

The `xparse` package

Document command parser*

The L^AT_EX3 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 L^AT_EX 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 \LaTeX 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 $\langle token1 \rangle$ and $\langle token2 \rangle$: `r $\langle token1 \rangle \langle token2 \rangle$` . If the opening $\langle token \rangle$ 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 $\langle default \rangle$, given as `R $\langle token1 \rangle \langle token2 \rangle \{ \langle default \rangle \}$` .
- u** Reads an argument “until” $\langle tokens \rangle$ are encountered, where the desired $\langle tokens \rangle$ are given as an argument to the specifier: `u $\{ \langle tokens \rangle \}$` .
- v** Reads an argument “verbatim”, between the following character and its next occurrence, in a way similar to the argument of the $\text{\LaTeX} 2_{\epsilon}$ 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 $\langle token1 \rangle$ and $\langle token2 \rangle$, which are given as arguments: `d $\langle token1 \rangle \langle token2 \rangle$` . As with **o**, if no value is given the special marker `-NoValue-` is returned.
- O** As for **o**, but returns $\langle default \rangle$ if no value is given. Should be given as `O $\{ \langle default \rangle \}$` .

- D As for `d`, but returns $\langle default \rangle$ if no value is given: `D $\langle token1 \rangle \langle token2 \rangle \{ \langle default \rangle \}$` . 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 $\langle token \rangle$, which will result in a value `\BooleanTrue` if $\langle token \rangle$ is present and `\BooleanFalse` otherwise. Given as `t $\langle token \rangle$` .
- 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 $\langle default \rangle$ if no value is given: `G $\{ \langle default \rangle \}$` .

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 $\langle balanced\ text \rangle$ 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 2ε`’s `\newcommand`, this applies on an argument-by-argument basis. So modifying the example to ‘`s o o +m O{default}`’ means that the mandatory argument is now `\long`, whereas the optional arguments are not.

Secondly, the 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 $\text{\LaTeX 2}_{\epsilon}$ `\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 L^AT_EX 3. All of the functions generated in this way are naturally robust (using the ε -T_EX `\protected` mechanism).

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

`\DeclareDocumentCommand` $\langle Function \rangle$ $\{\langle arg\ spec \rangle\}$ $\{\langle code \rangle\}$

This family of commands are used to create a document-level $\langle function \rangle$. The argument specification for the function is given by $\langle arg\ spec \rangle$, and expanding to be replaced by the $\langle code \rangle$.

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 L^AT_EX 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 $\langle function \rangle$ is already defined.

- `\DeclareDocumentCommand` will always create the new definition, irrespective of any existing $\langle function \rangle$ with the same name.
- `\NewDocumentCommand` will issue an error if $\langle function \rangle$ has already been defined.
- `\RenewDocumentCommand` will issue an error if $\langle function \rangle$ has not previously been defined.
- `\ProvideDocumentCommand` creates a new definition for $\langle function \rangle$ only if one has not already been given.

T_EXhackers note: Unlike L^AT_EX 2_ε's `\newcommand` and relatives, the `\DeclareDocumentCommand` function do not prevent creation of functions with names starting `\end...`

<code>\DeclareDocumentEnvironment</code>	<code>\DeclareDocumentEnvironment {⟨environment⟩} {⟨arg spec⟩}</code>
<code>\NewDocumentEnvironment</code>	<code>{⟨start code⟩} {⟨end code⟩}</code>
<code>\RenewDocumentEnvironment</code>	
<code>\ProvideDocumentEnvironment</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.

<code>\IfNoValueTF</code> ★	<code>\IfNoValueTF {⟨argument⟩} {⟨true code⟩} {⟨false code⟩}</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`.

<code>\IfValueTF</code> ★	<code>\IfValueTF {⟨argument⟩} {⟨true code⟩} {⟨false code⟩}</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.

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

<code>\IfBooleanTF</code> ★	<code>\IfBooleanTF <argument> {\true code} {\false code}</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.

<code>\ProcessedArgument</code>	
---------------------------------	--

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`.

<code>\ReverseBoolean</code>	<code>\ReverseBoolean</code>
------------------------------	------------------------------

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 T_EX 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*!

<code>\DeclareExpandableDocumentCommand</code>	<code>\DeclareExpandableDocumentCommand</code> <code>⟨function⟩ {⟨arg spec⟩} {⟨code⟩}</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.

<code>\GetDocumentCommandArgSpec</code>	<code>\GetDocumentCommandArgSpec ⟨function⟩</code>
<code>\GetDocumentEnvironmentArgSpec</code>	<code>\GetDocumentEnvironmentArgSpec ⟨environment⟩</code>

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.

<code>\ShowDocumentCommandArgSpec</code>	<code>\ShowDocumentCommandArgSpec ⟨function⟩</code>
<code>\ShowDocumentEnvironmentArgSpec</code>	<code>\ShowDocumentEnvironmentArgSpec ⟨environment⟩</code>

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

```
1 <*package>
2 <@@=xparse>
3 \ProvidesExplPackage
4   {\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.

```
5 \group_begin:
6 \char_set_lccode:nn { '\Q } { '\- }
7 \char_set_lccode:nn { '\N } { '\N }
8 \char_set_lccode:nn { '\V } { '\V }
9 \tl_to_lowercase:n
10 {
11   \group_end:
12   \tl_const:Nn \c__xparse_no_value_tl { QNoValue- }
13 }
```

(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.

```
14 \prop_new:N \c__xparse_shorthands_prop
15 \prop_put:Nnn \c__xparse_shorthands_prop { o } { d[] }
16 \prop_put:Nnn \c__xparse_shorthands_prop { 0 } { D[] }
17 \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.

```
18 <*initex>
19 \seq_new:N \c__xparse_special_chars_seq
20 \seq_set_split:Nnn \c__xparse_special_chars_seq { }
21   { \ \ \ \ { \} \# \^ \_ \% \~ }
22 </initex>
```

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

<code>\l__xparse_all_long_bool</code>	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. <code>23 \bool_new:N \l__xparse_all_long_bool</code> <i>(End definition for \l__xparse_all_long_bool This variable is documented on page ??.)</i>
<code>\l__xparse_args_tl</code>	Token list variable for grabbed arguments. <code>24 \tl_new:N \l__xparse_args_tl</code> <i>(End definition for \l__xparse_args_tl This variable is documented on page ??.)</i>
<code>\l__xparse_command_arg_specs_prop</code>	Used to record all document commands created, and the argument specifications that go with these. <code>25 \prop_new:N \l__xparse_command_arg_specs_prop</code> <i>(End definition for \l__xparse_command_arg_specs_prop This variable is documented on page ??.)</i>
<code>\l__xparse_current_arg_int</code>	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. <code>26 \int_new:N \l__xparse_current_arg_int</code> <i>(End definition for \l__xparse_current_arg_int This variable is documented on page ??.)</i>
<code>\l__xparse_environment_bool</code>	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. <code>27 \bool_new:N \l__xparse_environment_bool</code> <i>(End definition for \l__xparse_environment_bool This variable is documented on page ??.)</i>
<code>\l__xparse_environment_arg_specs_prop</code>	Used to record all document environment created, and the argument specifications that go with these. <code>28 \prop_new:N \l__xparse_environment_arg_specs_prop</code> <i>(End definition for \l__xparse_environment_arg_specs_prop This variable is documented on page ??.)</i>
<code>\l__xparse_expandable_bool</code>	Used to indicate if an expandable command is begin generated, as this affects both the acceptable argument types and how they are implemented. <code>29 \bool_new:N \l__xparse_expandable_bool</code> <i>(End definition for \l__xparse_expandable_bool This variable is documented on page ??.)</i>
<code>\l__xparse_expandable_aux_name_tl</code>	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. <code>30 \tl_new:N \l__xparse_expandable_aux_name_tl</code> <code>31 \tl_set:Nn \l__xparse_expandable_aux_name_tl</code> <code>32 {</code> <code>33 \l__xparse_function_tl \c_space_tl</code> <code>34 (arg~ \int_use:N \l__xparse_current_arg_int)</code> <code>35 }</code> <i>(End definition for \l__xparse_expandable_aux_name_tl This variable is documented on page ??.)</i>
<code>\l__xparse_fn_tl</code>	For passing the pre-formed name of the auxiliary to be used as the parsing function. <code>36 \tl_new:N \l__xparse_fn_tl</code>

(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`.

`37 \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.

`38 \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.

`39 \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.

`40 \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.

`41 \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.

`42 \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.

`43 \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.

`44 \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.

```

45 \cs_new_protected_nopar:Npn \__xparse_declare_cmd:Nnn
46   {
47     \bool_set_false:N \l__xparse_expandable_bool
48     \__xparse_declare_cmd_aux:Nnn
49   }
50 \cs_new_protected_nopar:Npn \__xparse_declare_expandable_cmd:Nnn
51   {
52     \bool_set_true:N \l__xparse_expandable_bool
53     \__xparse_declare_cmd_aux:Nnn
54   }

```

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.

```

55 \cs_new_protected:Npn \__xparse_declare_cmd_aux:Nnn #1#2
56   {
57     \cs_if_exist:NTF #1
58     {
59       \__msg_kernel_warning:nxxx { xparse } { redefine-command }
60       { \token_to_str:N #1 } { \tl_to_str:n {#2} }
61     }
62     {
63       \__msg_kernel_info:nxxx { xparse } { define-command }
64       { \token_to_str:N #1 } { \tl_to_str:n {#2} }
65     }
66     \prop_put:Nnn \l__xparse_command_arg_specs_prop {#1} {#2}
67     \bool_set_false:N \l__xparse_environment_bool
68     \__xparse_declare_cmd_internal:Nnn #1 {#2}
69   }

```

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

```

70 \cs_new_protected:Npn \__xparse_declare_cmd_internal:Nnn #1#2#3
71   {
72     \tl_set:Nx \l__xparse_function_tl { \cs_to_str:N #1 }
73     \__xparse_count_mandatory:n {#2}
74     \__xparse_prepare_signature:n {#2}
75     \int_compare:nNnTF \l__xparse_current_arg_int = \l__xparse_m_args_int
76     {
77       \bool_if:NTF \l__xparse_environment_bool
78       { \__xparse_declare_cmd_mixed:Nn #1 {#3} }
79       { \__xparse_declare_cmd_all_m:Nn #1 {#3} }
80     }
81     { \__xparse_declare_cmd_mixed:Nn #1 {#3} }
82     \__xparse_break_point:n {#2}
83   }
84 \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.

```
85 \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).

```
86 \cs_new_protected:Npn \_xparse_declare_cmd_all_m:Nn #1#2
87 {
88   \cs_generate_from_arg_count:Ncnn #1
89   {
90     cs_set
91     \bool_if:NF \l__xparse_expandable_bool { _protected }
92     \bool_if:NF \l__xparse_all_long_bool { _nopar }
93     :Npn
94   }
95   \l__xparse_current_arg_int {#2}
96 }
```

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

```
97 \cs_new_protected:Npn \_xparse_declare_cmd_mixed:Nn
98 {
99   \bool_if:NFT \l__xparse_expandable_bool
100   { \_xparse_declare_cmd_mixed_expandable:Nn }
101   { \_xparse_declare_cmd_mixed_aux:Nn }
102 }
```

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.

```
103 \cs_new_protected:Npn \_xparse_declare_cmd_mixed_aux:Nn #1#2
104 {
105   \_xparse_flush_m_args:
106   \cs_generate_from_arg_count:cNnn
107   { \l__xparse_function_tl \c_space_tl code }
108   \cs_set_protected:Npn \l__xparse_current_arg_int {#2}
109   \cs_set_protected_nopar:Npx #1
110   {
111     \int_zero:N \l__xparse_processor_int
112     \tl_set:Nn \exp_not:N \l__xparse_args_tl
113     { \exp_not:c { \l__xparse_function_tl \c_space_tl code } }
114     \tl_set:Nn \exp_not:N \l__xparse_fn_tl
115     { \exp_not:c { \l__xparse_function_tl \c_space_tl } }
116     \exp_not:o \l__xparse_signature_tl
117     \exp_not:N \l__xparse_args_tl
118   }
```

```

119 }
120 \cs_new_protected:Npn \__xparse_declare_cmd_mixed_expandable:Nn #1#2
121 {
122   \cs_generate_from_arg_count:cNnn
123   { \l__xparse_function_tl \c_space_tl code }
124   \cs_set:Npn \l__xparse_current_arg_int {#2}
125   \cs_set_nopar:Npx #1
126   {
127     \exp_not:o \l__xparse_signature_tl
128     \exp_not:N \__xparse_grab_expandable_end:wN
129     \exp_not:c { \l__xparse_function_tl \c_space_tl code }
130     \exp_not:N \q__xparse
131     \exp_not:c { \l__xparse_function_tl \c_space_tl }
132   }
133   \bool_if:NTF \l__xparse_all_long_bool
134   { \cs_set:cpx }
135   { \cs_set_nopar:cpx }
136   { \l__xparse_function_tl \c_space_tl } ##1##2 { ##1 {##2} }
137 }

```

(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.

```

138 \cs_new_protected:Npn \__xparse_declare_env:nnnn #1#2
139 {
140   <*initex>
141   \cs_if_exist:cTF { environment~ #1 }
142   </initex>
143   <*package>
144   \cs_if_exist:cTF {#1}
145   </package>
146   {
147     \__msg_kernel_warning:nnxx { xparse } { redefine-environment }
148     {#1} { \tl_to_str:n {#2} }
149   }
150   {
151     \__msg_kernel_info:nnxx { xparse } { define-environment }
152     {#1} { \tl_to_str:n {#2} }
153   }
154   \prop_put:Nnn \l__xparse_environment_arg_specs_prop {#1} {#2}
155   \bool_set_true:N \l__xparse_environment_bool
156   \__xparse_declare_env_internal:nnnn {#1} {#2}
157 }

```

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.

```

158 \cs_new_protected:Npn \__xparse_declare_env_internal:nnnn #1#2#3#4
159 {
160   \__xparse_declare_cmd_internal:cnx { environment~ #1 } {#2}
161   {
162     \cs_set_protected_nopar:Npx \exp_not:c { environment~ #1 ~end~aux }
163     {
164       \exp_not:c { environment~ #1~end~aux~ }
165       \exp_not:n { \tl_tail:N \l__xparse_args_tl }
166     }
167     \exp_not:n {#3}
168   }
169   \cs_set_protected_nopar:cpx { environment~ #1 ~end }
170   { \exp_not:c { environment~ #1 ~end~aux } }
171   \cs_generate_from_arg_count:cNnn
172   { environment~ #1 ~end~aux~ } \cs_set_protected:Npn
173   \l__xparse_current_arg_int {#4}
174   (*package)
175   \cs_set_eq:cc {#1} { environment~ #1 }
176   \cs_set_eq:cc { end #1 } { environment~ #1 ~end }
177   </package>
178   }

```

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

2.3 Counting mandatory arguments

__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.

```

179 \cs_new_protected:Npn \__xparse_count_mandatory:n #1
180 {
181   \int_zero:N \l__xparse_mandatory_args_int
182   \__xparse_count_mandatory:N #1
183   \q_recursion_tail \q_recursion_tail \q_recursion_tail \q_recursion_stop
184 }
185 \cs_new_protected:Npn \__xparse_count_mandatory:N #1
186 {
187   \quark_if_recursion_tail_stop:N #1
188   \prop_get:NnNTF \c__xparse_shorthands_prop {#1} \l__xparse_tmp_tl
189   { \exp_after:wN \__xparse_count_mandatory:N \l__xparse_tmp_tl }
190   { \__xparse_count_mandatory_aux:N #1 }
191 }
192 \cs_new_protected:Npn \__xparse_count_mandatory_aux:N #1
193 {
194   \cs_if_free:cTF { __xparse_count_type_ \token_to_str:N #1 :w }

```

```

195     { \__xparse_count_type_m:w }
196     { \use:c { __xparse_count_type_ \token_to_str:N #1 :w } }
197 }

```

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

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

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

```

198 \cs_new_protected:cpn { __xparse_count_type_>:w } #1
199 {
200   \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
201   \__xparse_count_mandatory:N
202 }
203 \cs_new_protected_nopar:cpn { __xparse_count_type_+:w }
204 { \__xparse_count_mandatory:N }
205 \cs_new_protected:Npn \__xparse_count_type_d:w #1#2
206 {
207   \__xparse_single_token_check:n {#1}
208   \__xparse_single_token_check:n {#2}
209   \quark_if_recursion_tail_stop_do:Nn #2 { \__xparse_bad_arg_spec:wn }
210   \__xparse_count_mandatory:N
211 }
212 \cs_new_protected:Npn \__xparse_count_type_D:w #1#2#3
213 {
214   \__xparse_single_token_check:n {#1}
215   \__xparse_single_token_check:n {#2}
216   \quark_if_recursion_tail_stop_do:nn {#3} { \__xparse_bad_arg_spec:wn }
217   \__xparse_count_mandatory:N
218 }
219 \cs_new_protected_nopar:Npn \__xparse_count_type_g:w
220 { \__xparse_count_mandatory:N }
221 \cs_new_protected:Npn \__xparse_count_type_G:w #1
222 {
223   \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
224   \__xparse_count_mandatory:N
225 }
226 \cs_new_protected_nopar:Npn \__xparse_count_type_m:w
227 {
228   \int_incr:N \l__xparse_mandatory_args_int
229   \__xparse_count_mandatory:N
230 }
231 \cs_new_protected:Npn \__xparse_count_type_r:w #1#2

```

```

232 {
233   \__xparse_single_token_check:n {#1}
234   \__xparse_single_token_check:n {#2}
235   \quark_if_recursion_tail_stop_do:Nn #2 { \__xparse_bad_arg_spec:wn }
236   \int_incr:N \l__xparse_mandatory_args_int
237   \__xparse_count_mandatory:N
238 }
239 \cs_new_protected:Npn \__xparse_count_type_R:w #1#2#3
240 {
241   \__xparse_single_token_check:n {#1}
242   \__xparse_single_token_check:n {#2}
243   \quark_if_recursion_tail_stop_do:nn {#3} { \__xparse_bad_arg_spec:wn }
244   \int_incr:N \l__xparse_mandatory_args_int
245   \__xparse_count_mandatory:N
246 }
247 \cs_new_protected:Npn \__xparse_count_type_t:w #1
248 {
249   \__xparse_single_token_check:n {#1}
250   \quark_if_recursion_tail_stop_do:Nn #1 { \__xparse_bad_arg_spec:wn }
251   \__xparse_count_mandatory:N
252 }
253 \cs_new_protected:Npn \__xparse_count_type_u:w #1
254 {
255   \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
256   \int_incr:N \l__xparse_mandatory_args_int
257   \__xparse_count_mandatory:N
258 }

```

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

__xparse_single_token_check:n A spin-out function to check that what should be single tokens really are single tokens.
 __xparse_single_token_check_aux:nwn

```

259 \cs_new_protected:Npn \__xparse_single_token_check:n #1
260 {
261   \tl_if_single:nF {#1}
262   { \__xparse_single_token_check_aux:nwn {#1} }
263 }
264 \cs_new_protected:Npn \__xparse_single_token_check_aux:nwn
265 #1#2 \__xparse_break_point:n #3
266 {
267   \__msg_kernel_error:nnx { xparse } { not-single-token }
268   { \tl_to_str:n {#1} } { \tl_to_str:n {#3} }
269 }

```

(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.

```

270 \cs_new_protected:Npn \__xparse_bad_arg_spec:wn #1 \__xparse_break_point:n #2
271 { \__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.

```

272 \cs_new_protected:Npn \__xparse_prepare_signature:n #1
273 {
274   \bool_set_false:N \l__xparse_all_long_bool
275   \int_zero:N \l__xparse_current_arg_int
276   \bool_set_false:N \l__xparse_long_bool
277   \int_zero:N \l__xparse_m_args_int
278   \bool_set_false:N \l__xparse_processor_bool
279   \tl_clear:N \l__xparse_signature_tl
280   \__xparse_prepare_signature:N #1 \q_recursion_tail \q_recursion_stop
281 }

```

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.

```

282 \cs_new_protected_nopar:Npn \__xparse_prepare_signature:N
283 {
284   \bool_set_false:N \l__xparse_processor_bool
285   \__xparse_prepare_signature_bypass:N
286 }
287 \cs_new_protected:Npn \__xparse_prepare_signature_bypass:N #1
288 {
289   \quark_if_recursion_tail_stop:N #1
290   \prop_get:NnNTF \c__xparse_shorthands_prop {#1} \l__xparse_tmp_tl
291   { \exp_after:wN \__xparse_prepare_signature:N \l__xparse_tmp_tl }
292   {
293     \int_incr:N \l__xparse_current_arg_int
294     \__xparse_prepare_signature_add:N #1
295   }
296 }

```

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.

```

297 \cs_new_protected:Npn \__xparse_prepare_signature_add:N #1
298 {
299   \cs_if_exist_use:cF
300   {
301     __xparse_add
302     \bool_if:NT \l__xparse_expandable_bool { _expandable }
303     _type_ \token_to_str:N #1 :w
304   }
305   {
306     \__msg_kernel_error:nxx { xparse } { unknown-argument-type }
307     { \token_to_str:N #1 }
308   }
309 }

```

```

308         \bool_if:NTF \l__xparse_expandable_bool
309         { \__xparse_add_expandable_type_m:w }
310         { \__xparse_add_type_m:w }
311     }
312 }

```

(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.

```

313 \cs_new_protected_nopar:cpn { \__xparse_add_type_+ :w }
314 {
315     \__xparse_flush_m_args:
316     \bool_set_true:N \l__xparse_long_bool
317     \int_decr:N \l__xparse_current_arg_int
318     \__xparse_prepare_signature:N
319 }

```

(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.

```

320 \cs_new_protected:cpn { \__xparse_add_type_> :w } #1
321 {
322     \bool_set_true:N \l__xparse_processor_bool
323     \__xparse_flush_m_args:
324     \int_decr:N \l__xparse_current_arg_int
325     \tl_put_right:Nn \l__xparse_signature_tl { \__xparse_process_arg:n {#1} }
326     \__xparse_prepare_signature_bypass:N
327 }

```

(End definition for __xparse_add_type_> :w This function is documented on page ??.)

`__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.

```

\__xparse_add_type_D:w
328 \cs_new_protected:Npn \__xparse_add_type_d:w #1#2
329 { \exp_args:NNNo \__xparse_add_type_D:w #1 #2 \c__xparse_no_value_tl }
330 \cs_new_protected:Npn \__xparse_add_type_D:w #1#2#3
331 {

```

```

332 \__xparse_flush_m_args:
333 \__xparse_add_grabber_optional:N D
334 \tl_put_right:Nn \l__xparse_signature_tl { #1 #2 {#3} }
335 \__xparse_prepare_signature:N
336 }

```

(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.

```

337 \cs_new_protected_nopar:Npn \__xparse_add_type_g:w
338 { \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.

```

339 \cs_new_protected:Npn \__xparse_add_type_G:w #1
340 {
341   \__xparse_flush_m_args:
342   \__xparse_add_grabber_optional:N G
343   \tl_put_right:Nn \l__xparse_signature_tl { {#1} }
344   \__xparse_prepare_signature:N
345 }

```

(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.

```

346 \cs_new_protected_nopar:Npn \__xparse_add_type_l:w
347 {
348   \__xparse_flush_m_args:
349   \__xparse_add_grabber_mandatory:N l
350   \__xparse_prepare_signature:N
351 }

```

(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:`.

```

352 \cs_new_protected_nopar:Npn \__xparse_add_type_m:w
353 {
354   \bool_if:nTF { \l__xparse_long_bool || \l__xparse_processor_bool }
355   {
356     \__xparse_flush_m_args:
357     \__xparse_add_grabber_mandatory:N m
358   }
359   { \int_incr:N \l__xparse_m_args_int }
360   \__xparse_prepare_signature:N
361 }

```

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

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

`__xparse_add_type_R:w`

```

362 \cs_new_protected:Npn \__xparse_add_type_r:w #1#2
363 { \exp_args:NNNo \__xparse_add_type_R:w #1 #2 \c__xparse_no_value_tl }
364 \cs_new_protected:Npn \__xparse_add_type_R:w #1#2#3
365 {
366   \__xparse_flush_m_args:
367   \__xparse_add_grabber_mandatory:N R
368   \tl_put_right:Nn \l__xparse_signature_tl { #1 #2 {#3} }
369   \__xparse_prepare_signature:N
370 }

```

(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.

```

371 \cs_new_protected:Npn \__xparse_add_type_t:w #1
372 {
373   \__xparse_flush_m_args:
374   \__xparse_add_grabber_optional:N t
375   \tl_put_right:Nn \l__xparse_signature_tl { #1 }
376   \__xparse_prepare_signature:N
377 }

```

(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.

```

378 \cs_new_protected:Npn \__xparse_add_type_u:w #1
379 {
380   \__xparse_flush_m_args:
381   \__xparse_add_grabber_mandatory:N u
382   \tl_put_right:Nn \l__xparse_signature_tl { {#1} }
383   \__xparse_prepare_signature:N
384 }

```

(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.

```

385 \cs_new_protected_nopar:Npn \__xparse_add_type_v:w
386 {
387   \__xparse_flush_m_args:
388   \__xparse_add_grabber_mandatory:N v
389   \__xparse_prepare_signature:N
390 }

```

(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.

```

391 \cs_new_protected_nopar:Npn \__xparse_flush_m_args:
392 {
393   \int_compare:nNnT \l__xparse_m_args_int > \c_zero
394   {
395     \tl_put_right:Nx \l__xparse_signature_tl
396       { \exp_not:c { __xparse_grab_m_ \int_use:N \l__xparse_m_args_int :w } }
397     \int_set:Nn \l__xparse_mandatory_args_int
398       { \l__xparse_mandatory_args_int - \l__xparse_m_args_int }
399   }
400   \int_zero:N \l__xparse_m_args_int
401 }

```

(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.

```

402 \cs_new_protected_nopar:Npn \__xparse_add_grabber_mandatory:N #1
403 {
404   \tl_put_right:Nx \l__xparse_signature_tl
405   {
406     \exp_not:c
407       { __xparse_grab_ #1 \bool_if:NT \l__xparse_long_bool { _long } :w }
408   }
409   \bool_set_false:N \l__xparse_long_bool
410   \int_decr:N \l__xparse_mandatory_args_int
411 }
412 \cs_new_protected_nopar:Npn \__xparse_add_grabber_optional:N #1
413 {
414   \tl_put_right:Nx \l__xparse_signature_tl
415   {
416     \exp_not:c
417       {
418         __xparse_grab_ #1
419         \bool_if:NT \l__xparse_long_bool { _long }
420         \int_compare:nNnF \l__xparse_mandatory_args_int > \c_zero
421         { _trailing }
422       :w
423     }
424   }
425   \bool_set_false:N \l__xparse_long_bool
426 }

```

(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.
427 \cs_new_protected_nopar:cpn { \_xparse_add_expandable_type_+ :w }
428 {
429   \bool_set_true:N \l__xparse_long_bool
430   \int_compare:nNnTF \l__xparse_current_arg_int = \c_one
431     { \bool_set_true:N \l__xparse_all_long_bool }
432     {
433       \bool_if:NF \l__xparse_all_long_bool
434       { \_msg_kernel_error:nn { xparse } { inconsistent-long } }
435     }
436   \int_decr:N \l__xparse_current_arg_int
437   \__xparse_prepare_signature:N
438 }
(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.
439 \cs_new_protected:cpn { \_xparse_add_expandable_type_> :w } #1
440 {
441   \_msg_kernel_error:nnx { xparse } { processor-in-expandable }
442   { \token_to_str:c { \l__xparse_function_tl } }
443   \int_decr:N \l__xparse_current_arg_int
444   \__xparse_prepare_signature:N
445 }
(End definition for \_xparse_add_expandable_type_> :w This function is documented on page ??.)
```

```
\_xparse_add_expandable_type_d:w The set up for d- and D-type arguments is the same, and involves constructing a rather
\_xparse_add_expandable_type_D:w complex auxiliary which is used repeatedly when grabbing. There is an auxiliary here so
\_xparse_add_expandable_type_D_aux:NNn that the R-type can share code readily.
\_xparse_add_expandable_type_D_aux:Nn
446 \cs_new_protected:Npn \_xparse_add_expandable_type_d:w #1#2
447 {
448   \exp_args:NNNo
449   \_xparse_add_expandable_type_D:w #1 #2 \c__xparse_no_value_tl
450 }
451 \cs_new_protected:Npn \_xparse_add_expandable_type_D:w #1#2
452 {
453   \token_if_eq_meaning:NNTF #1 #2
454   {
455     \_xparse_add_expandable_grabber_optional:n { D_alt }
456     \_xparse_add_expandable_type_D_aux:Nn #1
457   }
458   {
459     \_xparse_add_expandable_grabber_optional:n { D }
460     \_xparse_add_expandable_type_D_aux:NNn #1#2
461   }
462 }
463 \cs_new_protected:Npn \_xparse_add_expandable_type_D_aux:NNn #1#2#3
464 {
```

```

465 \bool_if:NTF \l__xparse_all_long_bool
466 { \cs_set:cpx }
467 { \cs_set_nopar:cpx }
468 { \l__xparse_expandable_aux_name_tl } ##1 ##2 #1 ##3 \q__xparse ##4 #2
469 { ##1 {##2} {##3} {##4} }
470 \tl_put_right:Nx \l__xparse_signature_tl
471 {
472   \exp_not:c { \l__xparse_expandable_aux_name_tl }
473   \exp_not:n { #1 #2 {#3} }
474 }
475 \bool_set_false:N \l__xparse_long_bool
476 \__xparse_prepare_signature:N
477 }

```

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.

```

478 \cs_new_protected:Npn \__xparse_add_expandable_type_D_aux:Nn #1#2
479 {
480   \bool_if:NTF \l__xparse_all_long_bool
481   { \cs_set:cpx }
482   { \cs_set_nopar:cpx }
483   { \l__xparse_expandable_aux_name_tl } ##1 #1 ##2 #1
484   { ##1 {##2} }
485   \tl_put_right:Nx \l__xparse_signature_tl
486   {
487     \exp_not:c { \l__xparse_expandable_aux_name_tl }
488     \exp_not:n { #1 {#2} }
489   }
490   \bool_set_false:N \l__xparse_long_bool
491   \__xparse_prepare_signature:N
492 }

```

(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.

```

493 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_g:w
494 {
495   \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { g }
496   \__xparse_add_expandable_type_m:w
497 }
498 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_G:w #1
499 {
500   \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { G }
501   \__xparse_add_expandable_type_m:w
502 }

```

(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).

```

503 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_l:w

```

```

504 {
505     \_msg_kernel_error:nxx { xparse } { invalid-expandable-argument-type } { 1 }
506     \_xparse_add_expandable_type_m:w
507 }

```

(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.

```

508 \cs_new_protected_nopar:Npn \_xparse_add_expandable_type_m:w
509 {
510     \int_incr:N \l__xparse_m_args_int
511     \_xparse_add_expandable_grabber_mandatory:n { m }
512     \bool_set_false:N \l__xparse_long_bool
513     \_xparse_prepare_signature:N
514 }

```

(End definition for _xparse_add_expandable_type_m:w This function is documented on page ??.)

_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.

```

515 \cs_new_protected:Npn \_xparse_add_expandable_type_r:w #1#2
516 {
517     \exp_args:NNNo
518     \_xparse_add_expandable_type_R:w #1 #2 \c__xparse_no_value_tl
519 }
520 \cs_new_protected:Npn \_xparse_add_expandable_type_R:w #1#2
521 {
522     \token_if_eq_meaning:NNTF #1 #2
523     {
524         \_xparse_add_expandable_grabber_optional:n { R_alt }
525         \_xparse_add_expandable_type_D_aux:Nn #1
526     }
527     {
528         \_xparse_add_expandable_grabber_optional:n { R }
529         \_xparse_add_expandable_type_D_aux:NNn #1#2
530     }
531 }

```

(End definition for _xparse_add_expandable_type_r:w This function is documented on page ??.)

_xparse_add_expandable_type_t:w

```

532 \cs_new_protected_nopar:Npn \_xparse_add_expandable_type_t:w #1
533 {
534     \_xparse_add_expandable_grabber_optional:n { t }
535     \bool_if:NNTF \l__xparse_all_long_bool
536     { \cs_set:cpn }
537     { \cs_set_nopar:cpn }
538     { \l__xparse_expandable_aux_name_tl } ##1 #1 {##1}
539     \tl_put_right:Nx \l__xparse_signature_tl
540     {

```

```

541         \exp_not:c { \l__xparse_expandable_aux_name_tl }
542         \exp_not:N #1
543     }
544     \bool_set_false:N \l__xparse_long_bool
545     \__xparse_prepare_signature:N
546 }

```

(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.

```

547 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_u:w #1
548 {
549     \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { u }
550     \__xparse_add_expandable_type_m:w
551 }

```

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

__xparse_add_expandable_type_v:w Another forbidden type.

```

552 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_v:w
553 {
554     \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { v }
555     \__xparse_add_expandable_type_m:w
556 }

```

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

__xparse_add_expandable_grabber_mandatory:n 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.

```

557 \cs_new_protected_nopar:Npn \__xparse_add_expandable_grabber_mandatory:n #1
558 {
559     \__xparse_add_expandable_long_check:
560     \tl_put_right:Nx \l__xparse_signature_tl
561         { \exp_not:c { __xparse_expandable_grab_ #1 :w } }
562     \bool_set_false:N \l__xparse_long_bool
563     \int_decr:N \l__xparse_mandatory_args_int
564 }
565 \cs_new_protected_nopar:Npn \__xparse_add_expandable_grabber_optional:n #1
566 {
567     \__xparse_add_expandable_long_check:
568     \int_compare:nNnF \l__xparse_mandatory_args_int > \c_zero
569         { \__msg_kernel_error:nn { xparse } { expandable-ending-optional } }
570     \tl_put_right:Nx \l__xparse_signature_tl
571         { \exp_not:c { __xparse_expandable_grab_ #1 :w } }
572     \bool_set_false:N \l__xparse_long_bool
573 }
574 \cs_new_protected_nopar:Npn \__xparse_add_expandable_long_check:
575 {
576     \bool_if:nT { \l__xparse_all_long_bool && ! ( \l__xparse_long_bool ) }
577         { \__msg_kernel_error:nn { xparse } { inconsistent-long } }

```

```
578 }
```

(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.

```
579 \cs_new_protected:Npn \__xparse_grab_arg:w { }
580 \cs_new_protected:Npn \__xparse_grab_arg_aux_i:w { }
581 \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.

```
582 \cs_new_protected:Npn \__xparse_grab_D:w #1#2#3#4 \l__xparse_args_tl
583 {
584   \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn
585     { _ignore_spaces }
586 }
587 \cs_new_protected:Npn \__xparse_grab_D_long:w #1#2#3#4 \l__xparse_args_tl
588 {
589   \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected:Npn
590     { _ignore_spaces }
591 }
592 \cs_new_protected:Npn \__xparse_grab_D_trailing:w #1#2#3#4 \l__xparse_args_tl
593 { \__xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn { } }
594 \cs_new_protected:Npn \__xparse_grab_D_long_trailing:w #1#2#3#4 \l__xparse_args_tl
595 { \__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.

```
596 \cs_new_protected:Npn \__xparse_grab_D_aux:NNnnNn #1#2#3#4#5#6
597 {
598   \__xparse_grab_D_aux:NNnN #1#2 {#4} #5
599   \use:c { peek_meaning_remove #6 :NTF } #1
```

```

600     { \__xparse_grab_arg:w }
601     {
602         \__xparse_add_arg:n {#3}
603         #4 \l__xparse_args_tl
604     }
605 }
606 \cs_new_protected:Npn \__xparse_grab_D_aux:NNnN #1#2#3#4
607 {
608     \cs_set_protected_nopar:Npn \__xparse_grab_arg:w
609     {
610         \exp_after:wN #4 \l__xparse_fn_tl #####1 #2
611         {
612             \tl_if_in:nnTF {#####1} {#1}
613             { \__xparse_grab_D_nested:NNnnN #1 #2 {#####1} {#3} #4 }
614             {
615                 \__xparse_add_arg:o { \use_none:n #####1 }
616                 #3 \l__xparse_args_tl
617             }
618         }
619     }

```

This section needs a little explanation. In order to avoid losing 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.)

```

619     \group_begin:
620     \token_if_eq_catcode:NNTF #1 ^
621     {
622         \char_set_lccode:nn { 'A } { '#1 }
623         \tl_to_lowercase:n
624         {
625             \group_end:
626             \l__xparse_fn_tl A
627         }
628     }
629     {
630         \char_set_lccode:nn { '^ } { '#1 }
631         \tl_to_lowercase:n
632         {
633             \group_end:
634             \l__xparse_fn_tl ^
635         }
636     }
637 }

```

```

638 }
(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.

```

639 \tl_new:N \l__xparse_nesting_a_tl
640 \tl_new:N \l__xparse_nesting_b_tl
641 \quark_new:N \q__xparse
642 \cs_new_protected:Npn \_xparse_grab_D_nested:NNnnN #1#2#3#4#5
643 {
644   \tl_clear:N \l__xparse_nesting_a_tl
645   \tl_clear:N \l__xparse_nesting_b_tl
646   \exp_after:wN #5 \l__xparse_fn_tl ##1 #1 ##2 \q__xparse ##3 #2
647   {
648     \tl_put_right:No \l__xparse_nesting_a_tl { \use_none:n ##1 #1 }
649     \tl_put_right:No \l__xparse_nesting_b_tl { \use_i:nn #2 ##3 }
650     \tl_if_in:nnTF {##2} {#1}
651     {
652       \l__xparse_fn_tl
653       \q_nil ##2 \q__xparse \ERROR
654     }
655     {
656       \tl_put_right:Nx \l__xparse_nesting_a_tl
657       { \_xparse_grab_D_nested:w \q_nil ##2 \q_stop }
658       \tl_if_in:NnTF \l__xparse_nesting_b_tl {#1}
659       {
660         \tl_set_eq:NN \l__xparse_tmp_tl \l__xparse_nesting_b_tl
661         \tl_clear:N \l__xparse_nesting_b_tl
662         \exp_after:wN \l__xparse_fn_tl \exp_after:wN
663         \q_nil \l__xparse_tmp_tl \q_nil \q__xparse \ERROR
664       }
665       {
666         \tl_put_right:No \l__xparse_nesting_a_tl
667         \l__xparse_nesting_b_tl
668         \_xparse_add_arg:V \l__xparse_nesting_a_tl
669         #4 \l__xparse_args_tl
670       }
671     }
672   }

```

```
673 \l__xparse_fn_tl #3 \q_nil \q__xparse \ERROR
```

```
674 }
```

```
675 \cs_new:Npn \__xparse_grab_D_nested:w #1 \q_nil \q_stop
```

```
676 { \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.

```
677 \cs_new_protected:Npn \__xparse_grab_G:w #1#2 \l__xparse_args_tl
```

```
678 {
```

```
679 \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected_nopar:Npn
```

```
680 { _ignore_spaces }
```

```
681 }
```

```
682 \cs_new_protected:Npn \__xparse_grab_G_long:w #1#2 \l__xparse_args_tl
```

```
683 {
```

```
684 \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected:Npn { _ignore_spaces }
```

```
685 }
```

```
686 \cs_new_protected:Npn \__xparse_grab_G_trailing:w #1#2 \l__xparse_args_tl
```

```
687 { \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected_nopar:Npn { } }
```

```
688 \cs_new_protected:Npn \__xparse_grab_G_long_trailing:w #1#2 \l__xparse_args_tl
```

```
689 { \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected:Npn { } }
```

```
690 \cs_new_protected:Npn \__xparse_grab_G_aux:nnNn #1#2#3#4
```

```
691 {
```

```
692 \exp_after:wN #3 \l__xparse_fn_tl ##1
```

```
693 {
```

```
694 \__xparse_add_arg:n {##1}
```

```
695 #2 \l__xparse_args_tl
```

```
696 }
```

```
697 \use:c { peek_meaning #4 :NTF } \c_group_begin_token
```

```
698 { \l__xparse_fn_tl }
```

```
699 {
```

```
700 \__xparse_add_arg:n {#1}
```

```
701 #2 \l__xparse_args_tl
```

```
702 }
```

```
703 }
```

(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.

```
704 \cs_new_protected:Npn \__xparse_grab_l:w #1 \l__xparse_args_tl
```

```
705 { \__xparse_grab_l_aux:nN {#1} \cs_set_protected_nopar:Npn }
```

```
706 \cs_new_protected:Npn \__xparse_grab_l_long:w #1 \l__xparse_args_tl
```

```
707 { \__xparse_grab_l_aux:nN {#1} \cs_set_protected:Npn }
```

```
708 \cs_new_protected:Npn \__xparse_grab_l_aux:nN #1#2
```

```
709 {
```

```
710 \exp_after:wN #2 \l__xparse_fn_tl ##1##
```

```
711 {
```

```
712 \__xparse_add_arg:n {##1}
```

```
713 #1 \l__xparse_args_tl
```

```
714 }
```



```

715     \l__xparse_fn_tl
716   }
(End definition for \__xparse_grab_1:w This function is documented on page ??.)

```

__xparse_grab_m:w Collecting a single mandatory argument is quite easy.
 __xparse_grab_m_long:w

```

717 \cs_new_protected:Npn \__xparse_grab_m:w #1 \l__xparse_args_tl
718 {
719   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl ##1
720   {
721     \__xparse_add_arg:n {##1}
722     #1 \l__xparse_args_tl
723   }
724   \l__xparse_fn_tl
725 }
726 \cs_new_protected:Npn \__xparse_grab_m_long:w #1 \l__xparse_args_tl
727 {
728   \exp_after:wN \cs_set_protected:Npn \l__xparse_fn_tl ##1
729   {
730     \__xparse_add_arg:n {##1}
731     #1 \l__xparse_args_tl
732   }
733   \l__xparse_fn_tl
734 }

```

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

__xparse_grab_m_1:w Grabbing 1–8 mandatory arguments. We don't need to worry about nine arguments as
 __xparse_grab_m_2:w this is only possible if everything is mandatory. Each function has an auxiliary so that
 __xparse_grab_m_3:w \par tokens from other arguments still work.

```

735 \cs_new_protected:cpn { __xparse_grab_m_1:w } #1 \l__xparse_args_tl
736 {
737   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl ##1
738   {
739     \tl_put_right:Nn \l__xparse_args_tl { {##1} }
740     #1 \l__xparse_args_tl
741   }
742   \l__xparse_fn_tl
743 }
744 \cs_new_protected:cpn { __xparse_grab_m_2:w } #1 \l__xparse_args_tl
745 {
746   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
747   ##1##2
748   {
749     \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} }
750     #1 \l__xparse_args_tl
751   }
752   \l__xparse_fn_tl
753 }
754 \cs_new_protected:cpn { __xparse_grab_m_3:w } #1 \l__xparse_args_tl
755 {

```

```

756 \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
757   ##1##2##3
758   {
759     \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} }
760     #1 \l__xparse_args_tl
761   }
762 \l__xparse_fn_tl
763 }
764 \cs_new_protected:cpn { __xparse_grab_m_4:w } #1 \l__xparse_args_tl
765 {
766   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
767     ##1##2##3##4
768     {
769       \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} {##4} }
770       #1 \l__xparse_args_tl
771     }
772   \l__xparse_fn_tl
773 }
774 \cs_new_protected:cpn { __xparse_grab_m_5:w } #1 \l__xparse_args_tl
775 {
776   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
777     ##1##2##3##4##5
778     {
779       \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} {##4} {##5} }
780       #1 \l__xparse_args_tl
781     }
782   \l__xparse_fn_tl
783 }
784 \cs_new_protected:cpn { __xparse_grab_m_6:w } #1 \l__xparse_args_tl
785 {
786   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
787     ##1##2##3##4##5##6
788     {
789       \tl_put_right:Nn \l__xparse_args_tl
790       { {##1} {##2} {##3} {##4} {##5} {##6} }
791       #1 \l__xparse_args_tl
792     }
793   \l__xparse_fn_tl
794 }
795 \cs_new_protected:cpn { __xparse_grab_m_7:w } #1 \l__xparse_args_tl
796 {
797   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
798     ##1##2##3##4##5##6##7
799     {
800       \tl_put_right:Nn \l__xparse_args_tl
801       { {##1} {##2} {##3} {##4} {##5} {##6} {##7} }
802       #1 \l__xparse_args_tl
803     }
804   \l__xparse_fn_tl
805 }

```

```

806 \cs_new_protected:cpn { __xparse_grab_m:8:w } #1 \l__xparse_args_tl
807 {
808   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
809     ##1##2##3##4##5##6##7##8
810   {
811     \tl_put_right:Nn \l__xparse_args_tl
812       { {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} }
813     #1 \l__xparse_args_tl
814   }
815   \l__xparse_fn_tl
816 }

```

(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.

```

817 \cs_new_protected:Npn \__xparse_grab_R:w #1#2#3#4 \l__xparse_args_tl
818 { \__xparse_grab_R_aux:NNnnN #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn }
819 \cs_new_protected:Npn \__xparse_grab_R_long:w #1#2#3#4 \l__xparse_args_tl
820 { \__xparse_grab_R_aux:NNnnN #1 #2 {#3} {#4} \cs_set_protected:Npn }
821 \cs_new_protected:Npn \__xparse_grab_R_aux:NNnnN #1#2#3#4#5
822 {
823   \__xparse_grab_D_aux:NNnn #1 #2 {#4} #5
824   \peek_meaning_remove_ignore_spaces:NTF #1
825     { \__xparse_grab_arg:w }
826     {
827       \msg_kernel_error:nnxx { xparse } { missing-required }
828       { \token_to_str:N #1 } { \tl_to_str:n {#3} }
829       \__xparse_add_arg:n {#3}
830       #4 \l__xparse_args_tl
831     }
832 }

```

(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.

```

833 \cs_new_protected:Npn \__xparse_grab_t:w #1#2 \l__xparse_args_tl
834 {
835   \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected_nopar:Npn
836     { _ignore_spaces }
837 }
838 \cs_new_protected:Npn \__xparse_grab_t_long:w #1#2 \l__xparse_args_tl
839 { \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected:Npn { _ignore_spaces } }
840 \cs_new_protected:Npn \__xparse_grab_t_trailing:w #1#2 \l__xparse_args_tl
841 { \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected_nopar:Npn { } }
842 \cs_new_protected:Npn \__xparse_grab_t_long_trailing:w #1#2 \l__xparse_args_tl
843 { \__xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected:Npn { } }
844 \cs_new_protected:Npn \__xparse_grab_t_aux:NnNn #1#2#3#4
845 {

```

```

846 \exp_after:wN #3 \l__xparse_fn_tl
847 {
848   \use:c { peek_meaning_remove #4 :NTF } #1
849   {
850     \__xparse_add_arg:n { \BooleanTrue }
851     #2 \l__xparse_args_tl
852   }
853   {
854     \__xparse_add_arg:n { \BooleanFalse }
855     #2 \l__xparse_args_tl
856   }
857 }
858 \l__xparse_fn_tl
859 }

```

(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.

```

860 \cs_new_protected:Npn \__xparse_grab_u:w #1#2 \l__xparse_args_tl
861 { \__xparse_grab_u_aux:nnN {#1} {#2} \cs_set_protected_nopar:Npn }
862 \cs_new_protected:Npn \__xparse_grab_u_long:w #1#2 \l__xparse_args_tl
863 { \__xparse_grab_u_aux:nnN {#1} {#2} \cs_set_protected:Npn }
864 \cs_new_protected:Npn \__xparse_grab_u_aux:nnN #1#2#3
865 {
866   \exp_after:wN #3 \l__xparse_fn_tl ##1 #1
867   {
868     \__xparse_add_arg:n {##1}
869     #2 \l__xparse_args_tl
870   }
871   \l__xparse_fn_tl
872 }

```

(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.

```

873 \tl_new:N \l__xparse_v_rest_of_signature_tl
874 \tl_new:N \l__xparse_v_arg_tl
875 \cs_new_protected_nopar:Npn \__xparse_grab_v:w
876 {
877   \bool_set_false:N \l__xparse_long_bool
878   \__xparse_grab_v_aux:w
879 }

```

```

880 \cs_new_protected_nopar:Npn \__xparse_grab_v_long:w
881 {
882   \bool_set_true:N \l__xparse_long_bool
883   \__xparse_grab_v_aux:w
884 }
885 \cs_new_protected:Npn \__xparse_grab_v_aux:w #1 \l__xparse_args_tl
886 {
887   \tl_set:Nn \l__xparse_v_rest_of_signature_tl {#1}
888   \group_begin:
889     \group_align_safe_begin:
890     \tex_escapechar:D = 92 \scan_stop:
891     \tl_clear:N \l__xparse_v_arg_tl
892     \peek_N_type:TF
893     { \__xparse_grab_v_aux_test:N }
894     {
895       \peek_meaning_remove:NTF \c_group_begin_token
896       {
897         \group_align_safe_end:
898         \__xparse_grab_v_bgroup:
899       }
900       { \__xparse_grab_v_aux_abort: }
901     }
902 }
903 \cs_new_protected_nopar:Npn \__xparse_grab_v_group_end:
904 {
905   \group_align_safe_end:
906   \exp_args:NNNo
907   \group_end:
908   \tl_set:Nn \l__xparse_v_arg_tl { \l__xparse_v_arg_tl }
909 }

```

(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.

```

910 \cs_new_protected:Npn \__xparse_grab_v_aux_test:N #1
911 {
912   \__xparse_grab_v_aux_put:N #1
913   \__xparse_grab_v_token_if_char:NTF #1
914   {
915     \__xparse_grab_v_aux_catcodes:

```

```

916         \__xparse_grab_v_aux_loop:N #1
917     }
918     { \__xparse_grab_v_aux_abort: }
919 }
920 \cs_new_protected:Npn \__xparse_grab_v_aux_loop:N #1
921 {
922     \peek_N_type:TF
923     { \__xparse_grab_v_aux_loop_ii:NN #1 }
924     { \__xparse_grab_v_aux_abort: }
925 }
926 \cs_new_protected:Npn \__xparse_grab_v_aux_loop_ii:NN #1 #2
927 {
928     \__xparse_grab_v_token_if_char:NTF #2
929     {
930         \token_if_eq_charcode:NNTF #1 #2
931         { \__xparse_grab_v_aux_loop_end: }
932         {
933             \__xparse_grab_v_aux_put:N #2
934             \__xparse_grab_v_aux_loop:N #1
935         }
936     }
937     { \__xparse_grab_v_aux_abort: #2 }
938 }
939 \cs_new_protected_nopar:Npn \__xparse_grab_v_aux_loop_end:
940 {
941     \__xparse_grab_v_group_end:
942     \exp_args:Nx \__xparse_add_arg:n { \tl_tail:N \l__xparse_v_arg_tl }
943     \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
944 }

```

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

__xparse_grab_v_bgroup: 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.

__xparse_grab_v_bgroup_loop: 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.

```

945 \int_new:N \l__xparse_v_nesting_int
946 \cs_new_protected_nopar:Npx \__xparse_grab_v_bgroup:
947 {
948     \exp_not:N \__xparse_grab_v_aux_catcodes:
949     \exp_not:n { \int_set_eq:NN \l__xparse_v_nesting_int \c_one }
950     \exp_not:N \__xparse_grab_v_aux_put:N \iow_char:N \{
951     \exp_not:N \__xparse_grab_v_bgroup_loop:
952 }
953 \cs_new_protected:Npn \__xparse_grab_v_bgroup_loop:
954 {
955     \peek_N_type:TF

```

```

956     { \_xparse_grab_v_bgroup_loop_ii:N }
957     { \_xparse_grab_v_aux_abort: }
958   }
959 \cs_new_protected:Npn \_xparse_grab_v_bgroup_loop_ii:N #1
960 {
961   \_xparse_grab_v_token_if_char:NNTF #1
962   {
963     \token_if_eq_charcode:NNTF \c_group_end_token #1
964     {
965       \int_decr:N \l__xparse_v_nesting_int
966       \int_compare:nNnTF \l__xparse_v_nesting_int > \c_zero
967       {
968         \_xparse_grab_v_aux_put:N #1
969         \_xparse_grab_v_bgroup_loop:
970       }
971       { \_xparse_grab_v_aux_loop_end: }
972     }
973     {
974       \token_if_eq_charcode:NNT \c_group_begin_token #1
975       { \int_incr:N \l__xparse_v_nesting_int }
976       \_xparse_grab_v_aux_put:N #1
977       \_xparse_grab_v_bgroup_loop:
978     }
979   }
980   { \_xparse_grab_v_aux_abort: #1 }
981 }

```

(End definition for _xparse_grab_v_bgroup: This function is documented on page ??.)

_xparse_grab_v_aux_catcodes: 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
_xparse_grab_v_aux_abort: verbatim arguments is to make the end-line character a macro parameter character: this
_xparse_grab_v_aux_abort_ii:w 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.

```

982 \cs_new_protected_nopar:Npn \_xparse_grab_v_aux_catcodes:
983 {
984   (*initex)
985   \seq_map_function:NN
986   \c__xparse_special_chars_seq
987   \char_set_catcode_other:N
988   (/initex)
989   (*package)
990   \cs_set_eq:NN \do \char_set_catcode_other:N
991   \dospecials
992   (/package)
993   \tex_endlinechar:D = ‘^M \scan_stop:
994   \bool_if:NNTF \l__xparse_long_bool
995   { \char_set_catcode_other:n { \tex_endlinechar:D } }
996   { \char_set_catcode_parameter:n { \tex_endlinechar:D } }
997 }

```

```

998 \cs_new_protected_nopar:Npn \__xparse_grab_v_aux_abort:
999 {
1000   \__xparse_grab_v_group_end:
1001   \__xparse_add_arg:o \c__xparse_no_value_tl
1002   \exp_after:wN \__xparse_grab_v_aux_abort_ii:w \l__xparse_args_tl \q_stop
1003 }
1004 \cs_new_protected:Npn \__xparse_grab_v_aux_abort_ii:w #1 #2 \q_stop
1005 {
1006   \group_begin:
1007   \char_set_lccode:nn { '\# } { \tex_endlinechar:D }
1008   \tl_to_lowercase:n
1009   { \group_end: \peek_meaning_remove:NTF ## }
1010   {
1011     \__msg_kernel_error:nxx { xparse } { verbatim-newline }
1012     { \token_to_str:N #1 }
1013     { \tl_to_str:N \l__xparse_v_arg_tl }
1014     \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
1015   }
1016   {
1017     \__msg_kernel_error:nxx { xparse } { verbatim-already-tokenized }
1018     { \token_to_str:N #1 }
1019     { \tl_to_str:N \l__xparse_v_arg_tl }
1020     \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
1021   }
1022 }

```

(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).

```

1023 \cs_new_protected:Npn \__xparse_grab_v_aux_put:N #1
1024 {
1025   \tl_put_right:Nx \l__xparse_v_arg_tl
1026   {
1027     \token_if_active:NTF #1
1028     { \exp_not:N #1 } { \token_to_str:N #1 }
1029   }
1030 }

```

(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.

```

1031 \cs_new_protected:Npn \__xparse_grab_v_token_if_char:NTF #1
1032 { \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.

```

1033 \cs_new_protected:Npn \__xparse_add_arg:n #1
1034 {
1035   \int_compare:nNnTF \l__xparse_processor_int = \c_zero
1036     { \tl_put_right:Nn \l__xparse_args_tl { {#1} } }
1037     {
1038       \tl_clear:N \ProcessedArgument
1039       \__xparse_if_no_value:nTF {#1}
1040       {
1041         \int_zero:N \l__xparse_processor_int
1042         \tl_put_right:Nn \l__xparse_args_tl { {#1} }
1043       }
1044       { \__xparse_add_arg_aux:n {#1} }
1045     }
1046 }
1047 \cs_generate_variant:Nn \__xparse_add_arg:n { V , o }
1048 \cs_new_protected:Npn \__xparse_add_arg_aux:n #1
1049 {
1050   \use:c { __xparse_processor_ \int_use:N \l__xparse_processor_int :n } {#1}
1051   \int_decr:N \l__xparse_processor_int
1052   \int_compare:nNnTF \l__xparse_processor_int = \c_zero
1053   {
1054     \tl_put_right:Nx \l__xparse_args_tl
1055       { { \exp_not:V \ProcessedArgument } }
1056   }
1057   { \__xparse_add_arg_aux:V \ProcessedArgument }
1058 }
1059 \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.

```

1060 \cs_new:Npn \__xparse_expandable_grab_D:w #1 \q__xparse #2
1061 { #2 { \__xparse_expandable_grab_D_i:NNNnwN #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

¹It 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>*.

```

1062 \cs_new:Npn \__xparse_expandable_grab_D_i:NNNwNn #1#2#3#4#5 \q__xparse #6#7
1063 {
1064   \str_if_eq:onTF
1065     { #1 { } { } #7 #2 \q__xparse #3 }
1066     { { } { #2 } { } }
1067     {
1068       #1
1069       { \__xparse_expandable_grab_D_ii:NNwNnnn #1#2#3#5 \q__xparse #6 }
1070       \q_nil { } #2 \ERROR \q__xparse \ERROR
1071     }
1072     { #5 {#4} \q__xparse #6 {#7} }
1073 }

```

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.

```

1074 \cs_new:Npn \__xparse_expandable_grab_D_ii:NNwNnnn #1#2#3#4 \q__xparse #5#6#7#8
1075 {
1076   \exp_args:Nof \__xparse_expandable_grab_D_iv:nnNNwN
1077   { \use_ii:nn #6 #2 }
1078   { \__xparse_expandable_grab_D_iii:Nw #3 \exp_stop_f: #7 #8 }
1079   #1#2#3 #4 \q__xparse #5
1080 }
1081 \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`.

```

1082 \cs_new:Npn \__xparse_expandable_grab_D_iv:nnNNwN #1#2#3#4#5#6 \q__xparse #7
1083 {
1084   \exp_args:No \tl_if_empty:oTF
1085   { #3 { \use_none:nnn } #2 \q__xparse #5 #4 \q__xparse #5 }
1086   {
1087     \__xparse_put_arg_expandable:ow { \use_none:nn #1#2 }
1088     #6 \q__xparse #7
1089   }
1090   {
1091     #3
1092     { \__xparse_expandable_grab_D_ii:NNNwNnnn #3#4#5#6 \q__xparse #7 }
1093     \q_nil {#1} #2 \ERROR \q__xparse \ERROR
1094   }
1095 }

```

(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 possible and a simplified approach is used. The test concept here is the same as for the case where the delimiters are different.

```

1096 \cs_new:Npn \__xparse_expandable_grab_D_alt:w #1 \q__xparse #2
1097 { #2 { \__xparse_expandable_grab_D_alt_i:NNnwNn #1 \q__xparse #2 } }
1098 \cs_new:Npn \__xparse_expandable_grab_D_alt_i:NNnwNn #1#2#3#4 \q__xparse #5#6
1099 {
1100   \str_if_eq:onTF
1101   { #1 { } #6 #2 #2 }
1102   { { } #2 }
1103   {
1104     #1
1105     { \__xparse_expandable_grab_D_alt_ii:Nwn #5 #4 \q__xparse }
1106     #6 \ERROR
1107   }
1108   { #4 {#3} \q__xparse #5 {#6} }
1109 }
1110 \cs_new:Npn \__xparse_expandable_grab_D_alt_ii:Nwn #1#2 \q__xparse #3
1111 { \__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.

```

1112 \cs_new:Npn \__xparse_expandable_grab_m:w #1 \q__xparse #2
1113 { #2 { \__xparse_expandable_grab_m_aux:wNn #1 \q__xparse #2 } }
1114 \cs_new:Npn \__xparse_expandable_grab_m_aux:wNn #1 \q__xparse #2#3
1115 { #1 {#3} \q__xparse #2 }

```

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

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

```

1116 \cs_new:Npn \_xparse_expandable_grab_R:w #1 \q__xparse #2
1117   { #2 { \_xparse_expandable_grab_R_aux:NNnwNn #1 \q__xparse #2 } }
1118 \cs_new:Npn \_xparse_expandable_grab_R_aux:NNnwNn #1#2#3#4#5 \q__xparse #6#7
1119   {
1120     \str_if_eq:onTF
1121     { #1 { } { } #7 #2 \q__xparse #3 }
1122     { { } { #2 } { } }
1123     {
1124       #1
1125       { \_xparse_expandable_grab_D_ii:NNwNnnn #1#2#3#5 \q__xparse #6 }
1126       \q_nil { } #2 \ERROR \q__xparse \ERROR
1127     }
1128     {
1129       \_msg_kernel_expandable_error:nnn
1130       { xparse } { missing-required } {#2}
1131       #5 {#4} \q__xparse #6 {#7}
1132     }
1133   }

```

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

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

```

1134 \cs_new:Npn \_xparse_expandable_grab_R_alt:w #1 \q__xparse #2
1135   { #2 { \_xparse_expandable_grab_R_alt_aux:NNnwNn #1 \q__xparse #2 } }
1136 \cs_new:Npn \_xparse_expandable_grab_R_alt_aux:NNnwNn #1#2#3#4 \q__xparse #5#6
1137   {
1138     \str_if_eq:onTF
1139     { #1 { } #6 #2 #2 }
1140     { { } #2 }
1141     {
1142       #1
1143       { \_xparse_expandable_grab_D_alt_ii:Nwn #5 #4 \q__xparse }
1144       #6 \ERROR
1145     }
1146     {
1147       \_msg_kernel_expandable_error:nnn
1148       { xparse } { missing-required } {#2}
1149       #4 {#3} \q__xparse #5 {#6}
1150     }
1151   }

```

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

`_xparse_expandable_grab_t:w` 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.

```

1152 \cs_new:Npn \_xparse_expandable_grab_t:w #1 \q__xparse #2
1153   { #2 { \_xparse_expandable_grab_t_aux:NNwn #1 \q__xparse #2 } }
1154 \cs_new:Npn \_xparse_expandable_grab_t_aux:NNwn #1#2#3 \q__xparse #4#5
1155   {

```

```

1156 \str_if_eq:onTF { #1 { } #5 #2 } { #2 }
1157 { #3 { \BooleanTrue } \q__xparse #4 }
1158 { #3 { \BooleanFalse } \q__xparse #4 {#5} }
1159 }

```

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

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

```

\__xparse_put_arg_expandable:ow 1160 \cs_new:Npn \__xparse_put_arg_expandable:nw #1#2 \q__xparse { #2 {#1} \q__xparse }
1161 \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.

```

1162 \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.

```

1163 \cs_new_protected:Npn \__xparse_process_arg:n #1
1164 {
1165 \int_incr:N \l__xparse_processor_int
1166 \cs_set:cpn { __xparse_processor_ \int_use:N \l__xparse_processor_int :n } ##1
1167 { #1 {##1} }
1168 }

```

(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.

```

1169 \cs_new_protected:Npn \__xparse_process_to_str:n #1
1170 { \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.

```

1171 \cs_new_protected:Npn \__xparse_bool_reverse:N #1
1172 {
1173 \bool_if:NTF #1
1174 { \tl_set:Nn \ProcessedArgument { \c_false_bool } }
1175 { \tl_set:Nn \ProcessedArgument { \c_true_bool } }
1176 }

```

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

\l__xparse_split_list_seq 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.

```

\l__xparse_split_list_tl 1177 \seq_new:N \l__xparse_split_list_seq
\__xparse_split_list:nn 1178 \tl_new:N \l__xparse_split_list_tl
\__xparse_split_list_multi:nn 1179 \cs_new_protected:Npn \__xparse_split_list:nn #1#2
\__xparse_split_list_multi:nV 1180 {
\__xparse_split_list_single:Nn

```

```

1181 \bool_if:nTF
1182 {
1183     \tl_if_single_p:n {#1} &&
1184     ! ( \token_if_cs_p:N #1 )
1185 }
1186 { \__xparse_split_list_single:Nn #1 {#2} }
1187 { \__xparse_split_list_multi:nn {#1} {#2} }
1188 }
1189 \cs_set_protected:Npn \__xparse_split_list_multi:nn #1#2
1190 {
1191     \seq_set_split:Nnn \l__xparse_split_list_seq {#1} {#2}
1192     \tl_clear:N \ProcessedArgument
1193     \seq_map_inline:Nn \l__xparse_split_list_seq
1194     { \tl_put_right:Nn \ProcessedArgument { {##1} } }
1195 }
1196 \cs_generate_variant:Nn \__xparse_split_list_multi:nn { nV }
1197 \group_begin:
1198 \char_set_catcode_active:N \@
1199 \cs_new_protected:Npn \__xparse_split_list_single:Nn #1#2
1200 {
1201     \tl_set:Nn \l__xparse_split_list_tl {#2}
1202     \group_begin:
1203     \char_set_lccode:nn { \@ } { '#1 }
1204     \tl_to_lowercase:n
1205     {
1206         \group_end:
1207         \tl_replace_all:Nnn \l__xparse_split_list_tl { @ } {#1}
1208     }
1209     \__xparse_split_list_multi:nV {#1} \l__xparse_split_list_tl
1210 }
1211 \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 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.

```

1212 \cs_new_protected:Npn \__xparse_split_argument:nnn #1#2#3
1213 {
1214     \__xparse_split_list:nn {#2} {#3}
1215     \exp_args:Nf \__xparse_split_argument_aux:nnnn
1216     { \tl_count:N \ProcessedArgument }
1217     {#1} {#2} {#3}
1218 }
1219 \cs_new_protected:Npn \__xparse_split_argument_aux:nnnn #1#2#3#4
1220 {
1221     \int_compare:nNnF {#1} = { #2 + \c_one }
1222     {

```

```

1223 \int_compare:nNnTF {#1} > { #2 + \c_one }
1224 {
1225   \tl_set:Nx \ProcessedArgument
1226   {
1227     \exp_last_unbraced:NnNo
1228     \__xparse_split_argument_aux:n
1229     { #2 + \c_one }
1230     \use_none_delimit_by_q_stop:w
1231     \ProcessedArgument
1232     \q_stop
1233   }
1234   \__msg_kernel_error:nxxxx { xparse } { split-excess-tokens }
1235   { \tl_to_str:n {#3} } { \int_eval:n { #2 + \c_one } }
1236   { \tl_to_str:n {#4} }
1237 }
1238 {
1239   \tl_put_right:Nx \ProcessedArgument
1240   {
1241     \prg_replicate:nn { #2 + \c_one - (#1) }
1242     { { \exp_not:V \c__xparse_no_value_tl } }
1243   }
1244 }
1245 }
1246 }

```

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

```

1247 \cs_new:Npn \__xparse_split_argument_aux:n #1
1248 { \prg_replicate:nn {#1} { \__xparse_split_argument_aux:wn } }
1249 \cs_new:Npn \__xparse_split_argument_aux:wn #1 \use_none_delimit_by_q_stop:w #2
1250 {
1251   \exp_not:n { {#2} }
1252   #1
1253   \use_none_delimit_by_q_stop:w
1254 }

```

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

__xparse_trim_spaces:n This one is almost trivial.

```

1255 \cs_new_protected:Npn \__xparse_trim_spaces:n #1
1256 { \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 Recovering the argument specification is also trivial, using the \tl_set_eq:cN function.

```

\__xparse_get_arg_spec:n
\ArgumentSpecification
1257 \cs_new_protected:Npn \__xparse_get_arg_spec:N #1
1258 {
1259   \prop_get:NnNF \l__xparse_command_arg_specs_prop {#1}
1260   \ArgumentSpecification
1261   {

```

```

1262     \_msg_kernel_error:nxx { xparse } { unknown-document-command }
1263     { \token_to_str:N #1 }
1264   }
1265 }
1266 \cs_new_protected:Npn \_xparse_get_arg_spec:n #1
1267 {
1268   \prop_get:NnNF \l__xparse_environment_arg_specs_prop {#1}
1269   \ArgumentSpecification
1270   {
1271     \_msg_kernel_error:nxx { xparse } { unknown-document-environment }
1272     { \tl_to_str:n {#1} }
1273   }
1274 }
1275 \tl_new:N \ArgumentSpecification

```

(End definition for _xparse_get_arg_spec:N This function is documented on page ??.)

_xparse_show_arg_spec:N Showing the argument specification simply means finding it and then calling the \tl_show:N function.

```

1276 \cs_new_protected:Npn \_xparse_show_arg_spec:N #1
1277 {
1278   \prop_get:NnNTF \l__xparse_command_arg_specs_prop {#1}
1279   \ArgumentSpecification
1280   { \tl_show:N \ArgumentSpecification }
1281   {
1282     \_msg_kernel_error:nxx { xparse } { unknown-document-command }
1283     { \token_to_str:N #1 }
1284   }
1285 }
1286 \cs_new_protected:Npn \_xparse_show_arg_spec:n #1
1287 {
1288   \prop_get:NnNTF \l__xparse_environment_arg_specs_prop {#1}
1289   \ArgumentSpecification
1290   { \tl_show:N \ArgumentSpecification }
1291   {
1292     \_msg_kernel_error:nxx { xparse } { unknown-document-environment }
1293     { \tl_to_str:n {#1} }
1294   }
1295 }

```

(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.

```

1296 \group_begin:
1297 \char_set_lccode:nn { '\Q } { '\- }
1298 \char_set_lccode:nn { '\F } { '\F }
1299 \char_set_lccode:nn { '\N } { '\N }
1300 \char_set_lccode:nn { '\T } { '\T }

```



```

1301 \char_set_lccode:nn { '\V } { '\V }
1302 \tl_to_lowercase:n
1303 {
1304   \group_end:
1305   \prg_new_conditional:Npnn \__xparse_if_no_value:n #1 { T , F , TF }
1306   {
1307     \str_if_eq:onTF
1308     { \__xparse_if_value_aux:w ? #1 { } QNoValue- }
1309     { ? { } QNoValue- }
1310     { \prg_return_true: }
1311     { \prg_return_false: }
1312   }
1313   \cs_new:Npn \__xparse_if_value_aux:w #1 QNoValue- { #1 }
1314 }

```

(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.

```

1315 \__msg_kernel_new:nnnn { xparse } { bad-arg-spec }
1316 { Bad~argument~specification~'#1'. }
1317 {
1318   \c_msg_coding_error_text_tl
1319   The~argument~specification~provided~was~not~valid:~
1320   one~or~more~mandatory~pieces~of~information~were~missing. \\\
1321   LaTeX~will~ignore~this~entire~definition.
1322 }
1323 \__msg_kernel_new:nnnn { xparse } { command-already-defined }
1324 { Command~'#1'~already~defined! }
1325 {
1326   You~have~used~\NewDocumentCommand
1327   with~a~command~that~already~has~a~definition. \\\
1328   The~existing~definition~of~'#1'~will~be~overwritten.
1329 }
1330 \__msg_kernel_new:nnnn { xparse } { command-not-yet-defined }
1331 { Command ~'#1'~not~yet~defined! }
1332 {
1333   You~have~used~\RenewDocumentCommand
1334   with~a~command~that~was~never~defined.\\
1335   A~new~command~'#1'~will~be~created.
1336 }
1337 \__msg_kernel_new:nnnn { xparse } { environment-already-defined }
1338 { Environment~'#1'~already~defined! }
1339 {
1340   You~have~used~\NewDocumentEnvironment
1341   with~an~environment~that~already~has~a~definition.\\
1342   The~existing~definition~of~'#1'~will~be~overwritten.

```

```

1343 }
1344 \_msg_kernel_new:nnnn { xparse } { environment-mismatch }
1345 { Mismatch~between~start~and~end~of~environment. }
1346 {
1347   The~current~environment~is~called~'#1',~but~you~have~tried~to~
1348   end~one~called~'#2'.~Environments~have~to~be~properly~nested.
1349 }
1350 \_msg_kernel_new:nnnn { xparse } { environment-not-yet-defined }
1351 { Environment~'#1'~not~yet~defined! }
1352 {
1353   You~have~used~\RenewDocumentEnvironment
1354   with~an~environment~that~was~never~defined.\\
1355   A~new~environment~'#1'~will~be~created.
1356 }
1357 \_msg_kernel_new:nnnn { xparse } { environment-unknown }
1358 { Environment~'#1'~undefined. }
1359 {
1360   You~have~tried~to~start~an~environment~called~'#1',~
1361   but~this~has~never~been~defined.\\
1362   The~command~will~be~ignored.
1363 }
1364 \_msg_kernel_new:nnnn { xparse } { expandable-ending-optional }
1365 { Argument~specification~for~expandable~command~ends~with~optional~argument. }
1366 {
1367   \c_msg_coding_error_text_tl
1368   Expandable~commands~must~have~a~final~mandatory~argument~
1369   (or~no~arguments~at~all).~You~cannot~have~a~terminal~optional~
1370   argument~with~expandable~commands.
1371 }
1372 \_msg_kernel_new:nnnn { xparse } { inconsistent-long }
1373 { Inconsistent~long~arguments~for~expandable~command. }
1374 {
1375   \c_msg_coding_error_text_tl
1376   The~arguments~for~an~expandable~command~must~either~all~be~
1377   short~or~all~be~long.~You~have~tried~to~mix~the~two~types.
1378 }
1379 \_msg_kernel_new:nnnn { xparse } { invalid-expandable-argument-type }
1380 { Argument~type~'#1'~not~available~for~an~expandable~function. }
1381 {
1382   \c_msg_coding_error_text_tl
1383   The~letter~'#1'~does~not~specify~an~argument~type~which~can~be~used~
1384   in~an~expandable~function.
1385   \\ \\
1386   LaTeX~will~assume~you~want~a~standard~mandatory~argument~(type~'m').
1387 }
1388 \_msg_kernel_new:nnnn { xparse } { missing-required }
1389 { Failed~to~find~required~argument~starting~with~'#1'. }
1390 {
1391   There~is~supposed~to~be~an~argument~to~the~current~function~starting~with~
1392   '#1'.~LaTeX~did~not~find~it,~and~will~insert~'#2'~as~the~value~to~be~

```

```

1393     processed.
1394 }
1395 \_msg_kernel_new:nnnn { xparse } { not-single-token }
1396 { Argument~delimiter~should~be~a~single~token::~#1'. }
1397 {
1398   \c_msg_coding_error_text_tl
1399   The~argument~specification~provided~was~not~valid::~
1400   in~a~place~where~a~single~token~is~required,,~LaTeX~found::~#1'. \\ \\
1401   LaTeX~will~ignore~this~entire~definition.
1402 }
1403 \_msg_kernel_new:nnnn { xparse } { processor-in-expandable }
1404 { Argument~processors~cannot~be~used~with~expandable~functions. }
1405 {
1406   \c_msg_coding_error_text_tl
1407   The~argument~specification~for~#1~contains~a~processor~function::~
1408   this~is~only~supported~for~standard~robust~functions.
1409 }
1410 \_msg_kernel_new:nnnn { xparse } { split-excess-tokens }
1411 { Too~many~'~#1'~tokens~when~trying~to~split~argument. }
1412 {
1413   LaTeX~was~asked~to~split~the~input~'#3'~
1414   at~each~occurrence~of~the~token~'#1',~up~to~a~maximum~of~#2~parts::~
1415   There~were~too~many~'#1'~tokens.
1416 }
1417 \_msg_kernel_new:nnnn { xparse } { unknown-argument-type }
1418 { Unknown~argument~type~'#1'~replaced~by~'m'. }
1419 {
1420   \c_msg_coding_error_text_tl
1421   The~letter~'#1'~does~not~specify~a~known~argument~type::~
1422   LaTeX~will~assume~you~want~a~standard~mandatory~argument~(type~'m').
1423 }
1424 \_msg_kernel_new:nnnn { xparse } { unknown-document-command }
1425 { Unknown~document~command~'#1'. }
1426 {
1427   You~have~asked~for~the~argument~specification~for~a~command~'#1',~
1428   but~this~is~not~a~document~command.
1429 }
1430 \_msg_kernel_new:nnnn { xparse } { unknown-document-environment }
1431 { Unknown~document~environment~'#1'. }
1432 {
1433   You~have~asked~for~the~argument~specification~for~a~command~'#1',~
1434   but~this~is~not~a~document~environment.
1435 }
1436 \_msg_kernel_new:nnnn { xparse } { verbatim-newline }
1437 { Verbatim~argument~of~#1~ended~by~end~of~line. }
1438 {
1439   The~verbatim~argument~of~#1~cannot~contain~more~than~one~line,,~but~the~end~
1440   of~the~current~line~has~been~reached::~You~have~probably~forgotten~the~
1441   closing~delimiter.
1442   \\ \\

```

```

1443 LaTeX~will~ignored~'~#2'.
1444 }
1445 \_msg_kernel_new:nnnn { xparse } { verbatim-already-tokenized }
1446 { Verbatim~command~#1~illegal~in~command~argument. }
1447 {
1448 The~command~#1~takes~a~verbatim~argument.~It~may~not~appear~within~
1449 the~argument~of~another~function.
1450 \\\ \\\
1451 LaTeX~will~ignore~'~#2'.
1452 }

```

Intended more for information.

```

1453 \_msg_kernel_new:nnn { xparse } { define-command }
1454 {
1455 Defining~document~command~#1~
1456 with~arg.~spec.~'~#2'~\msg_line_context:.
1457 }
1458 \_msg_kernel_new:nnn { xparse } { define-environment }
1459 {
1460 Defining~document~environment~'~#1'~
1461 with~arg.~spec.~'~#2'~\msg_line_context:.
1462 }
1463 \_msg_kernel_new:nnn { xparse } { redefine-command }
1464 {
1465 Redefining~document~command~#1~
1466 with~arg.~spec.~'~#2'~\msg_line_context:.
1467 }
1468 \_msg_kernel_new:nnn { xparse } { redefine-environment }
1469 {
1470 Redefining~document~environment~'~#1'~
1471 with~arg.~spec.~'~#2'~\msg_line_context:.
1472 }

```

2.14 User functions

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

\BooleanFalse Design-space names for the Boolean values.

\BooleanTrue

```

1473 \cs_new_eq:NN \BooleanFalse \c_false_bool
1474 \cs_new_eq:NN \BooleanTrue \c_true_bool

```

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

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

```

\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand
1475 \cs_new_protected:Npn \DeclareDocumentCommand #1#2#3
1476 { \_xparse_declare_cmd:Nnn #1 {#2} {#3} }
1477 \cs_new_protected:Npn \NewDocumentCommand #1#2#3
1478 {
1479 \cs_if_exist:NTF #1
1480 {

```

```

1481     \_msg_kernel_error:nxx { xparse } { command-already-defined }
1482     { \token_to_str:N #1 }
1483   }
1484   { \_xparse_declare_cmd:Nnn #1 {#2} {#3} }
1485 }
1486 \cs_new_protected:Npn \RenewDocumentCommand #1#2#3
1487 {
1488   \cs_if_exist:NTF #1
1489   { \_xparse_declare_cmd:Nnn #1 {#2} {#3} }
1490   {
1491     \_msg_kernel_error:nxx { xparse } { command-not-yet-defined }
1492     { \token_to_str:N #1 }
1493   }
1494 }
1495 \cs_new_protected:Npn \ProvideDocumentCommand #1#2#3
1496 { \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.

```

1497 \cs_new_protected:Npn \DeclareDocumentEnvironment #1#2#3#4
1498 { \_xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
1499 \cs_new_protected:Npn \NewDocumentEnvironment #1#2#3#4
1500 {
1501   \cs_if_exist:cTF {#1}
1502   { \_msg_kernel_error:nxx { xparse } { environment-already-defined } {#1} }
1503   { \_xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
1504 }
1505 \cs_new_protected:Npn \RenewDocumentEnvironment #1#2#3#4
1506 {
1507   \cs_if_exist:cTF {#1}
1508   { \_xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
1509   { \_msg_kernel_error:nxx { xparse } { environment-not-yet-defined } {#1} }
1510 }
1511 \cs_new_protected:Npn \ProvideDocumentEnvironment #1#2#3#4
1512 { \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.

```

1513 \cs_new_protected:Npn \DeclareExpandableDocumentCommand #1#2#3
1514 { \_xparse_declare_expandable_cmd:Nnn #1 {#2} {#3} }

```

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

`\IfBooleanTF`

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`.

```

1515 \cs_new_eq:NN \IfBooleanTF \bool_if:NTF
1516 \cs_new_eq:NN \IfBooleanT \bool_if:NT
1517 \cs_new_eq:NN \IfBooleanF \bool_if:NF

```

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

\IfNoValueTF Simple re-naming.

```
1518 \cs_new_eq:NN \IfNoValueF  \__xparse_if_no_value:nF
1519 \cs_new_eq:NN \IfNoValueT  \__xparse_if_no_value:nT
1520 \cs_new_eq:NN \IfNoValueTF \__xparse_if_no_value:nTF
(End definition for \IfNoValueTF This function is documented on page 6.)
```

\IfValueTF Inverted logic.

```
1521 \cs_set:Npn \IfValueF { \__xparse_if_no_value:nT }
1522 \cs_set:Npn \IfValueT { \__xparse_if_no_value:nF }
1523 \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.

```
1524 \tl_new:N \ProcessedArgument
(End definition for \ProcessedArgument This function is documented on page 7.)
```

\ReverseBoolean Simple copies.

```
\SplitArgument 1525 \cs_new_eq:NN \ReverseBoolean \__xparse_bool_reverse:N
\SplitList      1526 \cs_new_eq:NN \SplitArgument  \__xparse_split_argument:nnn
\TrimSpaces     1527 \cs_new_eq:NN \SplitList      \__xparse_split_list:nn
                1528 \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.

```
1529 \cs_new_eq:NN \ProcessList \tl_map_function:nN
(End definition for \ProcessList This function is documented on page 8.)
```

\GetDocumentCommandArgSpec More simple mappings.

```
\GetDocumentEnvironmentArgSpec 1530 \cs_new_eq:NN \GetDocumentCommandArgSpec \__xparse_get_arg_spec:N
\ShowDocumentCommandArgSpec    1531 \cs_new_eq:NN \GetDocumentEnvironmentArgSpec \__xparse_get_arg_spec:n
\ShowDocumentEnvironmentArgSpec 1532 \cs_new_eq:NN \ShowDocumentCommandArgSpec \__xparse_show_arg_spec:N
                                1533 \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.

```
1534 \DeclareOption { log-declarations = true } { }
1535 \DeclareOption { log-declarations = false }
1536 {
1537   \msg_redirect_module:nnn { LaTeX / xparse } { info } { none }
1538   \msg_redirect_module:nnn { LaTeX / xparse } { warning } { none }
1539 }
1540 \DeclareOption { log-declarations } { }
1541 \ProcessOptions \scan_stop:
1542 \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.

Symbols	
\#	21, 1007
\%	21
\-	6, 1297
\@	1198, 1203
\\	21, 1320, 1327, 1334, 1341, 1354, 1361, 1385, 1400, 1442, 1450
\{	21, 950
\}	21
\^	21, 993
_	21
_msg_kernel_error:nn	434, 569, 577
_msg_kernel_error:nnx	267, 271, 306, 441, 495, 500, 505, 549, 554, 1262, 1271, 1282, 1292, 1481, 1491, 1502, 1509
_msg_kernel_error:nnxx	827, 1011, 1017
_msg_kernel_error:nnxxx	1234
_msg_kernel_expandable_error:nnn	1129, 1147
_msg_kernel_info:nnxx	63, 151
_msg_kernel_new:nnn	1453, 1458, 1463, 1468
_msg_kernel_new:nnnn	1315, 1323, 1330, 1337, 1344, 1350, 1357, 1364, 1372, 1379, 1388, 1395, 1403, 1410, 1417, 1424, 1430, 1436, 1445
_msg_kernel_warning:nnxx	59, 147
_xparse_add_arg:V	668, 1033
_xparse_add_arg:n	602, 694, 700, 712, 721, 730, 829, 850, 854, 868, 942, 1033, 1033, 1047
_xparse_add_arg:o	615, 1001, 1033
_xparse_add_arg_aux:V	1033, 1057
_xparse_add_arg_aux:n	1033, 1044, 1048, 1059
_xparse_add_expandable_grabber_mandatory	511, 557, 557
_xparse_add_expandable_grabber_optional	455, 459, 524, 528, 534, 557, 565
_xparse_add_expandable_long_check	557, 559, 567, 574
_xparse_add_expandable_type_+:w	427
_xparse_add_expandable_type_>:w	439
_xparse_add_expandable_type_D:w	446, 449, 451
_xparse_add_expandable_type_D_aux:NNn	446, 460, 463, 529
_xparse_add_expandable_type_D_aux:Nn	446, 456, 478, 525
_xparse_add_expandable_type_G:w	493, 498
_xparse_add_expandable_type_R:w	515, 518, 520
_xparse_add_expandable_type_d:w	446, 446
_xparse_add_expandable_type_g:w	493, 493
_xparse_add_expandable_type_l:w	503, 503
_xparse_add_expandable_type_m:w	309, 496, 501, 506, 508, 508, 550, 555
_xparse_add_expandable_type_r:w	515, 515
_xparse_add_expandable_type_t:w	532, 532
_xparse_add_expandable_type_u:w	547, 547
_xparse_add_expandable_type_v:w	552, 552
_xparse_add_grabber_mandatory:N	349, 357, 367, 381, 388, 402, 402
_xparse_add_grabber_optional:N	333, 342, 374, 402, 412
_xparse_add_type_+:w	313
_xparse_add_type_>:w	320
_xparse_add_type_D:w	328, 329, 330
_xparse_add_type_G:w	338, 339, 339
_xparse_add_type_R:w	362, 363, 364
_xparse_add_type_d:w	328, 328
_xparse_add_type_g:w	337, 337
_xparse_add_type_l:w	346, 346
_xparse_add_type_m:w	310, 352, 352
_xparse_add_type_r:w	362, 362
_xparse_add_type_t:w	371, 371
_xparse_add_type_u:w	378, 378
_xparse_add_type_v:w	385, 385

__xparse_bad_arg_spec:wn . . .	200, 209, 216, 223, 235, 243, 250, 255, 270, 270	__xparse_expandable_grab_D_alt_ii:Nwn	1105, 1110, 1143
__xparse_bool_reverse:N	1171, 1171, 1525	__xparse_expandable_grab_D_i:NNNnwN	1060
__xparse_break_point:n	82, 85, 85, 265, 270	__xparse_expandable_grab_D_i:NNNnwNn	1061, 1062
__xparse_count_mandatory:N . . .	179, 179, 182, 185, 189, 201, 204, 210, 217, 220, 224, 229, 237, 245, 251, 257	__xparse_expandable_grab_D_ii:NNNwNnnn	1060, 1069, 1074, 1092, 1125
__xparse_count_mandatory:n	73, 179, 179	__xparse_expandable_grab_D_iii:Nw .	1060, 1078, 1081
__xparse_count_mandatory_aux:N	190, 192	__xparse_expandable_grab_D_iv:nnNNwN	1060, 1076, 1082
__xparse_count_type_+:w	198	__xparse_expandable_grab_R:w	1116, 1116
__xparse_count_type_>:w	198	__xparse_expandable_grab_R_alt:w . .	1134, 1134
__xparse_count_type_D:w	198, 212	__xparse_expandable_grab_R_alt_aux:NNnwNn	1134, 1135, 1136
__xparse_count_type_G:w	198, 221	__xparse_expandable_grab_R_aux:NNNwNn	1117, 1118
__xparse_count_type_R:w	239	__xparse_expandable_grab_R_aux:NNwn	1116
__xparse_count_type_d:w	198, 205	__xparse_expandable_grab_m:w	1112, 1112
__xparse_count_type_g:w	198, 219	__xparse_expandable_grab_m_aux:wNn	1112, 1113, 1114
__xparse_count_type_m:w . .	195, 198, 226	__xparse_expandable_grab_t:w	1152, 1152
__xparse_count_type_r:w	231	__xparse_expandable_grab_t_aux:NNwn	1152, 1153, 1154
__xparse_count_type_t:w	198, 247	__xparse_flush_m_args:	105, 315, 323, 332, 341, 348, 356, 366, 373, 380, 387, 391, 391
__xparse_count_type_u:w	198, 253	__xparse_get_arg_spec:N	1257, 1257, 1530
__xparse_declare_cmd:Nnn	45, 45, 1476, 1484, 1489, 1496	__xparse_get_arg_spec:n	1257, 1266, 1531
__xparse_declare_cmd_all_m:Nn	79, 86, 86	__xparse_grab_D:w	582, 582
__xparse_declare_cmd_aux:Nnn	45, 48, 53, 55	__xparse_grab_D_aux:NNnN	596, 598, 606, 823
__xparse_declare_cmd_internal:Nnn	45, 68, 70, 84	__xparse_grab_D_aux:NNnnNn	584, 589, 593, 595, 596, 596
__xparse_declare_cmd_internal:cnx	45, 160	__xparse_grab_D_long:w	582, 587
__xparse_declare_cmd_mixed:Nn	78, 81, 86, 97	__xparse_grab_D_long_trailing:w	582, 594
__xparse_declare_cmd_mixed_aux:Nn	86, 101, 103	__xparse_grab_D_nested:NNnnN	613, 639, 642
__xparse_declare_cmd_mixed_expandable:Nn	86, 100, 120	__xparse_grab_D_nested:w	639, 657, 675
__xparse_declare_env:nnnn	138, 138, 1498, 1503, 1508, 1512	__xparse_grab_D_trailing:w	582, 592
__xparse_declare_env_internal:nnnn	138, 156, 158	__xparse_grab_G:w	677, 677
__xparse_declare_expandable_cmd:Nnn	45, 50, 1514	__xparse_grab_G_aux:nnNn	677, 679, 684, 687, 689, 690
__xparse_expandable_grab_D:w	1060, 1060	__xparse_grab_G_long:w	677, 682
__xparse_expandable_grab_D_alt:w . .	1096, 1096	__xparse_grab_G_long_trailing:w	677, 688
__xparse_expandable_grab_D_alt_i:NNnwNn	1096, 1097, 1098		
__xparse_expandable_grab_D_alt_ii:Nw	1096		

_xparse_grab_G_trailing:w ...	677, 686	_xparse_grab_v_aux_test:N	893, 910, 910
_xparse_grab_R:w	817, 817	_xparse_grab_v_bgroup: ..	898, 945, 946
_xparse_grab_R_aux:NNnnN		_xparse_grab_v_bgroup_loop:	
.....	817, 818, 820, 821	945, 951, 953, 969, 977
_xparse_grab_R_long:w	817, 819	_xparse_grab_v_bgroup_loop_ii:N ..	
_xparse_grab_arg:w 579, 579, 600, 608, 825		945, 956, 959
_xparse_grab_arg_aux_i:w	579, 580	_xparse_grab_v_group_end:	
_xparse_grab_arg_aux_ii:w ...	579, 581	873, 903, 941, 1000
_xparse_grab_expandable_end:wN ...		_xparse_grab_v_long:w	873, 880
.....	128, 1162, 1162	_xparse_grab_v_token_if_char:NTF .	
_xparse_grab_l:w	704, 704	913, 928, 961, 1031, 1031
_xparse_grab_l_aux:nN 704, 705, 707, 708		_xparse_if_no_value:n	1305
_xparse_grab_l_long:w	704, 706	_xparse_if_no_value:nF	1518, 1522
_xparse_grab_m:w	717, 717	_xparse_if_no_value:nT	1519, 1521
_xparse_grab_m_1:w	735	_xparse_if_no_value:nTF	
_xparse_grab_m_2:w	735	1039, 1296, 1520, 1523
_xparse_grab_m_3:w	735	_xparse_if_value_aux:w	1308, 1313
_xparse_grab_m_4:w	735	_xparse_prepare_signature:N	
_xparse_grab_m_5:w	735	272, 280, 282, 291,
_xparse_grab_m_6:w	735		318, 335, 344, 350, 360, 369, 376,
_xparse_grab_m_7:w	735		383, 389, 437, 444, 476, 491, 513, 545
_xparse_grab_m_8:w	735	_xparse_prepare_signature:n	
_xparse_grab_m_long:w	717, 726	74, 272, 272
_xparse_grab_t:w	833, 833	_xparse_prepare_signature_add:N ..	
_xparse_grab_t_aux:NnNn	272, 294, 297
.....	833, 835, 839, 841, 843, 844	_xparse_prepare_signature_bypass:N	
_xparse_grab_t_long:w	833, 838	272, 285, 287, 326
_xparse_grab_t_long_trailing:w ...		_xparse_process_arg:n .	325, 1163, 1163
.....	833, 842	_xparse_process_to_str:n ..	1169, 1169
_xparse_grab_t_trailing:w ...	833, 840	_xparse_put_arg_expandable:nw	
_xparse_grab_u:w	860, 860	1160, 1160, 1161
_xparse_grab_u_aux:nnN		_xparse_put_arg_expandable:ow	
.....	860, 861, 863, 864	1087, 1111, 1160
_xparse_grab_u_long:w	860, 862	_xparse_show_arg_spec:N 1276, 1276, 1532	
_xparse_grab_v:w	873, 875	_xparse_show_arg_spec:n 1276, 1286, 1533	
_xparse_grab_v_aux:w 873, 878, 883, 885		_xparse_single_token_check:n	
_xparse_grab_v_aux_abort:	207, 208, 214,
900, 918, 924, 937, 957, 980, 982, 998			215, 233, 234, 241, 242, 249, 259, 259
_xparse_grab_v_aux_abort_ii:w		_xparse_single_token_check_aux:nnn	
.....	982, 1002, 1004	259, 262, 264
_xparse_grab_v_aux_catcodes:		_xparse_split_argument:nnn	
.....	915, 948, 982, 982	1212, 1212, 1526
_xparse_grab_v_aux_loop:N		_xparse_split_argument_aux:n	
.....	910, 916, 920, 934	1212, 1228, 1247
_xparse_grab_v_aux_loop_end:		_xparse_split_argument_aux:nnnn ..	
.....	910, 931, 939, 971	1212, 1215, 1219
_xparse_grab_v_aux_loop_ii:NN		_xparse_split_argument_aux:wn	
.....	910, 923, 926	1212, 1248, 1249
_xparse_grab_v_aux_put:N		_xparse_split_list:nn	
..	912, 933, 950, 968, 976, 1023, 1023	1177, 1179, 1214, 1527

_xparse_split_list_multi:nV	1177, 1209	\char_set_catcode_other:n	995
_xparse_split_list_multi:nn	1177, 1187, 1189, 1196	\char_set_catcode_parameter:n	996
_xparse_split_list_single:Nn	1177, 1186, 1199	\char_set_lccode:nn	6, 7, 8, 622, 630, 1007, 1203, 1297, 1298, 1299, 1300, 1301
_xparse_trim_spaces:n	1255, 1255, 1528	\cs_generate_from_arg_count:cNnn	106, 122, 171
\~	21	\cs_generate_from_arg_count:Ncnn	88
		\cs_generate_variant:Nn	84, 1047, 1059, 1161, 1196
_	21	\cs_if_exist:cF	1512
		\cs_if_exist:cTF	141, 144, 1501, 1507
A		\cs_if_exist:NF	1496
\ArgumentSpecification	1257, 1260, 1269, 1275, 1279, 1280, 1289, 1290	\cs_if_exist:NTF	57, 1479, 1488
B		\cs_if_exist_use:cF	299
\bool_if:NF	91, 92, 433, 1517	\cs_if_free:cTF	194
\bool_if:NT	302, 407, 419, 1516	\cs_new:Npn	675, 1060, 1062, 1074, 1081, 1082, 1096, 1098, 1110, 1112, 1114, 1116, 1118, 1134, 1136, 1152, 1154, 1160, 1162, 1247, 1249, 1313
\bool_if:nT	576	\cs_new_eq:NN	85, 1473, 1474, 1515, 1516, 1517, 1518, 1519, 1520, 1525, 1526, 1527, 1528, 1529, 1530, 1531, 1532, 1533
\bool_if:NTF	77, 99, 133, 308, 465, 480, 535, 994, 1173, 1515	\cs_new_protected:cpn	198, 320, 439, 735, 744, 754, 764, 774, 784, 795, 806
\bool_if:nTF	354, 1181	\cs_new_protected:Npn	55, 70, 86, 97, 103, 120, 138, 158, 179, 185, 192, 205, 212, 221, 231, 239, 247, 253, 259, 264, 270, 272, 287, 297, 328, 330, 339, 362, 364, 371, 378, 446, 451, 463, 478, 515, 520, 579, 580, 581, 582, 587, 592, 594, 596, 606, 642, 677, 682, 686, 688, 690, 704, 706, 708, 717, 726, 817, 819, 821, 833, 838, 840, 842, 844, 860, 862, 864, 885, 910, 920, 926, 953, 959, 1004, 1023, 1031, 1033, 1048, 1163, 1169, 1171, 1179, 1199, 1212, 1219, 1255, 1257, 1266, 1276, 1286, 1475, 1477, 1486, 1495, 1497, 1499, 1505, 1511, 1513
\bool_new:N	23, 27, 29, 38, 41	\cs_new_protected_nopar:cpn	203, 313, 427
\bool_set_false:N	47, 67, 274, 276, 278, 284, 409, 425, 475, 490, 512, 544, 562, 572, 877	\cs_new_protected_nopar:Npn	45, 50, 219, 226, 282, 337, 346, 352, 385, 391, 402, 412, 493, 498, 503, 508, 532, 547, 552, 557, 565, 574, 875, 880, 903, 939, 982, 998
\bool_set_true:N	52, 155, 316, 322, 429, 431, 882	\cs_new_protected_nopar:Npx	946
\BooleanFalse	6, 854, 1158, 1473, 1473	\cs_set:cpn	536, 1166
\BooleanTrue	6, 850, 1157, 1473, 1474		
C			
\c_xparse_no_value_tl	5, 12, 329, 338, 363, 449, 518, 1001, 1242		
\c_xparse_shorthands_prop	14, 14, 15, 16, 17, 188, 290		
\c_xparse_special_chars_seq	18, 19, 20, 986		
\c_false_bool	1174, 1473		
\c_group_begin_token	697, 895, 974		
\c_group_end_token	963		
\c_msg_coding_error_text_tl	1318, 1367, 1375, 1382, 1398, 1406, 1420		
\c_one	430, 949, 1221, 1223, 1229, 1235, 1241		
\c_space_tl	33, 107, 113, 115, 123, 129, 131, 136		
\c_true_bool	1175, 1474		
\c_zero	393, 420, 568, 966, 1035, 1052		
\char_set_catcode_active:N	1198		
\char_set_catcode_other:N	987, 990		

\cs_set:cpx	134, 466, 481	\ExplFileName	4
\cs_set:Npn	124, 1521, 1522, 1523	\ExplFileVersion	4
\cs_set_eq:cc	175, 176		
\cs_set_eq:NN	990	F	
\cs_set_nopar:cpn	537	\F	1298
\cs_set_nopar:cpx	135, 467, 482		
\cs_set_nopar:Npx	125	G	
\cs_set_protected:Npn		\GetDocumentCommandArgSpec	10, 1530, 1530
..... 108, 172, 589, 595, 684,		\GetDocumentEnvironmentArgSpec	10, 1530
689, 707, 728, 820, 839, 843, 863, 1189		\GetDocumentEnvironmmmentArgSpec ..	1531
\cs_set_protected_nopar:cpx	169	\group_align_safe_begin:	889
\cs_set_protected_nopar:Npn		\group_align_safe_end:	897, 905
..... 584, 593, 608, 679,		\group_begin:	
687, 705, 719, 737, 746, 756, 766,		.. 5, 619, 888, 1006, 1197, 1202, 1296	
776, 786, 797, 808, 818, 835, 841, 861		\group_end:	11,
\cs_set_protected_nopar:Npx ...	109, 162	625, 633, 907, 1009, 1206, 1211, 1304	
\cs_to_str:N	72		
		I	
D		\IfBooleanF	1517
\DeclareDocumentCommand ..	5, 1475, 1475	\IfBooleanT	1516
\DeclareDocumentEnvironment	6, 1497, 1497	\IfBooleanTF	7, 1515, 1515
\DeclareExpandableDocumentCommand ..		\IfNoValueF	1518
..... 10, 1513, 1513		\IfNoValueT	1519
\DeclareOption	1534, 1535, 1540	\IfNoValueTF	6, 1518, 1520
\do	990	\IfValueF	1521
\dospecials	991	\IfValueT	1522
		\IfValueTF	6, 1521, 1523
E		\int_compare:nNnF	420, 568, 1221
\ERROR	653, 663,	\int_compare:nNnT	393
673, 1070, 1081, 1093, 1106, 1126, 1144		\int_compare:nNnTF	
\exp_after:wN .	189, 291, 610, 646, 662, 75, 430, 966, 1035, 1052, 1223	
692, 710, 719, 728, 737, 746, 756,		\int_decr:N	
766, 776, 786, 797, 808, 846, 866, 1002		317, 324, 410, 436, 443, 563, 965, 1051	
\exp_args:Nf	1215	\int_eval:n	1235
\exp_args:NNNo ...	329, 363, 448, 517, 906	\int_incr:N	228,
\exp_args:No	338, 1084	236, 244, 256, 293, 359, 510, 975, 1165	
\exp_args:Nof	1076	\int_new:N	26, 39, 40, 42, 945
\exp_args:Nx	942	\int_set:Nn	397
\exp_last_unbraced:NnNo	1227	\int_set_eq:NN	949
\exp_not:c		\int_use:N	34, 396, 1050, 1166
. 113, 115, 129, 131, 162, 164, 170,		\int_zero:N ..	111, 181, 275, 277, 400, 1041
396, 406, 416, 472, 487, 541, 561, 571		\iow_char:N	950
\exp_not:N	112, 114,		
117, 128, 130, 542, 948, 950, 951, 1028		L	
\exp_not:n ..	165, 167, 473, 488, 949, 1251	\l__xparse_all_long_bool ..	23, 23, 92,
\exp_not:o	116, 127, 676	133, 274, 431, 433, 465, 480, 535, 576	
\exp_not:V	1055, 1242	\l__xparse_args_tl	
\exp_stop_f:	1078 24, 24, 112, 117, 165, 582,	
\ExplFileName	4	587, 592, 594, 603, 616, 669, 677,	
\ExplFileDescription	4	682, 686, 688, 695, 701, 704, 706,	

713, 717, 722, 726, 731, 735, 739, 740, 744, 749, 750, 754, 759, 760, 764, 769, 770, 774, 779, 780, 784, 789, 791, 795, 800, 802, 806, 811, 813, 817, 819, 830, 833, 838, 840, 842, 851, 855, 860, 862, 869, 885, 943, 1002, 1014, 1020, 1036, 1042, 1054	\l__xparse_split_list_seq
\l__xparse_command_arg_specs_prop 1177, 1177, 1191, 1193
..... 25, 25, 66, 1259, 1278	\l__xparse_split_list_tl
\l__xparse_current_arg_int 1177, 1178, 1201, 1207, 1209
..... 26, 26, 34, 75, 95, 108, 124, 173, 275, 293, 317, 324, 430, 436, 443	\l__xparse_tmp_tl
\l__xparse_environment_arg_specs_prop	.. 44, 44, 188, 189, 290, 291, 660, 663
..... 28, 28, 154, 1268, 1288	\l__xparse_v_arg_tl
\l__xparse_environment_bool	873, 874, 891, 908, 942, 1013, 1019, 1025
..... 27, 27, 67, 77, 155	\l__xparse_v_nesting_int
\l__xparse_expandable_aux_name_tl 30, 945, 945, 949, 965, 966, 975
30, 31, 468, 472, 483, 487, 538, 541	\l__xparse_v_rest_of_signature_tl ..
\l__xparse_expandable_bool 873, 873, 887, 943, 1014, 1020
..... 29, 29, 47, 52, 91, 99, 302, 308	log-declarations (option)
\l__xparse_fn_tl	11
..... 36, 36, 114, 610, 626, 634, 646, 652, 662, 673, 692, 698, 710, 715, 719, 724, 728, 733, 737, 742, 746, 752, 756, 762, 766, 772, 776, 782, 786, 793, 797, 804, 808, 815, 846, 858, 866, 871	
\l__xparse_function_tl 33, 37, 37, 72, 107, 113, 115, 123, 129, 131, 136, 442	
\l__xparse_long_bool	
..... 38, 38, 276, 316, 354, 407, 409, 419, 425, 429, 475, 490, 512, 544, 562, 572, 576, 877, 882, 994	
\l__xparse_m_args_int	
..... 39, 39, 75, 277, 359, 393, 396, 398, 400, 510	
\l__xparse_mandatory_args_int	
..... 40, 40, 181, 228, 236, 244, 256, 397, 398, 410, 420, 563, 568	
\l__xparse_nesting_a_tl	
..... 639, 639, 644, 648, 656, 666, 668	
\l__xparse_nesting_b_tl	
..... 639, 640, 645, 649, 658, 660, 661, 667	
\l__xparse_processor_bool	
..... 41, 41, 278, 284, 322, 354	
\l__xparse_processor_int	
..... 42, 42, 111, 1035, 1041, 1050, 1051, 1052, 1165, 1166	
\l__xparse_signature_tl	
..... 43, 43, 116, 127, 279, 325, 334, 343, 368, 375, 382, 395, 404, 414, 470, 485, 539, 560, 570	
	M
	\msg_line_context: 1456, 1461, 1466, 1471
	\msg_redirect_module:nnn 1537, 1538
	N
	\N 7, 1299
	\NewDocumentCommand .. 5, 1326, 1475, 1477
	\NewDocumentEnvironment
 6, 1340, 1497, 1499
	O
	log-declarations
	11
	P
	\peek_meaning_remove:NTF 895, 1009
	\peek_meaning_remove_ignore_spaces:NTF
 824
	\peek_N_type:TF 892, 922, 955
	\prg_new_conditional:Nppnn 1305
	\prg_replicate:nn 1241, 1248
	\prg_return_false: 1311
	\prg_return_true: 1310
	\ProcessedArgument 7, 1038, 1055, 1057, 1170, 1174, 1175, 1192, 1194, 1216, 1225, 1231, 1239, 1256, 1524, 1524
	\ProcessList 8, 1529, 1529
	\ProcessOptions 1541
	\prop_get:NnNF 1259, 1268
	\prop_get:NnNTF 188, 290, 1278, 1288
	\prop_new:N 14, 25, 28
	\prop_put:Nnn 15, 16, 17, 66, 154
	\ProvideDocumentCommand .. 5, 1475, 1495
	\ProvideDocumentEnvironment 6, 1497, 1511
	\ProvidesExplPackage 3
	Q
	\Q 6, 1297

\q_xparse	130, 468, 639, 641, 646, 653, 663, 673, 1060, 1061, 1062, 1065, 1069, 1070, 1072, 1074, 1079, 1082, 1085, 1088, 1092, 1093, 1096, 1097, 1098, 1105, 1108, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1121, 1125, 1126, 1131, 1134, 1135, 1136, 1143, 1149, 1152, 1153, 1154, 1157, 1158, 1160, 1162	\tl_if_empty:oTF	1084
\q_nil	653, 657, 663, 673, 675, 1070, 1093, 1126	\tl_if_in:NnTF	658
\q_recursion_stop	183, 280	\tl_if_in:nnTF	612, 650
\q_recursion_tail	183, 280	\tl_if_single:nF	261
\q_stop	657, 675, 1002, 1004, 1232	\tl_if_single_p:n	1183
\quark_if_recursion_tail_stop:N	187, 289	\tl_map_function:nN	1529
\quark_if_recursion_tail_stop_do:Nn	209, 235, 250	\tl_new:N	24, 30, 36, 37, 43, 44, 639, 640, 873, 874, 1178, 1275, 1524
\quark_if_recursion_tail_stop_do:nn	200, 216, 223, 243, 255	\tl_put_right:Nn	325, 334, 343, 368, 375, 382, 739, 749, 759, 769, 779, 789, 800, 811, 1036, 1042, 1194
\quark_new:N	641	\tl_put_right:No	648, 649, 666
R		\tl_put_right:Nx	395, 404, 414, 470, 485, 539, 560, 570, 656, 1025, 1054, 1239
\RenewDocumentCommand	5, 1333, 1475, 1486	\tl_replace_all:Nnn	1207
\RenewDocumentEnvironment	6, 1353, 1497, 1505	\tl_set:Nn	31, 112, 114, 887, 908, 1174, 1175, 1201
\ReverseBoolean	7, 1525, 1525	\tl_set:Nx	72, 1170, 1225, 1256
S		\tl_set_eq:NN	660
\scan_stop:	890, 993, 1541	\tl_show:N	1280, 1290
\seq_map_function:NN	985	\tl_tail:N	165, 942
\seq_map_inline:Nn	1193	\tl_to_lowercase:n	9, 623, 631, 1008, 1204, 1302
\seq_new:N	19, 1177	\tl_to_str:N	1013, 1019
\seq_set_split:Nnn	20, 1191	\tl_to_str:n	60, 64, 148, 152, 268, 271, 828, 1170, 1235, 1236, 1272, 1293
\ShowDocumentCommandArgSpec	10, 1530, 1532	\tl_trim_spaces:n	1256
\ShowDocumentEnvironmentArgSpec	10, 1530, 1533	\token_if_active:NnTF	1027
\SplitArgument	8, 1525, 1526	\token_if_cs_p:N	1184
\SplitList	8, 1525, 1527	\token_if_eq_catcode:NNTF	620
\str_if_eq:onTF	1064, 1100, 1120, 1138, 1156, 1307	\token_if_eq_charcode:NNT	974
\str_if_eq_x:nnTF	1032	\token_if_eq_charcode:NNTF	930, 963
\str_tail:n	1032	\token_if_eq_meaning:NNTF	453, 522
T		\token_to_str:c	442
\T	1300	\token_to_str:N	60, 64, 194, 196, 303, 307, 828, 1012, 1018, 1028, 1263, 1283, 1482, 1492
\tex_endlinechar:D	993, 995, 996, 1007	\TrimSpaces	9, 1525, 1528
\tex_escapechar:D	890	U	
\tl_clear:N	279, 644, 645, 661, 891, 1038, 1192	\use:c	196, 599, 697, 848, 1050
\tl_const:Nn	12	\use_i:nn	649
\tl_count:N	1216	\use_ii:nn	1077
		\use_none:n	85, 615, 648, 676, 1111
		\use_none:nn	1087
		\use_none:nnn	1085
		\use_none_delimit_by_q_stop:w	1230, 1249, 1253
		V	
		\V	8, 1301