The xparse package
Document command parser[*]

The LaTeX3 Project[†]

Released 2012/09/05

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 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`
- `\DeclareExpandableDocumentCommand`
- `\IfNoValue(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.

[*]This file describes v4205, last revised 2012/09/05.
[†]E-mail: latex-team@latex-project.org

0.1 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.

`r` Reads a "required" delimited argument, where the delimiters are given as ⟨*token1*⟩ and ⟨*token2*⟩: `r`⟨*token1*⟩⟨*token2*⟩. If the opening ⟨*token*⟩ is missing, the default marker `-NoValue-` will be inserted after a suitable error.

`R` As for `r`, this is a "required" delimited argument but has a user-definable recovery ⟨*default*⟩, given as `R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`.

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

`v` Reads an argument "verbatim", between the following character and its next occurrence, in a way similar to the argument of the LATEX 2$_\varepsilon$ command `\verb`. Thus a `v`-type argument is read between two matching tokens, which cannot be any of `%`, `\`, `#`, `{`, `}`, `^` or `␣`. The verbatim argument can also be enclosed between braces, `{` and `}`. A command with a verbatim argument will not work when it appears within an argument of another function.

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` 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 marker `-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.

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

```
\DeclareDocumentCommand{\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. Arguments to be grabbed inside these tokens must be created as either m- or g-type arguments.

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 0.7.

By default, an argument of type `v` must be at most one line. Prefixing with `+` allows line breaks within the argument. The argument is given as a string of characters with category codes 12 or 13, except spaces, which have category code 10.

0.2 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.

0.3 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

```
\DeclareDocumentCommand\foo{r()m}
\foo{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.

Users should note that support for required delimited arguments is somewhat experimental. Feedback is therefore very welcome on the `LaTeX-L` mailing list.

0.4 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.

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.

Users should note that support for verbatim arguments is somewhat experimental. Feedback is therefore very welcome on the `LaTeX-L` mailing list.

0.5 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 expanding to be replaced by the ⟨*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 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 `\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 2_ε'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*⟩.

0.6 Testing special values

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

`\IfNoValue`*TF* ⋆

`\IfNoValueTF {`⟨*argument*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

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

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`.

`\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.

`\IfBooleanTF` ⋆

`\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.

0.7 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`.

`\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
Updated: 2012-02-12

`\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 placed inside ⟨*number*⟩ + 1 sets of braces for further use. If there are fewer 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. 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.

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

If only a single ⟨*token*⟩ is used for the split, any category code 13 (active) ⟨*token*⟩ will be replaced before the split takes place.

\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

```
\DeclareDocumentCommand \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

```
\DeclareDocumentCommand \foo
  { > { \TrimSpaces } }
  { \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.

0.8 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 last argument (if any are present) must be one of the mandatory types `m` or `r`.
- All arguments are either short or long: it is not possible to mix short and long argument types.
- The mandatory argument types `l` and `u` are not available.
- The "optional group" argument types `g` and `G` are not available.
- The "verbatim" argument type `v` is 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.

0.9 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.

1 Load-time options

`log-declarations` The package recognises 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 `true`, meaning that each command or environment declared is logged. By loading xparse using

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

this may be suppressed and no information messages are produced.

2 xparse implementation

```
⟨*package⟩
⟨@@=xparse⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

2.1 Variables and constants

`\c__xparse_no_value_tl` A special "awkward" token list: it contains two `-` tokens with different category codes. This is used as the marker for nothing being returned when no optional argument is given.

```
\group_begin:
\char_set_lccode:nn { ‘\Q } { ‘\- }
\char_set_lccode:nn { ‘\N } { ‘\N }
\char_set_lccode:nn { ‘\V } { ‘\V }
\tl_to_lowercase:n
  {
    \group_end:
    \tl_const:Nn \c__xparse_no_value_tl { QNoValue- }
  }
```

(*End definition for* `\c__xparse_no_value_tl` *This variable is documented on page* **??**.)

`\c__xparse_shorthands_prop` Shorthands are stored as a property list: this is set up here as it is a constant.

```
\prop_new:N \c__xparse_shorthands_prop
\prop_put:Nnn \c__xparse_shorthands_prop { o } { d[] }
\prop_put:Nnn \c__xparse_shorthands_prop { O } { D[] }
\prop_put:Nnn \c__xparse_shorthands_prop { s } { t* }
```

(*End definition for* `\c__xparse_shorthands_prop` *This variable is documented on page* **??**.)

`\c__xparse_special_chars_seq` In iniTeX mode, we store special characters in a sequence. Maybe `$` or `&` will have to be added later.

```
⟨*initex⟩
\seq_new:N \c__xparse_special_chars_seq
\seq_set_split:Nnn \c__xparse_special_chars_seq { }
  { \  \\ \{ \} \# \^ \_ \% \~ }
⟨/initex⟩
```

(*End definition for* `\c__xparse_special_chars_seq` *This variable is documented on page* **??**.)

`\l__xparse_all_long_bool` For expandable commands, all arguments have the same long status, but this needs to be checked. A flag is therefore needed to track whether arguments are long at all.

```
\bool_new:N \l__xparse_all_long_bool
```

(*End definition for* `\l__xparse_all_long_bool` *This variable is documented on page* **??**.)

`\l__xparse_args_tl` Token list variable for grabbed arguments.

```
\tl_new:N \l__xparse_args_tl
```

(*End definition for* `\l__xparse_args_tl` *This variable is documented on page* **??**.)

`\l__xparse_command_arg_specs_prop` Used to record all document commands created, and the argument specifications that go with these.

```
\prop_new:N \l__xparse_command_arg_specs_prop
```

(*End definition for* `\l__xparse_command_arg_specs_prop` *This variable is documented on page* **??**.)

`\l__xparse_current_arg_int` The number of the current argument being set up: this is used for creating the expandable auxiliary functions, and also to indicate if all arguments are `m`-type.

```
\int_new:N \l__xparse_current_arg_int
```

(*End definition for* `\l__xparse_current_arg_int` *This variable is documented on page* **??**.)

`\l__xparse_environment_bool` Generating environments uses the same mechanism as generating functions. However, full processing of arguments is always needed for environments, and so the function-generating code needs to know this.

```
\bool_new:N \l__xparse_environment_bool
```

(*End definition for* `\l__xparse_environment_bool` *This variable is documented on page* **??**.)

`\l__xparse_environment_arg_specs_prop` Used to record all document environment created, and the argument specifications that go with these.

```
\prop_new:N \l__xparse_environment_arg_specs_prop
```

(*End definition for* `\l__xparse_environment_arg_specs_prop` *This variable is documented on page* **??**.)

`\l__xparse_expandable_bool` Used to indicate if an expandable command is begin generated, as this affects both the acceptable argument types and how they are implemented.

```
\bool_new:N \l__xparse_expandable_bool
```

(*End definition for* `\l__xparse_expandable_bool` *This variable is documented on page* **??**.)

`\l__xparse_expandable_aux_name_tl` Used to create pretty-printing names for the auxiliaries: although the immediate definition does not vary, the full expansion does and so it does not count as a constant.

```
\tl_new:N \l__xparse_expandable_aux_name_tl
\tl_set:Nn \l__xparse_expandable_aux_name_tl
  {
    \l__xparse_function_tl \c_space_tl
    ( arg~ \int_use:N \l__xparse_current_arg_int )
  }
```

(*End definition for* `\l__xparse_expandable_aux_name_tl` *This variable is documented on page* **??**.)

`\l__xparse_fn_tl` For passing the pre-formed name of the auxiliary to be used as the parsing function.

```
\tl_new:N \l__xparse_fn_tl
```

(*End definition for* `\l__xparse_fn_tl` *This variable is documented on page* **??**.)

`\l__xparse_function_tl` Holds the control sequence name of the function currently being defined: used to avoid passing this as an argument and to avoid repeated use of `\cs_to_str:N`.

```
\tl_new:N \l__xparse_function_tl
```

(*End definition for* `\l__xparse_function_tl` *This variable is documented on page* **??**.)

`\l__xparse_long_bool` Used to indicate that an argument is long: this is used on a per-argument basis for non-expandable functions, or for the entire set of arguments when working expandably.

```
\bool_new:N \l__xparse_long_bool
```

(*End definition for* `\l__xparse_long_bool` *This variable is documented on page* **??**.)

`\l__xparse_m_args_int` The number of `m` arguments: if this is the same as the total number of arguments, then a short-cut can be taken in the creation of the grabber code.

```
\int_new:N \l__xparse_m_args_int
```

(*End definition for* `\l__xparse_m_args_int` *This variable is documented on page* **??**.)

`\l__xparse_mandatory_args_int` Holds the total number of mandatory arguments for a function, which is needed to tell whether further mandatory arguments follow an optional one.

```
\int_new:N \l__xparse_mandatory_args_int
```

(*End definition for* `\l__xparse_mandatory_args_int` *This variable is documented on page* **??**.)

`\l__xparse_processor_bool` Indicates that the current argument will be followed by one or more processors.

```
\bool_new:N \l__xparse_processor_bool
```

(*End definition for* `\l__xparse_processor_bool` *This variable is documented on page* **??**.)

`\l__xparse_processor_int` In the grabber routine, each processor is saved with a number recording the order it was found in. The total is then used to work back through the grabbers so they apply to the argument right to left.

```
\int_new:N \l__xparse_processor_int
```

(*End definition for* `\l__xparse_processor_int` *This variable is documented on page* **??**.)

`\l__xparse_signature_tl` Used when constructing the signature (code for argument grabbing) to hold what will become the implementation of the main function.

```
\tl_new:N \l__xparse_signature_tl
```

(*End definition for* `\l__xparse_signature_tl` *This variable is documented on page* **??**.)

`\l__xparse_tmp_tl` Scratch space.

```
\tl_new:N \l__xparse_tmp_tl
```

(*End definition for* `\l__xparse_tmp_tl` *This variable is documented on page* **??**.)

2.2 Declaring commands and environments

__xparse_declare_cmd:Nnn
__xparse_declare_expandable_cmd:Nnn
__xparse_declare_cmd_aux:Nnn
__xparse_declare_cmd_internal:Nnn
__xparse_declare_cmd_internal:cnx

The main functions for creating commands set the appropriate flag then use the same internal code to do the definition.

```
\cs_new_protected_nopar:Npn \__xparse_declare_cmd:Nnn
  {
    \bool_set_false:N \l__xparse_expandable_bool
    \__xparse_declare_cmd_aux:Nnn
  }
\cs_new_protected_nopar:Npn \__xparse_declare_expandable_cmd:Nnn
  {
    \bool_set_true:N \l__xparse_expandable_bool
    \__xparse_declare_cmd_aux:Nnn
  }
```

The first stage is to log information, both for the user in the log and for programmatic use in a property list of all declared commands.

```
\cs_new_protected:Npn \__xparse_declare_cmd_aux:Nnn #1#2
  {
    \cs_if_exist:NTF #1
      {
        \__msg_kernel_warning:nnxx { xparse } { redefine-command }
          { \token_to_str:N #1 } { \tl_to_str:n {#2} }
      }
      {
        \__msg_kernel_info:nnxx { xparse } { define-command }
          { \token_to_str:N #1 } { \tl_to_str:n {#2} }
      }
    \prop_put:Nnn \l__xparse_command_arg_specs_prop {#1} {#2}
    \bool_set_false:N \l__xparse_environment_bool
    \__xparse_declare_cmd_internal:Nnn #1 {#2}
  }
```

The real business of defining a document command starts with setting up the appropriate name, then counting up the number of mandatory arguments.

```
\cs_new_protected:Npn \__xparse_declare_cmd_internal:Nnn #1#2#3
  {
    \tl_set:Nx \l__xparse_function_tl { \cs_to_str:N #1 }
    \__xparse_count_mandatory:n {#2}
    \__xparse_prepare_signature:n {#2}
    \int_compare:nNnTF \l__xparse_current_arg_int = \l__xparse_m_args_int
      {
        \bool_if:NTF \l__xparse_environment_bool
          { \__xparse_declare_cmd_mixed:Nn #1 {#3} }
          { \__xparse_declare_cmd_all_m:Nn #1 {#3} }
      }
      { \__xparse_declare_cmd_mixed:Nn #1 {#3} }
    \__xparse_break_point:n {#2}
  }
\cs_generate_variant:Nn \__xparse_declare_cmd_internal:Nnn { cnx }
```

(*End definition for* `\__xparse_declare_cmd:Nnn` *and* `\__xparse_declare_expandable_cmd:Nnn` *These functions are documented on page* **??**.)

`\__xparse_break_point:n` A marker used to escape from creating a definition if necessary.

```
\cs_new_eq:NN \__xparse_break_point:n \use_none:n
```

(*End definition for* `\__xparse_break_point:n` *This function is documented on page* **??**.)

`\__xparse_declare_cmd_all_m:Nn`
`\__xparse_declare_cmd_mixed:Nn`
`\__xparse_declare_cmd_mixed_aux:Nn`
`\__xparse_declare_cmd_mixed_expandable:Nn`

When all of the arguments to grab are simple `m`-type, a short cut can be taken to provide only a single function. In the case of expandable commands, this can also happen for `+m` (as all arguments in this case must be long).

```
\cs_new_protected:Npn \__xparse_declare_cmd_all_m:Nn #1#2
  {
    \cs_generate_from_arg_count:Ncnn #1
      {
        cs_set
        \bool_if:NF \l__xparse_expandable_bool { _protected }
        \bool_if:NF \l__xparse_all_long_bool { _nopar }
        :Npn
      }
      \l__xparse_current_arg_int {#2}
  }
```

In the case of mixed arguments, any remaining `m`-type ones are first added to the signature, then the appropriate auxiliary is called.

```
\cs_new_protected:Npn \__xparse_declare_cmd_mixed:Nn
  {
    \bool_if:NTF \l__xparse_expandable_bool
      { \__xparse_declare_cmd_mixed_expandable:Nn }
      { \__xparse_declare_cmd_mixed_aux:Nn }
   }
```

Creating standard functions with mixed arg. specs sets up the main function to zero the number of processors, set the name of the function (for the grabber) and clears the list of grabbed arguments.

```
\cs_new_protected:Npn \__xparse_declare_cmd_mixed_aux:Nn #1#2
  {
    \__xparse_flush_m_args:
    \cs_generate_from_arg_count:cNnn
      { \l__xparse_function_tl \c_space_tl code }
      \cs_set_protected:Npn \l__xparse_current_arg_int {#2}
    \cs_set_protected_nopar:Npx #1
      {
        \int_zero:N \l__xparse_processor_int
        \tl_set:Nn \exp_not:N \l__xparse_args_tl
          { \exp_not:c { \l__xparse_function_tl \c_space_tl code } }
        \tl_set:Nn \exp_not:N \l__xparse_fn_tl
          { \exp_not:c { \l__xparse_function_tl \c_space_tl } }
        \exp_not:o \l__xparse_signature_tl
        \exp_not:N \l__xparse_args_tl
      }
  }
\cs_new_protected:Npn \__xparse_declare_cmd_mixed_expandable:Nn #1#2
  {
    \cs_generate_from_arg_count:cNnn
      { \l__xparse_function_tl \c_space_tl code }
      \cs_set:Npn \l__xparse_current_arg_int {#2}
    \cs_set_nopar:Npx #1
      {
        \exp_not:o \l__xparse_signature_tl
        \exp_not:N \__xparse_grab_expandable_end:wN
        \exp_not:c { \l__xparse_function_tl \c_space_tl code }
        \exp_not:N \q__xparse
        \exp_not:c { \l__xparse_function_tl \c_space_tl }
      }
    \bool_if:NTF \l__xparse_all_long_bool
      { \cs_set:cpx }
      { \cs_set_nopar:cpx }
      { \l__xparse_function_tl \c_space_tl } ##1##2 { ##1 {##2} }
  }
```

(*End definition for `\__xparse_declare_cmd_all_m:Nn` and `\__xparse_declare_cmd_mixed:Nn` These functions are documented on page* **??**.)

`\__xparse_declare_env:nnnn`
`\__xparse_declare_env_internal:nnnn`

The lead-off to creating an environment is much the same as that for creating a command: issue the appropriate message, store the argument specification then hand off to an internal function.

```
\cs_new_protected:Npn \__xparse_declare_env:nnnn #1#2
  {
⟨*initex⟩
    \cs_if_exist:cTF { environment~ #1 }
⟨/initex⟩
⟨*package⟩
    \cs_if_exist:cTF {#1}
⟨/package⟩
      {
        \__msg_kernel_warning:nnxx { xparse } { redefine-environment }
          {#1} { \tl_to_str:n {#2} }
      }
      {
        \__msg_kernel_info:nnxx { xparse } { define-environment }
          {#1} { \tl_to_str:n {#2} }
      }
    \prop_put:Nnn \l__xparse_environment_arg_specs_prop {#1} {#2}
    \bool_set_true:N \l__xparse_environment_bool
    \__xparse_declare_env_internal:nnnn {#1} {#2}
  }
```

Creating a document environment requires a few more steps than creating a single command. In order to pass the arguments of the command to the end of the function, it is necessary to store the grabbed arguments. To do that, the function used at the end of the environment has to be redefined to contain the appropriate information. To minimize the amount of expansion at point of use, the code here is expanded now as well as when used.

```
\cs_new_protected:Npn \__xparse_declare_env_internal:nnnn #1#2#3#4
  {
    \__xparse_declare_cmd_internal:cnx { environment~ #1 } {#2}
      {
        \cs_set_protected_nopar:Npx \exp_not:c { environment~ #1 ~end~aux }
          {
            \exp_not:c { environment~ #1~end~aux~ }
            \exp_not:n { \tl_tail:N \l__xparse_args_tl }
          }
        \exp_not:n {#3}
      }
    \cs_set_protected_nopar:cpx { environment~ #1 ~end }
      { \exp_not:c { environment~ #1 ~end~aux } }
    \cs_generate_from_arg_count:cNnn
      { environment~ #1 ~end~aux~ } \cs_set_protected:Npn
      \l__xparse_current_arg_int {#4}
⟨*package⟩
    \cs_set_eq:cc {#1}       { environment~ #1 }
    \cs_set_eq:cc { end #1 } { environment~ #1 ~end }
⟨/package⟩
  }
```

(*End definition for* `\__xparse_declare_env:nnnn` *This function is documented on page* **??**.)

2.3 Counting mandatory arguments

`\__xparse_count_mandatory:n`
`\__xparse_count_mandatory:N`
`\__xparse_count_mandatory:N`

To count up mandatory arguments before the main parsing run, the same approach is used. First, check if the current token is a short-cut for another argument type. If it is, expand it and loop again. If not, then look for a "counting" function to check the argument type. No error is raised here if one is not found as one will be raised by later code.

```
\cs_new_protected:Npn \__xparse_count_mandatory:n #1
  {
    \int_zero:N \l__xparse_mandatory_args_int
    \__xparse_count_mandatory:N #1
      \q_recursion_tail \q_recursion_tail \q_recursion_tail \q_recursion_stop
  }
\cs_new_protected:Npn \__xparse_count_mandatory:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \prop_get:NnNTF \c__xparse_shorthands_prop {#1} \l__xparse_tmp_tl
      { \exp_after:wN \__xparse_count_mandatory:N \l__xparse_tmp_tl }
      { \__xparse_count_mandatory_aux:N #1 }
  }
\cs_new_protected:Npn \__xparse_count_mandatory_aux:N #1
  {
    \cs_if_free:cTF { __xparse_count_type_ \token_to_str:N #1 :w }
```

```
      { \__xparse_count_type_m:w }
      { \use:c { __xparse_count_type_ \token_to_str:N #1 :w } }
  }
```

(*End definition for* `\__xparse_count_mandatory:n` *This function is documented on page* **??**.)

`\__xparse_count_type_>:w`
`\__xparse_count_type_+:w`
`\__xparse_count_type_d:w`
`\__xparse_count_type_D:w`
`\__xparse_count_type_g:w`
`\__xparse_count_type_G:w`
`\__xparse_count_type_m:w`
`\__xparse_count_type_t:w`
`\__xparse_count_type_u:w`

For counting the mandatory arguments, a function is provided for each argument type that will mop any extra arguments and call the loop function. Only the counting functions for mandatory arguments actually do anything: the rest are simply there to ensure the loop continues correctly. There are no count functions for `l` or `v` argument types as they are exactly the same as `m`, and so a little code can be saved.

The second thing that can be done here is to check that the signature is actually valid, such that all of the various argument types have the correct number of data items associated with them. If this fails to be the case, the entire set up is abandoned to avoid any strange internal errors. The opportunity is also taken to make sure that where a single token is required, one has actually been supplied.

```
\cs_new_protected:cpn { __xparse_count_type_>:w } #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
    \__xparse_count_mandatory:N
  }
\cs_new_protected_nopar:cpn { __xparse_count_type_+:w }
  { \__xparse_count_mandatory:N }
\cs_new_protected:Npn \__xparse_count_type_d:w #1#2
  {
    \__xparse_single_token_check:n {#1}
    \__xparse_single_token_check:n {#2}
    \quark_if_recursion_tail_stop_do:Nn #2 { \__xparse_bad_arg_spec:wn }
    \__xparse_count_mandatory:N
  }
\cs_new_protected:Npn \__xparse_count_type_D:w #1#2#3
  {
    \__xparse_single_token_check:n {#1}
    \__xparse_single_token_check:n {#2}
    \quark_if_recursion_tail_stop_do:nn {#3} { \__xparse_bad_arg_spec:wn }
    \__xparse_count_mandatory:N
  }
\cs_new_protected_nopar:Npn \__xparse_count_type_g:w
  { \__xparse_count_mandatory:N }
\cs_new_protected:Npn \__xparse_count_type_G:w #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
    \__xparse_count_mandatory:N
  }
\cs_new_protected_nopar:Npn \__xparse_count_type_m:w
  {
    \int_incr:N \l__xparse_mandatory_args_int
    \__xparse_count_mandatory:N
  }
\cs_new_protected:Npn \__xparse_count_type_r:w #1#2
```

```
  {
    \__xparse_single_token_check:n {#1}
    \__xparse_single_token_check:n {#2}
    \quark_if_recursion_tail_stop_do:Nn #2 { \__xparse_bad_arg_spec:wn }
    \int_incr:N \l__xparse_mandatory_args_int
    \__xparse_count_mandatory:N
  }
\cs_new_protected:Npn \__xparse_count_type_R:w #1#2#3
  {
    \__xparse_single_token_check:n {#1}
    \__xparse_single_token_check:n {#2}
    \quark_if_recursion_tail_stop_do:nn {#3} { \__xparse_bad_arg_spec:wn }
    \int_incr:N \l__xparse_mandatory_args_int
    \__xparse_count_mandatory:N
  }
\cs_new_protected:Npn \__xparse_count_type_t:w #1
  {
    \__xparse_single_token_check:n {#1}
    \quark_if_recursion_tail_stop_do:Nn #1 { \__xparse_bad_arg_spec:wn }
    \__xparse_count_mandatory:N
  }
\cs_new_protected:Npn \__xparse_count_type_u:w #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
    \int_incr:N \l__xparse_mandatory_args_int
    \__xparse_count_mandatory:N
  }
```

(*End definition for* `\__xparse_count_type_>:w` *and others. These functions are documented on page* **??**.)

`\__xparse_single_token_check:n`
`\__xparse_single_token_check_aux:nwn`

A spin-out function to check that what should be single tokens really are single tokens.

```
\cs_new_protected:Npn \__xparse_single_token_check:n #1
  {
    \tl_if_single:nF {#1}
      { \__xparse_single_token_check_aux:nwn {#1} }
  }
\cs_new_protected:Npn \__xparse_single_token_check_aux:nwn
  #1#2 \__xparse_break_point:n #3
  {
    \__msg_kernel_error:nnx { xparse } { not-single-token }
      { \tl_to_str:n {#1} } { \tl_to_str:n {#3} }
  }
```

(*End definition for* `\__xparse_single_token_check:n` *This function is documented on page* **??**.)

`\__xparse_bad_arg_spec:wn`

If the signature is wrong, this provides an escape from the entire definition process.

```
\cs_new_protected:Npn \__xparse_bad_arg_spec:wn #1 \__xparse_break_point:n #2
  { \__msg_kernel_error:nnx { xparse } { bad-arg-spec } { \tl_to_str:n {#2} } }
```

(*End definition for* `\__xparse_bad_arg_spec:wn` *This function is documented on page* **??**.)

2.4 Preparing the signature: general mechanism

__xparse_prepare_signature:n
__xparse_prepare_signature:N
__xparse_prepare_signature_bypass:N
__xparse_prepare_signature_add:N

Actually creating the signature uses the same loop approach as counting up mandatory arguments. There are first a number of variables which need to be set to track what is going on.

```
\cs_new_protected:Npn \__xparse_prepare_signature:n #1
  {
    \bool_set_false:N \l__xparse_all_long_bool
    \int_zero:N \l__xparse_current_arg_int
    \bool_set_false:N \l__xparse_long_bool
    \int_zero:N \l__xparse_m_args_int
    \bool_set_false:N \l__xparse_processor_bool
    \tl_clear:N \l__xparse_signature_tl
    \__xparse_prepare_signature:N #1 \q_recursion_tail \q_recursion_stop
  }
```

The main looping function does not take an argument,but carries out the reset on the processor boolean. This is split off from the rest of the process so that when actually setting up processors the flag-reset can be bypassed.

```
\cs_new_protected_nopar:Npn \__xparse_prepare_signature:N
  {
    \bool_set_false:N \l__xparse_processor_bool
    \__xparse_prepare_signature_bypass:N
  }
\cs_new_protected:Npn \__xparse_prepare_signature_bypass:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \prop_get:NnNTF \c__xparse_shorthands_prop {#1} \l__xparse_tmp_tl
      { \exp_after:wN \__xparse_prepare_signature:N \l__xparse_tmp_tl }
      {
        \int_incr:N \l__xparse_current_arg_int
        \__xparse_prepare_signature_add:N #1
      }
  }
```

For each known argument type there is an appropriate function to actually do the addition to the signature. These are separate for expandable and standard functions, as the approaches are different. Of course, if the type is not known at all then a fall-back is needed.

```
\cs_new_protected:Npn \__xparse_prepare_signature_add:N #1
  {
    \cs_if_exist_use:cF
      {
         __xparse_add
         \bool_if:NT \l__xparse_expandable_bool { _expandable }
         _type_  \token_to_str:N #1 :w
      }
      {
        \__msg_kernel_error:nnx { xparse } { unknown-argument-type }
          { \token_to_str:N #1 }
        \bool_if:NTF \l__xparse_expandable_bool
          { \__xparse_add_expandable_type_m:w }
          { \__xparse_add_type_m:w }
      }
  }
```
(*End definition for* `\__xparse_prepare_signature:n` *This function is documented on page* **??**.)

2.5 Setting up a standard signature

All of the argument-adding functions work in essentially the same way, except the one for `m` arguments. Any collected `m` arguments are added to the signature, then the appropriate grabber is added to the signature. Some of the adding functions also pick up one or more arguments, and are also added to the signature. All of the functions then call the loop function `\__xparse_prepare_signature:N`.

`\__xparse_add_type_+:w`

Making the next argument long means setting the flag and knocking one back off the total argument count. The `m` arguments are recorded here as this has to be done for every case where there is then a long argument.

```
\cs_new_protected_nopar:cpn { __xparse_add_type_+:w }
  {
    \__xparse_flush_m_args:
    \bool_set_true:N \l__xparse_long_bool
    \int_decr:N \l__xparse_current_arg_int
    \__xparse_prepare_signature:N
  }
```
(*End definition for* `\__xparse_add_type_+:w` *This function is documented on page* **??**.)

`\__xparse_add_type_>:w`

When a processor is found, the function `\__xparse_process_arg:n` is added to the signature along with the processor code itself. When the signature is used, the code will be added to an execution list by `\__xparse_process_arg:n`. Here, the loop calls `\__xparse_prepare_signature_bypass:N` rather than `\__xparse_prepare_signature:N` so that the flag is not reset.

```
\cs_new_protected:cpn { __xparse_add_type_>:w } #1
  {
    \bool_set_true:N \l__xparse_processor_bool
    \__xparse_flush_m_args:
    \int_decr:N \l__xparse_current_arg_int
    \tl_put_right:Nn \l__xparse_signature_tl { \__xparse_process_arg:n {#1} }
    \__xparse_prepare_signature_bypass:N
  }
```
(*End definition for* `\__xparse_add_type_>:w` *This function is documented on page* **??**.)

`\__xparse_add_type_d:w`
`\__xparse_add_type_D:w`

To save on repeated code, `d` is actually turned into the same grabber as is used by `D`, by putting the `-NoValue-` default in the correct place.

```
\cs_new_protected:Npn \__xparse_add_type_d:w #1#2
  { \exp_args:NNNo \__xparse_add_type_D:w #1 #2 \c__xparse_no_value_tl }
\cs_new_protected:Npn \__xparse_add_type_D:w #1#2#3
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_optional:N D
    \tl_put_right:Nn \l__xparse_signature_tl { #1 #2 {#3} }
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_d:w` *and* `\__xparse_add_type_D:w` *These functions are documented on page* **??**.)

`\__xparse_add_type_g:w` The `g` type is simply an alias for `G` with the correct default built-in.

```
\cs_new_protected_nopar:Npn \__xparse_add_type_g:w
  { \exp_args:No \__xparse_add_type_G:w \c__xparse_no_value_tl }
```

(*End definition for* `\__xparse_add_type_g:w` *This function is documented on page* **??**.)

`\__xparse_add_type_G:w` For the `G` type, the grabber and the default are added to the signature.

```
\cs_new_protected:Npn \__xparse_add_type_G:w #1
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_optional:N G
    \tl_put_right:Nn \l__xparse_signature_tl { {#1} }
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_G:w` *This function is documented on page* **??**.)

`\__xparse_add_type_l:w` Finding `l` arguments is very simple: there is nothing to do other than add the grabber.

```
\cs_new_protected_nopar:Npn \__xparse_add_type_l:w
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_mandatory:N l
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_l:w` *This function is documented on page* **??**.)

`\__xparse_add_type_m:w` The `m` type is special as short arguments which are not post-processed are simply counted at this stage. Thus there is a check to see if either of these cases apply. If so, a one-argument grabber is added to the signature. On the other hand, if a standard short argument is required it is simply counted at this stage, to be added later using `\__xparse_flush_m_args:`.

```
\cs_new_protected_nopar:Npn \__xparse_add_type_m:w
  {
    \bool_if:nTF { \l__xparse_long_bool || \l__xparse_processor_bool }
      {
        \__xparse_flush_m_args:
        \__xparse_add_grabber_mandatory:N m
      }
      { \int_incr:N \l__xparse_m_args_int }
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_m:w` *This function is documented on page* **??**.)

`\__xparse_add_type_r:w`
`\__xparse_add_type_R:w`

The `r`- and `R`-type arguments are very similar to the `d`- and `D`-types.

```
\cs_new_protected:Npn \__xparse_add_type_r:w #1#2
  { \exp_args:NNNo \__xparse_add_type_R:w #1 #2 \c__xparse_no_value_tl }
\cs_new_protected:Npn \__xparse_add_type_R:w #1#2#3
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_mandatory:N R
    \tl_put_right:Nn \l__xparse_signature_tl { #1 #2 {#3} }
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_r:w` *and* `\__xparse_add_type_R:w` *These functions are documented on page* **??**.)

`\__xparse_add_type_t:w`

Setting up a `t` argument means collecting one token for the test, and adding it along with the grabber to the signature.

```
\cs_new_protected:Npn \__xparse_add_type_t:w #1
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_optional:N t
    \tl_put_right:Nn \l__xparse_signature_tl { #1 }
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_t:w` *This function is documented on page* **??**.)

`\__xparse_add_type_u:w`

At the set up stage, the `u` type argument is identical to the `G` type except for the name of the grabber function.

```
\cs_new_protected:Npn \__xparse_add_type_u:w #1
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_mandatory:N u
    \tl_put_right:Nn \l__xparse_signature_tl { {#1} }
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_u:w` *This function is documented on page* **??**.)

`\__xparse_add_type_v:w`

At this stage, the `v` argument is identical to `l`.

```
\cs_new_protected_nopar:Npn \__xparse_add_type_v:w
  {
    \__xparse_flush_m_args:
    \__xparse_add_grabber_mandatory:N v
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_type_v:w` *This function is documented on page* **??**.)

`\__xparse_flush_m_args:`

As `m` arguments are simply counted, there is a need to add them to the token register in a block. As this function can only be called if something other than `m` turns up, the flag can be switched here. The total number of mandatory arguments added to the signature is also decreased by the appropriate amount.

```
\cs_new_protected_nopar:Npn \__xparse_flush_m_args:
  {
    \int_compare:nNnT \l__xparse_m_args_int > \c_zero
      {
        \tl_put_right:Nx \l__xparse_signature_tl
           { \exp_not:c { __xparse_grab_m_ \int_use:N \l__xparse_m_args_int :w } }
        \int_set:Nn \l__xparse_mandatory_args_int
          { \l__xparse_mandatory_args_int - \l__xparse_m_args_int }
      }
    \int_zero:N \l__xparse_m_args_int
  }
```

(*End definition for* `\__xparse_flush_m_args:` *This function is documented on page* **??**.)

`\__xparse_add_grabber_mandatory:N`
`\__xparse_add_grabber_optional:N`

To keep the various checks needed in one place, adding the grabber to the signature is done here. For mandatory arguments, the only question is whether to add a long grabber. For optional arguments, there is also a check to see if any mandatory arguments are still to be added. This is used to determine whether to skip spaces or not where searching for the argument.

```
\cs_new_protected_nopar:Npn \__xparse_add_grabber_mandatory:N #1
  {
    \tl_put_right:Nx \l__xparse_signature_tl
      {
        \exp_not:c
          { __xparse_grab_ #1 \bool_if:NT \l__xparse_long_bool { _long } :w }
      }
    \bool_set_false:N \l__xparse_long_bool
    \int_decr:N \l__xparse_mandatory_args_int
  }
\cs_new_protected_nopar:Npn \__xparse_add_grabber_optional:N #1
  {
    \tl_put_right:Nx \l__xparse_signature_tl
      {
        \exp_not:c
          {
            __xparse_grab_ #1
            \bool_if:NT \l__xparse_long_bool { _long }
            \int_compare:nNnF \l__xparse_mandatory_args_int > \c_zero
              { _trailing }
            :w
          }
      }
    \bool_set_false:N \l__xparse_long_bool
  }
```

(*End definition for* `\__xparse_add_grabber_mandatory:N` *This function is documented on page* **??**.)

2.6 Setting up expandable types

The approach here is not dissimilar to that for standard types, although types which are not supported in expandable functions give an error. There is also a need to define the per-function auxiliaries: this is done here, while the general grabbers are dealt with later.

__xparse_add_expandable_type_+:w

Check that a plus is given only if it occurs for every argument.

```
\cs_new_protected_nopar:cpn { __xparse_add_expandable_type_+:w }
  {
    \bool_set_true:N \l__xparse_long_bool
    \int_compare:nNnTF \l__xparse_current_arg_int = \c_one
      { \bool_set_true:N \l__xparse_all_long_bool }
      {
        \bool_if:NF \l__xparse_all_long_bool
          { \__msg_kernel_error:nn { xparse } { inconsistent-long } }
      }
    \int_decr:N \l__xparse_current_arg_int
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_expandable_type_+:w` *This function is documented on page* **??**.)

__xparse_add_expandable_type_>:w

No processors in expandable arguments, so this issues an error.

```
\cs_new_protected:cpn { __xparse_add_expandable_type_>:w } #1
  {
    \__msg_kernel_error:nnx { xparse } { processor-in-expandable }
      { \token_to_str:c { \l__xparse_function_tl } }
    \int_decr:N \l__xparse_current_arg_int
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_expandable_type_>:w` *This function is documented on page* **??**.)

__xparse_add_expandable_type_d:w
__xparse_add_expandable_type_D:w
__xparse_add_expandable_type_D_aux:NNn
__xparse_add_expandable_type_D_aux:Nn

The set up for `d`- and `D`-type arguments is the same, and involves constructing a rather complex auxiliary which is used repeatedly when grabbing. There is an auxiliary here so that the `R`-type can share code readily.

```
\cs_new_protected:Npn \__xparse_add_expandable_type_d:w #1#2
  {
    \exp_args:NNNo
      \__xparse_add_expandable_type_D:w #1 #2 \c__xparse_no_value_tl
  }
\cs_new_protected:Npn \__xparse_add_expandable_type_D:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 #2
      {
        \__xparse_add_expandable_grabber_optional:n { D_alt }
        \__xparse_add_expandable_type_D_aux:Nn #1
      }
      {
        \__xparse_add_expandable_grabber_optional:n { D }
        \__xparse_add_expandable_type_D_aux:NNn #1#2
      }
  }
\cs_new_protected:Npn \__xparse_add_expandable_type_D_aux:NNn #1#2#3
  {
    \bool_if:NTF \l__xparse_all_long_bool
      { \cs_set:cpx }
      { \cs_set_nopar:cpx }
      { \l__xparse_expandable_aux_name_tl } ##1 ##2 #1 ##3 \q__xparse ##4 #2
      { ##1 {##2} {##3} {##4} }
    \tl_put_right:Nx \l__xparse_signature_tl
      {
        \exp_not:c  { \l__xparse_expandable_aux_name_tl }
        \exp_not:n { #1 #2 {#3} }
      }
    \bool_set_false:N \l__xparse_long_bool
    \__xparse_prepare_signature:N
  }
```

This route is needed if the two delimiting tokens are identical: in contrast to the non-expandable route, the grabber here has to act differently for this case.

```
\cs_new_protected:Npn \__xparse_add_expandable_type_D_aux:Nn #1#2
  {
    \bool_if:NTF \l__xparse_all_long_bool
      { \cs_set:cpx }
      { \cs_set_nopar:cpx }
      { \l__xparse_expandable_aux_name_tl } ##1 #1 ##2 #1
      { ##1 {##2} }
    \tl_put_right:Nx \l__xparse_signature_tl
      {
        \exp_not:c  { \l__xparse_expandable_aux_name_tl }
        \exp_not:n { #1 {#2} }
      }
    \bool_set_false:N \l__xparse_long_bool
    \__xparse_prepare_signature:N
  }
```

(*End definition for* `\__xparse_add_expandable_type_d:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_g:w`
`\__xparse_add_expandable_type_G:w`

These are not allowed at all, so there is a complaint and a fall-back.

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_g:w
  {
    \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { g }
    \__xparse_add_expandable_type_m:w
  }
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_G:w #1
  {
    \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { G }
    \__xparse_add_expandable_type_m:w
  }
```

(*End definition for* `\__xparse_add_expandable_type_g:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_l:w`

Invalid in expandable contexts (as the next left brace may have been inserted by xparse due to a failed search for an optional argument).

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_l:w
```

```
  {
    \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { l }
    \__xparse_add_expandable_type_m:w
  }
```
(*End definition for* `\__xparse_add_expandable_type_l:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_m:w`

Unlike the standard case, when working expandably each argument is always grabbed separately unless the function takes only `m`-type arguments. To deal with the latter case, the value of `\l__xparse_m_args_int` needs to be increased appropriately.

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_m:w
  {
    \int_incr:N \l__xparse_m_args_int
    \__xparse_add_expandable_grabber_mandatory:n { m }
    \bool_set_false:N \l__xparse_long_bool
    \__xparse_prepare_signature:N
  }
```
(*End definition for* `\__xparse_add_expandable_type_m:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_r:w`
`\__xparse_add_expandable_type_R:w`

The `r`- and `R`-types are very similar to `D`-type arguments, and so the same internals are used.

```
\cs_new_protected:Npn \__xparse_add_expandable_type_r:w #1#2
  {
    \exp_args:NNNo
      \__xparse_add_expandable_type_R:w #1 #2 \c__xparse_no_value_tl
  }
\cs_new_protected:Npn \__xparse_add_expandable_type_R:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 #2
      {
        \__xparse_add_expandable_grabber_optional:n { R_alt }
        \__xparse_add_expandable_type_D_aux:Nn #1
      }
      {
        \__xparse_add_expandable_grabber_optional:n { R }
        \__xparse_add_expandable_type_D_aux:NNn #1#2
      }
  }
```
(*End definition for* `\__xparse_add_expandable_type_r:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_t:w`

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_t:w #1
  {
    \__xparse_add_expandable_grabber_optional:n { t }
    \bool_if:NTF \l__xparse_all_long_bool
      { \cs_set:cpn }
      { \cs_set_nopar:cpn }
      { \l__xparse_expandable_aux_name_tl } ##1 #1 {##1}
    \tl_put_right:Nx \l__xparse_signature_tl
      {
```

```
      \exp_not:c  { \l__xparse_expandable_aux_name_tl }
      \exp_not:N #1
    }
  \bool_set_false:N \l__xparse_long_bool
  \__xparse_prepare_signature:N
}
```

(*End definition for* `\__xparse_add_expandable_type_t:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_u:w`

Invalid in an expandable context as any preceding optional argument may wrap part of the delimiter up in braces.

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_u:w #1
  {
    \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { u }
    \__xparse_add_expandable_type_m:w
  }
```

(*End definition for* `\__xparse_add_expandable_type_u:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_type_v:w`

Another forbidden type.

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_type_v:w
  {
    \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { v }
    \__xparse_add_expandable_type_m:w
  }
```

(*End definition for* `\__xparse_add_expandable_type_v:w` *This function is documented on page* **??**.)

`\__xparse_add_expandable_grabber_mandatory:n`
`\__xparse_add_expandable_grabber_optional:n`
`\__xparse_add_expandable_long_check:`

Adding a grabber to the signature is very simple here, with only a test to ensure that optional arguments still have mandatory ones to follow. This is also a good place to check on the consistency of the long status of arguments.

```
\cs_new_protected_nopar:Npn \__xparse_add_expandable_grabber_mandatory:n #1
  {
    \__xparse_add_expandable_long_check:
    \tl_put_right:Nx \l__xparse_signature_tl
      { \exp_not:c { __xparse_expandable_grab_ #1 :w } }
    \bool_set_false:N \l__xparse_long_bool
    \int_decr:N \l__xparse_mandatory_args_int
  }
\cs_new_protected_nopar:Npn \__xparse_add_expandable_grabber_optional:n #1
  {
    \__xparse_add_expandable_long_check:
    \int_compare:nNnF \l__xparse_mandatory_args_int > \c_zero
      { \__msg_kernel_error:nn { xparse } { expandable-ending-optional } }
    \tl_put_right:Nx \l__xparse_signature_tl
      { \exp_not:c { __xparse_expandable_grab_ #1 :w } }
    \bool_set_false:N \l__xparse_long_bool
  }
\cs_new_protected_nopar:Npn \__xparse_add_expandable_long_check:
  {
    \bool_if:nT { \l__xparse_all_long_bool && ! ( \l__xparse_long_bool ) }
      { \__msg_kernel_error:nn { xparse } { inconsistent-long } }
  }
```

(*End definition for* `\__xparse_add_expandable_grabber_mandatory:n` *and* `\__xparse_add_expandable_grabber_optional:n` *These functions are documented on page* **??**.)

2.7 Grabbing arguments

All of the grabbers follow the same basic pattern. The initial function sets up the appropriate information to define `\parse_grab_arg:w` to grab the argument. This means determining whether to use `\cs_set:Npn` or `\cs_set_nopar:Npn`, and for optional arguments whether to skip spaces. In all cases, `\__xparse_grab_arg:w` is then called to actually do the grabbing.

`\__xparse_grab_arg:w`
`\__xparse_grab_arg_aux_i:w`
`\__xparse_grab_arg_aux_ii:w`

Each time an argument is actually grabbed, `xparse` defines a function to do it. In that way, long arguments from previous functions can be included in the definition of the grabber function, so that it does not raise an error if not long. The generic function used for this is reserved here. A couple of auxiliary functions are also needed in various places.

```
\cs_new_protected:Npn \__xparse_grab_arg:w { }
\cs_new_protected:Npn \__xparse_grab_arg_aux_i:w { }
\cs_new_protected:Npn \__xparse_grab_arg_aux_ii:w { }
```

(*End definition for* `\__xparse_grab_arg:w` *This function is documented on page* **??**.)

`\__xparse_grab_D:w`
`\__xparse_grab_D_long:w`
`\__xparse_grab_D_trailing:w`
`\__xparse_grab_D_long_trailing:w`

The generic delimited argument grabber. The auxiliary function does a peek test before calling `\__xparse_grab_arg:w`, so that the optional nature of the argument works as expected.

```
\cs_new_protected:Npn \__xparse_grab_D:w #1#2#3#4 \l__xparse_args_tl
  {
    \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn
      { _ignore_spaces }
  }
\cs_new_protected:Npn \__xparse_grab_D_long:w #1#2#3#4 \l__xparse_args_tl
  {
    \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected:Npn
      { _ignore_spaces }
  }
\cs_new_protected:Npn \__xparse_grab_D_trailing:w #1#2#3#4 \l__xparse_args_tl
  { \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn { } }
\cs_new_protected:Npn \__xparse_grab_D_long_trailing:w #1#2#3#4 \l__xparse_args_tl
  { \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected:Npn { } }
```

`\__xparse_grab_D_aux:NNnnNn`
`\__xparse_grab_D_aux:NNnN`

This is a bit complicated. The idea is that, in order to check for nested optional argument tokens (`[[...]]` and so on) the argument needs to be grabbed without removing any braces at all. If this is not done, then cases like `[{[}]` fail. So after testing for an optional argument, it is collected piece-wise. Inserting a quark prevents loss of braces, and there is then a test to see if there are nested delimiters to handle.

```
\cs_new_protected:Npn \__xparse_grab_D_aux:NNnnNn #1#2#3#4#5#6
  {
    \__xparse_grab_D_aux:NNnN #1#2 {#4} #5
    \use:c { peek_meaning_remove #6 :NTF } #1
```

```
      { \__xparse_grab_arg:w }
      {
        \__xparse_add_arg:n {#3}
        #4 \l__xparse_args_tl
      }
  }
\cs_new_protected:Npn \__xparse_grab_D_aux:NNnN #1#2#3#4
  {
    \cs_set_protected_nopar:Npn \__xparse_grab_arg:w
      {
        \exp_after:wN #4 \l__xparse_fn_tl ####1 #2
          {
            \tl_if_in:nnTF {####1} {#1}
              { \__xparse_grab_D_nested:NNnnN #1 #2 {####1} {#3} #4 }
              {
                \__xparse_add_arg:o { \use_none:n ####1 }
                #3 \l__xparse_args_tl
              }
          }
```

This section needs a little explanation. In order to avoid loosing any braces, a token needs to be inserted before the argument to be grabbed. If the argument runs away because the closing token is missing then this inserted token shows up in the terminal. Ideally, #1 would therefore be used directly, but that is no good as it will mess up the rest of the grabber. Instead, a copy of #1 with an altered category code is used, as this will look right in the terminal but will not mess up the grabber. The only issue then is that the category code of #1 is unknown. So there is a quick test to ensure that the inserted token can never be matched by the grabber. (This assumes that #1 and #2 are not the same character with different category codes, but that really should not happen in any sensible document-level syntax.)

```
        \group_begin:
          \token_if_eq_catcode:NNTF #1 ^
            {
              \char_set_lccode:nn { `A } { `#1 }
              \tl_to_lowercase:n
                {
                  \group_end:
                  \l__xparse_fn_tl A
                }
            }
            {
              \char_set_lccode:nn { `^ } { `#1 }
              \tl_to_lowercase:n
                {
                  \group_end:
                  \l__xparse_fn_tl ^
                }
            }
      }
```

```
  }
```

(*End definition for* `\__xparse_grab_D:w` *This function is documented on page* **??**.)

__xparse_grab_D_nested:NNnnN
__xparse_grab_D_nested:w
`\l__xparse_nesting_a_tl`
`\l__xparse_nesting_b_tl`
`\q__xparse`

Catching nested optional arguments means more work. The aim here is to collect up each pair of optional tokens without TEX helping out, and without counting anything. The code above will already have removed the leading opening token and a closing token, but the wrong one. The aim is then to work through the the material grabbed so far and divide it up on each opening token, grabbing a closing token to match (thus working in pairs). Once there are no opening tokens, then there is a second check to see if there are any opening tokens in the second part of the argument (for things like `[][]`). Once everything has been found, the entire collected material is added to the output as a single argument. The only tricky part here is ensuring that any grabbing function that might run away is named after the function currently being parsed and not after xparse. That leads to some rather complex nesting! There is also a need to prevent the loss of any braces, hence the insertion and removal of quarks along the way.

```
\tl_new:N \l__xparse_nesting_a_tl
\tl_new:N \l__xparse_nesting_b_tl
\quark_new:N \q__xparse
\cs_new_protected:Npn \__xparse_grab_D_nested:NNnnN #1#2#3#4#5
  {
    \tl_clear:N \l__xparse_nesting_a_tl
    \tl_clear:N \l__xparse_nesting_b_tl
    \exp_after:wN #5 \l__xparse_fn_tl ##1 #1 ##2 \q__xparse ##3 #2
      {
        \tl_put_right:No \l__xparse_nesting_a_tl { \use_none:n ##1 #1 }
        \tl_put_right:No \l__xparse_nesting_b_tl { \use_i:nn #2 ##3 }
        \tl_if_in:nnTF {##2} {#1}
          {
            \l__xparse_fn_tl
              \q_nil ##2 \q__xparse \ERROR
          }
          {
            \tl_put_right:Nx \l__xparse_nesting_a_tl
              { \__xparse_grab_D_nested:w \q_nil ##2 \q_stop }
            \tl_if_in:NnTF \l__xparse_nesting_b_tl {#1}
              {
                \tl_set_eq:NN \l__xparse_tmp_tl \l__xparse_nesting_b_tl
                \tl_clear:N \l__xparse_nesting_b_tl
                \exp_after:wN \l__xparse_fn_tl \exp_after:wN
                  \q_nil \l__xparse_tmp_tl \q_nil \q__xparse \ERROR
              }
              {
                \tl_put_right:No \l__xparse_nesting_a_tl
                  \l__xparse_nesting_b_tl
                \__xparse_add_arg:V \l__xparse_nesting_a_tl
                #4 \l__xparse_args_tl
              }
          }
      }
    \l__xparse_fn_tl #3 \q_nil \q__xparse \ERROR
  }
\cs_new:Npn \__xparse_grab_D_nested:w #1 \q_nil \q_stop
  { \exp_not:o { \use_none:n #1 } }
```

(*End definition for* `\__xparse_grab_D_nested:NNnnN` *This function is documented on page* **??**.)

`\__xparse_grab_G:w`
`\__xparse_grab_G_long:w`
`\__xparse_grab_G_trailing:w`
`\__xparse_grab_G_long_trailing:w`
`\__xparse_grab_G_aux:nnNn`

Optional groups are checked by meaning, so that the same code will work with, for example, ConTEXt-like input.

```
\cs_new_protected:Npn \__xparse_grab_G:w #1#2 \l__xparse_args_tl
  {
    \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected_nopar:Npn
      { _ignore_spaces }
  }
\cs_new_protected:Npn \__xparse_grab_G_long:w #1#2 \l__xparse_args_tl
  {
    \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected:Npn { _ignore_spaces }
  }
\cs_new_protected:Npn \__xparse_grab_G_trailing:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected_nopar:Npn { } }
\cs_new_protected:Npn \__xparse_grab_G_long_trailing:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected:Npn { } }
\cs_new_protected:Npn \__xparse_grab_G_aux:nnNn #1#2#3#4
  {
    \exp_after:wN #3 \l__xparse_fn_tl ##1
      {
        \__xparse_add_arg:n {##1}
        #2 \l__xparse_args_tl
      }
    \use:c { peek_meaning #4 :NTF } \c_group_begin_token
      { \l__xparse_fn_tl }
      {
        \__xparse_add_arg:n {#1}
        #2 \l__xparse_args_tl
      }
  }
```

(*End definition for* `\__xparse_grab_G:w` *This function is documented on page* **??**.)

`\__xparse_grab_l:w`
`\__xparse_grab_l_long:w`
`\__xparse_grab_l_aux:nN`

Argument grabbers for mandatory TEX arguments are pretty simple.

```
\cs_new_protected:Npn \__xparse_grab_l:w #1 \l__xparse_args_tl
  { \__xparse_grab_l_aux:nN {#1} \cs_set_protected_nopar:Npn }
\cs_new_protected:Npn \__xparse_grab_l_long:w #1 \l__xparse_args_tl
  { \__xparse_grab_l_aux:nN {#1} \cs_set_protected:Npn }
\cs_new_protected:Npn \__xparse_grab_l_aux:nN #1#2
  {
    \exp_after:wN #2 \l__xparse_fn_tl ##1##
      {
        \__xparse_add_arg:n {##1}
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
```

(*End definition for* `\__xparse_grab_l:w` *This function is documented on page* **??**.)

`\__xparse_grab_m:w`
`\__xparse_grab_m_long:w`

Collecting a single mandatory argument is quite easy.

```
\cs_new_protected:Npn \__xparse_grab_m:w #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl ##1
      {
        \__xparse_add_arg:n {##1}
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:Npn \__xparse_grab_m_long:w #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected:Npn \l__xparse_fn_tl ##1
      {
        \__xparse_add_arg:n {##1}
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
```

(*End definition for* `\__xparse_grab_m:w` *This function is documented on page* **??**.)

`\__xparse_grab_m_1:w`
`\__xparse_grab_m_2:w`
`\__xparse_grab_m_3:w`
`\__xparse_grab_m_4:w`
`\__xparse_grab_m_5:w`
`\__xparse_grab_m_6:w`
`\__xparse_grab_m_7:w`
`\__xparse_grab_m_8:w`

Grabbing 1–8 mandatory arguments. We don't need to worry about nine arguments as this is only possible if everything is mandatory. Each function has an auxiliary so that `\par` tokens from other arguments still work.

```
\cs_new_protected:cpn { __xparse_grab_m_1:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl ##1
      {
        \tl_put_right:Nn \l__xparse_args_tl { {##1} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:cpn { __xparse_grab_m_2:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2
      {
        \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:cpn { __xparse_grab_m_3:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2##3
      {
        \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:cpn { __xparse_grab_m_4:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2##3##4
      {
        \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} {##4} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:cpn { __xparse_grab_m_5:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2##3##4##5
      {
        \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} {##4} {##5} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:cpn { __xparse_grab_m_6:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2##3##4##5##6
      {
        \tl_put_right:Nn \l__xparse_args_tl
          { {##1} {##2} {##3} {##4} {##5} {##6} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
\cs_new_protected:cpn { __xparse_grab_m_7:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2##3##4##5##6##7
      {
        \tl_put_right:Nn \l__xparse_args_tl
          { {##1} {##2} {##3} {##4} {##5} {##6} {##7} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
```

```
\cs_new_protected:cpn { __xparse_grab_m_8:w } #1 \l__xparse_args_tl
  {
    \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
      ##1##2##3##4##5##6##7##8
      {
        \tl_put_right:Nn \l__xparse_args_tl
          { {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} }
        #1 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
```

(*End definition for* `\__xparse_grab_m_1:w` *This function is documented on page* **??**.)

`\__xparse_grab_R:w`
`\__xparse_grab_R_long:w`
`\__xparse_grab_R_aux:NNnnN`

The grabber for R-type arguments is basically the same as that for D-type ones, but always skips spaces (as it is mandatory) and has a hard-coded error message.

```
\cs_new_protected:Npn \__xparse_grab_R:w #1#2#3#4 \l__xparse_args_tl
  { \__xparse_grab_R_aux:NNnnN #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn }
\cs_new_protected:Npn \__xparse_grab_R_long:w #1#2#3#4 \l__xparse_args_tl
  { \__xparse_grab_R_aux:NNnnN #1 #2 {#3} {#4} \cs_set_protected:Npn }
\cs_new_protected:Npn \__xparse_grab_R_aux:NNnnN #1#2#3#4#5
  {
    \__xparse_grab_D_aux:NNnN #1 #2 {#4} #5
    \peek_meaning_remove_ignore_spaces:NTF #1
      { \__xparse_grab_arg:w }
      {
        \__msg_kernel_error:nnxx { xparse } { missing-required }
          { \token_to_str:N #1 } { \tl_to_str:n {#3} }
        \__xparse_add_arg:n {#3}
        #4 \l__xparse_args_tl
      }
  }
```

(*End definition for* `\__xparse_grab_R:w` *and* `\__xparse_grab_R_long:w` *These functions are documented on page* **??**.)

`\__xparse_grab_t:w`
`\__xparse_grab_t_long:w`
`\__xparse_grab_t_trailing:w`
`\__xparse_grab_t_long_trailing:w`
`\__xparse_grab_t_aux:NnNn`

Dealing with a token is quite easy. Check the match, remove the token if needed and add a flag to the output.

```
\cs_new_protected:Npn \__xparse_grab_t:w #1#2 \l__xparse_args_tl
  {
    \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected_nopar:Npn
      { _ignore_spaces }
  }
\cs_new_protected:Npn \__xparse_grab_t_long:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected:Npn { _ignore_spaces } }
\cs_new_protected:Npn \__xparse_grab_t_trailing:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected_nopar:Npn { } }
\cs_new_protected:Npn \__xparse_grab_t_long_trailing:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected:Npn { } }
\cs_new_protected:Npn \__xparse_grab_t_aux:NnNn #1#2#3#4
  {
    \exp_after:wN #3 \l__xparse_fn_tl
      {
        \use:c { peek_meaning_remove #4 :NTF } #1
          {
            \__xparse_add_arg:n { \BooleanTrue }
            #2 \l__xparse_args_tl
          }
          {
            \__xparse_add_arg:n { \BooleanFalse }
            #2 \l__xparse_args_tl
          }
      }
    \l__xparse_fn_tl
  }
```

(*End definition for* `\__xparse_grab_t:w` *This function is documented on page* **??**.)

`\__xparse_grab_u:w`
`\__xparse_grab_u_long:w`
`\__xparse_grab_u_aux:nnN`

Grabbing up to a list of tokens is quite easy: define the grabber, and then collect.

```
\cs_new_protected:Npn \__xparse_grab_u:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_u_aux:nnN {#1} {#2} \cs_set_protected_nopar:Npn }
\cs_new_protected:Npn \__xparse_grab_u_long:w #1#2 \l__xparse_args_tl
  { \__xparse_grab_u_aux:nnN {#1} {#2} \cs_set_protected:Npn }
\cs_new_protected:Npn \__xparse_grab_u_aux:nnN #1#2#3
  {
    \exp_after:wN #3 \l__xparse_fn_tl ##1 #1
      {
        \__xparse_add_arg:n {##1}
        #2 \l__xparse_args_tl
      }
    \l__xparse_fn_tl
  }
```

(*End definition for* `\__xparse_grab_u:w` *This function is documented on page* **??**.)

`\__xparse_grab_v:w`
`\__xparse_grab_v_long:w`
`\__xparse_grab_v_aux:w`
`\__xparse_grab_v_group_end:`
`\l__xparse_v_rest_of_signature_tl`
`\l__xparse_v_arg_tl`

The opening delimiter is never read verbatim, for consistency: if the preceeding argument was optional and absent, then TEX has already read that token when looking for the optional argument. The first thing to check is that this delimiter is a character, and distinguish the case of a left brace (in that case, `\group_align_safe_end:` is needed to compensate for the begin-group character that was just seen). Then set verbatim catcodes with `\__xparse_grab_v_aux_catcodes:`.

The group keep catcode changes local, and `\group_align_safe_begin/end:` allow to use a character with category code 4 (normally `&`) as the delimiter. It is ended by `\__xparse_grab_v_group_end:`, which smuggles the collected argument out of the group.

```
\tl_new:N \l__xparse_v_rest_of_signature_tl
\tl_new:N \l__xparse_v_arg_tl
\cs_new_protected_nopar:Npn \__xparse_grab_v:w
  {
    \bool_set_false:N \l__xparse_long_bool
    \__xparse_grab_v_aux:w
  }
```

```
\cs_new_protected_nopar:Npn \__xparse_grab_v_long:w
  {
    \bool_set_true:N \l__xparse_long_bool
    \__xparse_grab_v_aux:w
  }
\cs_new_protected:Npn \__xparse_grab_v_aux:w #1 \l__xparse_args_tl
  {
    \tl_set:Nn \l__xparse_v_rest_of_signature_tl {#1}
    \group_begin:
      \group_align_safe_begin:
        \tex_escapechar:D = 92 \scan_stop:
        \tl_clear:N \l__xparse_v_arg_tl
        \peek_N_type:TF
          { \__xparse_grab_v_aux_test:N }
          {
            \peek_meaning_remove:NTF \c_group_begin_token
              {
                \group_align_safe_end:
                \__xparse_grab_v_bgroup:
              }
              { \__xparse_grab_v_aux_abort: }
          }
  }
\cs_new_protected_nopar:Npn \__xparse_grab_v_group_end:
  {
        \group_align_safe_end:
        \exp_args:NNNo
      \group_end:
    \tl_set:Nn \l__xparse_v_arg_tl { \l__xparse_v_arg_tl }
  }
```

(*End definition for* `\__xparse_grab_v:w` *This function is documented on page* **??**.)

`\__xparse_grab_v_aux_test:N`
`\__xparse_grab_v_aux_loop:N`
`\__xparse_grab_v_aux_loop_ii:NN`
`\__xparse_grab_v_aux_loop_end:`

Check that the opening delimiter is a character, setup category codes, then start reading tokens one by one, keeping the delimiter as an argument. If the verbatim was not nested, we will be grabbing one character at each step. Unfortunately, it can happen that what follows the verbatim argument is already tokenized. Thus, we check at each step that the next token is indeed a "nice" character, *i.e.*, is not a character with category code 1 (begin-group), 2 (end-group) or 6 (macro parameter), nor the space character, with category code 10 and character code 32, nor a control sequence. The partially built argument is stored in `\l__xparse_v_arg_tl`. If we ever meet a token which we cannot grab (non-N-type), or which is not a character according to `\__xparse_grab_v_token_-if_char:NTF`, then we bail out with `\__xparse_grab_v_aux_abort:`. Otherwise, we stop at the first character matching the delimiter.

```
\cs_new_protected:Npn \__xparse_grab_v_aux_test:N #1
  {
    \__xparse_grab_v_aux_put:N #1
    \__xparse_grab_v_token_if_char:NTF #1
      {
        \__xparse_grab_v_aux_catcodes:
```

```
        \__xparse_grab_v_aux_loop:N #1
      }
      { \__xparse_grab_v_aux_abort: }
  }
\cs_new_protected:Npn \__xparse_grab_v_aux_loop:N #1
  {
    \peek_N_type:TF
      { \__xparse_grab_v_aux_loop_ii:NN #1 }
      { \__xparse_grab_v_aux_abort: }
  }
\cs_new_protected:Npn \__xparse_grab_v_aux_loop_ii:NN #1 #2
  {
    \__xparse_grab_v_token_if_char:NTF #2
      {
        \token_if_eq_charcode:NNTF #1 #2
          { \__xparse_grab_v_aux_loop_end: }
          {
            \__xparse_grab_v_aux_put:N #2
            \__xparse_grab_v_aux_loop:N #1
          }
      }
      { \__xparse_grab_v_aux_abort: #2 }
  }
\cs_new_protected_nopar:Npn \__xparse_grab_v_aux_loop_end:
  {
    \__xparse_grab_v_group_end:
    \exp_args:Nx \__xparse_add_arg:n { \tl_tail:N \l__xparse_v_arg_tl }
    \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
  }
```

(*End definition for `\__xparse_grab_v_aux_test:N` This function is documented on page* **??**.)

`\__xparse_grab_v_bgroup:`
`\__xparse_grab_v_bgroup_loop:`
`\__xparse_grab_v_bgroup_loop_ii:N`
`\l__xparse_v_nesting_int`

If the opening delimiter is a left brace, we keep track of how many left and right braces were encountered so far in `\l__xparse_v_nesting_int` (the methods used for optional arguments cannot apply here), and stop as soon as it reaches 0.

Some care was needed when removing the opening delimiter, which has already been assigned category code 1: using `\peek_meaning_remove:NTF` in the `\__xparse_grab_v_aux:w` function would break within alignments. Instead, we first convert that token to a string, and remove the result as a normal undelimited argument.

```
\int_new:N \l__xparse_v_nesting_int
\cs_new_protected_nopar:Npx \__xparse_grab_v_bgroup:
  {
    \exp_not:N \__xparse_grab_v_aux_catcodes:
    \exp_not:n { \int_set_eq:NN \l__xparse_v_nesting_int \c_one }
    \exp_not:N \__xparse_grab_v_aux_put:N \iow_char:N \{
    \exp_not:N \__xparse_grab_v_bgroup_loop:
  }
\cs_new_protected:Npn \__xparse_grab_v_bgroup_loop:
  {
    \peek_N_type:TF
```

```
      { \__xparse_grab_v_bgroup_loop_ii:N }
      { \__xparse_grab_v_aux_abort: }
  }
\cs_new_protected:Npn \__xparse_grab_v_bgroup_loop_ii:N #1
  {
    \__xparse_grab_v_token_if_char:NTF #1
      {
        \token_if_eq_charcode:NNTF \c_group_end_token #1
          {
            \int_decr:N \l__xparse_v_nesting_int
            \int_compare:nNnTF \l__xparse_v_nesting_int > \c_zero
              {
                \__xparse_grab_v_aux_put:N #1
                \__xparse_grab_v_bgroup_loop:
              }
              { \__xparse_grab_v_aux_loop_end: }
          }
          {
            \token_if_eq_charcode:NNT \c_group_begin_token #1
              { \int_incr:N \l__xparse_v_nesting_int }
            \__xparse_grab_v_aux_put:N #1
            \__xparse_grab_v_bgroup_loop:
          }
      }
      { \__xparse_grab_v_aux_abort: #1 }
  }
```

(*End definition for* `\__xparse_grab_v_bgroup`*: This function is documented on page* **??**.)

__xparse_grab_v_aux_catcodes:
__xparse_grab_v_aux_abort:
__xparse_grab_v_aux_abort_ii:w

In a standalone format, the list of special characters is kept as a sequence, `\c__xparse_-special_chars_seq`, and we use `\dospecials` in package mode. The approach for short verbatim arguments is to make the end-line character a macro parameter character: this is forbidden by the rest of the code. Then the error branch can check what caused the bail out and give the appropriate error message.

```
\cs_new_protected_nopar:Npn \__xparse_grab_v_aux_catcodes:
  {
⟨*initex⟩
    \seq_map_function:NN
      \c__xparse_special_chars_seq
      \char_set_catcode_other:N
⟨/initex⟩
⟨*package⟩
    \cs_set_eq:NN \do \char_set_catcode_other:N
    \dospecials
⟨/package⟩
    \tex_endlinechar:D = `\^^M \scan_stop:
    \bool_if:NTF \l__xparse_long_bool
      { \char_set_catcode_other:n { \tex_endlinechar:D } }
      { \char_set_catcode_parameter:n { \tex_endlinechar:D } }
  }
```

```
\cs_new_protected_nopar:Npn \__xparse_grab_v_aux_abort:
  {
    \__xparse_grab_v_group_end:
    \__xparse_add_arg:o \c__xparse_no_value_tl
    \exp_after:wN \__xparse_grab_v_aux_abort_ii:w \l__xparse_args_tl \q_stop
  }
\cs_new_protected:Npn \__xparse_grab_v_aux_abort_ii:w #1 #2 \q_stop
  {
    \group_begin:
    \char_set_lccode:nn { `\# } { \tex_endlinechar:D }
    \tl_to_lowercase:n
      { \group_end: \peek_meaning_remove:NTF ## }
      {
        \__msg_kernel_error:nnxx { xparse } { verbatim-newline }
          { \token_to_str:N #1 }
          { \tl_to_str:N \l__xparse_v_arg_tl }
        \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
      }
      {
        \__msg_kernel_error:nnxx { xparse } { verbatim-already-tokenized }
          { \token_to_str:N #1 }
          { \tl_to_str:N \l__xparse_v_arg_tl }
        \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
      }
  }
```

(*End definition for* `\__xparse_grab_v_aux_catcodes:` *This function is documented on page* **??***.*)

`\__xparse_grab_v_aux_put:N` Storing one token in the collected argument. Most tokens are converted to category code 12, with the exception of active characters, and spaces (not sure what should be done for those).

```
\cs_new_protected:Npn \__xparse_grab_v_aux_put:N #1
  {
    \tl_put_right:Nx \l__xparse_v_arg_tl
      {
        \token_if_active:NTF #1
          { \exp_not:N #1 } { \token_to_str:N #1 }
      }
  }
```

(*End definition for* `\__xparse_grab_v_aux_put:N` *This function is documented on page* **??***.*)

`\__xparse_grab_v_token_if_char:NTF` This function assumes that the escape character is printable. Then the string representation of control sequences is at least two characters, and `\str_tail:n` only removes the escape character. Macro parameter characters are doubled by `\tl_to_str:n`, and will also yield a non-empty result, hence are not considered as characters.

```
\cs_new_protected:Npn \__xparse_grab_v_token_if_char:NTF #1
  { \str_if_eq_x:nnTF { } { \str_tail:n {#1} } }
```

(*End definition for* `\__xparse_grab_v_token_if_char:NTF` *This function is documented on page* **??***.*)

__xparse_add_arg:n
__xparse_add_arg:V
__xparse_add_arg:o
__xparse_add_arg_aux:n
__xparse_add_arg_aux:V

The argument-storing system provides a single point for interfacing with processors. They are done in a loop, counting downward. In this way, the processor which was found last is executed first. The result is that processors apply from right to left, as intended. Notice that a set of braces are added back around the result of processing so that the internal function will correctly pick up one argument for each input argument.

```
\cs_new_protected:Npn \__xparse_add_arg:n #1
  {
    \int_compare:nNnTF \l__xparse_processor_int = \c_zero
      { \tl_put_right:Nn \l__xparse_args_tl { {#1} } }
      {
        \tl_clear:N \ProcessedArgument
        \__xparse_if_no_value:nTF {#1}
          {
            \int_zero:N \l__xparse_processor_int
            \tl_put_right:Nn \l__xparse_args_tl { {#1} }
          }
          { \__xparse_add_arg_aux:n {#1} }
      }
  }
\cs_generate_variant:Nn \__xparse_add_arg:n { V , o }
\cs_new_protected:Npn \__xparse_add_arg_aux:n #1
  {
    \use:c { __xparse_processor_ \int_use:N \l__xparse_processor_int :n } {#1}
    \int_decr:N \l__xparse_processor_int
    \int_compare:nNnTF \l__xparse_processor_int = \c_zero
      {
        \tl_put_right:Nx \l__xparse_args_tl
          { { \exp_not:V \ProcessedArgument } }
      }
      { \__xparse_add_arg_aux:V \ProcessedArgument }
}
\cs_generate_variant:Nn \__xparse_add_arg_aux:n { V }
```

(*End definition for* `\__xparse_add_arg:n` *,* `\__xparse_add_arg:V` *, and* `\__xparse_add_arg:o` *These functions are documented on page* **??***.*)

2.8 Grabbing arguments expandably

__xparse_expandable_grab_D:w
__xparse_expandable_grab_D_i:NNNnwN
__xparse_expandable_grab_D_ii:NNNwNnnn
__xparse_expandable_grab_D_iii:Nw
__xparse_expandable_grab_D_iv:nnNNNwN

The first step is to grab the first token or group. The generic grabber `\⟨function⟩␣` is just after `\q__xparse`, we go and find it.

```
\cs_new:Npn \__xparse_expandable_grab_D:w #1 \q__xparse #2
  { #2 { \__xparse_expandable_grab_D_i:NNNnwNn #1 \q__xparse #2 } }
```

We then wish to test whether `#7`, which we just grabbed, is exactly `#2`. Expand the only grabber function we have, `#1`, once: the two strings below are equal if and only if `#7` matches `#2` exactly.[1] If `#7` does not match `#2`, then the optional argument is missing, we

[1] It is obvious that if `#7` matches `#2` then the strings are equal. We must check the converse. The right-hand-side of `\str_if_eq:onTF` does not end with `#3`, implying that the grabber function took everything as its arguments. The first brace group can only be empty if `#7` starts with `#2`, otherwise the brace group preceding `#7` would not vanish. The third brace group is empty, thus the `\q__xparse` that was used by

use the default #4, and put back the argument #7 in the input stream.

If it does match, then interesting things need to be done. We will grab the argument piece by piece, with the following pattern:

> ⟨*grabber*⟩ {⟨*tokens*⟩}
> \q_nil {⟨*piece 1*⟩} ⟨*piece 2*⟩ \ERROR \q__xparse
> \q_nil ⟨*input stream*⟩

The ⟨*grabber*⟩ will find an opening delimiter in ⟨*piece 2*⟩, take the \q__xparse as a second delimiter, and find more material delimited by the closing delimiter in the ⟨*input stream*⟩. We then move the part before the opening delimiter from ⟨*piece 2*⟩ to ⟨*piece 1*⟩, and the material taken from the ⟨*input stream*⟩ to the ⟨*piece 2*⟩. Thus, the argument moves gradually from the ⟨*input stream*⟩ to the ⟨*piece 2*⟩, then to the ⟨*piece 1*⟩ when we have made sure to find all opening and closing delimiters. This two-step process ensures that nesting works: the number of opening delimiters minus closing delimiters in ⟨*piece 1*⟩ is always equal to the number of closing delimiters in ⟨*piece 2*⟩. We stop grabbing arguments once the ⟨*piece 2*⟩ contains no opening delimiter any more, hence the balance is reached, and the final argument is ⟨*piece 1*⟩ ⟨*piece 2*⟩.

```
\cs_new:Npn \__xparse_expandable_grab_D_i:NNNnwNn #1#2#3#4#5 \q__xparse #6#7
  {
    \str_if_eq:onTF
      { #1 { } { } #7 #2 \q__xparse #3 }
      { { } { #2 } { } }
      {
        #1
          { \__xparse_expandable_grab_D_ii:NNNwNnnn #1#2#3#5 \q__xparse #6 }
          \q_nil { } #2 \ERROR \q__xparse \ERROR
      }
      { #5 {#4} \q__xparse #6 {#7} }
  }
```

At this stage, #6 is \q_nil {⟨*piece 1*⟩} ⟨*more for piece 1*⟩, and we want to concatenate all that, removing \q_nil, and keeping the opening delimiter #2. Simply use \use_ii:nn. Also, #7 is ⟨*remainder of piece 2*⟩ \ERROR, and #8 is \ERROR ⟨*more for piece 2*⟩. We concatenate those, replacing the two \ERROR by the closing delimiter #3.

```
\cs_new:Npn \__xparse_expandable_grab_D_ii:NNNwNnnn #1#2#3#4 \q__xparse #5#6#7#8
  {
    \exp_args:Nof \__xparse_expandable_grab_D_iv:nnNNNwN
      { \use_ii:nn #6 #2 }
      { \__xparse_expandable_grab_D_iii:Nw #3 \exp_stop_f: #7 #8 }
    #1#2#3 #4 \q__xparse #5
  }
\cs_new:Npn \__xparse_expandable_grab_D_iii:Nw #1#2 \ERROR \ERROR { #2 #1 }
```

Armed with our two new ⟨*pieces*⟩, we are ready to loop. However, we must first see if ⟨*piece 2*⟩ (here #2) contains any opening delimiter #4. Again, we expand #3, this time

our grabber #1 must be the one that we inserted (not some token in #7), hence the second brace group contains the end of #7 followed by #2. Since this is #2 on the right-hand-side, and no brace can be lost there, #7 must contain nothing else than its leading #2.

removing its whole output with `\use_none:nnn`. The test is similar to `\tl_if_in:nnTF`. The token list is empty if and only if `#2` does not contain the opening delimiter. In that case, we are done, and put the argument (from which we remove a spurious pair of delimiters coming from how we started the loop). Otherwise, we go back to looping with `\__xparse_expandable_grab_D_ii:NNNwNnnn`.

```
\cs_new:Npn \__xparse_expandable_grab_D_iv:nnNNNwN #1#2#3#4#5#6 \q__xparse #7
  {
    \exp_args:No \tl_if_empty:oTF
      { #3 { \use_none:nnn } #2 \q__xparse #5 #4 \q__xparse #5 }
      {
        \__xparse_put_arg_expandable:ow { \use_none:nn #1#2 }
          #6 \q__xparse #7
      }
      {
        #3
          { \__xparse_expandable_grab_D_ii:NNNwNnnn #3#4#5#6 \q__xparse #7 }
          \q_nil {#1} #2 \ERROR \q__xparse \ERROR
      }
  }
```

(*End definition for* `\__xparse_expandable_grab_D:w` *This function is documented on page* **??**.)

`\__xparse_expandable_grab_D_alt:w`
`\__xparse_expandable_grab_D_alt_i:NNnwNn`
`\__xparse_expandable_grab_D_alt_ii:Nw`

When the delimiters are identical, nesting is not possble and a simplified approach is used. The test concept here is the same as for the case where the delimiters are different.

```
\cs_new:Npn \__xparse_expandable_grab_D_alt:w #1 \q__xparse #2
  { #2 { \__xparse_expandable_grab_D_alt_i:NNnwNn #1 \q__xparse #2 } }
\cs_new:Npn \__xparse_expandable_grab_D_alt_i:NNnwNn #1#2#3#4 \q__xparse #5#6
  {
    \str_if_eq:onTF
      { #1 { } #6 #2 #2 }
      { { } #2 }
      {
        #1
          { \__xparse_expandable_grab_D_alt_ii:Nwn #5 #4 \q__xparse }
          #6 \ERROR
      }
      { #4 {#3} \q__xparse #5 {#6} }
  }
\cs_new:Npn \__xparse_expandable_grab_D_alt_ii:Nwn #1#2 \q__xparse #3
  { \__xparse_put_arg_expandable:ow { \use_none:n #3 } #2 \q__xparse #1 }
```

(*End definition for* `\__xparse_expandable_grab_D_alt:w` *This function is documented on page* **??**.)

`\__xparse_expandable_grab_m:w`
`\__xparse_expandable_grab_m_aux:wNn`

The mandatory case is easy: find the auxiliary after the `\q__xparse`, and use it directly to grab the argument.

```
\cs_new:Npn \__xparse_expandable_grab_m:w #1 \q__xparse #2
  { #2 { \__xparse_expandable_grab_m_aux:wNn #1 \q__xparse #2 } }
\cs_new:Npn \__xparse_expandable_grab_m_aux:wNn #1 \q__xparse #2#3
  { #1 {#3} \q__xparse #2 }
```

(*End definition for* `\__xparse_expandable_grab_m:w` *This function is documented on page* **??**.)

__xparse_expandable_grab_R:w
__xparse_expandable_grab_R_aux:NNNwn

Much the same as for the D-type argument, with only the lead-off function varying.

```
\cs_new:Npn \__xparse_expandable_grab_R:w #1 \q__xparse #2
  { #2 { \__xparse_expandable_grab_R_aux:NNNnwNn #1 \q__xparse #2 } }
\cs_new:Npn \__xparse_expandable_grab_R_aux:NNNnwNn #1#2#3#4#5 \q__xparse #6#7
  {
    \str_if_eq:onTF
      { #1 { } { } #7 #2 \q__xparse #3 }
      { { } { #2 } { } }
      {
        #1
          { \__xparse_expandable_grab_D_ii:NNNwNnnn #1#2#3#5 \q__xparse #6 }
          \q_nil { } #2 \ERROR \q__xparse \ERROR
      }
      {
        \__msg_kernel_expandable_error:nnn
          { xparse } { missing-required } {#2}
        #5 {#4} \q__xparse #6 {#7}
      }
  }
```

(*End definition for* `\__xparse_expandable_grab_R:w` *This function is documented on page* **??**.)

__xparse_expandable_grab_R_alt:w
__xparse_expandable_grab_R_alt_aux:NNnwNn

When the delimiters are identical, nesting is not possble and a simplified approach is used. The test concept here is the same as for the case where the delimiters are different.

```
\cs_new:Npn \__xparse_expandable_grab_R_alt:w #1 \q__xparse #2
  { #2 { \__xparse_expandable_grab_R_alt_aux:NNnwNn #1 \q__xparse #2 } }
\cs_new:Npn \__xparse_expandable_grab_R_alt_aux:NNnwNn #1#2#3#4 \q__xparse #5#6
  {
    \str_if_eq:onTF
      { #1 { } #6 #2 #2 }
      { { } #2 }
      {
        #1
          { \__xparse_expandable_grab_D_alt_ii:Nwn #5 #4 \q__xparse }
          #6 \ERROR
      }
      {
        \__msg_kernel_expandable_error:nnn
          { xparse } { missing-required } {#2}
        #4 {#3} \q__xparse #5 {#6}
      }
  }
```

(*End definition for* `\__xparse_expandable_grab_R_alt:w` *This function is documented on page* **??**.)

__xparse_expandable_grab_t:w
__xparse_expandable_grab_t_aux:NNwn

As for a D-type argument, here we compare the grabbed tokens using the only parser we have in order to work out if #2 is exactly equal to the output of the grabber.

```
\cs_new:Npn \__xparse_expandable_grab_t:w #1 \q__xparse #2
  { #2 { \__xparse_expandable_grab_t_aux:NNwn #1 \q__xparse #2 } }
\cs_new:Npn \__xparse_expandable_grab_t_aux:NNwn #1#2#3 \q__xparse #4#5
  {
    \str_if_eq:onTF { #1 { } #5 #2 } { #2 }
      { #3 { \BooleanTrue } \q__xparse #4 }
      { #3 { \BooleanFalse } \q__xparse #4 {#5} }
  }
```

(*End definition for* `\__xparse_expandable_grab_t:w` *This function is documented on page* **??**.)

`\__xparse_put_arg_expandable:nw`
`\__xparse_put_arg_expandable:ow`

A useful helper, to store arguments when they are ready.

```
\cs_new:Npn \__xparse_put_arg_expandable:nw #1#2 \q__xparse { #2 {#1} \q__xparse }
\cs_generate_variant:Nn \__xparse_put_arg_expandable:nw { o }
```

(*End definition for* `\__xparse_put_arg_expandable:nw` *and* `\__xparse_put_arg_expandable:ow` *These functions are documented on page* **??**.)

`\__xparse_grab_expandable_end:wN`

For the end of the grabbing sequence: get rid of the generic grabber and insert the code function followed by its arguments.

```
\cs_new:Npn \__xparse_grab_expandable_end:wN #1 \q__xparse #2 {#1}
```

(*End definition for* `\__xparse_grab_expandable_end:wN` *This function is documented on page* **??**.)

2.9 Argument processors

`\__xparse_process_arg:n`

Processors are saved for use later during the grabbing process.

```
\cs_new_protected:Npn \__xparse_process_arg:n #1
  {
    \int_incr:N \l__xparse_processor_int
    \cs_set:cpn { __xparse_processor_ \int_use:N \l__xparse_processor_int :n } ##1
      { #1 {##1} }
  }
```

(*End definition for* `\__xparse_process_arg:n` *This function is documented on page* **??**.)

`\__xparse_process_to_str:n`

A basic argument processor: as much an example as anything else.

```
\cs_new_protected:Npn \__xparse_process_to_str:n #1
  { \tl_set:Nx \ProcessedArgument { \tl_to_str:n {#1} } }
```

(*End definition for* `\__xparse_process_to_str:n` *This function is documented on page* **??**.)

`\__xparse_bool_reverse:N`

A simple reversal.

```
\cs_new_protected:Npn \__xparse_bool_reverse:N #1
  {
    \bool_if:NTF #1
      { \tl_set:Nn \ProcessedArgument { \c_false_bool } }
      { \tl_set:Nn \ProcessedArgument { \c_true_bool } }
  }
```

(*End definition for* `\__xparse_bool_reverse:N` *This function is documented on page* **??**.)

`\l__xparse_split_list_seq`
`\l__xparse_split_list_tl`
`\__xparse_split_list:nn`
`\__xparse_split_list_multi:nn`
`\__xparse_split_list_multi:nV`
`\__xparse_split_list_single:Nn`

Splitting can take place either at a single token or at a longer identifier. To deal with single active tokens, a two-part procedure is needed.

```
\seq_new:N \l__xparse_split_list_seq
\tl_new:N \l__xparse_split_list_tl
\cs_new_protected:Npn \__xparse_split_list:nn #1#2
  {
    \bool_if:nTF
      {
          \tl_if_single_p:n {#1} &&
        ! ( \token_if_cs_p:N #1 )
      }
      { \__xparse_split_list_single:Nn #1 {#2} }
      { \__xparse_split_list_multi:nn {#1} {#2} }
    }
\cs_set_protected:Npn \__xparse_split_list_multi:nn #1#2
  {
    \seq_set_split:Nnn \l__xparse_split_list_seq {#1} {#2}
    \tl_clear:N \ProcessedArgument
    \seq_map_inline:Nn \l__xparse_split_list_seq
      { \tl_put_right:Nn \ProcessedArgument { {##1} } }
  }
\cs_generate_variant:Nn \__xparse_split_list_multi:nn { nV }
\group_begin:
\char_set_catcode_active:N \@
\cs_new_protected:Npn \__xparse_split_list_single:Nn #1#2
  {
    \tl_set:Nn \l__xparse_split_list_tl {#2}
    \group_begin:
    \char_set_lccode:nn { `\@ } { `#1 }
    \tl_to_lowercase:n
      {
         \group_end:
        \tl_replace_all:Nnn \l__xparse_split_list_tl { @ } {#1}
      }
     \__xparse_split_list_multi:nV {#1}  \l__xparse_split_list_tl
   }
\group_end:
```

(*End definition for* `\l__xparse_split_list_seq` *and* `\l__xparse_split_list_tl` *These functions are documented on page* **??**.)

`\__xparse_split_argument:nnn`
`\__xparse_split_argument_aux:nnnn`
`\__xparse_split_argument_aux:n`
`\__xparse_split_argument_aux:wn`

Splitting to a known number of items is a special version of splitting a list, in which the limit is hard-coded and where there will always be exactly the correct number of output items. An auxiliary function is used to save on working out the token list length several times.

```
\cs_new_protected:Npn \__xparse_split_argument:nnn #1#2#3
  {
    \__xparse_split_list:nn {#2} {#3}
    \exp_args:Nf \__xparse_split_argument_aux:nnnn
      { \tl_count:N \ProcessedArgument }
      {#1} {#2} {#3}
  }
\cs_new_protected:Npn \__xparse_split_argument_aux:nnnn #1#2#3#4
  {
    \int_compare:nNnF {#1} = { #2 + \c_one }
      {
        \int_compare:nNnTF {#1} > { #2 + \c_one }
          {
            \tl_set:Nx \ProcessedArgument
              {
                \exp_last_unbraced:NnNo
                  \__xparse_split_argument_aux:n
                  { #2 + \c_one }
                  \use_none_delimit_by_q_stop:w
                  \ProcessedArgument
                  \q_stop
              }
            \__msg_kernel_error:nnxxx { xparse } { split-excess-tokens }
              { \tl_to_str:n {#3} } { \int_eval:n { #2 + \c_one } }
              { \tl_to_str:n {#4} }
          }
          {
            \tl_put_right:Nx \ProcessedArgument
              {
                \prg_replicate:nn { #2 + \c_one - (#1) }
                  { { \exp_not:V \c__xparse_no_value_tl } }
              }
          }
      }
  }
```

Auxiliaries to leave exactly the correct number of arguments in \ProcessedArgument.

```
\cs_new:Npn \__xparse_split_argument_aux:n #1
  { \prg_replicate:nn {#1} { \__xparse_split_argument_aux:wn } }
\cs_new:Npn \__xparse_split_argument_aux:wn #1 \use_none_delimit_by_q_stop:w #2
  {
    \exp_not:n { {#2} }
    #1
    \use_none_delimit_by_q_stop:w
  }
```

(*End definition for* __xparse_split_argument:nnn *This function is documented on page* **??**.)

__xparse_trim_spaces:n

This one is almost trivial.

```
\cs_new_protected:Npn \__xparse_trim_spaces:n #1
  { \tl_set:Nx \ProcessedArgument { \tl_trim_spaces:n {#1} } }
```

(*End definition for* __xparse_trim_spaces:n *This function is documented on page* **??**.)

2.10 Access to the argument specification

__xparse_get_arg_spec:N
__xparse_get_arg_spec:n
\ArgumentSpecification

Recovering the argument specification is also trivial, using the \tl_set_eq:cN function.

```
\cs_new_protected:Npn \__xparse_get_arg_spec:N #1
  {
    \prop_get:NnNF \l__xparse_command_arg_specs_prop {#1}
      \ArgumentSpecification
      {
        \__msg_kernel_error:nnx { xparse } { unknown-document-command }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \__xparse_get_arg_spec:n #1
  {
    \prop_get:NnNF \l__xparse_environment_arg_specs_prop {#1}
      \ArgumentSpecification
      {
        \__msg_kernel_error:nnx { xparse } { unknown-document-environment }
          { \tl_to_str:n {#1} }
      }
  }
\tl_new:N \ArgumentSpecification
```

(*End definition for* `\__xparse_get_arg_spec:N` *This function is documented on page* **??**.)

`\__xparse_show_arg_spec:N`
`\__xparse_show_arg_spec:n`

Showing the argument specification simply means finding it and then calling the `\tl_-show:N` function.

```
\cs_new_protected:Npn \__xparse_show_arg_spec:N #1
  {
    \prop_get:NnNTF \l__xparse_command_arg_specs_prop {#1}
      \ArgumentSpecification
      { \tl_show:N \ArgumentSpecification }
      {
        \__msg_kernel_error:nnx { xparse } { unknown-document-command }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \__xparse_show_arg_spec:n #1
  {
    \prop_get:NnNTF \l__xparse_environment_arg_specs_prop {#1}
      \ArgumentSpecification
      { \tl_show:N \ArgumentSpecification }
      {
        \__msg_kernel_error:nnx { xparse } { unknown-document-environment }
          { \tl_to_str:n {#1} }
      }
  }
```

(*End definition for* `\__xparse_show_arg_spec:N` *This function is documented on page* **??**.)

2.11 Utilities

`\__xparse_if_no_value:nTF`

Tests for `-NoValue-`: this is similar to `\tl_if_in:nn` but set up to be expandable. The question mark prevents the auxiliary from losing braces.

```
\group_begin:
\char_set_lccode:nn { `\Q } { `\- }
\char_set_lccode:nn { `\F } { `\F }
\char_set_lccode:nn { `\N } { `\N }
\char_set_lccode:nn { `\T } { `\T }
```

```
\char_set_lccode:nn { `\V } { `\V }
\tl_to_lowercase:n
  {
    \group_end:
    \prg_new_conditional:Npnn \__xparse_if_no_value:n #1 { T ,  F , TF }
      {
        \str_if_eq:onTF
          { \__xparse_if_value_aux:w ? #1 { } QNoValue- }
          { ? { } QNoValue- }
          { \prg_return_true: }
          { \prg_return_false: }
      }
    \cs_new:Npn \__xparse_if_value_aux:w #1 QNoValue- { #1 }
  }
```

(*End definition for* `\__xparse_if_no_value:nTF` *This function is documented on page* **??**.)

2.12 Messages

2.13 Messages

Some messages intended as errors.

```
\__msg_kernel_new:nnnn { xparse } { bad-arg-spec }
  { Bad~argument~specification~'#1'. }
  {
    \c_msg_coding_error_text_tl
    The~argument~specification~provided~was~not~valid:~
    one~or~more~mandatory~pieces~of~information~were~missing. \\ \\
    LaTeX~will~ignore~this~entire~definition.
  }
\__msg_kernel_new:nnnn { xparse } { command-already-defined }
  { Command~'#1'~already~defined! }
  {
    You~have~used~\NewDocumentCommand
    with~a~command~that~already~has~a~definition. \\
    The~existing~definition~of~'#1'~will~be~overwritten.
  }
\__msg_kernel_new:nnnn { xparse } { command-not-yet-defined }
  { Command ~'#1'~not~yet~defined! }
  {
    You~have~used~\RenewDocumentCommand
    with~a~command~that~was~never~defined.\\
    A~new~command~'#1'~will~be~created.
  }
\__msg_kernel_new:nnnn { xparse } { environment-already-defined }
  { Environment~'#1'~already~defined! }
  {
    You~have~used~\NewDocumentEnvironment
    with~an~environment~that~already~has~a~definition.\\
    The~existing~definition~of~'#1'~will~be~overwritten.
  }
\__msg_kernel_new:nnnn { xparse } { environment-mismatch }
  { Mismatch~between~start~and~end~of~environment. }
  {
    The~current~environment~is~called~'#1',~but~you~have~tried~to~
    end~one~called~'#2'.~Environments~have~to~be~properly~nested.
  }
\__msg_kernel_new:nnnn { xparse } { environment-not-yet-defined }
  { Environment~'#1'~not~yet~defined! }
  {
    You~have~used~\RenewDocumentEnvironment
    with~an~environment~that~was~never~defined.\\
    A~new~environment~'#1'~will~be~created.
  }
\__msg_kernel_new:nnnn { xparse } { environment-unknown }
  { Environment~'#1'~undefined. }
  {
    You~have~tried~to~start~an~environment~called~'#1',~
    but~this~has~never~been~defined.\\
    The~command~will~be~ignored.
  }
\__msg_kernel_new:nnnn { xparse } { expandable-ending-optional }
  { Argument~specification~for~expandable~command~ends~with~optional~argument. }
  {
    \c_msg_coding_error_text_tl
    Expandable~commands~must~have~a~final~mandatory~argument~
    (or~no~arguments~at~all).~You~cannot~have~a~terminal~optional~
    argument~with~expandable~commands.
  }
\__msg_kernel_new:nnnn { xparse } { inconsistent-long }
  { Inconsistent~long~arguments~for~expandable~command. }
  {
    \c_msg_coding_error_text_tl
    The~arguments~for~an~expandable~command~must~either~all~be~
    short~or~all~be~long.~You~have~tried~to~mix~the~two~types.
  }
\__msg_kernel_new:nnnn { xparse } { invalid-expandable-argument-type }
  { Argument~type~'#1'~not~available~for~an~expandable~function. }
  {
    \c_msg_coding_error_text_tl
    The~letter~'#1'~does~not~specify~an~argument~type~which~can~be~used~
    in~an~expandable~function.
    \\ \\
    LaTeX~will~assume~you~want~a~standard~mandatory~argument~(type~'m').
  }
\__msg_kernel_new:nnnn { xparse } { missing-required }
  { Failed~to~find~required~argument~starting~with~'#1'. }
  {
    There~is~supposed~to~be~an~argument~to~the~current~function~starting~with~
    '#1'.~LaTeX~did~not~find~it,~and~will~insert~'#2'~as~the~value~to~be~
    processed.
  }
\__msg_kernel_new:nnnn { xparse } { not-single-token }
  { Argument~delimiter~should~be~a~single~token:~'#1'. }
  {
    \c_msg_coding_error_text_tl
    The~argument~specification~provided~was~not~valid:~
    in~a~place~where~a~single~token~is~required,~LaTeX~found~'#1'. \\ \\
    LaTeX~will~ignore~this~entire~definition.
  }
\__msg_kernel_new:nnnn { xparse } { processor-in-expandable }
  { Argument~processors~cannot~be~used~with~expandable~functions. }
  {
    \c_msg_coding_error_text_tl
    The~argument~specification~for~#1~contains~a~processor~function:~
    this~is~only~supported~for~standard~robust~functions.
  }
\__msg_kernel_new:nnnn { xparse } { split-excess-tokens }
  { Too~many~'#1'~tokens~when~trying~to~split~argument. }
  {
    LaTeX~was~asked~to~split~the~input~'#3'~
    at~each~occurrence~of~the~token~'#1',~up~to~a~maximum~of~#2~parts.~
    There~were~too~many~'#1'~tokens.
  }
\__msg_kernel_new:nnnn { xparse } { unknown-argument-type }
  { Unknown~argument~type~'#1'~replaced~by~'m'. }
  {
    \c_msg_coding_error_text_tl
    The~letter~'#1'~does~not~specify~a~known~argument~type.~
    LaTeX~will~assume~you~want~a~standard~mandatory~argument~(type~'m').
  }
\__msg_kernel_new:nnnn { xparse } { unknown-document-command }
  { Unknown~document~command~'#1'. }
  {
    You~have~asked~for~the~argument~specification~for~a~command~'#1',~
    but~this~is~not~a~document~command.
  }
\__msg_kernel_new:nnnn { xparse } { unknown-document-environment }
  { Unknown~document~environment~'#1'. }
  {
    You~have~asked~for~the~argument~specification~for~a~command~'#1',~
    but~this~is~not~a~document~environment.
  }
\__msg_kernel_new:nnnn { xparse } { verbatim-newline }
  { Verbatim~argument~of~#1~ended~by~end~of~line. }
  {
    The~verbatim~argument~of~#1~cannot~contain~more~than~one~line,~but~the~end~
    of~the~current~line~has~been~reached.~You~have~probably~forgotten~the~
    closing~delimiter.
    \\ \\
    LaTeX~will~ignored~'#2'.
  }
\__msg_kernel_new:nnnn { xparse } { verbatim-already-tokenized }
  { Verbatim~command~#1~illegal~in~command~argument. }
  {
    The~command~#1~takes~a~verbatim~argument.~It~may~not~appear~within~
    the~argument~of~another~function.
    \\ \\
    LaTeX~will~ignore~'#2'.
  }
```

Intended more for information.

```
\__msg_kernel_new:nnn { xparse } { define-command }
  {
    Defining~document~command~#1~
    with~arg.~spec.~'#2'~\msg_line_context:.
  }
\__msg_kernel_new:nnn { xparse } { define-environment }
  {
    Defining~document~environment~'#1'~
    with~arg.~spec.~'#2'~\msg_line_context:.
  }
\__msg_kernel_new:nnn { xparse } { redefine-command }
  {
    Redefining~document~command~#1~
    with~arg.~spec.~'#2'~\msg_line_context:.
  }
\__msg_kernel_new:nnn { xparse } { redefine-environment }
  {
    Redefining~document~environment~'#1'~
    with~arg.~spec.~'#2'~\msg_line_context:.
  }
```

2.14 User functions

The user functions are more or less just the internal functions renamed.

\BooleanFalse
\BooleanTrue

Design-space names for the Boolean values.

```
\cs_new_eq:NN \BooleanFalse \c_false_bool
\cs_new_eq:NN \BooleanTrue  \c_true_bool
```

(*End definition for* `\BooleanFalse` *This function is documented on page 6.*)

\DeclareDocumentCommand
\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand

The user macros are pretty simple wrappers around the internal ones.

```
\cs_new_protected:Npn \DeclareDocumentCommand #1#2#3
  { \__xparse_declare_cmd:Nnn #1 {#2} {#3} }
\cs_new_protected:Npn \NewDocumentCommand #1#2#3
  {
    \cs_if_exist:NTF #1
      {
        \__msg_kernel_error:nnx { xparse } { command-already-defined }
          { \token_to_str:N #1 }
      }
      { \__xparse_declare_cmd:Nnn #1 {#2} {#3} }
  }
\cs_new_protected:Npn \RenewDocumentCommand #1#2#3
  {
    \cs_if_exist:NTF #1
      { \__xparse_declare_cmd:Nnn #1 {#2} {#3} }
      {
        \__msg_kernel_error:nnx { xparse } { command-not-yet-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \ProvideDocumentCommand #1#2#3
  { \cs_if_exist:NF #1 { \__xparse_declare_cmd:Nnn #1 {#2} {#3} } }
```

(*End definition for* `\DeclareDocumentCommand` *This function is documented on page 5.*)

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

Very similar for environments.

```
\cs_new_protected:Npn \DeclareDocumentEnvironment #1#2#3#4
  { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
\cs_new_protected:Npn \NewDocumentEnvironment #1#2#3#4
  {
    \cs_if_exist:cTF {#1}
      { \__msg_kernel_error:nnx { xparse } { environment-already-defined } {#1} }
      { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
}
\cs_new_protected:Npn \RenewDocumentEnvironment #1#2#3#4
  {
    \cs_if_exist:cTF {#1}
      { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
      { \__msg_kernel_error:nnx { xparse } { environment-not-yet-defined } {#1} }
  }
\cs_new_protected:Npn \ProvideDocumentEnvironment #1#2#3#4
  { \cs_if_exist:cF { #1 } { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} } }
```

(*End definition for* `\DeclareDocumentEnvironment` *This function is documented on page 6.*)

`\DeclareExpandableDocumentCommand`

The expandable version of the basic function is essentially the same.

```
\cs_new_protected:Npn \DeclareExpandableDocumentCommand #1#2#3
  { \__xparse_declare_expandable_cmd:Nnn #1 {#2} {#3} }
```

(*End definition for* `\DeclareExpandableDocumentCommand` *This function is documented on page 10.*)

`\IfBoolean`*TF*

The logical ⟨*true*⟩ and ⟨*false*⟩ statements are just the normal `\c_true_bool` and `\c_false_bool`, so testing for them is done with the `\bool_if:NTF` functions from l3prg.

```
\cs_new_eq:NN \IfBooleanTF \bool_if:NTF
\cs_new_eq:NN \IfBooleanT  \bool_if:NT
\cs_new_eq:NN \IfBooleanF  \bool_if:NF
```

(*End definition for* `\IfBooleanTF` *This function is documented on page 7.*)

\IfNoValue*TF* Simple re-naming.

```
\cs_new_eq:NN \IfNoValueF  \__xparse_if_no_value:nF
\cs_new_eq:NN \IfNoValueT  \__xparse_if_no_value:nT
\cs_new_eq:NN \IfNoValueTF \__xparse_if_no_value:nTF
```

(*End definition for* `\IfNoValueTF` *This function is documented on page 6.*)

\IfValue*TF* Inverted logic.

```
\cs_set:Npn \IfValueF { \__xparse_if_no_value:nT }
\cs_set:Npn \IfValueT { \__xparse_if_no_value:nF }
\cs_set:Npn \IfValueTF #1#2#3 { \__xparse_if_no_value:nTF {#1} {#3} {#2} }
```

(*End definition for* `\IfValueTF` *This function is documented on page 6.*)

\ProcessedArgument Processed arguments are returned using this name, which is reserved here although the definition will change.

```
\tl_new:N \ProcessedArgument
```

(*End definition for* `\ProcessedArgument` *This function is documented on page 7.*)

\ReverseBoolean
\SplitArgument
\SplitList
\TrimSpaces
Simple copies.

```
\cs_new_eq:NN \ReverseBoolean \__xparse_bool_reverse:N
\cs_new_eq:NN \SplitArgument  \__xparse_split_argument:nnn
\cs_new_eq:NN \SplitList      \__xparse_split_list:nn
\cs_new_eq:NN \TrimSpaces     \__xparse_trim_spaces:n
```

(*End definition for* `\ReverseBoolean` *and others. These functions are documented on page 9.*)

\ProcessList To support `\SplitList`.

```
\cs_new_eq:NN \ProcessList \tl_map_function:nN
```

(*End definition for* `\ProcessList` *This function is documented on page 8.*)

\GetDocumentCommandArgSpec
\GetDocumentEnvironmentArgSpec
\ShowDocumentCommandArgSpec
\ShowDocumentEnvironmentArgSpec
More simple mappings.

```
\cs_new_eq:NN \GetDocumentCommandArgSpec      \__xparse_get_arg_spec:N
\cs_new_eq:NN \GetDocumentEnvironmmentArgSpec \__xparse_get_arg_spec:n
\cs_new_eq:NN \ShowDocumentCommandArgSpec     \__xparse_show_arg_spec:N
\cs_new_eq:NN \ShowDocumentEnvironmentArgSpec \__xparse_show_arg_spec:n
```

(*End definition for* `\GetDocumentCommandArgSpec` *This function is documented on page 10.*)

2.15 Package options

A faked key–value option to keep the log clean. Not yet perfect, but better than nothing.

```
\DeclareOption { log-declarations = true } { }
\DeclareOption { log-declarations = false }
  {
    \msg_redirect_module:nnn { LaTeX / xparse } { info }    { none }
    \msg_redirect_module:nnn { LaTeX / xparse } { warning } { none }
  }
\DeclareOption { log-declarations } { }
\ProcessOptions \scan_stop:
```

⟨/package⟩

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.