LaTeX for authors — current version

2023-05-23

Contents

1 Introduction

LaTeX 2$_\varepsilon$ was released in 1994 and added a number of then-new concepts to LaTeX. These are described in `usrguide-historic`, which has largely remained unchanged. Since then, the LaTeX team have worked on a number of ideas, firstly a programming language for LaTeX (expl3) and then a range of tools for document authors which build on that language. Here, we describe *stable* and *widely-usable* concepts that have resulted from that work. These 'new' ideas have been transferred from development packages into the LaTeX 2$_\varepsilon$ kernel. As such, they are now available to *all* LaTeX users and have the *same stability* as any other part of the kernel. The fact that 'behind the scenes' they are built on expl3 is useful for the development team, but is not directly important to users.

2 Creating document commands and environments

2.1 Overview

Creating document commands and environments using the LaTeX3 toolset is based around the idea that a common set of descriptions can be used to cover almost all argument types used in real documents. Thus parsing is reduced to a simple description of which arguments a command takes: this description provides the 'glue' between the document syntax and the implementation of the command.

First, we will describe the argument types, then move on to explain how these can be used to create both document commands and environments. Various more specialized features are then described, which allow an even richer application of a simple interface set up.

The details here are intended to help users create document commands in general. More technical detail, suitable for TeX programmers, is included in `interface3`.

2.2 Describing argument types

In order to allow each argument to be defined independently, the parser does not simply need to know the number of arguments for a function, but also the nature of each one. This is done by constructing an *argument specification*, which defines the number of arguments, the type of each argument and any additional information needed for the parser to read the user input and properly pass it through to internal functions.

The basic form of the argument specifier is a list of letters, where each letter defines a type of argument. As will be described below, some of the types need additional information, such as default values. The argument types can be divided into two, those which define arguments that are mandatory (potentially raising an error if not found) and those which define optional arguments. The mandatory types

`m` A standard mandatory argument, which can either be a single token alone or multiple tokens surrounded by curly braces `{}`. Regardless of the input, the argument will be passed to the internal code without the outer braces. This is the type specifier for a normal TEX argument.

`r` Given as `r`⟨*token1*⟩⟨*token2*⟩, this denotes a 'required' delimited argument, where the delimiters are ⟨*token1*⟩ and ⟨*token2*⟩. If the opening delimiter ⟨*token1*⟩ is missing, the default marker `-NoValue-` will be inserted after a suitable error.

`R` Given as `R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`, this is a 'required' delimited argument as for `r`, but it has a user-definable recovery ⟨*default*⟩ instead of `-NoValue-`.

`v` Reads an argument 'verbatim', between the following character and its next occurrence, in a way similar to the argument of the LATEX 2_ε command `\verb`. Thus a `v`-type argument is read between two identical characters, which cannot be any of `%`, `\`, `#`, `{`, `}` or ␣. The verbatim argument can also be enclosed between braces, `{` and `}`. A command with a verbatim argument will produce an error when it appears within an argument of another function.

`b` Only suitable in the argument specification of an environment, it denotes the body of the environment, between `\begin{`⟨*environment*⟩`}` and `\end{`⟨*environment*⟩`}`. See Section 2.11 for details.

The types which define optional arguments are:

`o` A standard LATEX optional argument, surrounded with square brackets, which will supply the special `-NoValue-` marker if not given (as described later).

`d` Given as `d`⟨*token1*⟩⟨*token2*⟩, an optional argument which is delimited by ⟨*token1*⟩ and ⟨*token2*⟩. As with `o`, if no value is given the special marker `-NoValue-` is returned.

`O` Given as `O{`⟨*default*⟩`}`, is like `o`, but returns ⟨*default*⟩ if no value is given.

`D` Given as `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`, it is as for `d`, but returns ⟨*default*⟩ if no value is given. Internally, the `o`, `d` and `O` types are short-cuts to an appropriated-constructed `D` type argument.

`s` An optional star, which will result in a value `\BooleanTrue` if a star is present and `\BooleanFalse` otherwise (as described later).

`t` An optional ⟨*token*⟩, which will result in a value `\BooleanTrue` if ⟨*token*⟩ is present and `\BooleanFalse` otherwise. Given as `t`⟨*token*⟩.

`e` Given as `e{`⟨*tokens*⟩`}`, a set of optional *embellishments*, each of which requires a *value*. If an embellishment is not present, `-NoValue-` is returned. Each embellishment gives one argument, ordered as for the list of ⟨*tokens*⟩ in the argument specification. All ⟨*tokens*⟩ must be distinct.

`E` As for `e` but returns one or more ⟨*defaults*⟩ if values are not given: `E{`⟨*tokens*⟩`}{`⟨*defaults*⟩`}`. See Section 2.7 for more details.

2.3 Modifying argument descriptions

In addition to the argument *types* discussed above, the argument description also gives special meaning to three other characters.

First, `+` is used to make an argument long (to accept paragraph tokens). In contrast to `\newcommand`, this applies on an argument-by-argument basis. So modifying the example to '`s o o +m O{default}`' means that the mandatory argument is now `\long`, whereas the optional arguments are not.

Secondly, `!` is used to control whether spaces are allowed before optional arguments. There are some subtleties to this, as TEX itself has some restrictions on where spaces can be 'detected': more detail is given in Section 2.6.

Thirdly, `=` is used to declare that the following argument should be interpreted as a series of keyvals. See Section 2.9 for more details.

Finally, the character `>` is used to declare so-called 'argument processors', which can be used to modify the contents of an argument before it is passed to the macro definition. The use of argument processors is a somewhat advanced topic, (or at least a less commonly used feature) and is covered in Section 2.10.

2.4 Creating document commands and environments

`\NewDocumentCommand {`⟨*cmd*⟩`} {`⟨*arg spec*⟩`} {`⟨*code*⟩`}`
`\RenewDocumentCommand {`⟨*cmd*⟩`} {`⟨*arg spec*⟩`} {`⟨*code*⟩`}`
`\ProvideDocumentCommand {`⟨*cmd*⟩`} {`⟨*arg spec*⟩`} {`⟨*code*⟩`}`
`\DeclareDocumentCommand {`⟨*cmd*⟩`} {`⟨*arg spec*⟩`} {`⟨*code*⟩`}`

This family of commands are used to create a ⟨*cmd*⟩. The argument specification for the function is given by ⟨*arg spec*⟩, and the command uses the ⟨*code*⟩ with `#1`, `#2`, etc. replaced by the arguments found by the parser.

An example:

```
\NewDocumentCommand\chapter{s o m}
  {%
    \IfBooleanTF{#1}%
      {\typesetstarchapter{#3}}%
      {\typesetnormalchapter{#2}{#3}}%
  }
```

would be a way to define a `\chapter` command which would essentially behave like the current LaTeX 2ε command (except that it would accept an optional argument even when a `*` was parsed). The `\typesetnormalchapter` could test its first argument for being `-NoValue-` to see if an optional argument was present. (See Section 2.8 for details of `\IfBooleanTF` and testing for `-NoValue-`.)

The difference between the `\New...` `\Renew...`, `\Provide...` and `\Declare...` versions is the behavior if ⟨*cmd*⟩ is already defined.

- `\NewDocumentCommand` will issue an error if ⟨*cmd*⟩ has already been defined.

- `\RenewDocumentCommand` will issue an error if ⟨*cmd*⟩ has not previously been defined.
- `\ProvideDocumentCommand` creates a new definition for ⟨*function*⟩ only if one has not already been given.
- `\DeclareDocumentCommand` will always create the new definition, irrespective of any existing ⟨*cmd*⟩ with the same name. This should be used sparingly.

If the ⟨*cmd*⟩ can't be provided as a single token but needs "constructing", you can use `\ExpandArgs` as explained in Section 4 which also gives an example in which this is needed.

`\NewDocumentEnvironment {`⟨*env*⟩`} {`⟨*arg spec*⟩`} {`⟨*beg-code*⟩`} {`⟨*end-code*⟩`}`
`\RenewDocumentEnvironment {`⟨*env*⟩`} {`⟨*arg spec*⟩`} {`⟨*beg-code*⟩`} {`⟨*end-code*⟩`}`
`\ProvideDocumentEnvironment {`⟨*env*⟩`} {`⟨*arg spec*⟩`} {`⟨*beg-code*⟩`} {`⟨*end-code*⟩`}`
`\DeclareDocumentEnvironment {`⟨*env*⟩`} {`⟨*arg spec*⟩`} {`⟨*beg-code*⟩`} {`⟨*end-code*⟩`}`

These commands work in the same way as `\NewDocumentCommand`, etc., but create environments (`\begin{`⟨*env*⟩`}` ... `\end{`⟨*env*⟩`}`). Both the ⟨*beg-code*⟩ and ⟨*end-code*⟩ may access the arguments as defined by ⟨*arg spec*⟩. The arguments will be given following `\begin{`⟨*env*⟩`}`.

2.5 Optional arguments

In contrast to commands created using LaTeX 2ε's `\newcommand`, optional arguments created using `\NewDocumentCommand` may safely be nested. Thus for example, following

```
\NewDocumentCommand\foo{om}{I grabbed ‘#1’ and ‘#2’}
\NewDocumentCommand\baz{o}{#1-#1}
```

using the command as

```
\foo[\baz[stuff]]{more stuff}
```

will print

> I grabbed ‘stuff-stuff’ and ‘more stuff’

This is particularly useful when placing a command with an optional argument *inside* the optional argument of a second command.

When an optional argument is followed by a mandatory argument with the same delimiter, the parser issues a warning because the optional argument could not be omitted by the user, thus becoming in effect mandatory. This can apply to `o`, `d`, `O`, `D`, `s`, `t`, `e`, and `E` type arguments followed by `r` or `R`-type required arguments.

The default for `O`, `D` and `E` arguments can be the result of grabbing another argument. Thus for example

```
\NewDocumentCommand\foo{O{#2} m}
```

would use the mandatory argument as the default for the leading optional one.

2.6 Spacing and optional arguments

TEX will find the first argument after a function name irrespective of any intervening spaces. This is true for both mandatory and optional arguments. So `\foo[arg]` and `\foo␣[arg]` are equivalent. Spaces are also ignored when collecting arguments up to the last mandatory argument to be collected (as it must exist). So after

```
\NewDocumentCommand\foo{m o m}{ ... }
```

the user input `\foo{arg1}[arg2]{arg3}` and `\foo{arg1}␣[arg2]␣{arg3}` will both be parsed in the same way.

The behavior of optional arguments *after* any mandatory arguments is selectable. The standard settings will allow spaces here, and thus with

```
\NewDocumentCommand\foobar{m o}{ ... }
```

both `\foobar{arg1}[arg2]` and `\foobar{arg1}␣[arg2]` will find an optional argument. This can be changed by giving the modified `!` in the argument specification:

```
\NewDocumentCommand\foobar{m !o}{ ... }
```

where `\foobar{arg1}␣[arg2]` will not find an optional argument.

There is one subtlety here due to the difference in handling by TEX of 'control symbols', where the command name is made up of a single character, such as '\\'. Spaces are not ignored by TEX here, and thus it is possible to require an optional argument directly follow such a command. The most common example is the use of \\ in amsmath environments, which in the terms here would be defined as

```
\NewDocumentCommand\\{!s !o}{ ... }
```

Also notable when using optional arguments in the last position is that TEX will necessarily look ahead for the argument opening token. This means that the value of `\inputlineno` will be 'out by one' if such a trailing optional argument is *not* present and the command ends a line; it will be one greater than the line number containing the last mandatory argument.

2.7 'Embellishments'

The E-type argument allows one default value per test token. This is achieved by giving a list of defaults for each entry in the list, for example:

```
E{^_}{{UP}{DOWN}}
```

If the list of default values is *shorter* than the list of test tokens, the special -NoValue- marker will be returned (as for the e-type argument). Thus for example

```
E{^_}{{UP}}
```

has default UP for the ^ test character, but will return the -NoValue- marker as a default for _. This allows mixing of explicit defaults with testing for missing values.

2.8 Testing special values

Optional arguments make use of dedicated variables to return information about the nature of the argument received.

```
\IfNoValueTF {⟨arg⟩} {⟨true code⟩} {⟨false code⟩}
\IfNoValueT {⟨arg⟩} {⟨true code⟩}
\IfNoValueF {⟨arg⟩} {⟨false code⟩}
```

The \IfNoValue(TF) tests are used to check if ⟨*argument*⟩ (#1, #2, *etc.*) is the special -NoValue- marker. For example

```
\NewDocumentCommand\foo{o m}
  {%
    \IfNoValueTF {#1}%
      {\DoSomethingJustWithMandatoryArgument{#2}}%
      {\DoSomethingWithBothArguments{#1}{#2}}%
  }
```

will use a different internal function if the optional argument is given than if it is not present.

Note that three tests are available, depending on which outcome branches are required: \IfNoValueTF, \IfNoValueT and \IfNoValueF.

As the \IfNoValue(TF) tests are expandable, it is possible to test these values later, for example at the point of typesetting or in an expansion context.

It is important to note that -NoValue- is constructed such that it will *not* match the simple text input -NoValue-, i.e. that

```
\IfNoValueTF{-NoValue-}
```

will be logically `false`. When two optional arguments follow each other (a syntax we typically discourage), it can make sense to allow users of the command to specify only the second argument by providing an empty first argument. Rather than testing separately for emptiness and for `-NoValue-` it is then best to use the argument type `O` with an empty default value, and then test for emptiness using the conditional `\IfBlankTF` (described below) instead.

New description 2022/06/01

`\IfValueTF {`⟨*arg*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfValueT {`⟨*arg*⟩`} {`⟨*true code*⟩`}`
`\IfValueF {`⟨*arg*⟩`} {`⟨*false code*⟩`}`

The reverse form of the `\IfNoValue(TF)` tests are also available as `\IfValue(TF)`. The context will determine which logical form makes the most sense for a given code scenario.

`\IfBlankTF {`⟨*arg*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfBlankT {`⟨*arg*⟩`} {`⟨*true code*⟩`}`
`\IfBlankF {`⟨*arg*⟩`} {`⟨*false code*⟩`}`

New feature 2022/06/01

The `\IfNoValueTF` command chooses the ⟨*true code*⟩ if the optional argument has not been used at all (and it returns the special `-NoValue-` marker), but not if it has been given an empty value. In contrast `\IfBlankTF` returns true if its argument is either truly empty or only contains one or more normal blanks. For example

```
\NewDocumentCommand\foo{m!o}{\par #1:
  \IfNoValueTF{#2}
    {No optional}%
    {%
      \IfBlankTF{#2}
        {Blanks in or empty}%
        {Real content in}%
    }%
  \space argument!}
\foo{1}[bar] \foo{2}[  ] \foo{3}[] \foo{4}[\space] \foo{5} [x]
```

results in the following output:

1: Real content in argument!
2: Blanks in or empty argument!
3: Blanks in or empty argument!
4: Real content in argument!
5: No optional argument! [x]

Note that the `\space` in (4) is considered real content—because it is a command and not a "space" character—even though it results in producing a space. You can also observe in (5) the effect of the `!` specifier, preventing the last `\foo` from interpreting `[x]` as its optional argument.

```
\BooleanFalse
\BooleanTrue
```

The `true` and `false` flags set when searching for an optional character (using `s` or `t`⟨*char*⟩) have names which are accessible outside of code blocks.

```
\IfBooleanTF {⟨arg⟩} {⟨true code⟩} {⟨false code⟩}
\IfBooleanT {⟨arg⟩} {⟨true code⟩}
\IfBooleanF {⟨arg⟩} {⟨false code⟩}
```

Used to test if ⟨*argument*⟩ (`#1`, `#2`, *etc.*) is `\BooleanTrue` or `\BooleanFalse`. For example

```
\NewDocumentCommand\foo{sm}
  {%
    \IfBooleanTF {#1}%
      {\DoSomethingWithStar{#2}}%
      {\DoSomethingWithoutStar{#2}}%
  }
```

checks for a star as the first argument, then chooses the action to take based on this information.

2.9 Auto-converting to key–value format

Some document commands have a long history of accepting a 'free text' optional argument, for example `\caption` and the sectioning commands `\section`, etc. Introducing more sophisticated (keyval) options to these commands therefore needs a method to interpret the optional argument *either* as free text *or* as a series of keyvals. This needs to take place during argument grabbing as there is a need for careful treatment of braces to obtain the correct result.

The `=` modifier is available to allow ltcmd to correctly implement this process. The modifier guarantees that the argument will be passed to further code as a series of keyvals. To do that, the `=` should be followed by an argument containing the default key name. This is used as the key in a key–value pair *if* the 'raw' argument does *not* have the correct form to be interpreted as a set of keyvals.

Taking `\caption` as an example, with the demonstration implementation

```
\DeclareDocumentCommand
  \caption
  {s ={short-text} +O{#3} +m}
  {%
    \showtokens{Grabbed arguments:^^J(#2)^^Jand^^J(#3)}%
  }
```

the default key name is `short-text`. When the command `\caption` is then used, if the optional argument is free text such as

```
\caption[Some short text]{A much longer and more detailed text for
  demonstration purposes}
```

then the output will be

```
Grabbed arguments:
(short-text={Some short text})
and
(A much longer and more detailed text for demonstration purposes)
```

On the other hand, if the caption is given with a keyval-form argument

```
\caption[label = cap:demo]%
  {A much longer and more detailed text for demonstration purposes}
```

then this will be respected

```
Grabbed arguments:
(label = cap:demo)
and
(A much longer and more detailed text for demonstration purposes)
```

Interpretation as keyval form is determined by the presence of `=` characters within the argument. Those in inline math mode (enclosed within `$...$` or `\(...\)`) are ignored. An argument can be forced to be read as keyvals by including an empty entry at the start

```
\caption[=,This is now a keyval]%
% ...
\caption[This is not $=$ keyval]%
```

This empty entry is *not* passed to the underlying code, so will not lead to issues with keyval parsers that do not allow an empty key name. Any text-mode `=` signs will need to be braced to avoid being misinterpreted: this is likely most conveniently handled by bracing the entire argument

```
\caption[{Not = to a keyval!}]%
```

which will be passed correctly as

```
Grabbed arguments:
(short-text = {Not = to a keyval!})
```

2.10 Argument processors

Argument processor are applied to an argument *after* it has been grabbed by the underlying system but before it is passed to ⟨*code*⟩. An argument processor can therefore be used to regularize input at an early stage, allowing the internal functions to be completely independent of input form. Processors are applied to user input and to default values for optional arguments, but *not* to the special `-NoValue-` marker.

Each argument processor is specified by the syntax `>{`⟨*processor*⟩`}` in the argument specification. Processors are applied from right to left, so that

```
>{\ProcessorB} >{\ProcessorA} m
```

would apply `\ProcessorA` followed by `\ProcessorB` to the tokens grabbed by the `m` argument.

`\SplitArgument {`⟨*number*⟩`} {`⟨*token(s)*⟩`}`

This processor splits the argument given at each occurrence of the ⟨*tokens*⟩ up to a maximum of ⟨*number*⟩ tokens (thus dividing the input into ⟨*number*⟩ + 1 parts). An error is given if too many ⟨*tokens*⟩ are present in the input. The processed input is placed inside ⟨*number*⟩ + 1 sets of braces for further use. If there are fewer than `{`⟨*number*⟩`}` of `{`⟨*tokens*⟩`}` in the argument then `-NoValue-` markers are added at the end of the processed argument.

```
\NewDocumentCommand \foo {>{\SplitArgument{2}{;}} m}
  {\InternalFunctionOfThreeArguments#1}
```

If only a single character ⟨*token*⟩ is used for the split, any category code 13 (active) character matching the ⟨*token*⟩ will be replaced before the split takes place. Spaces are trimmed at each end of each item parsed.

The `E` argument type is somewhat special, because with a single `E` in the command declaration you may end up with several arguments in a command (one formal argument per embellishment token). Therefore, when an argument processor is applied to an `E`-type argument, all the arguments pass through that processor before being fed to the ⟨*code*⟩. For example, this command

```
\NewDocumentCommand \foo { >{\TrimSpaces} e{_^} }
  { [#1](#2) }
```

applies `\TrimSpaces` to both arguments.

`\SplitList {`⟨*token(s)*⟩`}`

This processor splits the argument given at each occurrence of the ⟨*token(s)*⟩ where the number of items is not fixed. Each item is then wrapped in braces within `#1`. The result is that the processed argument can be further processed using a mapping function (see below).

```
\NewDocumentCommand \foo {>{\SplitList{;}} m}
  {\MappingFunction#1}
```

If only a single character ⟨*token*⟩ is used for the split, any category code 13 (active) character matching the ⟨*token*⟩ will be replaced before the split takes place. Spaces are trimmed at each end of each item parsed. Exactly one set of braces will be stripped if an entire item is surrounded by them, i.e. the following inputs and outputs result (each separte item as a brace group).

```
a      ==> {a}
{a}    ==> {a}
{a}b   ==> {{a}b}
a,b    ==> {a}{b}
{a},b  ==> {a}{b}
a,{b}  ==> {a}{b}
a,{b}c ==> {a}{{b}c}
```

`\ProcessList {`⟨*list*⟩`} {`⟨*function*⟩`}`

To support `\SplitList`, the function `\ProcessList` is available to apply a ⟨*function*⟩ to every entry in a ⟨*list*⟩. The ⟨*function*⟩ should absorb one argument: the list entry. For example

```
\NewDocumentCommand \foo {>{\SplitList{;}} m}
  {\ProcessList{#1}{\SomeDocumentCommand}}
```

`\ReverseBoolean`

This processor reverses the logic of `\BooleanTrue` and `\BooleanFalse`, so that the example from earlier would become

```
\NewDocumentCommand\foo{>{\ReverseBoolean} s m}
  {%
    \IfBooleanTF#1%
      {\DoSomethingWithoutStar{#2}}%
      {\DoSomethingWithStar{#2}}%
  }
```

`\TrimSpaces`

Removes any leading and trailing spaces (tokens with character code 32 and category code 10) for the ends of the argument. Thus for example declaring a function

```
\NewDocumentCommand\foo {>{\TrimSpaces} m}
  {\showtokens{#1}}
```

and using it in a document as

```
  \foo{␣hello␣world␣}
```

will show '`hello␣world`' at the terminal, with the space at each end removed. `\TrimSpaces` will remove multiple spaces from the ends of the input in cases where these have been included such that the standard TEX conversion of multiple spaces to a single space does not apply.

2.11 Body of an environment

While environments `\begin{`⟨*environment*⟩`}` ... `\end{`⟨*environment*⟩`}` are typically used in cases where the code implementing the ⟨*environment*⟩ does not need to access the contents of the environment (its 'body'), it is sometimes useful to have the body as a standard argument.

This is achieved by ending the argument specification with `b`, which is a dedicated argument type for this situation. For instance

```
\NewDocumentEnvironment{twice} {O{\ttfamily} +b}
  {#2#1#2} {}
\begin{twice}[\itshape]
  Hello world!
\end{twice}
```

typesets 'Hello world!*Hello world!*'.

The prefix `+` is used to allow multiple paragraphs in the environment's body. Argument processors can also be applied to `b` arguments. By default, spaces are trimmed at both ends of the body: in the example there would otherwise be spaces coming from the ends the lines after `[\itshape]` and `world!`. Putting the prefix `!` before `b` suppresses space-trimming.

When `b` is used in the argument specification, the last argument of the environment declaration (e.g., `\NewDocumentEnvironment`), which consists of an ⟨*end code*⟩ to insert at `\end{`⟨*environment*⟩`}`, is redundant since one can simply put that code at the end of the ⟨*start code*⟩. Nevertheless this (empty) ⟨*end code*⟩ must be provided.

Environments that use this feature can be nested.

2.12 Fully-expandable document commands

Document commands created using `\NewDocumentCommand`, etc., are normally created so that they do not expand unexpectedly. This is done using engine features, so is more powerful than LATEX 2ε's `\protect` mechanism. There are *very rare* occasion when it may be useful to create functions using a expansion-only grabber. This imposes a number of restrictions on the nature of the arguments accepted by a function, and the code it implements. This facility should only be used when *necessary*.

```
\NewExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
\RenewExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
\ProvideExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
\DeclareExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
```

This family of commands is used to create a document-level ⟨*function*⟩, which will grab its arguments in a fully-expandable manner. The argument specification for the function is given by ⟨*arg spec*⟩, and the function will execute ⟨*code*⟩. In general, ⟨*code*⟩ will also be fully expandable, although it is possible that this will not be the case (for example, a function for use in a table might expand so that `\omit` is the first non-expandable non-space token).

Parsing arguments by pure expansion imposes a number of restrictions on both the type of arguments that can be read and the error checking available:

- The last argument (if any are present) must be one of the mandatory types `m`, `r` or `R`.
- The 'verbatim' argument type `v` is not available.
- Argument processors (using `>`) are not available.
- It is not possible to differentiate between, for example `\foo[` and `\foo{[}`: in both cases the `[` will be interpreted as the start of an optional argument. As a result, checking for optional arguments is less robust than in the standard version.

2.13 Commands at the start of tabular cells

Creating commands that are used at the start of tabular cells imposes some restrictions on the underlying implementation. The standard LaTeX tabular environments (`tabular`, etc.) use a mechanism which requires that any command wrapping `\multicolumn` or similar must be 'expandable'. This is *not* the case for commands created using `\NewDocumentCommand`, etc., which as detailed in Section 2.12 use an engine feature which prevents such 'expansion'. Therefore, to create such wrappers for use at the start of tabular cells, you must use `\NewExpandableDocumentCommand`, for example

```
\NewExpandableDocumentCommand\MyMultiCol{m}{\multicolumn{3}{c}{#1}}
\begin{tabular}{lcr}
a & b & c \\
\MyMultiCol{stuff} \\
\end{tabular}
```

2.14 Details about argument delimiters

In normal (non-expandable) commands, the delimited types look for the initial delimiter by peeking ahead (using expl3's `\peek_...` functions) looking for the delimiter token. The token has to have the same meaning and 'shape' of the token defined as delimiter. There are three possible cases of delimiters: character

tokens, control sequence tokens, and active character tokens. For all practical purposes of this description, active character tokens will behave exactly as control sequence tokens.

2.14.1 Character tokens

A character token is characterized by its character code, and its meaning is the category code (`\catcode`). When a command is defined, the meaning of the character token is fixed into the definition of the command and cannot change. A command will correctly see an argument delimiter if the open delimiter has the same character and category codes as at the time of the definition. For example in:

```
\NewDocumentCommand { \foobar } { D<>{default} } {(#1)}
\foobar <hello> \par
\char_set_catcode_letter:N <
\foobar <hello>
```

the output would be:

```
(hello)
(default)<hello>
```

as the open-delimiter `<` changed in meaning between the two calls to `\foobar`, so the second one doesn't see the `<` as a valid delimiter. Commands assume that if a valid open-delimiter was found, a matching close-delimiter will also be there. If it is not (either by being omitted or by changing in meaning), a low-level TEX error is raised and the command call is aborted.

2.14.2 Control sequence tokens

A control sequence (or control character) token is characterized by is its name, and its meaning is its definition. A token cannot have two different meanings at the same time. When a control sequence is defined as delimiter in a command, it will be detected as delimiter whenever the control sequence name is found in the document regardless of its current definition. For example in:

```
\cs_set:Npn \x { abc }
\NewDocumentCommand { \foobar } { D\x\y{default} } {(#1)}
\foobar \x hello\y \par
\cs_set:Npn \x { def }
\foobar \x hello\y
```

the output would be:

```
(hello)
(hello)
```

with both calls to the command seeing the delimiter `\x`.

2.15 Creating new argument processors

`\ProcessedArgument`

Argument processors allow manipulation of a grabbed argument before it is passed to the underlying code. New processor implementations may be created as functions which take one trailing argument, and which leave their result in the `\ProcessedArgument` variable. For example, `\ReverseBoolean` is defined as

```
\ExplSyntaxOn
\cs_new_protected:Npn \ReverseBoolean #1
  {
    \bool_if:NTF #1
      { \tl_set:Nn \ProcessedArgument { \c_false_bool } }
      { \tl_set:Nn \ProcessedArgument { \c_true_bool } }
  }
\ExplSyntaxOff
```

[As an aside: the code is written in expl3, so we don't have to worry about spaces creeping into the definition.]

2.16 Access to the argument specification

The argument specifications for document commands and environments are available for examination and use.

`\GetDocumentCommandArgSpec {`⟨*cmd*⟩`}`
`\GetDocumentEnvironmentArgSpec {`⟨*environment*⟩`}`

These functions transfer the current argument specification for the requested ⟨*cmd*⟩ or ⟨*environment*⟩ into the token list variable `\ArgumentSpecification`. If the ⟨*cmd*⟩ or ⟨*environment*⟩ has no known argument specification then an error is issued. The assignment to `\ArgumentSpecification` is local to the current TEX group.

`\ShowDocumentCommandArgSpec {`⟨*cmd*⟩`}`
`\ShowDocumentEnvironmentArgSpec {`⟨*environment*⟩`}`

These functions show the current argument specification for the requested ⟨*cmd*⟩ or ⟨*environment*⟩ at the terminal. If the ⟨*cmd*⟩ or ⟨*environment*⟩ has no known argument specification then an error is issued.

3 Copying and showing (robust) commands and environments

If you want to (slightly) alter an existing command you may want to save the current definition under a new name and then use that in a new definition. If

the existing command is robust, then the old trick of using the low-level `\let` for this doesn't work, because it only copies the top-level definition, but not the part that actually does the work. As most LaTeX commands are nowadays robust, LaTeX now offers some high-level declarations for this instead.

However, please note that it is usually better to make use of available hooks (e.g., the generic command or environment hooks), instead of copying the current definition and thereby freezing it; see the hook management documentation `lthooks-doc.pdf` for details.

`\NewCommandCopy {`⟨*cmd*⟩`} {`⟨*existing-cmd*⟩`}`
`\RenewCommandCopy {`⟨*cmd*⟩`} {`⟨*existing-cmd*⟩`}`
`\DeclareCommandCopy {`⟨*cmd*⟩`} {`⟨*existing-cmd*⟩`}`

This copies the definition of ⟨*existing-cmd*⟩ to ⟨*cmd*⟩. After this ⟨*existing-cmd*⟩ can be redefined and ⟨*cmd*⟩ still works! This allows you to then provide a new definition for ⟨*existing-cmd*⟩ that makes use of ⟨*cmd*⟩ (i.e., of its old definition). For example, after

```
\NewCommandCopy\LaTeXorig\LaTeX
\RenewDocumentCommand\LaTeX{}{\textcolor{blue}{\LaTeXorig}}
```

all LaTeX logos generated with `\LaTeX` will come out in blue (assuming you have a color package loaded).

The differences between `\New...` and `\Renew...` are as elsewhere: i.e., you get an error depending on whether or not ⟨*cmd*⟩ already exists, or in case of `\Declare...` it is copied regardless. Note that there is no `\Provide...` declaration, because that would be of limited value.

If the ⟨*cmd*⟩ or ⟨*existing-cmd*⟩ can't be provided as a single token but need "constructing", you can use `\ExpandArgs` as explained in Section 4.

`\ShowCommand {`⟨*cmd*⟩`}`

This displays the meaning of the ⟨*cmd*⟩ on the terminal and then stops (just like the primitive `\show`). The difference is that it correctly shows the meaning of more complex commands, e.g., in case of robust commands it displays not only the top-level definition but also the actual payload code and in case of commands declared with `\NewDocumentCommand`, etc. it also gives you detailed information about the argument signature.

`\NewEnvironmentCopy {`⟨*env*⟩`} {`⟨*existing-env*⟩`}`
`\RenewEnvironmentCopy {`⟨*env*⟩`} {`⟨*existing-env*⟩`}`
`\DeclareEnvironmentCopy {`⟨*env*⟩`} {`⟨*existing-env*⟩`}`

This copies the definition for environment ⟨*existing-env*⟩ to ⟨*env*⟩ (both the beginning and end code), i.e., it is simply applying `\NewCommandCopy` twice to the internal commands that define an environment, i.e., `\`⟨*env*⟩ and `\end`⟨*env*⟩. The differences between `\New...`, `\Renew...`, and `\Declare...` are the usual ones.

`\ShowEnvironment {`⟨*env*⟩`}`

This displays the meaning of the begin end end code for environment ⟨*env*⟩.

4 Preconstructing command names (or otherwise expanding arguments)

When declaring new commands with `\NewDocumentCommand` or `\NewCommandCopy` or similar, it is sometimes necessary to "construct" the csname. As a general mechanism the L3 programming layer has `\exp_args:N...` for this, but there is no mechanism for it if `\ExplSyntaxOn` is not active (and mixing programming and user interface level commands is not a good approach anyhow). We therefore offer a mechanism to access this ability using CamelCase naming.

`\UseName {`⟨*string*⟩`}`
`\ExpandArgs {`⟨*spec*⟩`} {`⟨*cmd*⟩`} {`⟨*arg1*⟩`}` ...

`\UseName` turns the ⟨*string*⟩ directly into a csname and then executes it: this is equivalent to the long-standing LaTeX 2ε internal command `\@nameuse`, or the L3 programming equivalent `\use:c`. `\ExpandArgs` takes a ⟨*spec*⟩ which describes how to expand the ⟨*arguments*⟩, carries out these operations then executes the ⟨*cmd*⟩. The ⟨*spec*⟩ uses the descriptions offered by the L3 programming layer, and the relevant `\exp_args:N...` function must exist. Common cases will have a ⟨*spec*⟩ of `c`, `cc` or `Nc`: see below.

As an example, the following declaration provides a method to generate copyedit commands:

```
\NewDocumentCommand\newcopyedit{mO{red}}
  {%
    \newcounter{todo#1}%
    \ExpandArgs{c}\NewDocumentCommand{#1}{s m}%
      {%
        \stepcounter{todo#1}%
        \IfBooleanTF {##1}%
          {\todo[color=#2!10]{\UseName{thetodo#1}: ##2}}%
          {\todo[inline,color=#2!10]{\UseName{thetodo#1}: ##2}}%
      }%
  }
```

Given that declaration you can then write `\newcopyedit{note}[blue]` which defines the command `\note` and the corresponding counter for you.

A second example is to copy a command by string name using `\NewCommandCopy`: here we might need to construct both command names.

```
\NewDocumentCommand\savebyname{m}
  {\ExpandArgs{cc}\NewCommandCopy{saved#1}{#1}}
```

In the ⟨*spec*⟩ each `c` stands for one argument that is turned into a '`c`'ommand. An `n` represents a '`n`'ormal argument that is not altered and `N` stands for a '`N`'ormal argument which is also left unchanged, but one consisting only of a single token (and usually unbraced). Thus, to construct a command from a string only for the second argument of `\NewCommandCopy` you would write

```
\ExpandArgs{Nc}\NewCommandCopy\mysectionctr{c@section}
```

There are several other single letters supported in the L3 programming layer that *could* be used in the ⟨*spec*⟩ to manipulate arguments in other ways. If you are interested, take a look at the "Argument expansion" section in the L3 programming layer documentation in `interface3.pdf`.

5 Expandable floating point (and other) calculations

The LaTeX3 programming layer which is part of the format offers a rich interface to manipulate floating point variables and values. To allow for (simpler) applications to use this on document-level or in packages otherwise not making use of the L3 programming layer a few interface commands are made available.

`\fpeval {`⟨*floating point expression*⟩`}`

The expandable command `\fpeval` takes as its argument a floating point expression and produces a result using the normal rules of mathematics. As this command is expandable it can be used where TeX requires a number and for example within a low-level `\edef` operation to give a purely numerical result.

Briefly, the floating point expressions may comprise:

- Basic arithmetic: addition x `+` y, subtraction x `-` y, multiplication x `*` y, division x `/` y, square root $\sqrt{x}$, and parentheses.
- Comparison operators: x `<` y, x `<=` y, x `>?` y, x `!=` y *etc.*
- Boolean logic: sign `sign` x, negation `!` x, conjunction x `&&` y, disjunction x `||` y, ternary operator x `?` y `:` z.
- Exponentials: `exp` x, `ln` x, x`^`y.
- Integer factorial: `fact` x.
- Trigonometry: `sin` x, `cos` x, `tan` x, `cot` x, `sec` x, `csc` x expecting their arguments in radians, and `sind` x, `cosd` x, `tand` x, `cotd` x, `secd` x, `cscd` x expecting their arguments in degrees.
- Inverse trigonometric functions: `asin` x, `acos` x, `atan` x, `acot` x, `asec` x, `acsc` x giving a result in radians, and `asind` x, `acosd` x, `atand` x, `acotd` x, `asecd` x, `acscd` x giving a result in degrees.

- Extrema: `max`$(x_1, x_2, \ldots)$, `min`$(x_1, x_2, \ldots)$, `abs`(x).
- Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, `NaN` by default):
  - `trunc`(x, n) rounds towards zero,
  - `floor`(x, n) rounds towards $-\infty$,
  - `ceil`(x, n) rounds towards $+\infty$,
  - `round`(x, n, t) rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.
- Random numbers: `rand`(), `randint`(m, n).
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\number`) of integer, dimension, and skip variables to floating points numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: $(x_1, \ldots, x_n)$ that can be added together, multiplied or divided by a floating point number, and nested.

An example of use could be the following:

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
  = \fpeval{sin(3.5)/2 + 2e-3} $.
```

which produces the following output:

LaTeX can now compute: $\frac{\sin(3.5)}{2} + 2 \cdot 10^{-3} = -0.1733916138448099$.

`\inteval {`⟨*integer expression*⟩`}`

The expandable command `\inteval` takes as its argument an integer expression and produces a result using the normal rules of mathematics with some restrictions, see below. The operations recognized are `+`, `-`, `*` and `/` plus parentheses. As this command is expandable it can be used where TeX requires a number and for example within a low-level `\edef` operation to give a purely numerical result.

This is basically a thin wrapper for the primitive `\numexpr` command and therefore has some syntax restrictions. These are:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large;

- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

An example of use could be the following.

```
\LaTeX{} can now compute: The sum of the numbers is $\inteval{1 + 2 + 3}$.
```

which results in "LaTeX can now compute: The sum of the numbers is 6."

`\dimeval {`⟨*dimen expression*⟩`}` `\skipeval {`⟨*skip expression*⟩`}`

Similar to `\inteval` but computing a length (`dimen`) or a rubber length (`skip`) value. Both are thin wrappers around the corresponding engine primitives, which makes them fast, but therefore shows the same syntax peculiars as discussed above. Nevertheless, in practice they are usually sufficient. For example

```
\newcommand\calculateheight[1]{%
  \setlength\textheight{\dimeval{\topskip+\baselineskip*\inteval{#1-1}}}}
```

sets the `\textheight` to the appropriate value if a page should hold a specific number of text lines. Thus after `\calculateheight{40}` it is set to 478.0pt, given the values `\topskip` (10.0pt) and `\baselineskip` (12.0pt) in the current document.

6 Case changing

TeX provides two primitives `\uppercase` and `\lowercase` for changing the case of text. However, these have a range of limitations: they only change the case of explicit characters, do not account for the surrounding context, do not support UTF-8 input with 8-bit engines, etc. To overcome this problem, LaTeX provides the commands `\MakeUppercase`, `\MakeLowercase` and `\MakeTitlecase`: these offer significant enhancement over the TeX primitives. These commands are engine-robust (`\protected`), and so can be used in moving arguments.

Upper- and lower-casing are well-understood in general conversation. Title-casing here follows the definition given by the Unicode Consortium: the first character of the input will be converted to (broadly) uppercase, and the rest of the input to lowercase. The full range of Unicode UTF-8 input can be supported, with the proviso that at present the characters set up with 8-bit engines match those available in standard input encodings (`T1`, `T2`, `LGR`).

`\MakeUppercase{hello WORLD ßüé}`	HELLO WORLD SSÜÉ
`\MakeLowercase{hello WORLD ßüé}`	hello world ßüé
`\MakeTitlecase{hello WORLD ßüé}`	Hello world ßüé

The case-changing commands take an optional argument which can be used to tailor the output. This optional argument accepts the key `locale`, also available

under the alias `lang`, which can be used to give a language identifier in BCP-47 format. This is then applied to select language-specific features during case-changing.

The input given to these commands is 'expanded' before case changing is applied. This means that any commands within the input that convert to pure text will be case changed. Mathematical content is automatically excluded, as are the arguments to the commands `\label`, `\ref`, `\cite`, `\begin` and `\end`. Additional exclusions can be added using the command `\AddToNoCaseChangeList`. Input can be excluded from case changing using the command `\NoCaseChange`.

```
\MakeUppercase{Some text $y = mx + c$}   SOME TEXT y = mx + c
 \MakeUppercase{\NoCaseChange{iPhone}}   iPhone
```

To allow robust commands to be used within case changing *and* to produce the expected output, two additional control commands are available. `\CaseSwitch` allows the user to specify the result for the four possible cases

- No case changing
- Uppercasing
- Lowercasing
- Titlecasing (only applies for the start of the input)

The command `\DeclareCaseChangeEquivalent` provides a way to substitute a command by an alternative version when it is found inside a case changing situation. There are three commands for customising the case changing of codepoints

```
\DeclareLowercaseMapping [⟨locale⟩] {⟨codepoint⟩} {⟨output⟩}
\DeclareTitlecaseMapping [⟨locale⟩] {⟨codepoint⟩} {⟨output⟩}
\DeclareUppercaseMapping [⟨locale⟩] {⟨codepoint⟩} {⟨output⟩}
```

All three take a ⟨*codepoint*⟩ (as an integer expression) and will result in the ⟨*output*⟩ being produced under the appropriate case changing operation. The optional ⟨*locale*⟩ can be given if the mapping should only apply to a specific one: this is given in BCP-47 format (https://en.wikipedia.org/wiki/IETF_language_tag). For example, the kernel customises the mapping for U+01F0 (ǰ) when uppercasing in 8-bit engines:

```
\DeclareUppercaseMapping{"01F0}{\v{J}}
```

as there is no pre-composed J̌ character, and this is problematic if the engine does not support Unicode natively. Similarly, to set a locale `xx` to behave in the same way as Turkish and retain the difference between dotted- and dotless-i, one could use for example

```
   \DeclareLowercaseMapping{xx}{"0049}{\i}
\DeclareLowercaseMapping{xx}{"0130}{i}
\DeclareUppercaseMapping{xx}{"0069}{\.{I}}
\DeclareUppercaseMapping{xx}{"0131}{I}
```