The xparse package
Document command parser

The LaTeX Project*

Released 2025-10-09

1 Introduction

This package is obsolete with the October 2020 LaTeX release.

With new formats, `\NewDocumentCommand`, etc., are available in the format. Other than for a small number of now-deprecated argument types and support functions, package authors should transition their code to avoid loading xparse. An updated version of the documentation covering the full set of functionality available in the kernel is available in `usrguide`.

2 Old introduction

The xparse package provides a high-level interface for producing document-level commands. In that way, it is intended as a replacement for the LaTeX 2_ε `\newcommand` macro. However, xparse works so that the interface to a function (optional arguments, stars and mandatory arguments, for example) is separate from the internal implementation. xparse provides a normalized input for the internal form of a function, independent of the document-level argument arrangement.

At present, the functions in xparse which are regarded as "stable" are:

- `\NewDocumentCommand`
  `\RenewDocumentCommand`
  `\ProvideDocumentCommand`
  `\DeclareDocumentCommand`
- `\NewDocumentEnvironment`
  `\RenewDocumentEnvironment`
  `\ProvideDocumentEnvironment`
  `\DeclareDocumentEnvironment`
- `\NewExpandableDocumentCommand`
  `\RenewExpandableDocumentCommand`
  `\ProvideExpandableDocumentCommand`
  `\DeclareExpandableDocumentCommand`

*E-mail: latex-team@latex-project.org

- `\IfNoValue(TF)`
- `\IfValue(TF)`
- `\IfBoolean(TF)`

with the other functions currently regarded as "experimental". Please try all of the commands provided here, but be aware that the experimental ones may change or disappear.

3 Specifying arguments

Before introducing the functions used to create document commands, the method for specifying arguments with xparse will be illustrated. In order to allow each argument to be defined independently, xparse 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 xparse 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 are:

`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 xparse 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 3.6 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. *This is an experimental type.*

`E` As for `e` but returns one or more ⟨*defaults*⟩ if values are not given: `E{`⟨*tokens*⟩`}{`⟨*defaults*⟩`}`. See Section 3.5 for more details.

Using these specifiers, it is possible to create complex input syntax very easily. For example, given the argument definition '`s o o m O{default}`', the input '`*[Foo]{Bar}`' would be parsed as:

- `#1` = `\BooleanTrue`
- `#2` = `Foo`
- `#3` = `-NoValue-`
- `#4` = `Bar`
- `#5` = `default`

whereas '`[One][Two]{}[Three]`' would be parsed as:

- `#1` = `\BooleanFalse`
- `#2` = `One`
- `#3` = `Two`
- `#4` =
- `#5` = `Three`

Delimited argument types (`d`, `o` and `r`) are defined such that they require matched pairs of delimiters when collecting an argument. For example

```
\NewDocumentCommand{\foo}{o}{#1}
\foo[[content]] % #1 = "[content]"
\foo[[]         % Error: missing closing "]"
```

Also note that `{` and `}` cannot be used as delimiters as they are used by TEX as grouping tokens. Implicit begin- or end-group tokens (*e.g.*, `\bgroup` and `\egroup`) are not allowed for delimited argument tipes. Arguments to be grabbed inside these tokens must be created as either `m`- or `g`-type arguments.

Within delimited arguments, non-balanced or otherwise awkward tokens may be included by protecting the entire argument with a brace pair

```
\NewDocumentCommand{\foobar}{o}{#1}
\foobar[{[}]         % Allowed as the "[" is 'hidden'
```

These braces will be stripped only if they surround the *entire* content of the optional argument

```
\NewDocumentCommand{\foobaz}{o}{#1}
\foobaz[{abc}]          % => "abc"
\foobaz[ {abc}]          % => " {abc}"
```

Two more characters have a special meaning when creating an argument specifier. First, `+` is used to make an argument long (to accept paragraph tokens). In contrast to LaTeX 2ε's `\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, 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 5.2.

When an optional argument is followed by a mandatory argument with the same delimiter, xparse 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, but also to `g` or `G` type arguments followed by `m` type arguments.

As xparse is also used to describe interfaces that have appeared in the wider LaTeX 2ε eco-system, it also defines additional argument types, described in Section 3.8: the mandatory types `l` and `u` and the optional brace group types `g` and `G`. Their use is not recommended because it is simpler for a user if all packages use a similar syntax. For the same reason, delimited arguments `r`, `R`, `d` and `D` should normally use delimiters that are naturally paired, such as `[` and `]` or `(` and `)`, or that are identical, such as `"` and `"`. A very common syntax is to have one optional argument `o` treated as a key–value list (using for instance l3keys) followed by some mandatory arguments `m` (or `+m`).

3.1 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. In xparse terms it has signature

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

3.2 Required delimited arguments

The contrast between a delimited (`D`-type) and "required delimited" (`R`-type) argument is that an error will be raised if the latter is missing. Thus for example

```
\NewDocumentCommand {\foobaz} {r()m} {}
\foobaz{oops}
```

will lead to an error message being issued. The marker `-NoValue-` (`r`-type) or user-specified default (for `R`-type) will be inserted to allow error recovery.

3.3 Verbatim arguments

Arguments of type `v` are read in verbatim mode, which will result in the grabbed argument consisting of tokens of category codes 12 ("other") and 13 ("active"), except spaces, which are given category code 10 ("space"). The argument is delimited in a similar manner to the LaTeX 2_ε `\verb` function, or by (correctly nested) pairs of braces.

Functions containing verbatim arguments cannot appear in the arguments of other functions. The `v` argument specifier includes code to check this, and will raise an error if the grabbed argument has already been tokenized by TEX in an irreversible way.

By default, an argument of type `v` must be at most one line. Prefixing with `+` allows line breaks within the argument.

3.4 Default values of arguments

Uppercase argument types (`O`, `D`, ...) allow to specify a default value to be used when the argument is missing; their lower-case counterparts use the special marker `-NoValue-`. The default value can be expressed in terms of the value of any other arguments by using `#1`, `#2`, and so on.

```
\NewDocumentCommand {\conjugate} { m O{#1ed} O{#2} } {(#1,#2,#3)}
\conjugate {walk}            % => (walk,walked,walked)
\conjugate {find} [found]    % => (find,found,found)
\conjugate {do} [did] [done] % => (do,did,done)
```

The default values may refer to arguments that appear later in the argument specification. For instance a command could accept two optional arguments, equal by default:

```
\NewDocumentCommand {\margins} { O{#3} m O{#1} m } {(#1,#2,#3,#4)}
\margins {a} {b}              % => {(-NoValue-,a,-NoValue-,b)}
\margins [1cm] {a} {b}        % => {(1cm,a,1cm,b)}
\margins {a} [1cm] {b}        % => {(1cm,a,1cm,b)}
\margins [1cm] {a} [2cm] {b}  % => {(1cm,a,2cm,b)}
```

3.5 Default values for "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.

3.6 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 in xparse by ending the argument specification with b. The approach taken in xparse is different from the earlier packages environ or newenviron: the body of the environment is provided to the code part as a usual argument #1, #2 etc., rather than stored in a macro such as \BODY.

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 \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.

3.7 Starred environments

Many packages define environments with and without * in their name, for instance tabular and tabular*. At present, xparse does not provide specific tools to define these: one should simply define the two environment separately, for instance

```
\NewDocumentEnvironment { tabular } { o +m } {...} {...}
\NewDocumentEnvironment { tabular* } { m o +m } {...} {...}
```

Of course the implementation of these two environments, denoted "`...`" in this example, can rely on the same internal commands.

Note that this situation is different from the `s` argument type: if the signature of an environment starts with `s` then the star is searched for after the argument of `\begin`. For instance, the following typesets `star`.

```
\NewDocumentEnvironment { envstar } { s }
  {\IfBooleanTF {#1} {star} {no star}} {}
\begin{envstar}*
\end{envstar}
```

3.8 Backwards Compatibility

One role of xparse is to describe existing LaTeX interfaces, including some that are rather unusual in LaTeX (as opposed to formats such as plain TeX) such as delimited arguments. As such, the package defines some argument specifiers that should largely be avoided nowadays as using them in packages leads to inconsistent user interfaces. The simplest syntax is often best, with argument specifications such as `mmmm` or `ommmm`, namely an optional argument followed by some standard mandatory ones. The optional argument can be made to support key–value syntax using tools from l3keys.

The argument types that are not recommended any longer are:

`l` A mandatory argument which reads everything up to the first begin-group token: in standard LaTeX this is a left brace.

`u` Reads a mandatory argument "until" ⟨*tokens*⟩ are encountered, where the desired ⟨*tokens*⟩ are given as an argument to the specifier: `u{`⟨*tokens*⟩`}`.

`g` An optional argument given inside a pair of TeX group tokens (in standard LaTeX, `{ ... }`), which returns `-NoValue-` if not present.

`G` As for `g` but returns ⟨*default*⟩ if no value is given: `G{`⟨*default*⟩`}`.

3.9 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.

3.9.1 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.

3.9.2 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`.

4 Declaring commands and environments

With the concept of an argument specifier defined, it is now possible to describe the methods available for creating both functions and environments using xparse.

The interface-building commands are the preferred method for creating document-level functions in LATEX3. All of the functions generated in this way are naturally robust (using the ε-TEX `\protected` mechanism).

`\NewDocumentCommand`
`\RenewDocumentCommand`
`\ProvideDocumentCommand`
`\DeclareDocumentCommand`

`\NewDocumentCommand` ⟨*function*⟩ {⟨*arg spec*⟩} {⟨*code*⟩}

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

As 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 2_ε 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.

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

- `\NewDocumentCommand` will issue an error if ⟨*function*⟩ has already been defined.
- `\RenewDocumentCommand` will issue an error if ⟨*function*⟩ 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 ⟨*function*⟩ with the same name. This should be used sparingly.

TEXhackers note: Unlike LaTeX 2_ε's `\newcommand` and relatives, the `\NewDocumentCommand` family of functions do not prevent creation of functions with names starting `\end...`.

`\NewDocumentEnvironment`
`\RenewDocumentEnvironment`
`\ProvideDocumentEnvironment`
`\DeclareDocumentEnvironment`

```
\NewDocumentEnvironment {⟨environment⟩} {⟨arg spec⟩}
  {⟨start code⟩} {⟨end code⟩}
```

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

5 Other xparse commands

5.1 Testing special values

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

\IfNoValueT ⋆ \IfNoValueTF {⟨*argument*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\IfNoValueF ⋆ \IfNoValueT {⟨*argument*⟩} {⟨*true code*⟩}
\IfNoValueTF ⋆ \IfNoValueF {⟨*argument*⟩} {⟨*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 simply test for emptiness using the expl3 conditional `\tl_if_blank:nTF` or its etoolbox analogue `\ifblank`.

\IfValueT ⋆ \IfValueTF {⟨*argument*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\IfValueF ⋆
\IfValueTF ⋆

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.

\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.

\IfBooleanT ⋆ \IfBooleanTF {⟨*argument*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\IfBooleanF ⋆
\IfBooleanTF ⋆

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

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

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

5.2 Argument processors

xparse introduces the idea of an argument processor, which is 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.

It might sometimes be useful to use the value of another argument as one of the arguments of a processor. For example, using the \SplitList processor defined below,

```
\NewDocumentCommand \foo { O{,} >{\SplitList{#1}} m } { \foobar{#2} }
\foo{a,b;c,d}
```

results in \foobar receiving the argument {a}{b;c}{d} because \SplitList receives as its two arguments the optional one (whose value here is the default, a comma) and the mandatory one. To summarize, first the arguments are searched for in the input, then any default argument is determined as explained in Section 3.4, then these default arguments are passed to any processor. When referring to arguments (through #1, #2 and so on) in a processor, the arguments used are always those before applying any processor.

\ProcessedArgument

xparse defines a very small set of processor functions. In the main, it is anticipated that code writers will want to create their own processors. These need to accept one argument, which is the tokens as grabbed (or as returned by a previous processor function). Processor functions should return the processed argument as the variable \ProcessedArgument.

\ReverseBoolean `\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} }
  }
```

\SplitArgument `\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.

\SplitList `\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.

```
\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.

\ProcessList ⋆ `\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} { \SomeDocumentFunction } }
```

This function is experimental.

`\TrimSpaces` `\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.

This function is experimental.

5.3 Fully-expandable document commands

There are *very rare* occasion when it may be useful to create functions using a fully-expandable argument grabber. To support this, xparse can create expandable functions as well as the usual robust ones. 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 *absolutely necessary*; if you do not understand when this might be, *do not use these functions*!

\NewExpandableDocumentCommand
\RenewExpandableDocumentCommand
\ProvideExpandableDocumentCommand
\DeclareExpandableDocumentCommand

```
\NewExpandableDocumentCommand
    ⟨function⟩ {⟨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 expandably 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`, `R`, `l` or `u`.
- All short arguments appear before long arguments.
- The mandatory argument types `l` and `u` may not be used after optional arguments.
- The optional argument types `g` and `G` are not available.
- 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.

xparse will issue an error if an argument specifier is given which does not conform to the first six requirements. The last item is an issue when the function is used, and so is beyond the scope of xparse itself.

5.4 Access to the argument specification

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

\GetDocumentCommandArgSpec
\GetDocumentEnvironmentArgSpec

```
\GetDocumentCommandArgSpec ⟨function⟩
\GetDocumentEnvironmentArgSpec {⟨environment⟩}
```

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

\ShowDocumentCommandArgSpec
\ShowDocumentEnvironmentArgSpec

```
\ShowDocumentCommandArgSpec ⟨function⟩
\ShowDocumentEnvironmentArgSpec {⟨environment⟩}
```

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

6 Load-time options

`log-declarations` The package recognizes the load-time option `log-declarations`, which is a key–value option taking the value `true` and `false`. By default, the option is set to `false`, meaning that no command or environment declared is logged. By loading xparse using

```
\usepackage[log-declarations=true]{xparse}
```

each new, declared or renewed command or environment is logged.

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.