Vancouver
August 1999

New Interfaces for LaTeX Class Design

LaTeX3 project

David Carlisle
Computational
Mathematics Group
NAG Limited, UK

Frank Mittelbach
Informationstechnologie
und Service
EDS, Germany

Chris Rowley
Faculty of Mathematics
and Computing
Open University, UK

These are the slides and speaker notes for the talk given at TUG99 at Vancouver BC, August 18 1999.

They were not originally intended for public distribution but are being made available at the request of several members of the audience.

$2\varepsilon^*$

Vancouver
August 1999

LaTeX $2\varepsilon^*$?

WWW

- What is LaTeX 2$_\varepsilon$*?
- Why LaTeX 2$_\varepsilon$*?
- When LaTeX 2$_\varepsilon$*?

- LaTeX 2ε* is a collection of packages that run on top of LaTeX 2ε and provide new concepts for various aspects of LaTeX 2ε including
  - Class design interface
  - Document syntax declaration
  - ...
- The biggest open issue for LaTeX3 is the missing design interface. Work on this showed that one has to overcome several internal flaws of LaTeX's kernel. Thus providing a prototype designer interface resulted in rewriting large parts of the LaTeX 2ε kernel at the same time.
- To allow for widespread use during the development phase the work is based (for the moment) on top of LaTeX 2ε.
- First (partial) release is expected this year; final release as Y2K software — perhaps by then it will become LaTeX3.

$2\varepsilon^*$

Vancouver
August 1999

Document syntax declaration

LaTeX 2.09 $\hookrightarrow$ LaTeX 2_ε $\hookrightarrow$ LaTeX 2_{ε}^*

Declaring document syntax in LaTeX 2.09

`\newcommand\foo[`⟨*num*⟩`]{...}`
`\newenvironment{baz}[`⟨*num*⟩`][{...}{...}`

- Declaration supports only commands and environments with mandatory arguments
- No support for declaring commands with optional arguments or star forms or picture coordinate arguments
- Newly declared commands always accept multiple paragraphs in their arguments

Declaring document syntax in LaTeX 2ε (0)

`\newcommand\foo[`⟨*num*⟩`][`⟨*default*⟩`]{...}`
`\renewcommand*\bar[`⟨*num*⟩`][`⟨*default*⟩`]{...}`
`\newenvironment{baz}[`⟨*num*⟩`][`⟨*default*⟩`]{...}{...}`

- Declarations of commands and environments with one optional argument possible (always the first argument optional)
- Declaration of commands possible that do *not* accept multiple paragraphs in their arguments

Declaring document syntax in LaTeX 2ε (1)

```
\def\parbox{%
  \@ifnextchar[%]
    \@iparbox
    {\@iiiparbox c\relax[s]}}
\def\@iparbox[#1]{%
  \@ifnextchar[%]
    {\@iiparbox{#1}}%
    {\@iiiparbox{#1}\relax[s]}}
\def\@iiparbox#1[#2]{%
  \@ifnextchar[%]
    {\@iiiparbox{#1}{#2}}%
    {\@iiiparbox{#1}{#2}[#1]}}
\long\def\@iiiparbox#1#2[#3]#4#5{\leavevmode ... }
```

The first example is from the LaTeX kernel itself. Two questions:

- You all know the `\parbox` command, but even so, can you tell the number of optional and mandatory arguments it offers?

  Can you tell from looking at this code (easily)?

- Do you consider `\@iiiparbox` an acceptable interface to its functionality?

Declaring document syntax in LaTeX 2ε (2)

```
\newcommand\chapter{\if@openright\cleardoublepage
                    \else\clearpage\fi
                    \thispagestyle{plain}%
                    \global\@topnum\z@
                    \@afterindentfalse
                    \secdef\@chapter\@schapter}
\def\@chapter[#1]#2{\ifnum \c@secnumdepth >\m@ne
                         \refstepcounter{chapter}%
                         \typeout{\@chapapp\space\thechapter.}%
                         ...  }
\def\@schapter#1{\if@twocolumn
                   \@topnewpage[\@makeschapterhead{#1}]%
                 \else
                   ...  }
```

Second example, this time from the `report` class.

Here we have a mixture of semantics (for example, "no top floats on this page" and "no indentation after this heading") intermixed with document command syntax, that is: "run `\@schapter` code if you parse a star", or "use mandatory argument as optional argument if none is given", etc.

So how can we do better? We could, for a start, try to fully separate input syntax from formatting and processing issues.

Declaring document syntax with `xparse` (1+2)

```
\DeclareDocumentCommand \parbox { O{c} o O{s} m m}
  {
    \ltx@parbox {#1} {#2} {#3} {#4} {#5}
  }

\DeclareDocumentCommand \chapter { s o o m }
  {
    \UseInstance {heading} {A-heading}
                 {#1} {#2} {#3} {#4}
  }
```

Without going into too much detail at this point: what do we have here? (Both are hypothetical examples)

In case of the `\parbox` declaration we have an internal command (`\ltx@parbox`) that does the actual formatting taking a fixed number of mandatory arguments. The document level syntax for `\parbox` is specified in the second argument to `\DeclareDocumentCommand` showing that `\parbox` will scan up to three optional arguments (providing certain defaults for some of them if missing) followed by two mandatory arguments.

You may wonder what happens with the second optional argument if not present: you will learn about this later on.

In case of the `\chapter` command we see a specification for the document level syntax of an optional star to parse, followed by up to two optional arguments (this time without any default values) followed by a mandatory argument. The internal command being called is some strange beast called `\UseInstance` which we will learn more about later on. What is important though is that this internal command again as a normalised number of arguments (always 4).

So let's look at that interface a little closer.

The `xparse` interface

To specify a command with a LaTeX-like argument syntax

`\DeclareDocumentCommand` ⟨*cmd*⟩ `{` ⟨*arg-spec*⟩ `}{` ⟨*code*⟩ `}`

- ⟨*cmd*⟩ is the name of the document-level command to be defined
- ⟨*arg-spec*⟩ specifies the syntax of this command's arguments
- ⟨*code*⟩ uses normal TeX arguments `#1`, `#2`, . . . , `#n`, where `n` is the number of 'implicit' arguments present.

Thus, when ⟨*cmd*⟩ is used, ⟨*code*⟩ is run as if a command with only mandatory arguments had been there.

The `xparse` interface (arg-spec)

`m` will return the parsed argument surrounded by a brace pair, i.e., will normally be the identity

`o` will return the parsed argument surrounded by a brace pair if present. Otherwise it will return the token `\NoValue`

`O{...}` will return the parsed argument surrounded by a brace pair if present. Otherwise it will return a default value which has to be specified within the brace group following the `O`

`s` will return either the token `\BooleanTrue` or `\BooleanFalse` depending on whether or not a star was parsed

`c` will parse the syntax (⟨*x*⟩,⟨*y*⟩), i.e., a coordinate pair and will return the values for the x- and y-coordinate as two arguments each surrounded by braces

The `xparse` interface (arg-spec)

How to transform the document-level arguments into mandatory arguments (alternate presentation by Mr. anonymous)

spec	**arg-type**	**transform**
`m`	mandatory	{⟨*doc-arg*⟩}
`o`	optional	{⟨*doc-arg*⟩} `\NoValue`
`O{`⟨`default`⟩`}`	optional	{⟨*doc-arg*⟩} {⟨*default*⟩}
`s`	star?	`\BooleanTrue` `\BooleanFalse`
`c`	co-ordinate pair (⟨*x*⟩,⟨*y*⟩)	{⟨*x*⟩}{⟨*y*⟩}

You will notice that all argument specifiers will result in returning either the parsed argument in a brace group or will return a single token. This means that there will always be the same number of arguments that can be passed to the function or functions that do the formatting. More precisely, the internal command has only mandatory arguments.

Since the parsed arguments are available as `#1`, `#2`, etc., there is a limit of up to nine such arguments (in case of coordinate specifications even less since they return two brace groups each).

However for practical purposes this should be by far enough. This could be changed so that, for example, the x- and y-bits of a coordinate are returned as (transformed to) `{{x}{y}}`.

Not supported by `xparse` (yet?)

- Optional coordinate specification
- Variant syntax forms, e.g,

  `\newtheorem{`⟨*name*⟩`}` `{`⟨*caption*⟩`}[`⟨*within*⟩`]`
  `\newtheorem{`⟨*name*⟩`}[`⟨*numbered-like*⟩`]{`⟨*caption*⟩`}`

  or

  `\section *` `{`⟨*heading*⟩`}`
  `\section` `[`⟨*toc*⟩`]{`⟨*heading*⟩`}`

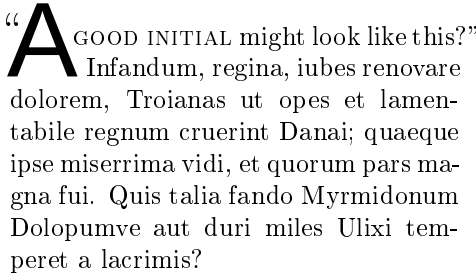
Perhaps some of these features will get added if it turns out that they are important enough.

While some of you may think that the specification possibilities offered by `xparse` are neat (I think they are) — they are not that important.

What is more important is that there is now a clear separation between functionality and document level syntax, which means that either can be individually replaced.

The document syntax can be completely new: for example, it can use an XML vocabulary, with elements and entities expressed in XML syntax.

A Finitial example :-)



produced by

```
\Initial[‘‘]{A}[good initial]  might look like this?’’ ...
```

A Finitial example :-))

A new document command (with no code yet):

```
\DeclareDocumentCommand \Initial {omo}
 {
  %% Produces a special treatment of the
  %% first letter, or words, of a paragraph
  %% #1 o : text that may be before the initial letter
  %% #2 m : the initial letter
  %% #3 o : other text that may require special treatment
  %% <<CODE GOES HERE>>
 }
```

with typical usage:

```
  \Initial[‘‘]{A}[good initial]  might look like this?’’ ...

  \Initial But it might look like this perhaps? ...
```

An Finitial example :-)))

Outline code:

```
\DeclareDocumentCommand \Initial {omo}
 {
  %% The command \MakeInitial is the formatter that produces
  %% an initial character (argument 2) at start of paragraph
  %%  - preceeded by ‘‘quote’’ character(s) (argument 1)
  %%    unless argument 1 is the token \NoValue
  %%  - followed by a number of characters in a special font
  %%    (argument 3) unless argument 3 is the token \NoValue
  \MakeInitial
    {#1}{#2}{#3} }
```

$2\varepsilon^*$

Vancouver
August 1999

Providing customisable layout

Templates and their Instances

Templates

Syntactically:

- A Type and a Template-Name
- Set of named attributes
- Fixed number of mandatory arguments
- Code

Semantically:

- Receives document input through mandatory arguments
- Manipulates this input (and possibly further parts of the document) by executing Code
- Processing of Code is controlled through the named attributes

The concept of templates and their instances is central to the new interface for class file design so we are going to look at this in some detail.

Template examples (1)

Type: `hyphenation`
Args: `none`
Semantics: Sets up hyphenation mechanism

Name: `TeX`
Keys: `uchyph hyphenpenalty exhyphenpenalty ...`

Name: `std`
Keys: `hyphen-disable-boolean`
`hyphen-uppercase-boolean ...`

Template examples (2)

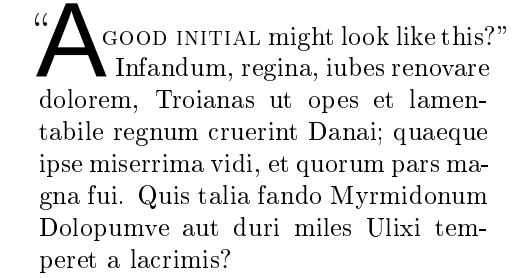
Type:	`initial`
Args:	`1,3 (\NoValue or string) 2 (string)`
Semantics:	Sets up an initial character at start of paragraph followed by a number of characters formatted in a special font. Handles quotes to the left of the initial if any.
Name:	`std`
Keys:	`initial-font initial-format text-font parshape-list v-adjust h-adjust ...`

Such templates can be directly applied, by providing (suitable) values for their keys and passing them the appropriate number of arguments.

Here we see an example. . .

Template usage (direct)

```
\UseTemplate{initial}{std}
 {initial-font  = \fontfamily{pop}\fontsize{40}{40}\selectfont,
  text-font     = \scshape,
  text-sep      = 3pt,
  parshape-list = {-5pt,0pt},
  v-adjust      = 0pt,              h-adjust  = -1pt,
  quote-sep     = -5pt,
  quote-format  = \LARGE #1,
 }
 {‘‘}{A}{good initial} might look like this?’’ ...
```



We see that

- the font for the initial character is chosen to be 40pt Optima
- that the text following it is set in small capitals
- that the initial is dropped two lines into the text and that the shape of the paragraph follows the shape of the letter A
- . . .

We also see that it would be a nightmare if we would have to specify this type of input over and over again in a source document.

Which brings us to the concept of named instances of a template . . .

Instances

Syntactically:

- A Type, a (new) Instance-Name, and a Template-Name
- Set of attribute/value pairs for the Template

Semantically:

- Evaluates attribute values at declaration-time using `calc` expression syntax for dimensions and counters
- Value evaluation can be explicitly delayed to happen at run-time
- At run-time applies Template-Code using stored attribute values to process document input

Instance declaration and usage

```
\DeclareInstance{initial}{A}{std}
 {initial-font  = \fontfamily{pop}\fontsize{40}{40}\selectfont,
  text-font     = \scshape,
  text-sep      = 3pt,
  parshape-list = {-5pt,0pt},
  v-adjust      = 0pt,              h-adjust  = -1pt,
  quote-sep     = -5pt,
  quote-format  = \LARGE #1,
 }

\UseInstance{initial}{A}{``}{A}{good initial}
   might look like this?'' ...
```

Still not very useful this way; the difference to the example with `\UseTemplate` is

- Frozen set of attributes so that the the particular incarnation of this template can be easily (and identically) replicated.
- The resulting "layout" is named rather than explicit and thus can be changed consistently by modifying it in one place, i.e., the instance declaration.
- The run-time usage is faster in processing as all the attribute values are preparsed and are at run-time assigned via primitive TEX operations rather than `calc` processing.

So to make further use of this concept we could apply `xparse` to provide a high-level document interface . . .

Instance usage with together with `xparse`

```
\DeclareDocumentCommand \Initial {omo}
 {
  \IfExistsInstanceTF{initial}{#2}
     {\UseInstance{initial}{#2}}
     {\UseInstance{initial}{default}}
  {#1}{#2}{#3}
 }

\Initial[‘‘]{A}[good initial]  might look like this?’’ ...

\Initial But it might look like this perhaps? ...
```

Explanation:

- The `\Initial` command takes one mandatory argument and tests if there exists an instance of type `initial` with the name of this argument (in case of accented characters this would need to be more elaborate).
- If so it invokes this instance otherwise it invokes an instance named `default`
- Any quote character to the left of the initial character and any text that should be typeset in a special font to the right can be specified through optional arguments to the left and the right of the mandatory argument.
- This means that for setting up initials of a certain type, e.g., dropped, etc., one has to declare a `default` instance to cover the general case and overwrite this for individual characters in case they need special treatment.

Instance declaration with delayed evaluation

```
\DeclareInstance{justification}{raggedright}{TeX}
   {leftskip    = 0pt,
    rightskip   = \DelayEvaluation { 0pt plus 2em },
    startskip   = 0pt,
    parfillskip = \fill,
    spaceskip   = \DelayEvaluation { \fontwordspace },
    xspaceskip  = \DelayEvaluation
                     { \fontwordspace + \fontextraspace },
}
```

This example shows the use of delayed evaluation:

- The `justification` type covers the typesetting of straight text with respect to the margins.
- The instance `raggedright` is supposed to produce unjustified text with a ragged right margin.
- For continuous text the behaviour of `\raggedright` as defined by LaTeX 2_ε is normally not desired as that command will always break text at word boundaries.
- Instead this instance will try to keep the "raggedness" within certain limits. These are defined with respect to the current font size and thus can't be fixed until run-time.
- For the same reason `spaceskip` and `xspaceskip` have to be declared delayed as they depend on the font used.

The Template Type

- Characterises the general behaviour of templates
- Templates of the same type are “exchangeable”
  - identical number of arguments
  - identical interpretation of arguments
  - comparable purpose (in a vague sense)
- For this reason has to be declared via `\DeclareTemplateType`

So far we haven't really given a good reason or example why there is a template type. So here is an explanation of it purpose.

Except for the number of arguments nothing is in practice enforced.

I.e., if people would disobey the restriction posed by the template type concept they will probably get documents that compile without errors. However the resulting documents will most likely show strange formatting.

So let's look at an example for the use of templates types . . .

The Template Type (example)

To format list structures such as `itemize` there might be a template type called `list` with the following characteristics:

- Starts a list structure embedding it into the surrounding formatting
- Sets up a command `\ListItem` (one argument) to start a new 'item' of the list
- Sets up a command `\EndThisList` to finish the current list structure properly
- Expects three arguments with the following semantics:
  - String to calculate width of left indentation, or `\NoValue`
  - Symbol/string to be used as item label, or `\Novalue`
  - Boolean to denote whether or not numbering continues

The arguments have to be present but are allowed to be ignored by templates of that type, e.g., for an inline list the argument specifying indentation is irrelevant.

Ditto non-counting lists do not care about the third argument.

Template examples (3)

Type: `list`
Args: `3 (\NoValue or string each)`
Semantics: Sets up a list environment

Name: `vertical`
Keys: `measure-setup pre-vmaterial-setup item-vmaterial-setup post-vmaterial-setup justification-setup ...`

Name: `inline`
Keys: `pre-hmaterial-setup item-hmaterial-setup post-hmaterial-setup item-label-text ...`

This slide now shows two templates named `vertical` and `inline` belonging to the template type `list`.

To build actual list layouts for a class one would produce instances from these templates.

The Template Type interfaced with `xparse`

```
\DeclareDocumentCommand \item  { o }  { \ListItem {#1} }
\DeclareDocumentEnvironment {enumerate} {  }
 {\UseInstance{list}{enumerate} \NoValue \NoValue \BooleanFalse}
 {\EndThisList}

\DeclareDocumentEnvironment{enumerate*} {  }
 {\UseInstance{list}{enumerate} \NoValue \NoValue \BooleanTrue}
 {\EndThisList}

\DeclareDocumentEnvironment{itemize}  { o }
 {\UseInstance{list}{itemize} \NoValue {#1} \BooleanFalse}
 {\EndThisList}
```

Here we see the mapping between the document level syntax and the layout definitions.

First example is a standard enumerate as in LaTeX 2_ε implemented using the `list` template with an instance named **`enumerate`**.

Note that this declaration only fixes the document interface not its layout! The **`enumerate`** instance might be derived from the template for vertical lists or it might be an instantiation of the inline list template or . . .

Second example defines a list that continues a previous enumeration.

Third example shows an 'itemize' list where the user can overwrite the item label being used at the start of the list.

Template Declaration (technically)

\DeclareTemplateType{⟨*type*⟩}{⟨*arg-no*⟩}

\DeclareTemplate{⟨*type*⟩}{⟨*name*⟩}{⟨*arg-no*⟩}
{
  ⟨*key-name*$_1$⟩ =⟨*key-type*$_1$⟩ ⟨*optional-default*$_1$⟩ ⟨*storage-bin*$_1$⟩,
  ⟨*key-name*$_2$⟩ =⟨*key-type*$_2$⟩ ⟨*optional-default*$_2$⟩ ⟨*storage-bin*$_2$⟩,
    ...
}
{ ⟨*initial-code*⟩ \DoParameterAssignments ⟨*action-code*⟩ }

Template Declaration (technically 2)

- The ⟨*key-name*⟩ is a string composed from the letters a–z, A–Z, digits and/or hyphen characters.
- The ⟨*key-type*⟩ is a string (typically a single letter) denoting the type of the attribute, e.g., a `l` means a LaTeX length register. It might be optionally preceeded by a + which denotes that the assignment is done globally.
- The ⟨*optional-default*⟩ is the value to assign if no value is given at instance declaration time. If present it is specified in brackets.
- The ⟨*storage-bin*⟩ is the register, command, or whatever that receives the value after evaluation.
- Within the code, the `\DoParameterAssigment` denotes the place where the run-time assignments to the ⟨*storage-bin*⟩s are happening.

. . .

Instead of working through all the possibilities in theory let's look again at an example. Here is the part of the actual template definition for initials (as of 31.7.99)

Template Declaration (technically 2)

```
\DeclareTemplate{initial}{std}{3}{
  initial-font   =f0             \initial@font,
  initial-format =f1 [#1]        \initial@boxhandling,
  parshape-list  =f0 [0pt]       \initial@parshape,
  v-adjust       =L              \initial@vadjust,
  ...
  text-font      =f0 [\scshape] \initial@shape,
}
{
  \let\initial@vadjust\z@
  ...
  \DoParameterAssignments
  ...
  \IfValueT{#3}{ {\initial@shape #3} }
}
```

- The key `initial-font` has no default specified. It is central to the template and if missing will result in an error.
- The key `v-adjust` has no default specified either. Instead it gets its default in the ⟨*initial-code*⟩ section before the `\DoParameterAssignments`. Both methods produce the same result but differ in spacing requirements and speed (as well as ease to read the code)
- Last line of the ⟨*action-code*⟩ shows the handling of the third argument to the template: test for the value not being `\NoValue` and if so applying the value of the `text-font` attribute.

Template Declaration (technically 3a)

- Attributes that receive names as values:

```
counter-id      =n  [\heading@id] \heading@counter,
```

- Attributes that receive functions as values:

```
initial-font   =f0             \initial@font,
initial-format =f1 [#1]        \initial@boxhandling,
```

- Attributes that receive dimensions as values:

```
pre-sep        =l   \topsep,
post-sep       =L   \botsep,
```

- The type `n` expects to receive a LaTeX name as a value. Used, for example, to specify the name of a LaTeX counter to use.
- The type `f`⟨*num*⟩ expects a function with ⟨*num*⟩ arguments as a value. The arguments are denoted by `#1`, `#2`, etc. In most cases either `f0` (for declarations) or `f1` (to format one argument) are needed.
- As far as specifying instances the `l` and `L` type behave identically. They differ only in the type of internal storage-bin they need: `l` expects a length register while `L` expects an ordinary macro name and assigns its value via `\def`.
- There might be a need to distinguish between TeX's dimen and skip registers. Right now this is not done and both `l` and `L` accepts what LaTeX calls "rubber length" specifications.

Template Declaration (technically 3b)

- Attributes that receive integers as values:

```
pre-penalty     =c   \@beginparpenalty,
penalty         =C   \hmaterial@penalty,
```

- Attributes that receive template instances as values:

```
justification-setup =i{justification} \list@justification,
```

Usage within an instance declaration is either

```
justification-setup = raggedright,
```

i.e., name of a declared instance or a call to `\UseTemplate`

```
justification-setup = \UseTemplate{justification}{TeX}
                            { startskip = 0pt, ... },
```

- The `c` and `C` type receive integers as values. Again either of them can be transparently used. In case of `c` the ⟨*storage-bin*⟩ has to be a TEX count register not a LATEX counter name, i.e., set up via `\newcount`. (LATEX counters can be used as well if they are accessed via their internal name, i.e., via `\c@`⟨*LATEX-counter*⟩)

- The type `i{`⟨*type*⟩`}` takes as value the name of a declared instance of that type. The ⟨*storage-bin*⟩ associated with the key will store a command essentially equivalent to a call to `\UseInstance{`⟨*type*⟩`}{`⟨*name*⟩`}`, but in a slightly optimised internal form.

  As an exception to this rule the replacement code may be of the form `\UseTemplate` followed by the key settings for the template but without the mandatory arguments. In this case the 'inner' instance declaration is 'pre compiled' and the token assigned to the store the value assigned to this key will execute an instance of the template directly, it will not re-parse the keyword settings each time the instance is used.

Template Declaration (technically 3c)

- Attributes that receive true or false values:

```
item-implicit-boolean  =s
    { \def\item@implicit@code{\item\relax} }{},
numbered-boolean       =b [true] @heading@nums,
```

- Attributes that accept any value:

```
generic-key   =g \typeout{#1},
extra-assigns =x \typeout{#1},
```

- The type `s` expects the strings `true` or `false` as values. In this case the declaration has no ⟨*storage-bin*⟩. Instead the declaration consists of two brace groups containing code. Depending on the value one of the groups gets copied verbatim into the internal parameter list of the instance and gets executed at run-time at the point where `\DoParameterAssignments` is seen.

- The type `b` can probably vanish. It is equivalent to specifying the mutators of a `\newif` command in the brace groups, e.g.

```
numbered-boolean =b [true] @heading@nums,
numbered-boolean =s [true] {\@heading@numstrue}
                           {\@heading@numsfalse},
```

- The type `g` is a low-level specification which contains arbitrary code in place of the ⟨*storage-bin*⟩. This code is evaluated at declaration time of the instance and by default *nothing* is passed to the internal parameter list (this has to happen explicitly from within the code). `#1` may be used to access the value specified.

- The main purpose for this type is of historical nature (originally most of the other types have been implemented internally using `g`).

- The type `x` also requires code in place of the ⟨*storage-bin*⟩. However with this type all of the code is copied unevaluated to the internal parameter list. There are some applications for this type when implementing customisable defaults. However, it is likely that it will not survive a final release.

Values depending on context

Attributes of type `l`, `L`, `c`, `C`, (registers) and `n` or `f0` also support a sort of case structure as their value of the following form:

⟨*key*⟩ = `\MultiSelection` ⟨*counter*⟩
`{` ⟨$value_1$⟩ `,`
⟨$value_2$⟩ `,`
...
⟨$value_n$⟩ `}`
`{` ⟨$value_{otherwise}$⟩ `}`

Values depending on context (example)

```
left-margin-width = \MultiSelection \@listdepth
                                {
                                  \DelayEvaluation {2.5em},
                                  \DelayEvaluation {2.2em},
                                  \DelayEvaluation {1.87em},
                                  \DelayEvaluation {1.7em}
                                 }
                                 { \DelayEvaluation {1em} },
```

The actual syntax for such a multi-selection is still under discussion (well everything to some extend is under discussion but . . .)

A possible alternative, or say variation, is to support a selection based on label strings and there might be other good ideas waiting to be discovered.

For the moment the way it is defined, it offers enough functionality to provide instances in the way we wanted them.

Restricted Templates

Syntactically:

- A Type, a (new) Template-Name, and an existing Template-Name
- Set of attribute/value pairs for the Template

Semantically:

- Evaluates attribute values like in Instance declaration
- The restricted template behaves like the original template but with some of the attributes (pre)set to fixed values
- Restricted templates can be used to build further restricted templates

Current implementation allows an instance to overwrite preset values (this may change).

Restricted Templates (example)

```
\DeclareRestrictedTemplate{list}{vertical-std}{vertical}
{
  left-margin-width    = 20pt,
  right-margin-width   = 0pt,
  ...
  justification-setup           = raggedright,
  item-accumulate-right-boolean = true,
  item-implicit-boolean         = false,
}

\DeclareInstance{list}{itemize}{vertical-std}{
  item-label-text =   \MultiSelection ...
}
```

Collections of Instances aka CollectionInstances

Syntactically:

- A command (`\DeclareCollectionInstance`) to declare template instances that belong to a named collection
- A command (`\UseCollection{`⟨*name*⟩`}{`⟨*type*⟩`}`) to activate the collection ⟨*name*⟩ for the template/instance type ⟨*type*⟩

Semantically:

- If a collection for a template/instance ⟨*type*⟩ is activated, a call to `\UseInstance{`⟨*type*⟩`}{`⟨*name*⟩`}` will first check if a corresponding collection instance is defined and if so use that instance.
- In no such instance is defined or no collection is active, the standard instance of type ⟨*type*⟩ and name ⟨*name*⟩ (i.e., the one defined via `\DeclareInstance`) is used.
- By default no collection is active.

- Collections provide a way to change the layout of certain document commands on a regional level, e.g., a different handling of front matter headings (while using the same document commands) could be implemented by providing a collection `frontmatter` for instances of type `head` (and perhaps others) and switch to this collection within the front matter.
- We will see an example for the use of collections when discussing 'page styles'.

$2\varepsilon^*$

Vancouver
August 1999

Galleys

Some problems with LaTeX 2ε galleys

Competition: What are the effects of

1. `\end{itemize} \vspace{3pt} \begin{itemize} ...`
2. `\section{HEAD} {\sc ABC} ...`
3. `\usepackage{hyperref}`
4. `{\section{HEAD}} para text \par \begin{itemize} ...`

Last question might look like silly input but it can happen in various ways one of which is `\twocolumn[\section{foo}]`!

Some problems with LaTeX 2ε galleys (answers)

1. `\end{itemize} \vspace{3pt} \begin{itemize} ...`

   ... not three points of extra space but 13 (with the `article` class)

2. `\section{HEAD} {\sc ABC} ...`

   ... page breaking restrictions apply to the second paragraph after the heading

Some problems with LaTeX 2ε galleys (answers)

3. `\usepackage{hyperref}`

   . . . the danger of completely changed vertical spacing because the `\specials` added by `hyperref` interfere with the spacing and penalty mechanisms

4. `{\section{HEAD}} para text \par \begin{itemize} ...`

   . . . one gets an indentation after the heading; a page break might happen after the first line in the paragraphs; and there will be no space above the itemize environment (though there will be space below) — because of this LaTeX 2ε claims this is incorrect input :-)

Some problems with LaTeX 2ε galleys (solution)

- Prohibit uncontrolled access to TeX's vertical mode
- Provide a data structure and mutator functions that access this data structure in controlled ways

Galley data structures (needs)

Examples of TEX-level commands that need to be controlled in vertical mode

- Anything that produces a 'whatsit node' —

  `\special` `\write` `\mark` `(and many pdftex commands)`

- penalties and glue

In fact, everything you cannot see!

Galley data structures (needs)

The ideal structure for a vertical list:

⟨*visible material: a box*⟩

⟨*whatsits*⟩ (not ideal: needed only for stuff that should be inside the last box but is 'too late')

⟨*single penalty*⟩

⟨*at most one glob of glue*⟩

⟨*visible material: a box*⟩

Galley data structures (solutions)

- Controlling vmode: `\par`
  - Insert `\nobreak`
- Controlling vmode: `\everypar`
  - Insert data structure, e.g., penalty, glue, whatsits
- Handling user-level stuff between paragraphs
  - Keep separate data structure for user spacing (`\vspace`) and layout spacing
  - Attach whatsits and writes either to previous or upcoming paragraph
- Handling user-level page-control within paragraphs
  - Control page breaking before and in paragraph

$2\varepsilon^*$

Vancouver
August 1999

Controlling Floats

LaTeX2.09 and LaTeX 2ε

- Each float is typeset in a box along with its captions.
- Floats of a given type always appear in sequence.
- A single float environment may contain multiple captions.

2ε*

Flexible caption formatting

- Neither the format nor the position of the caption is fixed at the time the float is specified:
  they may depend on the final area in which the float is positioned.
- Templates will be provided that enable the specification of differing layouts depending on the relative sizes of the float body and caption, and the float area into which a float is being placed.

Flexible caption formatting (consequences)

- Only one caption is allowed per float environment.
- The float placement algorithm can now try differing caption positions (below the body, to the side, in the margin,. . .) This flexibility lessens the chance of a float 'failing to fit' and being deferred to the end of the document.
- This allows for layouts where the formatting depends on the positioning of the float, e.g., caption always in the outer margin.

$2\varepsilon^*$

LaTeX $2_\varepsilon{}^*$: Finer float positioning control

- More general scheme that allows the positioning to depend on the relation to the call-out, for example `[hb|tp]`.
- Mechanism to override all automatic float placement via an external file that specifies exactly the page and area in which each float should be placed.

- `[hb|tp]` allows ‘here’ or ‘bottom’ on the page of the call-out; but ‘top’ or ‘float page’ on all subsequent pages.
- This may be used to give manual control for *final* editing, or for making revisions to a document without danger of triggering a totally different float placement affecting all pages.

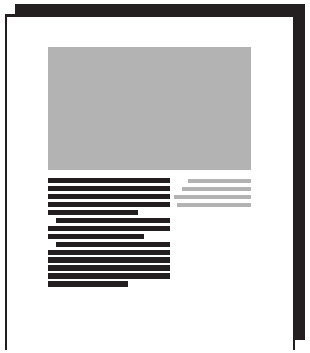
2ε*

More float areas and different handling

Possible extensions being considered include:

- Floats in the margin
- Bottom floats in two column setting
- Multi-column layout with column floats
- Retry last float page as text floats to save space

Caption Positioning examples



$2\varepsilon^*$

Vancouver
August 1999

Page Layout

LaTeX2.09 & LaTeX 2ε

Document level declarations accessing named page ‘styles’.

- `\pagestyle{headings}`
- `\thispagestyle{empty}`

Interactions with document structure are documented,
but cause surprises. . .

- `\pagestyle{empty}` . . . `\maketitle`

The ‘surprise’ is that `\maketitle` expresses the semantic that ‘this is a title page’ by running the command `\thispagestyle{plain}` thus overriding the user specified page style.

Solution is to separate out the declaration of the ‘page type’ from the specification of the formatting required.

Declaring a new page style in LaTeX 2ε

The following is taken from the source for the article class.

```
\if@twoside
  \def\ps@headings{%
      \let\@oddfoot\@empty\let\@evenfoot\@empty
      \def\@evenhead{\thepage\hfil\slshape\leftmark}%
      \def\@oddhead{{\slshape\rightmark}\hfil\thepage}%
      \let\@mkboth\markboth
    \def\sectionmark##1{. . . }%
    \def\subsectionmark##1{. . .}}
\else
  \def\ps@headings{%
    . . . }
\fi
```

Currently LaTeX gives no support for defining new page styles.

To define a page style to be accessed by `\pagestyle{new}` essentially arbitrary code is used to define an internal command, `\ps@new`.

There are packages available to overcome some of the deficiencies, e.g., `fancyhdr`.

Declaring Page Styles in LaTeX 2ε*

- Structural commands such as `\maketitle` and `\chapter` specify the *Page Type* for a page (or range of pages).
- *Page Styles* define a standard page format, and for any of the page types, may specify a different format.
- Page formats are specified as collections of instances of templates of type `pagestyle`.

Page Types

The code for a document structure command can specify the sequence of *types* of pages that should be used.

Some page types:

- standard
- title
- preopening
- opening
- postopening
- . . .

For example a chapter heading may produce a page of type ‘pre-opening’ in order to force the current page to be a recto (or verso) page, followed by a page of type ‘opening’ that contains the title of the chapter, and possibly the beginning of the chapter text.

Page Styles (1)

```
\DeclareInstance{pagestyle}{standard}{3part}
 {  recto-head-left-action  = \rightmark,
    recto-head-right-action = \thepage,
    head-rule-width         = .4pt,
    . . .   }

\DeclareInstance{pagestyle}{opening}{3part}
 {  recto-foot-center-action = \thepage ,
    verso-foot-center-action = \thepage ,
    . . .   }

\DeclareInstance{pagestyle}{preopening}{3part} {}
```

The ‘normal’ page style is implemented as a collection of instances of type `pagestyle`. In this case all are implemented as instances of a template ‘3part’ that allows specification of head and foot sections in three parts, with possible decorations such as rules. (The popular `fancyhdr` package gives an interface to these types of page styles for LaTeX 2ε.)

The standard layout is used on most pages, but opening pages (for example chapter title pages) omit the headline information and center the footline.

If a pre-opening page is produced to bring the opening page on to a recto (odd) page then this will have blank head and foot.

Page Styles (2)

The examples used a template ‘3part’ which may be used to specify simple ‘3 part’ structure of headline and footline.

More general templates will also be provided with more advanced features.

- Head and foot area extending into margins.
- Altering the height of individual pages.
- Altering the width (measure) of individual pages. (This feature may only be used in restricted contexts due to limitations in TEX’s page breaking algorithm.
- Altering the position of text columns within the page.
- . . .

Page Styles (3)

```
\DeclareDocumentCommand \pagestyle { m }
   { \UseCollection{pagestyle}{#1} }

\DeclareCollectionInstance{empty}{pagestyle}{standard}{3part}{}
\DeclareCollectionInstance{empty}{pagestyle}{opening}{3part}{}
\DeclareCollectionInstance{empty}{pagestyle}{preopening}{3part}{}
...
```

- For classes with many layout variation possibilities a `\pagestyle` command for changing the page style could be implemented by changing the current collection of instances of type `pagestyle`.
- An `empty` page style can, for example, be defined as a collection of this type. (The example assumes that the `3part` template has default values for all of its keys that result in empty headers and footers.)
- In more complicated classes with 'front matter' and 'back matter' parts the different regions could overwrite the page type definitions by providing collections with suitable names and internally switching to them.

$2\varepsilon^*$

Vancouver
August 1999

Front Matter (Journal Articles)

LaTeX2.09 & LaTeX 2ε

The standard article class offers only very simple front matter declarations, and in particular offers a single `\author` command that is to be used to declare all authors and their addresses.

This model does not really support a typical multi-author, multi-address journal article.

Many journal-specific classes support extended interfaces, but this offers no document portability between classes, and there is no interface to help the class designer implement an interface to front matter declarations.

Front Matter Layout Templates (1)

By providing a range of templates for specifying front matter formatting, LaTeX 2_{ε}* enables a class designer to easily specify a wide range of layouts that use the same document level commands.

The following slide shows one of the prototype template declarations developed this spring.

Front Matter Layout Templates (2)

```
\DeclareTemplate{titlesetup}{std}{0}{

  title-format          =f1 \maketitle@title,
  subtitle-format       =f1 \maketitle@subtitle,

  authors-format        =f1 \maketitle@author,
  addresses-format      =f1 \maketitle@address,
  abstract-format       =f1 \maketitle@abstract,

  author-setup          =i{author} [simple] \do@author,

  author-address-handling =n [grouped] \title@address@handling,

  and-text              =f0 [and] \andname,
  formatting-sequence   =f0 [{title,author,address,abstract}]
                            \maketitle@sequence,
 }
```

Each attribute such as `title-format` receives as argument data from the document level commands, in this case the argument from the command `\title`.

When declaring an instance of this template the class designer specifies, for each such attribute, the commands code that format the text and adds any required vertical space *above* the item.

The order in which the various fields are output may be specified by giving a sequence, otherwise the default sequence is used.

The `author-address-handling` attribute controls whether authors are grouped by address.

The `author-setup` attribute takes an instance of type `author`. The data structure for authors is more complicated as each author is associated not just with a name, but with one or more addresses, email, web home pages, and possibly other information.

Templates of type `author` will be provided to handle this data structure.

Front Matter Document Commands

The document level commands for such a class, would be similar to the current usage in amslatex classes, or revtex 4. However the template interface makes it much easier to specify a range of typographic styles for the same document markup.

In brief the document level syntax has one `\author` command for *each* author, followed by `\address`, `\email` and similar commands. Mechanisms to specify multiple authors sharing address, and also multiple addresses for a single author will be implemented but are not detailed here.

Vancouver
August 1999

The Status of LaTeX 2_{ε}*

Functional

- Document command interface
- Template/Instance interface
- Galley data structure
- Boxes with several reference points
- List templates
- Hyphenation and justification templates
- Templates for initials

Proto-types (under the knife)

- Heading templates
- Table of contents templates
- Front matter handling
- Output routine redesign (float handling)
- Data structure for multiple marks
- Page layout handling