@font \@noligs
  \everypar \expandafter{\the\everypar \unpenalty}%
```

Should next line be hidden in a tagging socket?

```
  \frenchspacing
  \AssignStructureRole {para/textblock}%
                       {\UseStructureName{block/verbatim/codeline}}%
```

If the argument is neither `visible` nor `invisible` nothing will happen—tough.

```
  \use:c { @setupverb #1 space }
  \@vobeyspaces
}
```

(*End of definition for* `\legacyverbatimsetup`. *This function is documented on page 37.*)

`verbatim/startline` (*socket*)

We might also need a hook for line numbers.

A socket that is executed at the start of each verbatim line, to be used, for example, to check for %␣ when the doc package is active.

```
\NewSocket{verbatim/startline}{0}
```

`\@setupverbinvisiblespace`

In the pdfTEX engine we need to use `\pdffakespace` chars for the invisible spaces. In luatex we do not want this as it would lead to doubling the number of real space chars. In dvi-mode we do not want that either: with pdftex it would error, with xetex it does nothing.

```
⟨@@=block⟩
\newcommand\@setupverbinvisiblespace{}
\bool_lazy_or:nnF
 { \sys_if_engine_luatex_p: }
 { \sys_if_output_dvi_p: }
 {
   \renewcommand\@setupverbinvisiblespace
     {\def\@xobeysp{\nobreakspace\pdffakespace}}
 }
```

(*End of definition for* `\@setupverbinvisiblespace`. *This function is documented on page 37.*)

The command `\@setupverbvisiblespace` is already defined in the kernel.

10.1.2 Helper commands for `alltt` and `alltt*`

`\legacyallttsetup`

The `alltt` environment also needs some special setup. We can reuse `\legacyverbatimsetup` but we have to take out `\`, `{`, and `}` from `\dospecials` as they should remain available with their normal catcodes and adjust `'` inside math. This is lifted straight from the original package code.

```
⟨@@=⟩
\ExplSyntaxOff
```

```
\def\legacyallttsetup #1{%
  \let\org@prime~%
  \everymath\expandafter{\the\everymath
    \catcode`\'=12 \let~\org@prime}%
  \everydisplay\expandafter{\the\everydisplay
    \catcode`\'=12 \let~\org@prime}%
```

This alters `\dospecials`:

```
  \let\org@dospecials\dospecials
  \g@remfrom@specials{\\}%
  \g@remfrom@specials{\{}%
  \g@remfrom@specials{\}}%
```

Then call `\legacyverbatimsetup`:

```
  \legacyverbatimsetup {#1}%
```

And afterwards restore `\dospecials`:

```
  \let\dospecials\org@dospecials
}
```

Copied from alltt.

\g@remfrom@specials

```
\def\g@remfrom@specials#1{%
  \def\@new@specials{}%
  \def\@remove##1{%
    \ifx##1#1\else
    \g@addto@macro\@new@specials{\do ##1}\fi}%
  \let\do\@remove\dospecials
  \let\dospecials\@new@specials
}
```

```
\ExplSyntaxOn
⟨@@=block⟩
```

(*End of definition for* `\legacyallttsetup` *and* `\g@remfrom@specials`*. These functions are documented on page 37.*)

10.1.3 Helper command for legacy `list` environment

\legacylistsetup

And here is the extra code for use in the list instance setup in the key `legacy-code`:

```
\cs_new_protected:Npn \legacylistsetup {
```

Reset values to defaults:

```
    \dim_zero:N \listparindent
    \dim_zero:N \rightmargin
    \dim_zero:N \itemindent
```

By default a `list` environment is not numbered, but this happens already in the block template.

```
%    \tl_set:Nn \@listctr {}
%    \legacy_if_set_false:n { @nmbrlist } % needed if lists are nested
```

By default there is a simple definition for `\makelabel`. It can be overwritten in the second mandatory argument to the list environment (stored in `\l__block_legacy_env_params_tl`) and is used if the instance sets the compatibility key to true.

```
   \let\makelabel\@mklab % TODO: customize
```

Now we use the argument with parameter settings to update some or all of the above defaults (this holds whatever was put into the second argument to the `list` environment):

```
\l__block_legacy_env_params_tl
```

As we don't know much about this list we can only make a guess about the nature of the list and the setting of the tag name (default `list` role-mapped to `<L>`) and any tag attributes may have to be overwritten in the optional key/value argument. But we do have some hints to play with.

```
\legacy_if:nTF { @nmbrlist }
  { \tl_set:Nn \l__tag_L_attr_class_tl {enumerate} }   % numbered list
  { \tl_if_empty:NTF \@itemlabel
      { \tl_set:Nn \l__tag_L_attr_class_tl {list}    } % no label
      { \tl_set:Nn \l__tag_L_attr_class_tl {itemize} } % unnumbered,
                                                       %   unordered
  }
}
```

(*End of definition for* `\legacylistsetup`. *This function is documented on page 37.*)

`\l__block_legacy_env_params_tl` The token list in which the declarations from the second argument of `list` are temporarily stored. This is then used in `\legacylistsetup`.

```
\tl_new:N\l__block_legacy_env_params_tl
```

(*End of definition for* `\l__block_legacy_env_params_tl`.)

10.2 Theorem-like environments

Theorem-like environments are defined in LaTeX with the help of `\newtheorem` declarations. Internally they used a list with a single item. Using lists was convenient back then, but in a tagged document you end up with a strange structure. We therefore alter the mechanism.

10.2.1 Declarations for theorem-like environments

`\newtheorem` We reimplement the extended amsthm version of the declaration which also supports a star form indicating that this theorem-like environment should not be numbered.

```
\RenewDocumentCommand \newtheorem { s m o m o } {
```

Is the environment definable at all? If not there is not much point in continuing.

```
  \expandafter\@ifdefinable\csname #2\endcsname
   {
```

undefined the old internal commands?

If a star was given then there is no need to set up a counter for this environment. Otherwise we do what LaTeX2e did, except that we do all the variations in one go, rather than using `\@ynthm`, `\@xnthm`, and `\@othm`.

```
    \IfBooleanF #1
      {
        \IfNoValueTF {#3}
```

If there was no counter to use (#2) then we set up a counter with the same name as the environment (#2).

```
          {
            \@definecounter {#2}
```

If there was no “counter within” the counter representation is simple, otherwise we build it up from the two counters:

```
                \IfNoValueTF {#5}
                  {  % @ynthm
                    \tl_gset:ce { the #2 }
                       {
                         \@thmcounter{#2}
                       }
                  }
                  {  % @xnthm
                    \@newctr{#2}[#5]
                    \tl_gset:ce { the #2 }
                       {
                         \expandafter\noexpand\csname the#5\endcsname
                         \@thmcountersep
                         \@thmcounter{#2}
                       }
                  }
              }
              {  % @othm
```

If we should reuse an existing counter (`#3` was given) we check that this counter actually exists and if so use it:

```
                \@ifundefined{c@#3}
                   { \@nocounterr{#3} }
                   {
                     \newcounteralias{#2}{#3}
                   }
              }
        }
```

With the counter defined we are ready to declare the environments. There is a slight complication though: the “theorem-like” environments have an optional argument which contains a possible note, but now we also want to use the first optional argument to hold a key/value list with parameter settings. We therefore define this argument via `={note}o` so that a simple note, if given is assigned to a `note` key. Further processing is then delegated to the command `\ParseLaTeXeTheoremlike` which, after sorting out the argument situation, eventually calls `\BlockEnv`.

```
      \NewDocumentEnvironment{#2}{ ={note}o }
         { \ParseLaTeXeTheoremlike {#2} \BooleanFalse {##1} }
         { \BlockEnvEnd }
```

The starred form of the environment suppresses the number so we pass it `\BooleanTrue`, otherwise it is identical to the previous definition.

```
      \NewDocumentEnvironment{#2*}{ ={note}o }
         { \ParseLaTeXeTheoremlike {#2} \BooleanTrue {##1} }
         { \BlockEnvEnd }
```

Now it is about time to provide all necessary template instances. They depend on the `\theoremstyle` specified by the user and possibly by a `\swapnumbers` declaration. We start by checking the requested `\theoremstyle` (if none was given then `plain` is the default and if it is an unknown name we also revert to `plain` after issuing a warning).

```
      \IfInstanceExistsF{thmstyle}{\l__block_thmstyle_tl}
                         { \@latex@warning{Unknown~ theoremstyle~
```

```
                                '\l__block_thmstyle_tl'~ using~ 'plain'}
                            \theoremstyle {plain}
                          }
```

So now we know that `\l__block_thmstyle_tl` holds a valid style. What we don't know is whether or not there are special `block` instances that go with that style (it might be a style that reuses the `thm-`⟨*style*⟩`block-...` instances).

```
\IfInstanceExistsTF{block} { thm- \l__block_thmstyle_tl -1 }
   { \__block_debug_typeout:n{...~ style~ \l__block_thmstyle_tl\space exists } }
   { \__block_debug_typeout:n{...~ style~ \l__block_thmstyle_tl\space
       does~ not ~exist;~ 'plain'~ used } }
```

So here is the `blockenv` instance for the new "theorem-like" environment. It uses a `captionedtext` instance for the `inner-instance` which we also have to declare.

```
\DeclareInstance{blockenv}{#2}{std}
  {
     name                 = theorem-like
    ,tag-name             = \UseStructureName{block/theorem-like}
    ,tag-attr-class       =
    ,tagging-recipe       = standalone
    ,inner-level-counter  =
    ,transparent-level    = true
    ,legacy-code          =
```

What block instance to use is determined by checking if a special one exists or whether we should use `plain`:

```
    ,block-instance:e     = thm-
                             \IfInstanceExistsTF{block}
                                {thm- \l__block_thmstyle_tl -1}
                                { \l__block_thmstyle_tl } { plain }
    ,inner-instance-type  = captionedtext
    ,inner-instance       = #2
```

decide

By default the body text is justified, but perhaps we should not set anything here and use whatever is current.

```
    ,para-instance        = justify
  }
```

The `captionedtext` instance is simple: the counter, if present, is either argument `#2` or `#3`; the `title` receives argument `#4`, and the `style` to use is stored in `\l__block_thmstyle_tl`. If `\swapnumbers` was requested we use a style variant with the suffix `-swap` appended.

```
\DeclareInstance{captionedtext}{#2}{thmlike}
  {
```

The counter to use is either none or `#2` based on the arguments given:

```
    ,counter:e = \IfBooleanF #1 { #2 }
    ,title     = #4
    ,style:e   = \l__block_thmstyle_tl
                 \bool_if:NT \l__block_swap_number_bool {-swap}
  }
```

We already know that the style `\l__block_thmstyle_tl` exists, since we have tested that earlier, but we don't know if that is also true for the `-swap` variant. So we have to check that and declare it if necessary.

```
\bool_if:NT \l__block_swap_number_bool {
```

```
      \IfInstanceExistsF{thmstyle}{\l__block_thmstyle_tl -swap}
      {
```

If it doesn't exist we first make a copy of the base instance.

```
        \DeclareInstanceCopy{thmstyle}
                            {\l__block_thmstyle_tl -swap}
                            {\l__block_thmstyle_tl}
```

Then we retrieve the value of the `order` key from that instance which is a clist.

```
        \clist_set:Ne \l__block_order_clist
            { \InstanceValue { thmstyle }
                             { \l__block_thmstyle_tl }
                             { order }
            }
```

Then we step through this clist and build a new one in `\l__block_tmp_clist` with the `title` and `number` swapped. That is done under the assumption that both actually exist in the clist which would be the case if the instance was declared with `\newtheoremstyle`, i.e., for legacy setups.

```
        \clist_clear:N \l__block_tmp_clist
        \clist_map_inline:Nn \l__block_order_clist
          {
            \clist_put_right:Ne \l__block_tmp_clist {
              \str_case:nnF {##1}
                { {title} {number}
                  {number} {title} }
                {##1}
            }
          }
```

Once that is done we put the new value for `order` in the new instance.

```
        \EditInstance {thmstyle}{\l__block_thmstyle_tl -swap}
          { order:e = \l__block_tmp_clist }
      }
    }
  }
}
```

(*End of definition for* `\newtheorem`.)

`\ParseLaTeXeTheoremlike` The arguments to `\ParseLaTeXeTheoremlike` are as follows:

#1: instance name to use (of type "blockenv")

#2: unnumbered? boolean normally provided by using the star form of the environment

#3: key/val for layout adjustments settings provided in the optional argument of the "theorem-like" environment. If a note to the theorem was given in that argument, then it has been turned into `note...=`

To be able to pick up the `note`, if provided, we make the following declaration:

```
\keys_define:nn {blockenv} {
  , note        .tl_set:N  = \l__block_note_tl
  , note        .groups:n  = { interface }
}
```

```
\cs_new_protected:Npn \ParseLaTeXeTheoremlike #1 #2 #3 {
%
%  \__block_debug_typeout:n{Parse~#1.~ Arguments~ found:~ \IfBooleanT{#2}{*}
%                \IfValueT{#3}{[\exp_not:n{#3}]}}
%
```

Normally, no note is provided, so that's the starting point:

```
  \tl_set:Nn \l__block_note_tl { \NoValue }
```

Then we check `#3` if it contains a key/val list containing `note`. This then sets `\l__block_note_tl` and puts all other key/vals in `\l__block_instance_keys_tl`. Otherwise the complete key/val list ends up there.

```
  \IfNoValueTF { #3 }
               { \tl_clear:N \l__block_instance_keys_tl }
               { \keys_set_groups:nnnN {blockenv} {interface} { #3 }
                                       \l__block_instance_keys_tl        }
```

We are now ready to invoke the `blockenv` instance for `#1` but we need to expand some of the values first, hence the `\use:e`.

```
  \use:e {
    \exp_not:N \BlockEnv { #1 }
        { \exp_not:o \l__block_instance_keys_tl }
        { #2 }
        { \exp_not:o \l__block_note_tl }
  }
```

We don't have any use for the last argument (the sub-caption) in the standard setup, so we pass `\NoValue`.

```
        \NoValue
}
```

(*End of definition for* `\ParseLaTeXeTheoremlike`*. This function is documented on page* ??.)

`\swapnumbers` Beside declaring theorem-like environments with `\newtheorem` and `\theoremstyle`, the amsthm package also introduced `\swapnumbers` to swap title and number in all environments (because that is a common requirement). The new implementation supports this approach as well.
It is implemented with a boolean `\l__block_swap_number_bool` which is toggled by `\swapnumbers`.

```
\bool_new:N \l__block_swap_number_bool
\cs_new_protected:Npn \swapnumbers { \bool_set_inverse:N \l__block_swap_number_bool }
```

(*End of definition for* `\swapnumbers`*. This function is documented on page* ??.)

`\newtheoremstyle` The `\newtheoremstyle` declaration was originally provided in the amsthm package. It has 9 mandatory arguments that sets some aspects of theorem styles. We map those to the template mechanism and generate `thmstyle` instances from them.
`\newtheoremstyle` has a bunch of argument conventions that haven't been fully implemented yet, e.g., `#8` can be a blank (meaning normal word space or `\newline`) or a skip. Those should eventually also be covered.

extend

```
\cs_set_protected:Npn \newtheoremstyle #1#2#3#4#5#6#7#8#9 {
```

First we build a `thmstyle` instance:

```
\DeclareInstance{thmstyle}{#1}{std}{
  ,caption-decls   = {#4}
  ,before-hspace:e = \tl_if_empty:nTF{#5}{0pt}{#5}
  ,body-decls      = {#6}
  ,punct           = {#7}
```

This setting doesn't cover all syntax possibilities.

```
  ,after-hspace:e  = \tl_if_empty:nTF{#8}{0pt}
                       {\tl_if_blank:nTF{#8}{3.3pt}{#8}}
}
```

If `#2` or `#3` are not empty we also have to set up a `block` instance to account for the fact that special vertical spacing is requested:

```
\tl_if_empty:nF { #2#3 }
  {
    \DeclareInstance{block}{thm-#1-1}{std}{
      ,begin-vspace:e = \tl_if_empty:nTF{#2}{0pt}{#2}
      ,end-vspace:e   = \tl_if_empty:nTF{#3}{0pt}{#3}
      ,left-margin    = 0pt
      ,para-indent    = \parindent
      ,para-vspace    = \parskip
    }
```

As elsewhere we provide two levels.

```
    \DeclareInstanceCopy{block}{thm-#1-2}{thm-#1-1}
  }
```

More complicated is argument `#9`. If not empty it can contain `\thmname`, `\thmnumber`, and/or `\thmnote` to define the layout of the theorem caption. All the spacing has to be given inside the arguments of these commands which means that this doesn't work together with `\swapnumbers`, but this is the way amsthm was defined. If the instances are manually defined then it is easy to make them work with and without a `\swapnumbers` declaration. So basically, this here is good for a subset of cases and for backwards compatibility with amsthm.
The approach used doesn't cover all circumstances, e.g., if the argument contains low-level programming on top of the interface commands that the translation below will fail, but most existing code should work and the rest would need a replacement using instances that are directly set up.

```
\tl_if_empty:oF { \exp_not:n{#9} }
  {
```

Give special definitions for the commands and then expand `#9` and use the result to edit the instance we defined earlier.

```
    \cs_set:Npn \thmname   ##1 {title-format={{\exp_not:n{##1}}},}
    \cs_set:Npn \thmnumber ##1 {number-format={{\exp_not:n{##1}}},}
    \cs_set:Npn \thmnote   ##1 {note-format={{\exp_not:n{##1}}},}
    \cs_set:Npn \__block_tmp:w ##1##2##3 {
      \exp_args:Nnne \EditInstance{thmstyle}{#1}{#9}}
    \__block_tmp:w {##1} {##1} {##1}
```

We then also use the commands to deduce a suitable `order` and put that into the instance as well.

```
    \cs_set:Npn \thmname   ##1 {title,}
    \cs_set:Npn \thmnumber ##1 {number,}
```

```
      \cs_set:Npn \thmnote   ##1 {punct,note}
      \cs_set:Npn \__block_tmp:w##1##2##3 {
        \exp_args:Nnne \EditInstance{thmstyle}{#1}{order={#9}}}
      \__block_tmp:w {##1} {##1} {##1}
    }
```

If block tracing is turned on we show the final result:

```
  \__block_debug:n { \ShowInstanceValues{thmstyle}{#1} }
}
```

(*End of definition for* `\newtheoremstyle`. *This function is documented on page 38.*)

10.2.2 Supporting QED in proofs

The amsthm package contains some elaborate code to support placing a QED symbol into the proof (by default at the end, but alternatively manually placed with `\qedhere`). This code is simply lifted and not adjusted in any way for now (and therefore also not documented—see the amsthm package for documentation for now).

```
\ExplSyntaxOff
  \def\math@qedhere{%
    \@ifundefined{\@currenvir @qed}{%
      \qed@warning\quad\hbox{\qedsymbol}%
    }{%
      \@xp\aftergroup\csname\@currenvir @qed\endcsname
    }%
  }
  \def\displaymath@qed{%
    \relax
    \ifmmode
      \ifinner \aftergroup\linebox@qed
      \else
        \eqno
        \let\eqno\relax \let\leqno\relax \let\veqno\relax
        \hbox{\qedsymbol}%
      \fi
    \else
       \aftergroup\linebox@qed
    \fi
  }
  \expandafter\let\csname equation*@qed\endcsname\displaymath@qed
  \def\equation@qed{%
    \iftagsleft@
      \hbox{\phantom{\quad\qedsymbol}}%
      \gdef\alt@tag{%
        \rlap{\hbox to\displaywidth{\hfil\qedsymbol}}%
        \global\let\alt@tag\@empty
      }%
    \else
      \gdef\alt@tag{%
        \global\let\alt@tag\@empty
        \vtop{\ialign{\hfil####\cr
                \tagform@\theequation\cr
                \qedsymbol\cr}}%
        \setbox\z@
      }%
```

```
    \fi
  }
  \def\qed@tag{%
    \global\tag@true \nonumber
    &\omit\setboxz@h {\strut@ \qedsymbol}\tagsleft@false
    \place@tag@gather
    \kern-\tabskip
    \ifst@rred \else \global\@eqnswtrue \fi \global\advance\row@\@ne \cr
  }
  \def\split@qed{%
    \def\endsplit{\crcr\egroup \egroup \ctagsplit@false \rendsplit@
      \aftergroup\align@qed
    }%
  }
  \def\align@qed{%
    \ifmeasuring@ \tag*{\qedsymbol}%
    \else \let\math@cr__block@\qed@tag
    \fi
  }
  \expandafter\let\csname align*@qed\endcsname\align@qed
  \expandafter\let\csname gather*@qed\endcsname\align@qed
  %
  \def\math@qedhere{\quad\hbox{\qedsymbol}}%
  %
  \DeclareRobustCommand{\qed}{%
    \ifmmode \mathqed
    \else
      \leavevmode\unskip\penalty9999 \hbox{}\nobreak\hfill
      \quad\hbox{\qedsymbol}%
    \fi
  }%
  \let\QED@stack\@empty
  \let\qed@elt\relax
  \newcommand{\pushQED}[1]{%
    \toks@{\qed@elt{#1}}\@temptokena\expandafter{\QED@stack}%
    \xdef\QED@stack{\the\toks@\the\@temptokena}%
  }%
  \newcommand{\popQED}{%
    \begingroup\let\qed@elt\popQED@elt \QED@stack\relax\relax\endgroup
  }%
  \def\popQED@elt#1#2\relax{#1\gdef\QED@stack{#2}}%
  \newcommand{\qedhere}{%
    \begingroup \let\mathqed\math@qedhere
      \let\qed@elt\setQED@elt \QED@stack\relax\relax \endgroup
  }%
  \def\setQED@elt#1#2\relax{%
    \ifmeasuring@
    \else \iffirstchoice@ \gdef\QED@stack{\qed@elt{}#2}\fi
    \fi
    #1%
  }%
  \def\qed@warning{%
    \PackageWarning{amsthm}{The \@nx\qedhere command may not work
      correctly here}%
```

```
  }%
  \newcommand{\mathqed}{\quad\hbox{\qedsymbol}}%
  \DeclareRobustCommand{\qed}{%
    \ifmmode \mathqed
    \else
      \leavevmode\unskip\penalty9999 \hbox{}\nobreak\hfill
      \quad\hbox{\qedsymbol}%
    \fi
  }
  \newcommand{\openbox}{\leavevmode
    \hbox to.77778em{%
    \hfil\vrule
    \vbox to.675em{\hrule width.6em\vfil\hrule}%
    \vrule\hfil}}
  \providecommand{\qedsymbol}{\openbox}
\ExplSyntaxOn
```

11 Support for other packages and classes

11.1 Replacement for alltt

The tools package alltt by Leslie Lamport has been completely implemented using the template approach and is therefore no longer necessary. In fact it has also been extended by providing `alltt*`.

```
\declare@file@substitution{alltt.sty}{null.tex}
```

11.2 Replacement for amsthm

The amsthm package is basically supported out of the box (though there are currently still a few limitation with `\newtheoremstyle` and perhaps also in other places). So this here is a bit premature, but for now we disable loading amsthm and wait to see how far this gets us. We may have to provide a bit more for better compatibility.

```
\declare@file@substitution{amsthm.sty}{null.tex}
```

11.3 Support for amsart and amsbook classes

Unfortunately, the amsart class contains a full implementation of amsthm inside the class (why ever) and they use `\newcommand`, sigh.

Thus, to make the new code work with this class we have to hide some definitions, load the class and only afterwards restore our own versions.

So first save some of the problematical definitions under some other names:

```
\let \amsnewtheorem      \newtheorem
\let \amsnewtheoremstyle \newtheoremstyle
\let \amstheoremstyle    \theoremstyle
\let \amsproof           \proof
\let \amsendproof        \endproof
```

Then undefine them just before the class gets loaded (quite a handful):

```
\AddToHook{class/amsart/before}[block]{
  \let \newtheoremstyle \relax
  \let \theoremstyle    \relax
```

```
  \let \proof           \relax
  \let \endproof        \relax
  \let \pushQED    \relax
  \let \popQED     \relax
  \let \qedhere    \relax
  \let \mathqed    \relax
  \let \openbox    \relax
}
```

Same for amsbook and amsproc:

```
\AddToHook{class/amsbook/before}[block]{
  \let \newtheoremstyle \relax
  \let \theoremstyle    \relax
  \let \proof           \relax
  \let \endproof        \relax
  \let \pushQED    \relax
  \let \popQED     \relax
  \let \qedhere    \relax
  \let \mathqed    \relax
  \let \openbox    \relax
}
\AddToHook{class/amsproc/before}[block]{
  \let \newtheoremstyle \relax
  \let \theoremstyle    \relax
  \let \proof           \relax
  \let \endproof        \relax
  \let \pushQED    \relax
  \let \popQED     \relax
  \let \qedhere    \relax
  \let \mathqed    \relax
  \let \openbox    \relax
}
```

And once the class is loaded restore our versions again. Note that we don't have to restored all the QED-related commands as ours are identical to those defined by the AMS.

```
\AddToHook{class/amsart/after}[block]{
  \let \newtheorem      \amsnewtheorem
  \let \newtheoremstyle \amsnewtheoremstyle
  \let \theoremstyle    \amstheoremstyle
  \let \proof           \amsproof
  \let \endproof        \amsendproof
}
\AddToHook{class/amsbook/after}[block]{
  \let \newtheorem      \amsnewtheorem
  \let \newtheoremstyle \amsnewtheoremstyle
  \let \theoremstyle    \amstheoremstyle
  \let \proof           \amsproof
  \let \endproof        \amsendproof
}
\AddToHook{class/amsproc/after}[block]{
  \let \newtheorem      \amsnewtheorem
  \let \newtheoremstyle \amsnewtheoremstyle
```

```
  \let \theoremstyle    \amstheoremstyle
  \let \proof           \amsproof
  \let \endproof        \amsendproof
}
```

11.4 Support for the enumitem interfaces

The current implementation incorporates most features of enumitem. The plan is that the enumitem interfaces are either natively available or are emulated and mapped to new interfaces, so that documents using enumitem work seamlessly.

Most (or even all of the enumitem keys have gotten new names, so there the task is to map old names to new names. One question to decide here is which (if any) of the original keys should remain natively available even if enumitem is not loaded, and which should only be supported if the document explicitly loads enumitem, i.e., support them only for compatibility with old documents. Providing the full set by default means one ends up with a fairly inconsistent interface, but not providing some of them may result in people unnecessarily loading enumitem in new documents just to get at, say, `nosep`.

decide

The enumitem package also provides declarations to build out new lists and adjust the layout of existing list using commands like `\newlist` or `\setlist`. With the new implementation this is normally done differently, e.g., defining instances and simple document level commands via `\SimpleBlockEnv`, etc. However, we probably also want a declaration such as `\newlist` (same name?) to provide a simple way to make this happen in the document preamble in one go.

I'm less sure about `\setlist` at least as far as its optional argument is concerned (even though we have to support it in an emulation.

decide

We put the code that emulates enumitem in a separate file to be loaded instead of the original package, but eventually some of the code from there has to move back to the kernel to be always present.

```
%\declare@file@substitution{enumitem.sty}{latex-lab-enumitem.sty}
```

But for now we simply disable enumitem loading and unconditionally load our replacement into the kernel for ease of testing. In the end we have to decide which parts of the interface (if any) we provide out of the box and which parts are only available if the document requests enumitem.

```
\declare@file@substitution{enumitem.sty}{null}
\RequirePackage{latex-lab-enumitem}
```

11.5 Support for the doc package

When the doc package is loaded it wants to remove a `%` sign from the start of each verbatim line. For this it uses `\check@percent` which we stick into the `verbatim/startline` socket.

doc (*plug*)

```
\NewSocketPlug {verbatim/startline}{doc}{ \check@percent }
\AddToHook{package/doc/after}{
  \AssignSocketPlug{verbatim/startline}{doc}
}
```

⟨/package-finish⟩

```
⟨*latex-lab⟩
\ProvidesFile{block-latex-lab-testphase.ltx}
        [\ltlabblockdate\space v\ltlabblockversion\space
                              blockenv implementation]
\RequirePackage{latex-lab-testphase-block}
⟨/latex-lab⟩
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T