Prototype reimplementation of $\LaTeX 2_\varepsilon$'s block environments using templates

LaTeX Project[*]

v0.9j 2025-07-09

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 2ε 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.

For example, a `quote` environment would make use of a (display) `block` and some `para` instnce while an standard `enumerate` would make use of a display `block`, a `list`, and an `item` and `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.

2 Template types and templates for blocks and lists

2.1 Template types

2.1.1 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, prohibitying or encuraging line breaks, and so forth.

2.1.2 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`.

2.1.3 The template type 'list'

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

Semantics:

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

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

2.1.4 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.

2.1.5 The template type 'blockenv'

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

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.

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 rolemapped and whether it adds additional attributes, etc.

2.2 Templates

2.2.1 The `blockenv` template 'display'

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`

`increment-level` (*boolean*) Does this `blockenv` increase the block level if it is nested in an outer block? Default: `true`

`setup-code` (*tokenlist*) Initial 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: `displayblock`

`para-instance` (*tokenlist*) Paragraph settings to use within the environment. If ⟨*`empty`*⟩ then outer values are retained. However, the block template resets some values, which may not be the right thing to do. 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. Default: `list`

`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: 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$_\varepsilon$ name).

It first checks that nothing is too deeply nested. If the level should increase then the increments the `\@listdepth` counter and calls the corresponding `\@list...` macro to update the legacy defaults. If `increment-level` is set to false this is bypassed.

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

Afterwards it calls the appropriate `block` instance based on `block-instance` and current level, e.g., `displayblock-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., `displayblock-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 `increment-level` is set to `false` then such an environment doesn't alter the nesting level and therefore you can nest those environments as often as you like (a typical example would be `flushleft` anywhere in the nesting hierarchy, that would have no effect on hitting the boundary).

2.2.2 The `block` template 'display'

Attributes:

`begin-vspace` (*skip*) Default: `\topsep`

`begin-extra-vspace` (*skip*) Default: `\partopsep`

`para-vspace` (*skip*) Default: `\parsep`

`end-vspace` (*skip*) Default: value from `begin-vspace`

`end-extra-vspace` (*skip*) 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*) Default: `\@beginparpenalty`

`end-penalty` (*integer*) Default: `\@endparpenalty`

`left-margin` (*length*) Default: `\leftmargin`

`right-margin` (*length*) Default: `\rightmargin`

`para-indent` (*length*) Default: `0pt`

Semantics & Comments: The idea of a `heading` key needs some further thoughts and therfore has been removed for now. Maybe instead the template type should accept a second argument and receive input for such a heading from the document level instead.

The names of the keys need further thoughts and some decision. Right now it is a mixture of those with hyphens and those that match legacy register names (the way enumitem did its keys).

2.2.3 The `para` template 'std'

Attributes:

`para-indent` (*length*)	Default: `\parindent`
`begin-hspace` (*skip*)	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` (*tokenlist*)	Default: `\@normalcr`
`para-attr-class` (*tokenlist*)	Default: `justify`

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

`item-label` (*tokenlist*) Label "string" for a fixed label or as generated from the current `counter` value

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

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

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

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-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.

3 Declaration of standard block environments

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

The `center` environment is defined through the `blockenv` instance `center` which makes use of the `block` instance `displayblock-`⟨*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. The default implementation is

```
\DeclareInstance{blockenv}{center}{display}
{
  name                   = center,
  tagging-recipe         = basic,
  tagging-suppress-paras = true ,
  increment-level        = false,
  block-instance         = displayblock ,
  para-instance          = center ,
}
```

The `flushleft` and `flushright` environments are defined in a similar way.

3.2 The `quote` and `quotation` environments

The `quote` environment is defined through the `blockenv` instance `quote` which makes use of the `block` instance `quoteblock-`⟨*level*⟩. The paragraph setup is inherited. The block nesting level is incremented. The default implementation is

```
\DeclareInstance{blockenv}{quote}{display}
{
  name                = quote,
  tag-name            = quote,
  tagging-recipe      = standard,
  increment-level     = true,
  block-instance      = quoteblock ,
}
```

The implementation of `quotation` is similar but uses `quotationblock-`⟨*level*⟩.

3.3 The `verbatim` and `verbatim*` environments

Both the `verbatim` environment is defined through the `blockenv` instance `verbatim` which makes use of the `block` instance `verbatimblock-`⟨*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. The default implementation is

```
\DeclareInstance{blockenv}{verbatim}{display}
{
  name                   = verbatim,
  tag-name               = verbatim,
  tagging-recipe         = standard,
  tagging-suppress-paras = true,
  increment-level        = false,
  block-instance         = verbatimblock ,
  para-instance          = justify ,
  final-code             = \legacyverbatimsetup
                             \@setupverbinvisiblespace \@vobeyspaces
                             \@xverbatim
}
```

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

3.4 The `itemize` environment

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. The ⟨*inner-level*⟩ is determined through `\@itemdepth`. The block nesting level and the inner list nesting level are incremented. The default implementation is

```
\DeclareInstance{blockenv}{itemize}{display}
{
  name                 = itemize,
  tag-name             = itemize,
  tag-attr-class       = itemize,
  tagging-recipe       = list,
  inner-level-counter  = \@itemdepth,
  increment-level      = true,
  max-inner-levels     = 4,
  block-instance       = listblock ,
  inner-instance       = itemize ,
}
```

3.5 The `enumerate` environment

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`.

3.6 The `description` environment

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.

3.7 The `list` environment

The generic `list` environment of LaTeX 2$_\varepsilon$ is modeled with a `blockenv` instance named `list`, a `block` instance named `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$_\varepsilon$) the `setup-code` key gets `\legacylistsetupcode` assigned, so the default setup (that should probably not be changed) looks as follows:

```
\DeclareInstance{blockenv}{list}{display}
{
  name                  = list,
  tag-name              = list,
  tagging-recipe        = list,
  increment-level       = true,
  setup-code            = \legacylistsetupcode ,
  block-instance        = listblock ,
  inner-instance        = legacy ,
}
```

3.8 The `verse` environment

The `verse` environment is currently still implemented as a list without real items (as in LaTeX 2$_\varepsilon$. That needs updating.

fix

3.9 The `trivlist` environment

In LaTeX 2$_\varepsilon$ `trivlist` was used to define various display environments that aren't really lists at all. To support such legacy definitions (even though they shuld 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.

3.10 Environments declared through `\newtheorem`

to document

4 Adjusting the layout of standard block environments

to document

5 Tagging support

5.1 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 live, 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>` (rollmapped 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-unit>
  <text>
    The paragraph text …
  </text>
</text-unit>
```

The `<text-unit>` structure is rollmapped 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>
      <Lbl> label </Lbl>
      <LBody>
        The text of the first item involving <text-unit> as necessary …
      </LBody>
    </LI>
    <LI>
      The second item …
    </LI>
    … further items …
  </itemize>
</text-unit>
```

The `<itemize>` is rollmapped 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>
```

5.2 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 rolemap 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 (`<Lbl>` for the item labels and `<LBody>` 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 rollmap.
- 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 `</LBody>`, `</LI>`, and `</L>` (or the tag name used) are closed.
- Then the lookahead for an empty line is done as described previously.

6 Debugging

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

These commands enable/disable debugging messages.

7 New and redefined kernel command

`\@doendpe` The original LaTeX 2_ε command is augmented to allow for tagging.

`\legacyverbatimsetup`
`\legacylistsetupcode`

to be documented

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

`endblockenv`
`\g_block_nesting_depth_int`

to be documented

`\newtheorem`
`\@thm`
`\@begintheorem`

Redefined to make theorems tagging aware.

`\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

`\para_end:` TODO: consider name, document

`para/begin` The para/begin hook is enhanced to support list ends

8 The Implementation

```
⟨*package⟩
⟨@@=block⟩
```

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

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
```

8.1 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 15.*)

`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
            {
              \__block_debug_typeout:n{Structure-end~
                  \l__tag_para_main_tag_tl\space
                  after~ displayblock \on@line     }
              \__tag_gincr_para_main_end_int:
              \tag_struct_end: %text-unit
            }
        }
    }
  }
\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`*.*)

8.2 Template types and template interfaces

`blockenv` (*templatetype*)
`block` (*templatetype*)
`para` (*templatetype*)
`list` (*templatetype*)
`item` (*templatetype*)

All template types expect a single key–value argument used to tweak template parameters specific to a given use in the document. This section is devoted to template interfaces, and the template code is covered later.

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

`blockenv display` (*templ.*)

```
\DeclareTemplateInterface{blockenv}{display}{1}
{
  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,
  increment-level        : boolean   = true ,
  setup-code             : tokenlist = ,
  block-instance         : tokenlist = displayblock ,
```

Paragraph instance is normally inherited so no default.

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

`block display` (*templ.*)

```
\DeclareTemplateInterface{block}{display}{1}
{
  begin-vspace       : skip = \topsep ,
  begin-extra-vspace : skip = \partopsep ,
  para-vspace        : skip = \parsep ,
  end-vspace         : skip = \KeyValue{begin-vspace} , % conflict with name below
  end-extra-vspace   : skip = \KeyValue{begin-extra-vspace} ,
  item-vspace          : skip = \itemsep ,
  begin-penalty      : integer = \UseName{@beginparpenalty} ,
  end-penalty        : integer = \UseName{@endparpenalty} ,
  left-margin        : length = \leftmargin ,
  right-margin       : length = \rightmargin ,
  para-indent        : length = 0pt ,
% maybe add? (or more general for fonts and color)
%  font            : tokenlist
}
```

`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           : tokenlist = \@normalcr ,
}
```

`list std` (*templ.*)

```
\DeclareTemplateInterface{list}{std}{1}
{
  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 ,
    text-font     : tokenlist ,
    compatibility : boolean = true ,
  }
```

8.3 Useful helper commands

This section collects expl3 commands that will be useful.

`\__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:`.)

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

8.3.1 Debugging

\g__block_debug_bool

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

(*End of definition for* \g__block_debug_bool.)

__block_debug:n
__block_debug_typeout:n

```
\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:

```
\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 { \typeout{[Blocks]~ ==>~ ##1} } }
  }
```

(*End of definition for* \block_debug_on:, \block_debug_off:, *and* __block_debug_gset:. *These functions are documented on page 15.*)

\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 15.*)

8.4 Implementation of templates

8.4.1 Implementation of blockenv templates ...

\g_block_nesting_depth_int LaTeX 2ε 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: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 15.*)

blockenv display (*templ.*)

```
\DeclareTemplateCode{blockenv}{display}{1}
{
  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 ,
  increment-level       = \l__block_level_incr_bool ,
  setup-code            = \l__block_setup_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 ,
}
{
  \__block_debug_typeout:n{\l__block_env_name_tl -env-start}
%
```

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

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

We need to know later if we have nested blockenvs 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.

```
  \bool_if:NT \l__block_level_incr_bool
     {
       \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.

```
             \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 overwritting 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 overwritting by the legacy setup code for the `list` environment in `\l__block_setup_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 setup code if any is given in the instance.

```
  \l__block_setup_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
    {
```

```
      \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
```

Again the instance may has processed a few keys from the sofar 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 realy clean.

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

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
}
```

`\l__block_unused_blockenv_keys_tl`

```
\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 blockenvs 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 15.*)

`\endblockenv` The code executed when a blockenv ends is 99% the same for all blockenvs (at least up to now). Small differences exist, though. They are accounted for first in the conditionals.

name is bad

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

If this block was incrementing the level we have to decrement it now again:

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

If this block was a list and there are still `\item` labels 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 enviornment is a list and not unconditionally.

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

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 2ε list environment 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* `\endblockenv`. *This function is documented on page 15.*)

`\__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 {listblock} }
```

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

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

```
\cs_new: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.

```
\socket_new_plug:nnn{block/endpe}{on}
```

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

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

```
\socket_assign_plug:nn{block/endpe}{on}
```

8.4.2 Implementation of para templates ...

para std (*templ.*)

```
\DeclareTemplateCode{para}{std}{1}
{
  para-indent         = \parindent ,
  begin-hspace        = \l__par_start_skip ,              % name??
  left-hspace         = \leftskip  ,
  right-hspace        = \rightskip  ,
  end-hspace          = \parfillskip ,
  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 ,
}
{
  \SetTemplateKeys{para}{std}{#1}
  \skip_set:Nn \@rightskip \rightskip
}
```

8.4.3 Implementation of block templates ...

block display (*templ.*)

In contrast to the LaTeX 2ε 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}{display}{1}
{
  begin-vspace        = \topsep ,
  begin-extra-vspace  = \partopsep ,
  para-vspace        = \parsep ,
  end-vspace        = \l__block_botsep_skip ,
  end-extra-vspace    = \l__block_parbotsep_skip ,
  item-vspace       = \itemsep ,
  begin-penalty    = \@beginparpenalty ,
  end-penalty      = \@endparpenalty ,
  right-margin     = \rightmargin ,
  left-margin      = \leftmargin ,
  para-indent       = \l__block_parindent_dim ,
}
{
    \SetKnownTemplateKeys{block}{display}{#1}
```

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

```
    \legacy_if:nT { @noskipsec } { \mode_leave_vertical: }
    \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
      }
```

Now we are back to legacy list implementation …

```
    \legacy_if:nTF { @inlabel }
      {
        \legacy_if_set_true:n { @noparitem }
        \legacy_if_set_true:n { @noparlist }
      }
      {
        \legacy_if:nT { @newlist } { \@noitemerr }
        \legacy_if_set_false:n { @noparlist }
        \skip_set_eq:NN \l__block_effective_top_skip \l__block_topsepadd_skip
      }
    \skip_add:Nn \l__block_effective_top_skip { \parskip }
```

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: }
        }
    }
  \skip_set_eq:NN \@outerparskip \parskip
  \skip_set_eq:NN \parskip \parsep


  \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 us suppressed. This is controlled through the legacy switches `@noparitem`, `minipage`, and `@nobreak`.

```
  \legacy_if:nTF { @noparitem }
    {
      \legacy_if_set_false:n { @noparitem }
      \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 - \parskip }
        }
    }
    {
      \legacy_if:nTF { @nobreak }
        { \addvspace{\skip_eval:n{\@outerparskip-\parskip}} }
        {
            \addpenalty \@beginparpenalty
            \addvspace \l__block_effective_top_skip
            \addvspace{-\parskip}
          }
      }
}
```

Extra keys to support enumitem conventions:

```
\keys_define:nn { template/block/display }
{
  ,topsep        .skip_set:N = \topsep
  ,partopsep     .skip_set:N = \partopsep
  ,listparindent .skip_set:N = \listparindent
}
```

__kernel_displayblock_begin:
__kernel_displayblock_beginpar_hmode:w
__kernel_displayblock_beginpar_vmode:

The internal kernel hooks for tagging.

```
\cs_new:Npn \__kernel_displayblock_begin: {
  \__block_debug_typeout:n
     {\detokenize{__kernel_displayblock_begin:}}
}

\cs_new:Npn \__kernel_displayblock_beginpar_hmode:w {
  \__block_debug_typeout:n
     {\detokenize{__kernel_displayblock_beginpar_hmode:w}}
}

\cs_new: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:`*.*)

8.4.4 Implementation of list templates ...

\@itemlabel
\@listctr

Both `\@itemlabel` and `\@listctr` from the LaTeX 2ε 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 15.*)

__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}{1}
{
  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
}
{
  \__block_debug_typeout:n{template:list:std}
%
```

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.

```
        \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.

```
      \int_compare:nNnF \l__block_counter_start_int = 1
        {
          \int_gset:cn{ c@ \@listctr }
              { \l__block_counter_start_int - 1 }
        }
    }
```

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.

```
    {
     \@nmbrlisttrue
     \tl_set_eq:NN \@listctr \l__block_counter_tl
     \bool_if:NF \l__block_resume_bool
       {
         \int_gset:cn{ c@ \@listctr }
             { \l__block_counter_start_int - 1 }
       }
    }
```

Does the current instance has 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.

```
  \tl_if_empty:NF \l__block_item_label_tl
    {
      \tl_set_eq:NN \@itemlabel \l__block_item_label_tl
    }
```

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.

```
    \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 setup up `\__block_item_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.

```
    \cs_set_eq:NN \__block_item_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.
  }
```

Extra keys to support enumitem conventions:

```
\keys_define:nn { template/list/std }
{
  ,nosep .code:n =
    \dim_zero:N \itemsep
    \dim_zero:N \parsep
    \dim_zero:N \topsep
    \dim_zero:N \l__block_botsep_skip
    \dim_zero:N \l__block_parbotsep_skip
  ,midsep    .skip_set:N = \topsep
}
```

8.4.5 Implementation of `\item` template(s)

`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 ,
    } ,
  }
```

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.

```
  {
    \__block_debug_typeout:n{template:item:std}
```

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 aplicable, 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
       }
```

If no optional argument was given then \l__block_label_given_tl is still equal to \c_novalue_tl and so we can distinuish 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
    \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$_\varepsilon$ 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` envrironment 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_item_everypar:` inside `para/begin`. By default this command does nothing, now we change it to attach the pending label or labels.

```
    \cs_set_eq:NN \__block_item_everypar: \__block_item_everypar_std:
  }
```

`\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 2ε'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*) A tagging socket to tag the label. It takes two arguments so that it can transparently pass the label content. Declaration is in lttagging.

```
\socket_if_exist:nF {tagsupport/block/list/label}
 {
   \NewTaggingSocket{block/list/label}{2}
 }
```

`default` (*plug*)

```
\def\LBody{LBody}
\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=Lbl}
   \tagmcbegin{tag=Lbl}
   %     }
   #2
   \tagmcend                                    % end mc-Lbl or artifact
   % FMi: unconditionally for now
   %  \tl_if_empty:oF \@itemlabel
   \tagstructend   % end   Lbl
   \tagstructbegin{tag=\LBody}
 }
\AssignTaggingSocketPlug{block/list/label}{default}
```

__block_item_everypar:
__block_item_everypar_std:
__block_item_everypar_first:

The __block_item_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_item_everypar: \prg_do_nothing:
```

```
\AddToHook{para/begin}[items]{\__block_item_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 or, better, into sockets.

```
\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_item_-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{item~ 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:

         \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_item_everypar: to do nothing.

```
          \cs_set_eq:NN \__block_item_everypar: \prg_do_nothing:
        }
}
```

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

```
\cs_new:Npn \__block_item_everypar_first: {
  \legacy_if:nT { @newlist }  { \@noitemerr }
}
```

(*End of definition for* `\__block_item_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$_\varepsilon$'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}{
  \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} }
```

Set the legacy switch that signals that we have a pending item label:

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

(*End of definition for* `\item`*. This function is documented on page 15.*)

`\__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 do 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:`*.*)

8.5 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
          {
            \__tag_gincr_para_main_begin_int:
            \tag_struct_begin:n
              {
                tag=\l__tag_para_main_tag_tl,
                attribute-class=\l__tag_para_main_attr_class_tl,
              }
            \__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 currently in tagpdf and uses sockets. The default plug plain is not good enough when lists get supported: we need to delay starting the paragraph tagging if we still have to place the list label. tagpdf assigns therefore a different socket in the `package/latex-lab-testphase-block/after` hook. This should move into lttagging (and then perhaps use kernel para hooks).

`/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
      {
        \__tag_gincr_para_main_begin_int:
        \tag_struct_begin:n
          {
            tag=\l__tag_para_main_tag_tl,
            attribute-class=\l__tag_para_main_attr_class_tl,
          }
        \__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 is currently handled by tagpdf which uses sockets there and assigns suitable sockets for the block code in a package/after hook. This code should move into lttagging and be active always. 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-}
```

`ort/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
    {
      \__tag_gincr_para_main_end_int:
      \tag_struct_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
  }
}
\cs_set_eq:NN \par     \para_end:
\cs_set_eq:NN \__blockpar   \para_end:
\cs_set_eq:NN \endgraf \para_end:
```

(*End of definition for* `\para_end:`. *This function is documented on page 16.*)

`\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 15.*)

`\__kernel_list_label_after:` 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.

```
\cs_new:Npn \__kernel_list_label_after: {
   \bool_lazy_and:nnT { \tag_if_active_p: } {\l__tag_para_bool }
     {
       \tag_socket_use:nn {block/startpara/direct} { LI- }
     }
}
```

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

`\__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: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:Npn \__block_inner_end: {
  \__block_debug_typeout:n{block-end \on@line}
  \legacy_if:nT { @endpe }
    {
      \__tag_gincr_para_main_end_int:
      \__block_debug_typeout:n{close~ /text-unit \on@line}
      \tagstructend
    }
  \tagstructend        % end inner structure
}
```

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

8.5.1 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},
  }
\def\LItag{LI}
```

`\__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=\LItag} }
```

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

`\__block_list_item_end:` When a list item ends we have to close `<LBody>` 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 }
    {
      \__tag_gincr_para_main_end_int:
      \tagstructend                              % text-unit
%      \__block_debug_typeout:n{Structure-end~ P~ at~ item-end \on@line }
    }
  \tagstructend \tagstructend   % end LBody, LI
}
```

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

`\__block_list_end:` Finally, at the list end we have to close the open `<LBody>`, `<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 `<LBody>` 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: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 }
        {
          \__tag_gincr_para_main_end_int:
          \tagstructend                       % text-unit
          \__block_debug_typeout:n{Structure-end~ P~ at~ list-end \on@line }
        }
      \tagstructend\tagstructend  % end LBody, LI
    }
  \tagstructend               % end L
}
```

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

End of tagging related declarations.

8.6 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: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: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 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: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 rolemapped to L-structure and arranges for handling list items, e.g., Li, Lbl and LBody 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: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 `\items`, 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:`.)

9 Implementation of document-level block environments

Most such environments are pretty simple: they take an optional argument and call a `blockenv` instance to do the work. At the end of environment we call `\endblockenv` to finish.

9.1 Displayblock environments

There are two basic block environment which are similar to LaTeX 2ε's `trivlist` except that there aren't degenerated lists and thus have no hidden `\item` inside.

`displayblock` (*env.*)

```
\NewDocumentEnvironment{displayblock}{ !O{} }
  { \UseInstance{blockenv}{displayblock} {#1} }
  { \endblockenv }
```

`displayblockflattened` (*env.*)

```
\NewDocumentEnvironment{displayblockflattened}{ !O{} }
  { \UseInstance{blockenv}{displayblockflattened} {#1} }
  { \endblockenv }
```

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

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}{
```

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

```
  \RenewDocumentEnvironment{center} { !O{} }
  { \UseInstance{blockenv}{center}{#1} }
  { \endblockenv }

  \RenewDocumentEnvironment{flushright} { !O{} }
  { \UseInstance{blockenv}{flushright}{#1} }
  { \endblockenv }

  \RenewDocumentEnvironment{flushleft} { !O{} }
  { \UseInstance{blockenv}{flushleft}{#1} }
  { \endblockenv }
}
```

9.3 Display quote environments

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

```
\AddToHook{begindocument/before}{
  \RenewDocumentEnvironment{quote}{ !O{} }
    { \UseInstance{blockenv}{quote} {#1} }
    { \endblockenv }

  \RenewDocumentEnvironment{quotation}{ !O{} }
    { \UseInstance{blockenv}{quotation} {#1} }
    { \endblockenv }
}
```

9.4 Verbatim environments

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

```
\AddToHook{begindocument/before}{
  \RenewDocumentEnvironment{verbatim}{ !O{} }
    { \UseInstance{blockenv}{verbatim} {#1} }
    { \endblockenv }


  \RenewDocumentEnvironment{verbatim*}{ !O{} }
    { \UseInstance{blockenv}{verbatim*} {#1} }
    { \endblockenv }
}
```

9.4.1 Helper commands for verbatim

\legacyverbatimsetup

This code resembles the LaTeX 2ε verbatim implementation with a slight twist: in LaTeX 2ε 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.

```
⟨@@=⟩
\def\legacyverbatimsetup{%
  \language\l@nohyphenation
  \@tempswafalse
  \def\par{%
    \if@tempswa
      \leavevmode \null {\@@par}\penalty\interlinepenalty
    \else
      \@tempswatrue
      \ifhmode{\@@par}\penalty\interlinepenalty\fi
    \fi}%
  \let\do\@makeother \dospecials
  \obeylines \verbatim@font \@noligs
  \everypar \expandafter{\the\everypar \unpenalty}%
  \frenchspacing
```

next should be a socket

```
  \tl_set:Nn \l__tag_para_tag_tl {codeline}
}
⟨@@=block⟩
```

(*End of definition for* `\legacyverbatimsetup`. *This function is documented on page* 15.)

`\@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.

```
\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* 15.)

9.5 Standard list environments

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

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

```
\AddToHook{begindocument/before}{
  \RenewDocumentEnvironment{itemize}{!O{}}
    { \UseInstance{blockenv}{itemize} {#1} }
    { \endblockenv }

  \RenewDocumentEnvironment{enumerate}{!O{}}
    { \UseInstance{blockenv}{enumerate} {#1} }
    { \endblockenv }

  \RenewDocumentEnvironment{description}{!O{}}
    { \UseInstance{blockenv}{description} {#1} }
    { \endblockenv }
}
```

9.6 verse environment

`verse` (*env.*) The verse environment has not special tagging currently. It is defined as a simple standard list and takes the tagging from there. But it must be redefined so that `\itemindent` is correctly set.

```
\AddToHook{begindocument/before}{
  \RenewDocumentEnvironment{verse}{ !O{} }
    {
      \let\\\@centercr
      \UseInstance{blockenv}{list}
        {
          item-indent=-1.5em,
          para-indent=-1.5em,
          item-vspace=0pt,
          right-margin = \leftmargin,
          left-margin  = \leftmargin+1.5em,
          #1
         }
      \item\relax
    }
    { \endblockenv }
  }
```

`list` (*env.*)

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

```
\AddToHook{begindocument/before}{
  \RenewDocumentEnvironment{list}{O{} m m }
    {
```

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

```
      \tl_set:Nn \l__block_legacy_env_params_tl
         {
           \tl_set:Nn \@itemlabel {#2}
           #3
         }

      \UseInstance{blockenv}{list} {#1}
    }
    { \endblockenv }
}
```

`\legacylistsetupcode`

And here is the extra code for use in the list instance setup inside the key `setup-code`.

```
\cs_new:Npn \legacylistsetupcode {
```

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:

```
    \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` rolemapped 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* `\legacylistsetupcode`. *This function is documented on page* 15.)

\l__block_legacy_env_params_tl

```
\tl_new:N\l__block_legacy_env_params_tl
```

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

Replace with code not using `\list`

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

9.7 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.

`\newtheorem` This is a slightly streamlined version of `\newtheorem`, but it still uses a lot of the 2e code for now. Eventually this will change.

```
\RenewDocumentCommand \newtheorem { m O{#1} m o }
{
  \expandafter\@ifdefinable\csname #1\endcsname
    {
      \str_if_eq:nnTF{#1}{#2}
         {
           \@definecounter {#2}
           \IfNoValueTF {#4}
             {  % @ynthm
               \tl_gset:ce { the #2 }
                  {
                    \@thmcounter{#2}
                  }
             }
             {  % @xnthm
               \@newctr{#1}[#4]
               \tl_gset:ce { the #2 }
                  {
                    \expandafter\noexpand\csname the#4\endcsname
                    \@thmcountersep
                    \@thmcounter{#2}
                  }
             }
         }
         {  % @othm
           \@ifundefined{c@#2}
              { \@nocounterr{#2} }
              {
                \tl_gset:cn { the #1 }
                   { \UseName { the #2 } }
              }
         }
      \global\@namedef{#1}   { \@thm{#2}{#3} }
      \global\@namedef{end#1}{ \@endtheorem  }
    }
}
```

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

`\@thm` `\@thm` executes `\refstepcounter` too early for hyperref and structure destinations: the generated target is outside the structure and can be separated from the theorem by a page break. We therefore move the anchor setting into `\@begintheorem`. `\@begintheorem` doesn't currently get the name of the counter as argument, so we store it in variable for now, to be able to pass it along.

```
\tl_new:N \l__block_thm_current_counter_tl
\def\@thm#1#2{%
  \@kernel@refstepcounter{#1}
  \tl_set:Nn \l__block_thm_current_counter_tl{#1}
  \@ifnextchar[{\@ythm{#1}{#2}}{\@xthm{#1}{#2}}}
```

To avoid that hyperref overwrites the definition again we must its patch:

```
\def\hyper@nopatch@thm{}
```

(*End of definition for* `\@thm`.)

`\@begintheorem`
`\@opargbegintheorem`

The `\@thm` command expands to either `\@beginthorem` or `\@opargbegintheorem`. For the moment we stick with this as it will help with the transition. But instead of using a `trivlist` we use a blockenv and some tagging for the title (as a Caption). We do not want potential tagging from `\textbf` here, so we use `\bfseries` to set the font. The commands set also the link targets which should be inside the main structure.

```
\def\@begintheorem#1#2{
  \UseInstance{blockenv}{theorem}{}
  \tagpdfparaOff


  \noindent
  \MakeLinkTarget{\l__block_thm_current_counter_tl}
  \tag_struct_begin:n{tag=Caption}
   \group_begin:
   \bfseries
   \tag_mc_begin:n {}
     #1\
   \tag_mc_end:
    \tag_struct_begin:n{tag=Lbl}
      \tag_mc_begin:n {}
          #2
      \tag_mc_end:
    \tag_struct_end:
    \group_end:
  \tag_struct_end:
  \tagpdfparaOn

  \tag_socket_use:nn { block/startpara/direct } { \PARALABEL }

  \itshape
  \hskip\labelsep
  \ignorespaces
}
\def\@opargbegintheorem#1#2#3{
  \UseInstance{blockenv}{theorem}{}
  \tagpdfparaOff
  \noindent
  \MakeLinkTarget{\l__block_thm_current_counter_tl}
  \tag_struct_begin:n{tag=Caption}
   \group_begin:
   \bfseries
   \tag_mc_begin:n {}
      #1\
   \tag_mc_end:
   \tag_struct_begin:n{tag=Lbl}
     \tag_mc_begin:n {}
       #2
     \tag_mc_end:
   \tag_struct_end:
     \tag_mc_begin:n {}
      \ (#3)
     \tag_mc_end:
   \group_end:
  \tag_struct_end:
  \tagpdfparaOn
 \tag_socket_use:nn { block/startpara/direct} { \PARALABEL }
  \itshape
  \hskip\labelsep
  \ignorespaces
}
\def\@endtheorem{\endblockenv}
```

(*End of definition for* `\@begintheorem` *and* `\@opargbegintheorem`*.*)

10 Instance declarations for environments

10.1 Blockenv instances

The blockenv instances handle overall setup for the document-level environments, i.e.,

- name of the environment for debugging purposes (`name`)
- how tagging should be performed (`tagging-recipe`, `tag-name`, `tag-attr-class`)
- does this environment changes the block level if nested (`increment-level`)
- any special setup code; normally not used (`setup-code`)
- what kind of block instance should be used (`block-instance`)
- what kind of para instance should be used; empty means inherit from the outside (`para-instance`)

Note that block does set some values — check if this is right!

- are inner paragraphs real paragraphs (default) or are they just `<text>` structures and part of an outer paragraph (`tagging-suppress-paras`)
- what kind of inner instance should be used, if any (`inner-instance`, `inner-instance-type`)

- the counter name of the inner level, if any (`inner-level-counter`)
- supported nesting depth of the inner level, if relevant (`max-inner-levels`)
- extra code executed at the end, by default `\ignorespaces` (`final-code`)

The blockenv displayblock instance below shows the full set with those using default values being commented out.

10.1.1 Basic instances

`blockenv displayblock` (*inst.*) This is like LaTeX 2$_\varepsilon$'s `trivlist`, i.e., it produces a vertical block with default setting, but doesn't put a list inside but uses a `<Div>` structure.

```
\DeclareInstance{blockenv}{displayblock}{display}
{
  name                = displayblock,
%  tagging-recipe      = standard,
%  tag-name            = ,
%  tag-attr-class      = ,
  increment-level     = false,
%  setup-code          = ,
%  block-instance      = displayblock ,
%  para-instance       = ,
%  tagging-suppress-paras = false ,
%  inner-instance      = ,
%  inner-instance-type = list , % 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 ,
}
```

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

```
\DeclareInstance{blockenv}{displayblockflattened}{display}
{
  name                   = displayblockflattened,
  tag-name               = ,
  tag-attr-class         = ,
  tagging-recipe         = basic,
  tagging-suppress-paras = true ,
  inner-level-counter    = ,
  increment-level        = false,
  setup-code             = ,
  block-instance         = displayblock ,
  inner-instance         = ,
}
```

10.1.2 Center, flushleft, and flushright instances

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

`blockenv center` (*inst.*) The `center` environment without using a list internally.

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

`blockenv flushleft` (*inst.*) The `flushleft` environment without using a list internally.

```
\DeclareInstance{blockenv}{flushleft}{display}
{
  name                   = flushleft,
  tag-name               = ,
  tag-attr-class         = ,
  tagging-recipe         = basic,
  tagging-suppress-paras = true ,
  inner-level-counter    = ,
  increment-level        = false,
  setup-code             = ,
  block-instance         = displayblock ,
  para-instance          = raggedright ,
  inner-instance         = ,
}
```

`blockenv flushright` (*inst.*)

The `flushright` environment without using a list internally.

```
\DeclareInstance{blockenv}{flushright}{display}
{
  name                   = flushleft,
  tag-name               = ,
  tag-attr-class         = ,
  tagging-recipe         = basic,
  tagging-suppress-paras = true ,
  inner-level-counter    = ,
  increment-level        = false,
  setup-code             = ,
  block-instance         = displayblock ,
  para-instance          = raggedleft ,
  inner-instance         = ,
}
```

10.1.3 Blockquote instances

LaTeX 2_ε has two environments for quoting: `quote` and `quoation`. By default they differ only in indentation of inner paragraphs. This is handled by using separate block instances.

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

`blockenv quotation` (*inst.*) For the `quotation` environment:

```
\DeclareInstance{blockenv}{quotation}{display}
{
  name                = quotation,
  tag-name            = quotation,
  tag-attr-class      = ,
  tagging-recipe      = standard,
  inner-level-counter = ,
  increment-level     = true,
  setup-code          = ,
  block-instance      = quotationblock ,
  inner-instance      =  ,
}
```

`blockenv quote` (*inst.*) For the `quote` environment:

```
\DeclareInstance{blockenv}{quote}{display}
{
  name                = quote,
  tag-name            = quote,
  tag-attr-class      = ,
  tagging-recipe      = standard,
  inner-level-counter = ,
  increment-level     = true,
  setup-code          = ,
  block-instance      = quoteblock ,
  inner-instance      =  ,
}
```

10.1.4 The theorem instance

`blockenv theorem` (*inst.*)

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

```
\DeclareInstance{blockenv}{theorem}{display}
{
  name                 = theorem-like,
  tag-name             = theorem-like,
  tag-attr-class       = ,
  tagging-recipe       = standalone,
  inner-level-counter  = ,
  increment-level      = false,
  setup-code           = ,
  block-instance       = theoremblock ,
}
```

10.1.5 The verbatim and verbatim* instances

The rolemapping 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.*)

```
\DeclareInstance{blockenv}{verbatim}{display}
{
  name                   = verbatim,
  tag-name               = verbatim,
  tag-attr-class         = ,
  tagging-recipe         = standard,
  tagging-suppress-paras = true,
  inner-level-counter    = ,
  increment-level        = false,
  setup-code             = ,
  block-instance         = verbatimblock ,
  inner-instance         = ,
  para-instance          = justify ,
  final-code             = \legacyverbatimsetup
```

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

```
                           \@setupverbinvisiblespace \@vobeyspaces
```

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

```
                           \@xverbatim
}
```

`blockenv verbatim*` (*inst.*)

```
\DeclareInstance{blockenv}{verbatim*}{display}
{
  name                   = verbatim,
  tag-name               = verbatim,
  tag-attr-class         = ,
  tagging-recipe         = standard,
  tagging-suppress-paras = true,
  inner-level-counter    = ,
  increment-level        = false,
  setup-code             = ,
  block-instance         = verbatimblock ,
  inner-instance         = ,
```

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

```
  para-instance          = justify ,
  final-code             = \legacyverbatimsetup
                             \@setupverbvisiblespace \@vobeyspaces
                             \@sxverbatim
}
```

10.1.6 Standard list instances

`blockenv itemize` (*inst.*) For the `itemize` environment:

```
\DeclareInstance{blockenv}{itemize}{display}
{
  name                 = itemize,
  tag-name             = itemize,
  tag-attr-class       = itemize,
  tagging-recipe       = list,
  inner-level-counter = \@itemdepth,
  increment-level     = true,
  max-inner-levels    = 4,
  setup-code          = ,
  block-instance      = listblock ,
  inner-instance      = itemize ,
}
```

`blockenv enumerate` (*inst.*)

For the `enumerate` environment:

```
\DeclareInstance{blockenv}{enumerate}{display}
{
  name                 = enumerate,
  tag-name             = enumerate,
  tag-attr-class       = enumerate,
  tagging-recipe       = list,
  increment-level     = true,
  max-inner-levels    = 4,
  setup-code          = ,
  block-instance      = listblock ,
  inner-level-counter = \@enumdepth,
  inner-instance      = enum ,
}
```

`blockenv description` (*inst.*)

For the `description` environment:

```
\DeclareInstance{blockenv}{description}{display}
{
  name                 = description,
  tag-name             = description,
  tag-attr-class       = description,
  tagging-recipe       = list,
  inner-level-counter = ,
  increment-level     = true,
  setup-code          = ,
  block-instance      = listblock ,
  inner-instance      = description ,
}
```

blockenv list (*inst.*) The general (legacy) `list` environment does some of its setup in the `setup-code` key.

```
\DeclareInstance{blockenv}{list}{display}
{
  name                = list,
  tag-name            = list,
  tag-attr-class      = ,
  tagging-recipe      = list,
  increment-level     = true,
  setup-code          = \legacylistsetupcode ,
  block-instance      = listblock ,
  inner-level-counter = ,
  inner-instance      = legacy ,
}
```

10.2 Block instances

Below, we declare the different block instances for the document-level environments. Some, such as the displayblock ones are shared between different environments (as long as one doesn't need to adjust individual values), other environments have their own instances (for precisely that reason).

10.2.1 Displayblock instances

We provide 6 nesting levels (as in LaTeX 2$_\varepsilon$). If you want to provide more you need to change the `maxblocklevels` counter, offer further `displayblock-xx` 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 displayblock-0 (*inst.*)
block displayblock-1 (*inst.*)
block displayblock-2 (*inst.*)
block displayblock-3 (*inst.*)
block displayblock-4 (*inst.*)
block displayblock-5 (*inst.*)
block displayblock-6 (*inst.*)

Here we need level zero as well in case a flattened displayblock (like the center env) it is used on top-level.

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

```
\DeclareInstance{block}{displayblock-0}{display}
  {
%    begin-vspace       = \topsep ,
%    begin-extra-vspace = \partopsep ,
%    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        = 0pt ,
%    right-margin       = \rightmargin ,
%    para-indent        = 0pt ,
  }
\DeclareInstanceCopy{block}{displayblock-1}{displayblock-0}
\DeclareInstanceCopy{block}{displayblock-2}{displayblock-0}
\DeclareInstanceCopy{block}{displayblock-3}{displayblock-0}
\DeclareInstanceCopy{block}{displayblock-4}{displayblock-0}
\DeclareInstanceCopy{block}{displayblock-5}{displayblock-0}
\DeclareInstanceCopy{block}{displayblock-6}{displayblock-0}
```

10.2.2 Verbatim instances

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

block verbatimblock-0 (*inst.*)
block verbatimblock-1 (*inst.*)
block verbatimblock-2 (*inst.*)
block verbatimblock-3 (*inst.*)
block verbatimblock-4 (*inst.*)
block verbatimblock-5 (*inst.*)
block verbatimblock-6 (*inst.*)

```
\DeclareInstance{block}{verbatimblock-0}{display}
  {
    left-margin      = 0pt ,
    para-vspace        = 0pt ,
  }
\DeclareInstanceCopy{block}{verbatimblock-1}{verbatimblock-0}
\DeclareInstanceCopy{block}{verbatimblock-2}{verbatimblock-0}
\DeclareInstanceCopy{block}{verbatimblock-3}{verbatimblock-0}
\DeclareInstanceCopy{block}{verbatimblock-4}{verbatimblock-0}
\DeclareInstanceCopy{block}{verbatimblock-5}{verbatimblock-0}
\DeclareInstanceCopy{block}{verbatimblock-6}{verbatimblock-0}
```

10.2.3 Quote/quotationblock instances

Quote and quotation are not flattened, i.e., they change levels, thus they start with level 1 not 0.

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

Default layout is to indent equally from both sides.

```
\DeclareInstance{block}{quoteblock-1}{display}
  { right-margin = \KeyValue{left-margin} }
\DeclareInstanceCopy{block}{quoteblock-2}{quoteblock-1}
\DeclareInstanceCopy{block}{quoteblock-3}{quoteblock-1}
\DeclareInstanceCopy{block}{quoteblock-4}{quoteblock-1}
\DeclareInstanceCopy{block}{quoteblock-5}{quoteblock-1}
\DeclareInstanceCopy{block}{quoteblock-6}{quoteblock-1}
```

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

Quotation additionally changes the para-indent.

```
\DeclareInstance{block}{quotationblock-1}{display}
  { para-indent = 1.5em , right-margin = \KeyValue{left-margin} }
\DeclareInstanceCopy{block}{quotationblock-2}{quotationblock-1}
\DeclareInstanceCopy{block}{quotationblock-3}{quotationblock-1}
\DeclareInstanceCopy{block}{quotationblock-4}{quotationblock-1}
\DeclareInstanceCopy{block}{quotationblock-5}{quotationblock-1}
\DeclareInstanceCopy{block}{quotationblock-6}{quotationblock-1}
```

10.2.4 Block instances for the theorems

block theoremblock-0 (*inst.*) Theorems do not support nesting, so we have only one to set up. The LaTeX default reused the general value of `\parindent` and `\parskip` and, of course, they start at the outer margin.

```
\DeclareInstance{block}{theoremblock-0}{display}
  {
    left-margin      = 0pt ,
    para-indent      = \parindent ,
    para-vspace      = \parskip ,
  }
```

There are, however, documents that put theorem-like environments inside of lists. While that is in most case somewhat dubious, it can make sense, for example, in `description` lists. So we support it somewhat by also providing theoremblock instances for level 1 and 2. If somebody really nests them further down, then more such instances need to be declared.

```
\DeclareInstanceCopy{block}{theoremblock-1}{theoremblock-0}
\DeclareInstanceCopy{block}{theoremblock-2}{theoremblock-0}
```

10.2.5 Block instances for the standard lists

block listblock-1 (*inst.*)
block listblock-2 (*inst.*)
block listblock-3 (*inst.*)
block listblock-4 (*inst.*)
block listblock-5 (*inst.*)
block listblock-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}{listblock-1}{display}{
%   begin-vspace       = \topsep ,
%   begin-extra-vspace = \partopsep ,
%   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 ,
}
\DeclareInstance{block}{listblock-2}{display}{}
\DeclareInstance{block}{listblock-3}{display}{}
\DeclareInstance{block}{listblock-4}{display}{}
\DeclareInstance{block}{listblock-5}{display}{}
\DeclareInstance{block}{listblock-6}{display}{}
```

10.3 List instances for the standard lists

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

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 ,
}
```

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 }
```

10.4 Item instances

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

There two item instances 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
  }
```

10.5 Para 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},
  }
```

`para center` (*inst.*)

```
\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.*)

```
\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.*)

```
\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 ,
}
```

`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 ,
}
```

\centering
\raggedleft
\raggedright
\justifying

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

```
\justifying
```

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

```
⟨/package⟩
```

```
⟨*latex-lab⟩
\ProvidesFile{block-latex-lab-testphase.ltx}
        [\ltlabblockdate\space v\ltlabblockversion\space
                              blockenv implementation]
\RequirePackage{latex-lab-testphase-block}
⟨/latex-lab⟩
```

A Documentation from first prototype implementations

A.1 Open questions

- Existing questions — moved to issues —

A.2 Code cleanup

- Actually implement what's announced.
- Encapsulate most uses of `\legacy_if…` into commands with expl3 syntax: we cannot rename these booleans for compatibility reasons but we can make the code cleaner nevertheless. — made issue —
- The `\topsep` and `\partopsep` business is tricky to reproduce exactly (see `\@topsepadd` and `\@topsep`) because of how it accumulates when lists are nested immediately.

A.3 Tasks

- Change author to LaTeX Team once it's nice enough to deserve that label.
- Reproducing exactly the standard layouts and examples in the enumitem documentation.

- Hooks, but do not duplicate those that already exist as environment hooks. Hence, mostly around items.
- Customization and interaction with LDB:
  - Allow arbitrary nesting depth with automatically defined styles for labels, counters etc.
  - Adapt everything to font size! (e.g. footnotes).
  - How to model the inheritance from trivlist to list to enumerate?
- Add key–value settings mimicking enumitem's ability to set any four of five horizontal parameters and deduce the fifth by `\leftmargin` + `\itemindent` = `\labelindent` + `\labelwidth` + `\labelsep`.
- Provide good ways to customize how overlong labels are dealt with.
- Use the `.aux` file.
  - Implement the `\ref` styles that enumitem provides.
  - Reverse enumerations, important in publication lists and the like. Somehow avoid needing 3 compilations for references to reverse enumerations to settle?
  - Ability to calculate `\labelwidth` from the label contents. Share calculated parameters between multiple environments (cf. `resume` option).
- Related to grabbing the whole list environment, and input syntax variations:
  - Other layouts: tabular (see listliketab vs typed-checklist), multicolumn and horizontally numbered (see tasks), inline lists, runin lists in the easy case where there is no intervening `\par`.
  - Formatting the item text in a box or similar (requires grabbing the whole list).
  - Filtering which items to show: hide certain items according to criteria (useful together with list reuse), see typed-checklist.
  - Shorthands `\iitem` for automatic nested lists, or `\1`, `\2` etc from outlines.
  - Support markdown input like asciilist.
- Check interaction with `babel` options such as `french` or `accadian` (see FrenchItemizeSpacing)
- RTL and vertical typesetting.

B Plan of attack of first prototype

Typesetting list environments involves a rather large number of parameters. They can be affected by the context such as the total list nesting level, the nesting level of the given type of list, and the font size. An environment like `enumerate` has two main aspects.

- It has a certain layout in the page, with vertical and horizontal spacing around it. This type of layout is shared with environments such as `quote`, `flushright`, or `tabbing`. This common layout is implemented in LaTeX 2_ε through `\trivlist` (or `\list`).

- It defines how each `\item` should be typeset: how to construct the label, in particular the `counter` name, and how to format the content of the item.

This suggests defining two template types, *block* and *item* covering these two aspects.[2] While the *item* type will perhaps have a single template, one could typeset a *block* template in several ways, for instance the standard LaTeX 2ε way or a fancy colored box.

The *general block* template should receive the following parameters. The *plain block* template is a restricted template that freezes all item-related parameters to dummy values (`counter`, `start`, `resume`, `label-width`, `label-sep` and all `item-*`). The *list block* template is a restricted template[3] that omits the `heading` parameter and whose default for `item-instance` is non-empty.

- Structural parameters: the `heading` to place before, `counter` name, `start` value, whether to `resume` a previous list, and the `item-instance` (an *item* instance) to use when typesetting items.
- Vertical spacing and penalties: `begin-penalty`, `begin-vspace`, `begin-extra-vspace`, `item-penalty`, `item-vspace`, `item-par-skip`, `end-penalty`, `end-vspace`, `end-extra-vspace`.
- Horizontal spacing: `right-margin`, `left-margin`, `para-indent`, `item-indent`, `label-width`, `label-sep`.

document class customizations

A document class should edit these templates (or define restricted templates) to set up default values that depend on `\g_block_nesting_depth_int`, namely how many lists are nested overall.[4] The document class should then set up an instance of these templates for each environment, with appropriate settings such as a `heading`, a suitable `item-instance`, or making `margin-right` equal to `margin-left` in a `quote` environment.

The *inline-list block* template receives many fewer parameters. Note that `begin-vspace`, `item-vspace`, `end-vspace` are now *horizontal* skips.

Text wrong and or concept with vspace and hspace questionable!

- Structural parameters: `counter`, `start`, `resume`, `item-instance`.
- Spacing and penalties: `begin-penalty`, `begin-vspace`, `item-penalty`, `item-vspace`, `end-penalty`, `end-vspace`.

revisit!

- Horizontal spacing: `label-width`, `label-sep`.

check!

The *std item* template should receive the following parameters. They depend on the type of list and its nesting level among lists of such type, but typically not on the total nesting level.

- Counter name (`counter`), shared with the parent *list block* template, but needed for incrementing.
- Label construction: a function `counter-label` that produces the label from the counter name, used if `\item` is given without argument.

[2] Possibly also *endblock* to deal with decorations at the end?

[3] A better approach could be to have a notion of inheritance for template types, so that we end up with two different *template types*. Then we can implement other template for the list template type: *table* for lists typeset as rows/columns of a table, *inline* for lists typeset in horizontal mode within a paragraph, and *runin* for run-in lists.

[4] Does xtemplate provide a way to specify default values that are only evaluated once an instance is used?

- References: a function `counter-ref` for how the label should be referred to when it is constructed from the counter, `label-ref` and `label-autoref` used when `\item` has an optional argument.
- Label formatting: `label-format` function, `label-strut` boolean.
- Label alignment (`label-align`, `label-boxed`, `next-line`).
- Content parameters: `text-font`.
- A `compatibility` boolean that controls for instance whether `\makelabel` is used.

document class customizations

The document class should set up an instance such as *enumiii* for each environment and nesting level.[5]

A given environment will adjust some nesting levels, then call the *block* instance appropriate to the environment type, passing it the *item* instance appropriate to the environment and depth. Additional context-dependence could be provided by l3ldb, but the main context-dependence should not rely on it for simplicity reasons and incidentally because l3ldb is not yet available.

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.

[5] This should be made easily extendible to deeper levels.