The xparse package*
Generic document command parser

The LaTeX3 Project†

2011/01/23

1 Creating document commands

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 2ε `\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 normalised 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:

- `\DeclareDocumentCommand`
- `\NewDocumentCommand`
- `\RenewDocumentCommand`
- `\ProvideDocumentCommand`
- `\DeclareDocumentEnvironment`
- `\NewDocumentEnvironment`
- `\RenewDocumentEnvironment`
- `\ProvideDocumentEnvironment`
- `\IfNoValue(TF)` (the need for `\IfValue(TF)` is currently an item of active discussion)

*This file has version number 2136, last revised 2011/01/23.

†Frank Mittelbach, Denys Duchier, Chris Rowley, Rainer Schöpf, Johannes Braams, Michael Downes, David Carlisle, Alan Jeffrey, Morten Høgholm, Thomas Lotze, Javier Bezos, Will Robertson, Joseph Wright

- `\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.

1.1 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 surrounded by a brace pair. This is the xparse type specifier for a normal TEX argument.

`l` An argument which reads everything up to the first open group token: in standard LATEX this is a left brace.

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

The types which define optional arguments are:

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

`d` An optional argument which is delimited by ⟨*token1*⟩ and ⟨*token2*⟩, which are given as arguments: `d`⟨*token1*⟩⟨*token2*⟩. As with `o`, if no value is given the special token `\NoValue` is returned.

`O` As for `o`, but returns ⟨*default*⟩ if no value is given. Should be given as `O{`⟨*default*⟩`}`.

`D` As for `d`, but returns ⟨*default*⟩ if no value is given: `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`. 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*⟩.

`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*⟩`}`.

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}`

Note that after parsing the input there will be always exactly the same number of ⟨*balanced text*⟩ arguments as the number of letters in the argument specifier. The `\BooleanTrue` and `\BooleanFalse` tokens are passed without braces; all other arguments are passed as brace groups.

Two more tokens 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 2$_\varepsilon$'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 token `>` 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 1.5.

1.2 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

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

the user input `\foo{arg1}[arg2]{arg3}` and `\foo{arg1}␣␣[arg2]␣␣␣{arg3}` will both be parsed in the same way. However, spaces are *not* ignored when parsing optional arguments after the last mandatory argument. Thus with

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

`\foo{arg1}[arg2]` will find an optional argument but `\foo{arg1}␣[arg2]` will not. This is so that trailing optional arguments are not picked up 'by accident' in input.

1.3 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).

`\DeclareDocumentCommand`
`\NewDocumentCommand`
`\RenewDocumentCommand`
`\ProvideDocumentCommand`
`\DeclareDocumentCommand` ⟨*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 will execute ⟨*code*⟩.

As an example:

```
\DeclareDocumentCommand \chapter { s o m } {
  \IfBooleanTF {#1} {
    \typesetnormalchapter {#2} {#3}
  }{
    \typesetstarchapter {#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.

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

- \DeclareDocumentCommand will always create the new definition, irrespective of any existing ⟨*function*⟩ with the same name.
- \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.

TEXhackers note: Unlike LATEX 2ε's \newcommand and relatives, the \DeclareDocumentCommand function do not prevent creation of functions with names starting \end....

\DeclareDocumentEnvironment
\NewDocumentEnvironment
\RenewDocumentEnvironment
\ProvideDocumentEnvironment

\DeclareDocumentEnvironment {⟨*environment*⟩} {⟨*arg spec*⟩} {⟨*start code*⟩} {⟨*end code*⟩}

These commands work in the same way as \DeclareDocumentCommand, etc., but create environments (\begin{⟨*function*⟩} ... \end{⟨*function*⟩}). Both the ⟨*start code*⟩ and ⟨*end code*⟩ may access the arguments as defined by ⟨*arg spec*⟩.

TEXhackers note: When loaded as part of a LATEX3 format, these, these commands do not create a pair of macros \⟨*environment*⟩ and \end⟨*environment*⟩. Thus LATEX3 environments have to be accessed using the \begin ... \end mechanism. When xparse is loaded as a LATEX 2ε package, \⟨*environment*⟩ and \end⟨*environment*⟩ are defined, as this is necessary to allow the new environment to work!

1.4 Testing special values

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

\NoValue

\NoValue is a special marker returned by xparse if no value is given for an optional argument. If typeset (which should not happen), it will print the value -NoValue-.

`\IfNoValue`*TF* ⋆ `\IfNoValueTF {`⟨*argument*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
The `\IfNoValue` tests are used to check if ⟨*argument*⟩ (`#1`, `#2`, etc.) is the special `\NoValue` token. For example

```
\DeclareDocumentCommand \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.

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.

`\IfValue`*TF* ⋆ `\IfValueTF {`⟨*argument*⟩`} {`⟨*true code*⟩`} {`⟨*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.

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

`\IfBoolean`*TF* ⋆ `\IfBooleanTF` ⟨*argument*⟩ `{`⟨*true code*⟩`} {`⟨*false code*⟩`}`
Used to test if ⟨*argument*⟩ (`#1`, `#2`, etc.) is `\BooleanTrue` or `\BooleanFalse`. For example

```
\DeclareDocumentCommand \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.

1.5 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 regularise 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.

`\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`.

`\xparse_process_to_str:n` `\xparse_process_to_str:n {`⟨*grabbed argument*⟩`}`

The `\xparse_process_to_str:n` processor applies the LaTeX3 `\tl_to_str:n` function to the ⟨*grabbed argument*⟩. For example

```
\DeclareDocumentCommand \foo { >{\xparse_arg_to_str:n} m } {
  #1 % Which is now detokenized
}
```

`\ReverseBoolean` `\ReverseBoolean`

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

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

`\SplitArgument` `\SplitArgument {`⟨*number*⟩`} {`⟨*token*⟩`}`

This processor splits the argument given at each occurrence of the ⟨*token*⟩ 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 places inside ⟨*number*⟩ + 1 sets of braces for further use. If there are less than `{`⟨*number*⟩`}` of `{`⟨*tokens*⟩`}` in the argument then empty brace groups are added at the end of the processed argument.

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

Any category code 13 (active) ⟨*tokens*⟩ will be replaced before the split takes place.

`\SplitList` `\SplitList {`⟨*token*⟩`}`

This processor splits the argument given at each occurrence of the ⟨*token*⟩ 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.

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

Any category code 13 (active) ⟨*tokens*⟩ will be replaced before the split takes place.

1.6 Separating interface and implementation

One *experimental* idea implemented in xparse is to separate out document command interfaces (the argument specification) from the implementation (code). This is carried out using a pair of functions, `\DeclareDocumentCommandInterface` and `\DeclareDocumentCommandImplementation`

`\DeclareDocumentCommandInterface` `\DeclareDocumentCommandInterface` ⟨*function*⟩ `{`⟨*implementation*⟩`} {`⟨*arg spec*⟩`}`

This declares a ⟨*function*⟩, which will take arguments as detailed in the ⟨*arg spec*⟩. When executed, the ⟨*function*⟩ will look for code stored as an ⟨*implementation*⟩.

`\DeclareDocumentCommandImplementation` `\DeclareDocumentCommandImplementation {`⟨*implementation*⟩`}` ⟨*args*⟩ `{`⟨*code*⟩`}`

Declares the ⟨*implementation*⟩ for a function to accept ⟨*args*⟩ arguments and expand to ⟨*code*⟩. An implementation must take the same number of arguments as a linked interface, although this is not enforced by the code.

1.7 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*!

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

This command 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 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 function must have at least one mandatory argument, and in particular the last argument must be one of the mandatory types (`l`, `m` or `u`).
- All arguments are either short or long: it is not possible to mix short and long argument types.
- The 'optional group' argument types `g` and `G` 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 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 three requirements. The last item is an issue when the function is used, and so is beyond the scope of xparse itself.

1.8 Access to the argument specification

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

`\GetDocumentCommandArgSpec` `\GetDocumentCommandArgSpec` ⟨*function*⟩
`\GetDocumentEnvironmentArgSpec` `\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.

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.