

The `xparse` package

Document command parser*

The L^AT_EX3 Project[†]

Released 2013/07/12

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 v4544, last revised 2013/07/12.

[†]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 , $\{ \dots \}$), 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`

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.

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

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

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

```
\DeclareDocumentCommand{\foo}{o}{#1}
\foo[{abc}]          % => "abc"
\foo[ {abc}]          % => " {abc}"
```

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 L^AT_EX 2_ε'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

T_EX 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 L^AT_EX-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 \LaTeX 3 . All of the functions generated in this way are naturally robust (using the ϵ - \TeX `\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}
  { \typesetstarchapter {#3} }
  { \typesetnormalchapter {#2} {#3} }
}

```

would be a way to define a `\chapter` command which would essentially behave like the current $\text{\LaTeX 2}_{\epsilon}$ 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.

\TeX hackers note: Unlike $\text{\LaTeX 2}_{\epsilon}$'s `\newcommand` and relatives, the `\DeclareDocumentCommand` function do not prevent creation of functions with names starting `\end...`

`\DeclareDocumentEnvironment` `\DeclareDocumentEnvironment` $\{\langle environment \rangle\}$ $\{\langle arg\ spec \rangle\}$
`\NewDocumentEnvironment` $\{\langle start\ code \rangle\}$ $\{\langle end\ code \rangle\}$
`\RenewDocumentEnvironment`
`\ProvideDocumentEnvironment`

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

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.

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

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

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

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

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

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

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

will be logically `false`.

`\IfValueTF` *★* `\IfValueTF {⟨argument⟩} {⟨true code⟩} {⟨false code⟩}`

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

`\BooleanFalse`
`\BooleanTrue`

The `true` and `false` flags set when searching for an optional token (using `s` or `t{token}`) have names which are accessible outside of code blocks.

`\IfBooleanTF` *★* `\IfBooleanTF {⟨argument⟩} {⟨true code⟩} {⟨false code⟩}`

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

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

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

0.7 Argument processors

`xparse` introduces the idea of an argument processor, which is applied to an argument *after* it has been grabbed by the underlying system but before it is passed to `⟨code⟩`. An

argument processor can therefore be used to regularise input at an early stage, allowing the internal functions to be completely independent of input form. Processors are applied to user input and to default values for optional arguments, but *not* to the special `\NoValue` marker.

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

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

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

`\ProcessedArgument`

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

`\ReverseBoolean`

`\ReverseBoolean`

This processor reverses the logic of `\BooleanTrue` and `\BooleanFalse`, so that the 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** { $\langle token(s) \rangle$ }

This processor splits the argument given at each occurrence of the $\langle token(s) \rangle$ 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 $\langle token \rangle$ is used for the split, any category code 13 (active) $\langle token \rangle$ will be replaced before the split takes place.

\ProcessList ★**\ProcessList** { $\langle list \rangle$ } { $\langle function \rangle$ }

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

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

This function is experimental.

\TrimSpaces**\TrimSpaces**

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

```
\DeclareDocumentCommand \foo
{ > { \TrimSpaces } }
{ \showtokens {#1} }
```

and using it in a document as

```
\foo{ hello world }
```

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

This function is experimental.

0.8 Fully-expandable document commands

There are *very rare* occasion when it may be useful to create functions using a fully-expandable argument grabber. To support this, `xparse` can create expandable functions as well as the usual robust ones. This imposes a number of restrictions on the nature of the arguments accepted by a function, and the code it implements. This facility should

only be used when *absolutely necessary*; if you do not understand when this might be, *do not use these functions!*

<code>\DeclareExpandableDocumentCommand</code>	<code>\DeclareExpandableDocumentCommand</code>
	<code>\langle function \rangle \{ \langle arg spec \rangle \} \{ \langle code \rangle \}</code>

This command is used to create a document-level $\langle function \rangle$, which will grab its arguments in a fully-expandable manner. The argument specification for the function is given by $\langle arg spec \rangle$, and the function will execute $\langle code \rangle$. In general, $\langle code \rangle$ 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</code>
<code>\GetDocumentEnvironmentArgSpec</code>	<code>\GetDocumentEnvironmentArgSpec</code>

These functions transfer the current argument specification for the requested $\langle function \rangle$ or $\langle environment \rangle$ into the token list variable `\ArgumentSpecification`. If the $\langle function \rangle$ or $\langle environment \rangle$ 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 ??.)

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

```

23 \bool_new:N \l__xparse_all_long_bool

```

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

`\l__xparse_args_tl` Token list variable for grabbed arguments.

```

24 \tl_new:N \l__xparse_args_tl

```

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

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

```

25 \prop_new:N \l__xparse_command_arg_specs_prop

```

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

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

```

26 \int_new:N \l__xparse_current_arg_int

```

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

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

```

27 \bool_new:N \l__xparse_environment_bool

```

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

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

```

28 \prop_new:N \l__xparse_environment_arg_specs_prop

```

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

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

```

29 \bool_new:N \l__xparse_expandable_bool

```

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

<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. <pre> 30 \tl_new:N \l__xparse_expandable_aux_name_tl 31 \tl_set:Nn \l__xparse_expandable_aux_name_tl 32 { 33 \l__xparse_function_tl \c_space_tl 34 (arg~ \int_use:N \l__xparse_current_arg_int) 35 } </pre> <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. <pre> 36 \tl_new:N \l__xparse_fn_tl </pre> <i>(End definition for \l__xparse_fn_tl. This variable is documented on page ??.)</i>
<code>\l__xparse_function_tl</code>	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 <code>\cs_to_str:N</code> . <pre> 37 \tl_new:N \l__xparse_function_tl </pre> <i>(End definition for \l__xparse_function_tl. This variable is documented on page ??.)</i>
<code>\l__xparse_long_bool</code>	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. <pre> 38 \bool_new:N \l__xparse_long_bool </pre> <i>(End definition for \l__xparse_long_bool. This variable is documented on page ??.)</i>
<code>\l__xparse_m_args_int</code>	The number of <code>m</code> 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. <pre> 39 \int_new:N \l__xparse_m_args_int </pre> <i>(End definition for \l__xparse_m_args_int. This variable is documented on page ??.)</i>
<code>\l__xparse_mandatory_args_int</code>	Holds the total number of mandatory arguments for a function, which is needed to tell whether further mandatory arguments follow an optional one. <pre> 40 \int_new:N \l__xparse_mandatory_args_int </pre> <i>(End definition for \l__xparse_mandatory_args_int. This variable is documented on page ??.)</i>
<code>\l__xparse_processor_bool</code>	Indicates that the current argument will be followed by one or more processors. <pre> 41 \bool_new:N \l__xparse_processor_bool </pre> <i>(End definition for \l__xparse_processor_bool. This variable is documented on page ??.)</i>
<code>\l__xparse_processor_int</code>	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. <pre> 42 \int_new:N \l__xparse_processor_int </pre> <i>(End definition for \l__xparse_processor_int. This variable is documented on page ??.)</i>
<code>\l__xparse_signature_tl</code>	Used when constructing the signature (code for argument grabbing) to hold what will become the implementation of the main function. <pre> 43 \tl_new:N \l__xparse_signature_tl </pre>

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

```
\__xparse_declare_expandable_cmd:Nnn
\__xparse_declare_cmd_aux:Nnn
\__xparse_declare_cmd_internal:Nnn
\__xparse_declare_cmd_internal:cnx
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 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).

```

\__xparse_declare_cmd_mixed:Nn
\__xparse_declare_cmd_mixed_aux:Nn
\__xparse_declare_cmd_mixed_expandable:Nn
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:NTF \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_false:N \l__xparse_expandable_bool
156 \bool_set_true:N \l__xparse_environment_bool
157 \__xparse_declare_env_internal:nnnn {#1} {#2}
158 }

```

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.

```

159 \cs_new_protected:Npn \__xparse_declare_env_internal:nnnn #1#2#3#4
160 {
161   \__xparse_declare_cmd_internal:cnx { environment~ #1 } {#2}
162   {
163     \cs_set_protected_nopar:Npx \exp_not:c { environment~ #1 ~end~aux }
164     {
165       \exp_not:c { environment~ #1~end~aux~ }
166       \exp_not:n { \tl_tail:N \l__xparse_args_tl }
167     }
168     \exp_not:n {#3}
169   }
170   \cs_set_protected_nopar:cpx { environment~ #1 ~end }
171   { \exp_not:c { environment~ #1 ~end~aux } }
172   \cs_generate_from_arg_count:cNnn
173   { environment~ #1 ~end~aux~ } \cs_set_protected:Npn
174   \l__xparse_current_arg_int {#4}
175   (*package)
176   \cs_set_eq:cc {#1} { environment~ #1 }
177   \cs_set_eq:cc { end #1 } { environment~ #1 ~end }
178   (/package)
179 }

```

(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.

```

180 \cs_new_protected:Npn \__xparse_count_mandatory:n #1
181 {
182   \int_zero:N \l__xparse_mandatory_args_int
183   \__xparse_count_mandatory:N #1
184   \q_recursion_tail \q_recursion_tail \q_recursion_tail \q_recursion_stop
185 }
186 \cs_new_protected:Npn \__xparse_count_mandatory:N #1
187 {

```

```

188 \quark_if_recursion_tail_stop:N #1
189 \prop_get:NnNTF \c__xparse_shorthands_prop {#1} \l__xparse_tmp_tl
190 { \exp_after:wN \__xparse_count_mandatory:N \l__xparse_tmp_tl }
191 { \__xparse_count_mandatory_aux:N #1 }
192 }
193 \cs_new_protected:Npn \__xparse_count_mandatory_aux:N #1
194 {
195   \cs_if_free:cTF { __xparse_count_type_ \token_to_str:N #1 :w }
196   { \__xparse_count_type_m:w }
197   { \use:c { __xparse_count_type_ \token_to_str:N #1 :w } }
198 }

```

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

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

```

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

```

```

225     \__xparse_count_mandatory:N
226   }
227 \cs_new_protected_nopar:Npn \__xparse_count_type_m:w
228 {
229     \int_incr:N \l__xparse_mandatory_args_int
230     \__xparse_count_mandatory:N
231 }
232 \cs_new_protected:Npn \__xparse_count_type_r:w #1#2
233 {
234     \__xparse_single_token_check:n {#1}
235     \__xparse_single_token_check:n {#2}
236     \quark_if_recursion_tail_stop_do:Nn #2 { \__xparse_bad_arg_spec:wn }
237     \int_incr:N \l__xparse_mandatory_args_int
238     \__xparse_count_mandatory:N
239 }
240 \cs_new_protected:Npn \__xparse_count_type_R:w #1#2#3
241 {
242     \__xparse_single_token_check:n {#1}
243     \__xparse_single_token_check:n {#2}
244     \quark_if_recursion_tail_stop_do:nn {#3} { \__xparse_bad_arg_spec:wn }
245     \int_incr:N \l__xparse_mandatory_args_int
246     \__xparse_count_mandatory:N
247 }
248 \cs_new_protected:Npn \__xparse_count_type_t:w #1
249 {
250     \__xparse_single_token_check:n {#1}
251     \quark_if_recursion_tail_stop_do:Nn #1 { \__xparse_bad_arg_spec:wn }
252     \__xparse_count_mandatory:N
253 }
254 \cs_new_protected:Npn \__xparse_count_type_u:w #1
255 {
256     \quark_if_recursion_tail_stop_do:nn {#1} { \__xparse_bad_arg_spec:wn }
257     \int_incr:N \l__xparse_mandatory_args_int
258     \__xparse_count_mandatory:N
259 }

```

(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

```

260 \cs_new_protected:Npn \__xparse_single_token_check:n #1
261 {
262     \tl_if_single:nF {#1}
263     { \__xparse_single_token_check_aux:nwn {#1} }
264 }
265 \cs_new_protected:Npn \__xparse_single_token_check_aux:nwn
266 #1#2 \__xparse_break_point:n #3
267 {
268     \__msg_kernel_error:nnx { xparse } { not-single-token }
269     { \tl_to_str:n {#1} } { \tl_to_str:n {#3} }
270 }

```

(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.

```

271 \cs_new_protected:Npn \_xparse_bad_arg_spec:wn #1 \_xparse_break_point:n #2
272 { \_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` 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.

`_xparse_prepare_signature:N`

`_xparse_prepare_signature_bypass:N`

`_xparse_prepare_signature_add:N`

```

273 \cs_new_protected:Npn \_xparse_prepare_signature:n #1
274 {
275   \bool_set_false:N \l__xparse_all_long_bool
276   \int_zero:N \l__xparse_current_arg_int
277   \bool_set_false:N \l__xparse_long_bool
278   \int_zero:N \l__xparse_m_args_int
279   \bool_set_false:N \l__xparse_processor_bool
280   \tl_clear:N \l__xparse_signature_tl
281   \_xparse_prepare_signature:N #1 \q_recursion_tail \q_recursion_stop
282 }

```

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.

```

283 \cs_new_protected_nopar:Npn \_xparse_prepare_signature:N
284 {
285   \bool_set_false:N \l__xparse_processor_bool
286   \_xparse_prepare_signature_bypass:N
287 }
288 \cs_new_protected:Npn \_xparse_prepare_signature_bypass:N #1
289 {
290   \quark_if_recursion_tail_stop:N #1
291   \prop_get:NnNTF \c__xparse_shorthands_prop {#1} \l__xparse_tmp_tl
292   { \exp_after:wN \_xparse_prepare_signature:N \l__xparse_tmp_tl }
293   {
294     \int_incr:N \l__xparse_current_arg_int
295     \_xparse_prepare_signature_add:N #1
296   }
297 }

```

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.

```

298 \cs_new_protected:Npn \_xparse_prepare_signature_add:N #1
299 {
300   \cs_if_exist_use:cF

```

```

301     {
302         __xparse_add
303         \bool_if:NT \l__xparse_expandable_bool { _expandable }
304         _type_ \token_to_str:N #1 :w
305     }
306     {
307         \__msg_kernel_error:nx { xparse } { unknown-argument-type }
308         { \token_to_str:N #1 }
309         \bool_if:NTF \l__xparse_expandable_bool
310         { \__xparse_add_expandable_type_m:w }
311         { \__xparse_add_type_m:w }
312     }
313 }

```

(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.

```

314 \cs_new_protected_nopar:cpn { __xparse_add_type_+:w }
315 {
316     \__xparse_flush_m_args:
317     \bool_set_true:N \l__xparse_long_bool
318     \int_decr:N \l__xparse_current_arg_int
319     \__xparse_prepare_signature:N
320 }

```

(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.

```

321 \cs_new_protected:cpn { __xparse_add_type_>:w } #1
322 {
323     \bool_set_true:N \l__xparse_processor_bool
324     \__xparse_flush_m_args:
325     \int_decr:N \l__xparse_current_arg_int
326     \tl_put_right:Nn \l__xparse_signature_tl { \__xparse_process_arg:n {#1} }
327     \__xparse_prepare_signature_bypass:N
328 }

```

(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
`__xparse_add_type_D:w` putting the `-NoValue-` default in the correct place.

```
329 \cs_new_protected:Npn \__xparse_add_type_d:w #1#2
330 { \exp_args:NNNo \__xparse_add_type_D:w #1 #2 \c__xparse_no_value_tl }
331 \cs_new_protected:Npn \__xparse_add_type_D:w #1#2#3
332 {
333   \__xparse_flush_m_args:
334   \__xparse_add_grabber_optional:N D
335   \tl_put_right:Nn \l__xparse_signature_tl { #1 #2 {#3} }
336   \__xparse_prepare_signature:N
337 }
```

(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.

```
338 \cs_new_protected_nopar:Npn \__xparse_add_type_g:w
339 { \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.

```
340 \cs_new_protected:Npn \__xparse_add_type_G:w #1
341 {
342   \__xparse_flush_m_args:
343   \__xparse_add_grabber_optional:N G
344   \tl_put_right:Nn \l__xparse_signature_tl { {#1} }
345   \__xparse_prepare_signature:N
346 }
```

(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.

```
347 \cs_new_protected_nopar:Npn \__xparse_add_type_l:w
348 {
349   \__xparse_flush_m_args:
350   \__xparse_add_grabber_mandatory:N l
351   \__xparse_prepare_signature:N
352 }
```

(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:`.

```
353 \cs_new_protected_nopar:Npn \__xparse_add_type_m:w
354 {
355   \bool_if:nTF { \l__xparse_long_bool || \l__xparse_processor_bool }
```

```

356     {
357         \__xparse_flush_m_args:
358         \__xparse_add_grabber_mandatory:N m
359     }
360     { \int_incr:N \l__xparse_m_args_int }
361     \__xparse_prepare_signature:N
362 }

```

(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

```

363 \cs_new_protected:Npn \__xparse_add_type_r:w #1#2
364 { \exp_args:NNNo \__xparse_add_type_R:w #1 #2 \c__xparse_no_value_tl }
365 \cs_new_protected:Npn \__xparse_add_type_R:w #1#2#3
366 {
367     \__xparse_flush_m_args:
368     \__xparse_add_grabber_mandatory:N R
369     \tl_put_right:Nn \l__xparse_signature_tl { #1 #2 {#3} }
370     \__xparse_prepare_signature:N
371 }

```

(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.

```

372 \cs_new_protected:Npn \__xparse_add_type_t:w #1
373 {
374     \__xparse_flush_m_args:
375     \__xparse_add_grabber_optional:N t
376     \tl_put_right:Nn \l__xparse_signature_tl { #1 }
377     \__xparse_prepare_signature:N
378 }

```

(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.

```

379 \cs_new_protected:Npn \__xparse_add_type_u:w #1
380 {
381     \__xparse_flush_m_args:
382     \__xparse_add_grabber_mandatory:N u
383     \tl_put_right:Nn \l__xparse_signature_tl { {#1} }
384     \__xparse_prepare_signature:N
385 }

```

(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.

```

386 \cs_new_protected_nopar:Npn \__xparse_add_type_v:w
387 {
388     \__xparse_flush_m_args:
389     \__xparse_add_grabber_mandatory:N v

```

```

390     \__xparse_prepare_signature:N
391 }

```

(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.

```

392 \cs_new_protected_nopar:Npn \__xparse_flush_m_args:
393 {
394     \int_compare:nNnT \l__xparse_m_args_int > \c_zero
395     {
396         \tl_put_right:Nx \l__xparse_signature_tl
397         { \exp_not:c { __xparse_grab_m_ \int_use:N \l__xparse_m_args_int :w } }
398         \int_set:Nn \l__xparse_mandatory_args_int
399         { \l__xparse_mandatory_args_int - \l__xparse_m_args_int }
400     }
401     \int_zero:N \l__xparse_m_args_int
402 }

```

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

__xparse_add_grabber_mandatory: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. __xparse_add_grabber_optional:N 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.

```

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

```



```

426     \bool_set_false:N \l__xparse_long_bool
427 }

```

(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.

```

428 \cs_new_protected_nopar:cpn { __xparse_add_expandable_type+:w }
429 {
430     \bool_set_true:N \l__xparse_long_bool
431     \int_compare:nNnTF \l__xparse_current_arg_int = \c_one
432     { \bool_set_true:N \l__xparse_all_long_bool }
433     {
434         \bool_if:NF \l__xparse_all_long_bool
435         { __msg_kernel_error:nn { xparse } { inconsistent-long } }
436     }
437     \int_decr:N \l__xparse_current_arg_int
438     \__xparse_prepare_signature:N
439 }

```

(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.

```

440 \cs_new_protected:cpn { __xparse_add_expandable_type>:w } #1
441 {
442     __msg_kernel_error:nnx { xparse } { processor-in-expandable }
443     { \token_to_str:c { \l__xparse_function_tl } }
444     \int_decr:N \l__xparse_current_arg_int
445     \__xparse_prepare_signature:N
446 }

```

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

_xparse_add_expandable_type_d:w
_xparse_add_expandable_type_D:w
_xparse_add_expandable_type_D_aux:NNn
_xparse_add_expandable_type_D_aux:Nn

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

```

447 \cs_new_protected:Npn __xparse_add_expandable_type_d:w #1#2
448 {
449     \exp_args:NNNo
450     \__xparse_add_expandable_type_D:w #1 #2 \c__xparse_no_value_tl
451 }
452 \cs_new_protected:Npn __xparse_add_expandable_type_D:w #1#2
453 {
454     \token_if_eq_meaning:NNTF #1 #2
455     {
456         \__xparse_add_expandable_grabber_optional:n { D_alt }

```

```

457     \__xparse_add_expandable_type_D_aux:Nn #1
458   }
459   {
460     \__xparse_add_expandable_grabber_optional:n { D }
461     \__xparse_add_expandable_type_D_aux:NNn #1#2
462   }
463 }
464 \cs_new_protected:Npn \__xparse_add_expandable_type_D_aux:NNn #1#2#3
465 {
466   \bool_if:NTF \l__xparse_all_long_bool
467   { \cs_set:cpx }
468   { \cs_set_nopar:cpx }
469   { \l__xparse_expandable_aux_name_tl } ##1 ##2 #1 ##3 \q__xparse ##4 #2
470   { ##1 {##2} {##3} {##4} }
471   \tl_put_right:Nx \l__xparse_signature_tl
472   {
473     \exp_not:c { \l__xparse_expandable_aux_name_tl }
474     \exp_not:n { #1 #2 {#3} }
475   }
476   \bool_set_false:N \l__xparse_long_bool
477   \__xparse_prepare_signature:N
478 }

```

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.

```

479 \cs_new_protected:Npn \__xparse_add_expandable_type_D_aux:Nn #1#2
480 {
481   \bool_if:NTF \l__xparse_all_long_bool
482   { \cs_set:cpx }
483   { \cs_set_nopar:cpx }
484   { \l__xparse_expandable_aux_name_tl } ##1 #1 ##2 #1
485   { ##1 {##2} }
486   \tl_put_right:Nx \l__xparse_signature_tl
487   {
488     \exp_not:c { \l__xparse_expandable_aux_name_tl }
489     \exp_not:n { #1 {#2} }
490   }
491   \bool_set_false:N \l__xparse_long_bool
492   \__xparse_prepare_signature:N
493 }

```

(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.

```

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

```

```

501     \__msg_kernel_error:nxx { xparse } { invalid-expandable-argument-type } { G }
502     \__xparse_add_expandable_type_m:w
503 }

```

(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).

```

504 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_l:w
505 {
506     \__msg_kernel_error:nxx { xparse } { invalid-expandable-argument-type } { l }
507     \__xparse_add_expandable_type_m:w
508 }

```

(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.

```

509 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_m:w
510 {
511     \int_incr:N \l__xparse_m_args_int
512     \__xparse_add_expandable_grabber_mandatory:n { m }
513     \bool_set_false:N \l__xparse_long_bool
514     \__xparse_prepare_signature:N
515 }

```

(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.
`__xparse_add_expandable_type_R:w`

```

516 \cs_new_protected:Npn \__xparse_add_expandable_type_r:w #1#2
517 {
518     \exp_args:NNNo
519     \__xparse_add_expandable_type_R:w #1 #2 \c__xparse_no_value_tl
520 }
521 \cs_new_protected:Npn \__xparse_add_expandable_type_R:w #1#2
522 {
523     \token_if_eq_meaning:NNTF #1 #2
524     {
525         \__xparse_add_expandable_grabber_optional:n { R_alt }
526         \__xparse_add_expandable_type_D_aux:Nn #1
527     }
528     {
529         \__xparse_add_expandable_grabber_optional:n { R }
530         \__xparse_add_expandable_type_D_aux:NNn #1#2
531     }
532 }

```

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

_xparse_add_expandable_type_t:w

```
533 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_t:w #1
534 {
535   \__xparse_add_expandable_grabber_optional:n { t }
536   \bool_if:NTF \l__xparse_all_long_bool
537     { \cs_set:cpn }
538     { \cs_set_nopar:cpn }
539     { \l__xparse_expandable_aux_name_tl } ##1 #1 {##1}
540   \tl_put_right:Nx \l__xparse_signature_tl
541     {
542       \exp_not:c { \l__xparse_expandable_aux_name_tl }
543       \exp_not:N #1
544     }
545   \bool_set_false:N \l__xparse_long_bool
546   \__xparse_prepare_signature:N
547 }
```

(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.

```
548 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_u:w #1
549 {
550   \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { u }
551   \__xparse_add_expandable_type_m:w
552 }
```

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

_xparse_add_expandable_type_v:w

Another forbidden type.

```
553 \cs_new_protected_nopar:Npn \__xparse_add_expandable_type_v:w
554 {
555   \__msg_kernel_error:nnx { xparse } { invalid-expandable-argument-type } { v }
556   \__xparse_add_expandable_type_m:w
557 }
```

(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.

_xparse_add_expandable_grabber_optional:n

_xparse_add_expandable_long_check:

```
558 \cs_new_protected_nopar:Npn \__xparse_add_expandable_grabber_mandatory:n #1
559 {
560   \__xparse_add_expandable_long_check:
561   \tl_put_right:Nx \l__xparse_signature_tl
562     { \exp_not:c { __xparse_expandable_grab_ #1 :w } }
563   \bool_set_false:N \l__xparse_long_bool
564   \int_decr:N \l__xparse_mandatory_args_int
565 }
566 \cs_new_protected_nopar:Npn \__xparse_add_expandable_grabber_optional:n #1
567 {
```

```

568 \__xparse_add_expandable_long_check:
569 \int_compare:nNnF \l__xparse_mandatory_args_int > \c_zero
570 { \__msg_kernel_error:nn { xparse } { expandable-ending-optional } }
571 \tl_put_right:Nx \l__xparse_signature_tl
572 { \exp_not:c { __xparse_expandable_grab_ #1 :w } }
573 \bool_set_false:N \l__xparse_long_bool
574 }
575 \cs_new_protected_nopar:Npn \__xparse_add_expandable_long_check:
576 {
577   \bool_if:nT { \l__xparse_all_long_bool && ! ( \l__xparse_long_bool ) }
578   { \__msg_kernel_error:nn { xparse } { inconsistent-long } }
579 }

```

(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` 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.

```

580 \cs_new_protected:Npn \__xparse_grab_arg:w { }
581 \cs_new_protected:Npn \__xparse_grab_arg_auxi:w { }
582 \cs_new_protected:Npn \__xparse_grab_arg_auxii:w { }

```

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

`__xparse_grab_D: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.

```

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

```

597 \cs_new_protected:Npn \__xparse_grab_D_aux:NNnnNn #1#2#3#4#5#6
598 {
599   \__xparse_grab_D_aux:NNnN #1#2 {#4} #5
600   \use:c { peek_meaning_remove #6 :NTF } #1
601   { \__xparse_grab_arg:w }
602   {
603     \__xparse_add_arg:n {#3}
604     #4 \l__xparse_args_tl
605   }
606 }

```

Inside the “standard” grabber, there is a test to see if the grabbed argument is entirely enclosed by braces. There are a couple of extra factors to allow for: the argument might be entirely empty, and spaces at the start and end of the input must be retained around a brace group.

```

607 \cs_new_protected:Npn \__xparse_grab_D_aux:NNnN #1#2#3#4
608 {
609   \cs_set_protected_nopar:Npn \__xparse_grab_arg:w
610   {
611     \exp_after:wN #4 \l__xparse_fn_tl #####1 #2
612     {
613       \tl_if_in:nnTF {#####1} {#1}
614       { \__xparse_grab_D_nested:NNnnN #1 #2 {#####1} {#3} #4 }
615       {
616         \tl_if_blank:oTF { \use_none:n #####1 }
617         { \__xparse_add_arg:o { \use_none:n #####1 } }
618         {
619           \str_if_eq_x:nnTF
620           { \exp_not:o { \use_none:n #####1 } }
621           { { \exp_not:o { \use_ii:nnn #####1 \q_nil } } }
622           { \__xparse_add_arg:o { \use_ii:nn #####1 } }
623           { \__xparse_add_arg:o { \use_none:n #####1 } }
624         }
625         #3 \l__xparse_args_tl
626       }
627     }
628   }

```

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.)

```

628     \group_begin:
629     \token_if_eq_catcode:NNTF #1 ^
630     {
631         \char_set_lccode:nn { 'A } { '#1 }
632         \tl_to_lowercase:n
633         {
634             \group_end:
635             \l__xparse_fn_tl A
636         }
637     }
638     {
639         \char_set_lccode:nn { '^ } { '#1 }
640         \tl_to_lowercase:n
641         {
642             \group_end:
643             \l__xparse_fn_tl ^
644         }
645     }
646 }
647 }

```

(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 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.

```

648 \tl_new:N \l__xparse_nesting_a_tl
649 \tl_new:N \l__xparse_nesting_b_tl
650 \quark_new:N \q__xparse
651 \cs_new_protected:Npn \__xparse_grab_D_nested:NNnnN #1#2#3#4#5
652 {
653     \tl_clear:N \l__xparse_nesting_a_tl
654     \tl_clear:N \l__xparse_nesting_b_tl
655     \exp_after:wN #5 \l__xparse_fn_tl ##1 #1 ##2 \q__xparse ##3 #2
656     {
657         \tl_put_right:No \l__xparse_nesting_a_tl { \use_none:n ##1 #1 }
658         \tl_put_right:No \l__xparse_nesting_b_tl { \use_i:nn #2 ##3 }

```

```

659 \tl_if_in:nnTF {##2} {#1}
660 {
661   \l__xparse_fn_tl
662   \q_nil ##2 \q__xparse \ERROR
663 }
664 {
665   \tl_put_right:Nx \l__xparse_nesting_a_tl
666   { \__xparse_grab_D_nested:w \q_nil ##2 \q_stop }
667   \tl_if_in:NnTF \l__xparse_nesting_b_tl {#1}
668   {
669     \tl_set_eq:NN \l__xparse_tmp_tl \l__xparse_nesting_b_tl
670     \tl_clear:N \l__xparse_nesting_b_tl
671     \exp_after:wN \l__xparse_fn_tl \exp_after:wN
672     \q_nil \l__xparse_tmp_tl \q_nil \q__xparse \ERROR
673   }
674   {
675     \tl_put_right:No \l__xparse_nesting_a_tl
676     \l__xparse_nesting_b_tl
677     \__xparse_add_arg:V \l__xparse_nesting_a_tl
678     #4 \l__xparse_args_tl
679   }
680 }
681 }
682 \l__xparse_fn_tl #3 \q_nil \q__xparse \ERROR
683 }
684 \cs_new:Npn \__xparse_grab_D_nested:w #1 \q_nil \q_stop
685 { \exp_not:o { \use_none:n #1 } }

```

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

__xparse_grab_G:w Optional groups are checked by meaning, so that the same code will work with, for
 __xparse_grab_G_long:w example, ConTeXt-like input.
 __xparse_grab_G_trailing:w
 __xparse_grab_G_long_trailing:w
 __xparse_grab_G_aux:nnNn

```

686 \cs_new_protected:Npn \__xparse_grab_G:w #1#2 \l__xparse_args_tl
687 {
688   \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected_nopar:Npn
689   { _ignore_spaces }
690 }
691 \cs_new_protected:Npn \__xparse_grab_G_long:w #1#2 \l__xparse_args_tl
692 {
693   \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected:Npn { _ignore_spaces }
694 }
695 \cs_new_protected:Npn \__xparse_grab_G_trailing:w #1#2 \l__xparse_args_tl
696 { \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected_nopar:Npn { } }
697 \cs_new_protected:Npn \__xparse_grab_G_long_trailing:w #1#2 \l__xparse_args_tl
698 { \__xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_protected:Npn { } }
699 \cs_new_protected:Npn \__xparse_grab_G_aux:nnNn #1#2#3#4
700 {
701   \exp_after:wN #3 \l__xparse_fn_tl ##1
702   {
703     \__xparse_add_arg:n {##1}

```



```

704         #2 \l__xparse_args_tl
705     }
706     \use:c { peek_meaning #4 :NTF } \c_group_begin_token
707     { \l__xparse_fn_tl }
708     {
709         \__xparse_add_arg:n {#1}
710         #2 \l__xparse_args_tl
711     }
712 }

```

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

__xparse_grab_l:w Argument grabbers for mandatory \TeX arguments are pretty simple.

```

\__xparse_grab_l_long:w 713 \cs_new_protected:Npn \__xparse_grab_l:w #1 \l__xparse_args_tl
\__xparse_grab_l_aux:nN 714 { \__xparse_grab_l_aux:nN {#1} \cs_set_protected_nopar:Npn }
715 \cs_new_protected:Npn \__xparse_grab_l_long:w #1 \l__xparse_args_tl
716 { \__xparse_grab_l_aux:nN {#1} \cs_set_protected:Npn }
717 \cs_new_protected:Npn \__xparse_grab_l_aux:nN #1#2
718 {
719     \exp_after:wN #2 \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 }

```

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

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

```

\__xparse_grab_m_long:w 726 \cs_new_protected:Npn \__xparse_grab_m:w #1 \l__xparse_args_tl
727 {
728     \exp_after:wN \cs_set_protected_nopar: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 }
735 \cs_new_protected:Npn \__xparse_grab_m_long:w #1 \l__xparse_args_tl
736 {
737     \exp_after:wN \cs_set_protected:Npn \l__xparse_fn_tl ##1
738     {
739         \__xparse_add_arg:n {##1}
740         #1 \l__xparse_args_tl
741     }
742     \l__xparse_fn_tl
743 }

```

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

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

```

\__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.
\__xparse_grab_m_4:w 744 \cs_new_protected:cpn { __xparse_grab_m_1:w } #1 \l__xparse_args_tl
\__xparse_grab_m_5:w 745 {
\__xparse_grab_m_6:w 746   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl ##1
\__xparse_grab_m_7:w 747   {
\__xparse_grab_m_8:w 748     \tl_put_right:Nn \l__xparse_args_tl { {##1} }
749     #1 \l__xparse_args_tl
750   }
751   \l__xparse_fn_tl
752 }
753 \cs_new_protected:cpn { __xparse_grab_m_2:w } #1 \l__xparse_args_tl
754 {
755   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
756   ##1##2
757   {
758     \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} }
759     #1 \l__xparse_args_tl
760   }
761   \l__xparse_fn_tl
762 }
763 \cs_new_protected:cpn { __xparse_grab_m_3:w } #1 \l__xparse_args_tl
764 {
765   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
766   ##1##2##3
767   {
768     \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} }
769     #1 \l__xparse_args_tl
770   }
771   \l__xparse_fn_tl
772 }
773 \cs_new_protected:cpn { __xparse_grab_m_4:w } #1 \l__xparse_args_tl
774 {
775   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
776   ##1##2##3##4
777   {
778     \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} {##4} }
779     #1 \l__xparse_args_tl
780   }
781   \l__xparse_fn_tl
782 }
783 \cs_new_protected:cpn { __xparse_grab_m_5:w } #1 \l__xparse_args_tl
784 {
785   \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
786   ##1##2##3##4##5
787   {
788     \tl_put_right:Nn \l__xparse_args_tl { {##1} {##2} {##3} {##4} {##5} }
789     #1 \l__xparse_args_tl

```

```

790     }
791     \l__xparse_fn_tl
792 }
793 \cs_new_protected:cpn { __xparse_grab_m_6:w } #1 \l__xparse_args_tl
794 {
795     \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
796     ##1##2##3##4##5##6
797     {
798         \tl_put_right:Nn \l__xparse_args_tl
799         { {##1} {##2} {##3} {##4} {##5} {##6} }
800         #1 \l__xparse_args_tl
801     }
802     \l__xparse_fn_tl
803 }
804 \cs_new_protected:cpn { __xparse_grab_m_7:w } #1 \l__xparse_args_tl
805 {
806     \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
807     ##1##2##3##4##5##6##7
808     {
809         \tl_put_right:Nn \l__xparse_args_tl
810         { {##1} {##2} {##3} {##4} {##5} {##6} {##7} }
811         #1 \l__xparse_args_tl
812     }
813     \l__xparse_fn_tl
814 }
815 \cs_new_protected:cpn { __xparse_grab_m_8:w } #1 \l__xparse_args_tl
816 {
817     \exp_after:wN \cs_set_protected_nopar:Npn \l__xparse_fn_tl
818     ##1##2##3##4##5##6##7##8
819     {
820         \tl_put_right:Nn \l__xparse_args_tl
821         { {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} }
822         #1 \l__xparse_args_tl
823     }
824     \l__xparse_fn_tl
825 }

```

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

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

```

826 \cs_new_protected:Npn \__xparse_grab_R:w #1#2#3#4 \l__xparse_args_tl
827 { \__xparse_grab_R_aux:NNnnN #1 #2 {#3} {#4} \cs_set_protected_nopar:Npn }
828 \cs_new_protected:Npn \__xparse_grab_R_long:w #1#2#3#4 \l__xparse_args_tl
829 { \__xparse_grab_R_aux:NNnnN #1 #2 {#3} {#4} \cs_set_protected:Npn }
830 \cs_new_protected:Npn \__xparse_grab_R_aux:NNnnN #1#2#3#4#5
831 {
832     \__xparse_grab_D_aux:NNnN #1 #2 {#4} #5
833     \peek_meaning_remove_ignore_spaces:NTF #1
834     { \__xparse_grab_arg:w }

```

```

835     {
836         \_msg_kernel_error:nnxx { xparse } { missing-required }
837         { \token_to_str:N #1 } { \tl_to_str:n {#3} }
838         \_xparse_add_arg:n {#3}
839         #4 \l__xparse_args_tl
840     }
841 }

```

(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.

```

842 \cs_new_protected:Npn \_xparse_grab_t:w #1#2 \l__xparse_args_tl
843 {
844     \_xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected_nopar:Npn
845     { _ignore_spaces }
846 }
847 \cs_new_protected:Npn \_xparse_grab_t_long:w #1#2 \l__xparse_args_tl
848 { \_xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected:Npn { _ignore_spaces } }
849 \cs_new_protected:Npn \_xparse_grab_t_trailing:w #1#2 \l__xparse_args_tl
850 { \_xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected_nopar:Npn { } }
851 \cs_new_protected:Npn \_xparse_grab_t_long_trailing:w #1#2 \l__xparse_args_tl
852 { \_xparse_grab_t_aux:NnNn #1 {#2} \cs_set_protected:Npn { } }
853 \cs_new_protected:Npn \_xparse_grab_t_aux:NnNn #1#2#3#4
854 {
855     \exp_after:wN #3 \l__xparse_fn_tl
856     {
857         \use:c { peek_meaning_remove #4 :NTF } #1
858         {
859             \_xparse_add_arg:n { \BooleanTrue }
860             #2 \l__xparse_args_tl
861         }
862         {
863             \_xparse_add_arg:n { \BooleanFalse }
864             #2 \l__xparse_args_tl
865         }
866     }
867     \l__xparse_fn_tl
868 }

```

(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.

```

869 \cs_new_protected:Npn \_xparse_grab_u:w #1#2 \l__xparse_args_tl
870 { \_xparse_grab_u_aux:nnN {#1} {#2} \cs_set_protected_nopar:Npn }
871 \cs_new_protected:Npn \_xparse_grab_u_long:w #1#2 \l__xparse_args_tl
872 { \_xparse_grab_u_aux:nnN {#1} {#2} \cs_set_protected:Npn }
873 \cs_new_protected:Npn \_xparse_grab_u_aux:nnN #1#2#3
874 {
875     \exp_after:wN #3 \l__xparse_fn_tl ##1 #1

```

```

876     {
877       \__xparse_add_arg:n {##1}
878       #2 \l__xparse_args_tl
879     }
880     \l__xparse_fn_tl
881   }

```

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

<pre> __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 </pre>	<p>The opening delimiter is never read verbatim, for consistency: if the preceding argument was optional and absent, then T_EX 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:.</p>
--	--

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.

```

882 \tl_new:N \l__xparse_v_rest_of_signature_tl
883 \tl_new:N \l__xparse_v_arg_tl
884 \cs_new_protected_nopar:Npn \__xparse_grab_v:w
885   {
886     \bool_set_false:N \l__xparse_long_bool
887     \__xparse_grab_v_aux:w
888   }
889 \cs_new_protected_nopar:Npn \__xparse_grab_v_long:w
890   {
891     \bool_set_true:N \l__xparse_long_bool
892     \__xparse_grab_v_aux:w
893   }
894 \cs_new_protected:Npn \__xparse_grab_v_aux:w #1 \l__xparse_args_tl
895   {
896     \tl_set:Nn \l__xparse_v_rest_of_signature_tl {#1}
897     \group_begin:
898     \group_align_safe_begin:
899     \tex_escapechar:D = 92 \scan_stop:
900     \tl_clear:N \l__xparse_v_arg_tl
901     \peek_N_type:TF
902     { \__xparse_grab_v_aux_test:N }
903     {
904       \peek_meaning_remove:NTF \c_group_begin_token
905       {
906         \group_align_safe_end:
907         \__xparse_grab_v_bgroup:
908       }
909       { \__xparse_grab_v_aux_abort: }
910     }
911   }
912 \cs_new_protected_nopar:Npn \__xparse_grab_v_group_end:
913   {

```

```

914         \group_align_safe_end:
915         \exp_args:NNNo
916         \group_end:
917         \tl_set:Nn \l__xparse_v_arg_tl { \l__xparse_v_arg_tl }
918     }

```

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

`__xparse_grab_v_aux_test:N` 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, `__xparse_grab_v_aux_loop:N` we will be grabbing one character at each step. Unfortunately, it can happen that what `__xparse_grab_v_aux_loop:NN` follows the verbatim argument is already tokenized. Thus, we check at each step that `__xparse_grab_v_aux_loop_end:` 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.

```

919 \cs_new_protected:Npn \__xparse_grab_v_aux_test:N #1
920 {
921     \__xparse_grab_v_aux_put:N #1
922     \__xparse_grab_v_token_if_char:NTF #1
923     {
924         \__xparse_grab_v_aux_catcodes:
925         \__xparse_grab_v_aux_loop:N #1
926     }
927     { \__xparse_grab_v_aux_abort: }
928 }
929 \cs_new_protected:Npn \__xparse_grab_v_aux_loop:N #1
930 {
931     \peek_N_type:TF
932     { \__xparse_grab_v_aux_loop:NN #1 }
933     { \__xparse_grab_v_aux_abort: }
934 }
935 \cs_new_protected:Npn \__xparse_grab_v_aux_loop:NN #1 #2
936 {
937     \__xparse_grab_v_token_if_char:NTF #2
938     {
939         \token_if_eq_charcode:NNTF #1 #2
940         { \__xparse_grab_v_aux_loop_end: }
941         {
942             \__xparse_grab_v_aux_put:N #2
943             \__xparse_grab_v_aux_loop:N #1
944         }
945     }
946     { \__xparse_grab_v_aux_abort: #2 }
947 }
948 \cs_new_protected_nopar:Npn \__xparse_grab_v_aux_loop_end:
949 {

```

```

950 \__xparse_grab_v_group_end:
951 \exp_args:Nx \__xparse_add_arg:n { \tl_tail:N \l__xparse_v_arg_tl }
952 \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
953 }

```

(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.

\l__xparse_v_nesting_int

```

954 \int_new:N \l__xparse_v_nesting_int
955 \cs_new_protected_nopar:Npx \__xparse_grab_v_bgroup:
956 {
957   \exp_not:N \__xparse_grab_v_aux_catcodes:
958   \exp_not:n { \int_set_eq:NN \l__xparse_v_nesting_int \c_one }
959   \exp_not:N \__xparse_grab_v_aux_put:N \iow_char:N \{
960   \exp_not:N \__xparse_grab_v_bgroup_loop:
961 }
962 \cs_new_protected:Npn \__xparse_grab_v_bgroup_loop:
963 {
964   \peek_N_type:TF
965   { \__xparse_grab_v_bgroup_loop:N }
966   { \__xparse_grab_v_aux_abort: }
967 }
968 \cs_new_protected:Npn \__xparse_grab_v_bgroup_loop:N #1
969 {
970   \__xparse_grab_v_token_if_char:NTF #1
971   {
972     \token_if_eq_charcode:NNTF \c_group_end_token #1
973     {
974       \int_decr:N \l__xparse_v_nesting_int
975       \int_compare:nNnTF \l__xparse_v_nesting_int > \c_zero
976       {
977         \__xparse_grab_v_aux_put:N #1
978         \__xparse_grab_v_bgroup_loop:
979       }
980       { \__xparse_grab_v_aux_loop_end: }
981     }
982     {
983       \token_if_eq_charcode:NNTF \c_group_begin_token #1
984       { \int_incr:N \l__xparse_v_nesting_int }
985       \__xparse_grab_v_aux_put:N #1
986       \__xparse_grab_v_bgroup_loop:
987     }
988   }
989   { \__xparse_grab_v_aux_abort: #1 }

```

```

990 }
(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_-
\__xparse_grab_v_aux_abort: special_chars_seq, and we use \dospecials in package mode. The approach for short
\__xparse_grab_v_aux_abort:w verbatim arguments is to make the end-line character a macro parameter character: this
is forbidden by the rest of the code. Then the error branch can check what caused the
bail out and give the appropriate error message.

991 \cs_new_protected_nopar:Npn \__xparse_grab_v_aux_catcodes:
992 {
993   \*initex
994     \seq_map_function:NN
995       \c__xparse_special_chars_seq
996       \char_set_catcode_other:N
997   \*initex
998   \*package
999     \cs_set_eq:NN \do \char_set_catcode_other:N
1000   \dospecials
1001 \*package
1002   \tex_endlinechar:D = ‘^^M \scan_stop:
1003   \bool_if:NTF \l__xparse_long_bool
1004     { \char_set_catcode_other:n { \tex_endlinechar:D } }
1005     { \char_set_catcode_parameter:n { \tex_endlinechar:D } }
1006 }
1007 \cs_new_protected_nopar:Npn \__xparse_grab_v_aux_abort:
1008 {
1009   \__xparse_grab_v_group_end:
1010   \__xparse_add_arg:o \c__xparse_no_value_tl
1011   \exp_after:wN \__xparse_grab_v_aux_abort:w \l__xparse_args_tl \q_stop
1012 }
1013 \cs_new_protected:Npn \__xparse_grab_v_aux_abort:w #1 #2 \q_stop
1014 {
1015   \group_begin:
1016   \char_set_lccode:nn { ‘\# } { \tex_endlinechar:D }
1017   \tl_to_lowercase:n
1018     { \group_end: \peek_meaning_remove:NTF ## }
1019     {
1020       \__msg_kernel_error:nxxx { xparse } { verbatim-newline }
1021       { \token_to_str:N #1 }
1022       { \tl_to_str:N \l__xparse_v_arg_tl }
1023       \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
1024     }
1025     {
1026       \__msg_kernel_error:nxxx { xparse } { verbatim-already-tokenized }
1027       { \token_to_str:N #1 }
1028       { \tl_to_str:N \l__xparse_v_arg_tl }
1029       \l__xparse_v_rest_of_signature_tl \l__xparse_args_tl
1030     }
1031 }

```


(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).

```

1032 \cs_new_protected:Npn \_xparse_grab_v_aux_put:N #1
1033 {
1034   \tl_put_right:Nx \l__xparse_v_arg_tl
1035   {
1036     \token_if_active:NTF #1
1037     { \exp_not:N #1 } { \token_to_str:N #1 }
1038   }
1039 }

```

(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.

```

1040 \cs_new_protected:Npn \_xparse_grab_v_token_if_char:NTF #1
1041 { \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 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.

_xparse_add_arg:V
_xparse_add_arg:o
_xparse_add_arg_aux:n
_xparse_add_arg_aux:V

```

1042 \cs_new_protected:Npn \_xparse_add_arg:n #1
1043 {
1044   \int_compare:nNnTF \l__xparse_processor_int = \c_zero
1045   { \tl_put_right:Nn \l__xparse_args_tl { {#1} } }
1046   {
1047     \tl_clear:N \ProcessedArgument
1048     \_xparse_if_no_value:NTF {#1}
1049     {
1050       \int_zero:N \l__xparse_processor_int
1051       \tl_put_right:Nn \l__xparse_args_tl { {#1} }
1052     }
1053     { \_xparse_add_arg_aux:n {#1} }
1054   }
1055 }
1056 \cs_generate_variant:Nn \_xparse_add_arg:n { V , o }
1057 \cs_new_protected:Npn \_xparse_add_arg_aux:n #1
1058 {
1059   \use:c { __xparse_processor_ \int_use:N \l__xparse_processor_int :n } {#1}
1060   \int_decr:N \l__xparse_processor_int
1061   \int_compare:nNnTF \l__xparse_processor_int = \c_zero

```

```

1062     {
1063         \tl_put_right:Nx \l__xparse_args_tl
1064         { { \exp_not:V \ProcessedArgument } }
1065     }
1066     { \__xparse_add_arg_aux:V \ProcessedArgument }
1067 }
1068 \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:NNNwN
\__xparse_expandable_grab_D:NNNwNnn
\__xparse_expandable_grab_D:Nw
\__xparse_expandable_grab_D: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.

```

1069 \cs_new:Npn \__xparse_expandable_grab_D:w #1 \q__xparse #2
1070   { #2 { \__xparse_expandable_grab_D:NNNwNn #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 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>`.

```

1071 \cs_new:Npn \__xparse_expandable_grab_D:NNNwNn #1#2#3#4#5 \q__xparse #6#7
1072   {
1073     \str_if_eq:onTF
1074     { #1 { } { } #7 #2 \q__xparse #3 }

```

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 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.

```

1075     { { } { #2 } { } }
1076     {
1077         #1
1078         { \__xparse_expandable_grab_D:NNNwNnnn #1#2#3#5 \q__xparse #6 }
1079         \q_nil { } #2 \ERROR \q__xparse \ERROR
1080     }
1081     { #5 {#4} \q__xparse #6 {#7} }
1082 }

```

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.

```

1083 \cs_new:Npn \__xparse_expandable_grab_D:NNNwNnnn #1#2#3#4 \q__xparse #5#6#7#8
1084 {
1085     \exp_args:Nof \__xparse_expandable_grab_D:nnNNNwN
1086     { \use_ii:nn #6 #2 }
1087     { \__xparse_expandable_grab_D:Nw #3 \exp_stop_f: #7 #8 }
1088     #1#2#3 #4 \q__xparse #5
1089 }
1090 \cs_new:Npn \__xparse_expandable_grab_D: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 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:NNNwNnnn`. The code to deal with brace stripping is much the same as for the non-expandable case.

```

1091 \cs_new:Npn \__xparse_expandable_grab_D:nnNNNwN #1#2#3#4#5#6 \q__xparse #7
1092 {
1093     \exp_args:No \tl_if_empty:oTF
1094     { #3 { \use_none:nnn } #2 \q__xparse #5 #4 \q__xparse #5 }
1095     {
1096         \tl_if_blank:oTF { \use_none:nn #1#2 }
1097         { \__xparse_put_arg_expandable:ow { } }
1098         {
1099             \str_if_eq_x:nnTF
1100             { \exp_not:o { \use_none:nn #1#2 } }
1101             { { \exp_not:o { \use_iii:nnnn #1#2 \q_nil } } }
1102             { \__xparse_put_arg_expandable:ow { \use_iii:nnn #1#2 } }
1103             { \__xparse_put_arg_expandable:ow { \use_none:nn #1#2 } }
1104         }
1105         #6 \q__xparse #7
1106     }
1107 }
1108 {
1109     #3
1110     { \__xparse_expandable_grab_D:NNNwNnnn #3#4#5#6 \q__xparse #7 }
1111     \q_nil {#1} #2 \ERROR \q__xparse \ERROR

```

```

1111     }
1112 }

```

(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:NNnwNn
\_xparse_expandable_grab_D_alt: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.

```

1113 \cs_new:Npn \_xparse_expandable_grab_D_alt:w #1 \q__xparse #2
1114 { #2 { \_xparse_expandable_grab_D_alt:NNnwNn #1 \q__xparse #2 } }
1115 \cs_new:Npn \_xparse_expandable_grab_D_alt:NNnwNn #1#2#3#4 \q__xparse #5#6
1116 {
1117   \str_if_eq:onTF
1118     { #1 { } #6 #2 #2 }
1119     { { } #2 }
1120     {
1121       #1
1122       { \_xparse_expandable_grab_D_alt:Nwn #5 #4 \q__xparse }
1123       #6 \ERROR
1124     }
1125     { #4 {#3} \q__xparse #5 {#6} }
1126 }
1127 \cs_new:Npn \_xparse_expandable_grab_D_alt:Nwn #1#2 \q__xparse #3
1128 {
1129   \tl_if_blank:oTF { \use_none:nn #1#2 }
1130   { \_xparse_put_arg_expandable:ow { } }
1131   {
1132     \str_if_eq:x:nnTF
1133       { \exp_not:o { \use_none:n #3 } }
1134       { { \exp_not:o { \use_ii:nnn #3 \q_nil } } }
1135       { \_xparse_put_arg_expandable:ow { \use_ii:nn #3 } }
1136       { \_xparse_put_arg_expandable:ow { \use_none:n #3 } }
1137     }
1138     #2 \q__xparse #1
1139 }

```

(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.

```

1140 \cs_new:Npn \_xparse_expandable_grab_m:w #1 \q__xparse #2
1141 { #2 { \_xparse_expandable_grab_m_aux:wNn #1 \q__xparse #2 } }
1142 \cs_new:Npn \_xparse_expandable_grab_m_aux:wNn #1 \q__xparse #2#3
1143 { #1 {#3} \q__xparse #2 }

```

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

```

\_xparse_expandable_grab_R:w
\_xparse_expandable_grab_R_aux:NNnwNn

```

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

```

1144 \cs_new:Npn \_xparse_expandable_grab_R:w #1 \q__xparse #2
1145 { #2 { \_xparse_expandable_grab_R_aux:NNnwNn #1 \q__xparse #2 } }
1146 \cs_new:Npn \_xparse_expandable_grab_R_aux:NNnwNn #1#2#3#4#5 \q__xparse #6#7
1147 {
1148   \str_if_eq:onTF

```

```

1149     { #1 { } { } #7 #2 \q__xparse #3 }
1150     { { } { #2 } { } }
1151     {
1152         #1
1153         { \__xparse_expandable_grab_D:NNwNnnn #1#2#3#5 \q__xparse #6 }
1154         \q_nil { } #2 \ERROR \q__xparse \ERROR
1155     }
1156     {
1157         \__msg_kernel_expandable_error:nnn
1158         { xparse } { missing-required } {#2}
1159         #5 {#4} \q__xparse #6 {#7}
1160     }
1161 }

```

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

__xparse_expandable_grab_R_alt:w
__xparse_expandable_grab_R_alt_aux:NNwNn

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.

```

1162 \cs_new:Npn \__xparse_expandable_grab_R_alt:w #1 \q__xparse #2
1163 { #2 { \__xparse_expandable_grab_R_alt_aux:NNwNn #1 \q__xparse #2 } }
1164 \cs_new:Npn \__xparse_expandable_grab_R_alt_aux:NNwNn #1#2#3#4 \q__xparse #5#6
1165 {
1166     \str_if_eq:onTF
1167     { #1 { } #6 #2 #2 }
1168     { { } #2 }
1169     {
1170         #1
1171         { \__xparse_expandable_grab_D_alt:Nwn #5 #4 \q__xparse }
1172         #6 \ERROR
1173     }
1174     {
1175         \__msg_kernel_expandable_error:nnn
1176         { xparse } { missing-required } {#2}
1177         #4 {#3} \q__xparse #5 {#6}
1178     }
1179 }

```

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

__xparse_expandable_grab_t:w
__xparse_expandable_grab_t_aux:NNwn

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

```

1180 \cs_new:Npn \__xparse_expandable_grab_t:w #1 \q__xparse #2
1181 { #2 { \__xparse_expandable_grab_t_aux:NNwn #1 \q__xparse #2 } }
1182 \cs_new:Npn \__xparse_expandable_grab_t_aux:NNwn #1#2#3 \q__xparse #4#5
1183 {
1184     \str_if_eq:onTF { #1 { } #5 #2 } { #2 }
1185     { #3 { \BooleanTrue } \q__xparse #4 }
1186     { #3 { \BooleanFalse } \q__xparse #4 {#5} }
1187 }

```

(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`

```

1188 \cs_new:Npn \_xparse_put_arg_expandable:nw #1#2 \q__xparse { #2 {#1} \q__xparse }
1189 \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.

```

1190 \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.

```

1191 \cs_new_protected:Npn \_xparse_process_arg:n #1
1192 {
1193   \int_incr:N \l__xparse_processor_int
1194   \cs_set:cpn { __xparse_processor_ \int_use:N \l__xparse_processor_int :n } ##1
1195   { #1 {##1} }
1196 }

```

(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.

```

1197 \cs_new_protected:Npn \_xparse_process_to_str:n #1
1198 { \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.

```

1199 \cs_new_protected:Npn \_xparse_bool_reverse:N #1
1200 {
1201   \bool_if:NTF #1
1202   { \tl_set:Nn \ProcessedArgument { \c_false_bool } }
1203   { \tl_set:Nn \ProcessedArgument { \c_true_bool } }
1204 }

```

(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
`\l__xparse_split_list_tl` single active tokens, a two-part procedure is needed.
`_xparse_split_list:nn`

```

1205 \seq_new:N \l__xparse_split_list_seq
1206 \tl_new:N \l__xparse_split_list_tl
1207 \cs_new_protected:Npn \_xparse_split_list:nn #1#2
1208 {
1209   \bool_if:NTF
1210   {
1211     \tl_if_single_p:n {#1} &&
1212     ! ( \token_if_cs_p:N #1 )
1213   }
1214   { \_xparse_split_list_single:Nn #1 {#2} }

```

```

1215     { \_xparse_split_list_multi:nn {#1} {#2} }
1216   }
1217 \cs_set_protected:Npn \_xparse_split_list_multi:nn #1#2
1218 {
1219   \seq_set_split:Nnn \l__xparse_split_list_seq {#1} {#2}
1220   \tl_clear:N \ProcessedArgument
1221   \seq_map_inline:Nn \l__xparse_split_list_seq
1222     { \tl_put_right:Nn \ProcessedArgument { {##1} } }
1223 }
1224 \cs_generate_variant:Nn \_xparse_split_list_multi:nn { nV }
1225 \group_begin:
1226 \char_set_catcode_active:N \@
1227 \cs_new_protected:Npn \_xparse_split_list_single:Nn #1#2
1228 {
1229   \tl_set:Nn \l__xparse_split_list_tl {#2}
1230   \group_begin:
1231   \char_set_lccode:nn { '\@ } { '#1 }
1232   \tl_to_lowercase:n
1233   {
1234     \group_end:
1235     \tl_replace_all:Nnn \l__xparse_split_list_tl { @ } {#1}
1236   }
1237   \_xparse_split_list_multi:nV {#1} \l__xparse_split_list_tl
1238 }
1239 \group_end:

```

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

_xparse_split_argument:nnn

_xparse_split_argument_aux:nnnn

_xparse_split_argument_aux:n

_xparse_split_argument_aux:wn

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

```

1240 \cs_new_protected:Npn \_xparse_split_argument:nnn #1#2#3
1241 {
1242   \_xparse_split_list:nn {#2} {#3}
1243   \exp_args:Nf \_xparse_split_argument_aux:nnnn
1244     { \tl_count:N \ProcessedArgument }
1245     {#1} {#2} {#3}
1246 }
1247 \cs_new_protected:Npn \_xparse_split_argument_aux:nnnn #1#2#3#4
1248 {
1249   \int_compare:nNnF {#1} = { #2 + \c_one }
1250   {
1251     \int_compare:nNnTF {#1} > { #2 + \c_one }
1252     {
1253       \tl_set:Nx \ProcessedArgument
1254       {
1255         \exp_last_unbraced:NnNo
1256         \_xparse_split_argument_aux:n

```

```

1257         { #2 + \c_one }
1258         \use_none_delimit_by_q_stop:w
1259         \ProcessedArgument
1260         \q_stop
1261     }
1262     \__msg_kernel_error:nnxxx { xparse } { split-excess-tokens }
1263     { \tl_to_str:n {#3} } { \int_eval:n { #2 + \c_one } }
1264     { \tl_to_str:n {#4} }
1265 }
1266 {
1267     \tl_put_right:Nx \ProcessedArgument
1268     {
1269         \prg_replicate:nn { #2 + \c_one - (#1) }
1270         { { \exp_not:V \c__xparse_no_value_tl } }
1271     }
1272 }
1273 }
1274 }

```

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

```

1275 \cs_new:Npn \__xparse_split_argument_aux:n #1
1276   { \prg_replicate:nn {#1} { \__xparse_split_argument_aux:wn } }
1277 \cs_new:Npn \__xparse_split_argument_aux:wn #1 \use_none_delimit_by_q_stop:w #2
1278   {
1279     \exp_not:n { {#2} }
1280     #1
1281     \use_none_delimit_by_q_stop:w
1282   }

```

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

__xparse_trim_spaces:n This one is almost trivial.

```

1283 \cs_new_protected:Npn \__xparse_trim_spaces:n #1
1284   { \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
1285 \cs_new_protected:Npn \__xparse_get_arg_spec:N #1
1286   {
1287     \prop_get:NnNF \l__xparse_command_arg_specs_prop {#1}
1288     \ArgumentSpecification
1289     {
1290       \__msg_kernel_error:nnx { xparse } { unknown-document-command }
1291       { \token_to_str:N #1 }
1292     }
1293   }
1294 \cs_new_protected:Npn \__xparse_get_arg_spec:n #1
1295   {

```



```

1296 \prop_get:NnNF \l__xparse_environment_arg_specs_prop {#1}
1297 \ArgumentSpecification
1298 {
1299   \__msg_kernel_error:nxx { xparse } { unknown-document-environment }
1300   { \tl_to_str:n {#1} }
1301 }
1302 }
1303 \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.

```

\__xparse_show_arg_spec:n
1304 \cs_new_protected:Npn \__xparse_show_arg_spec:N #1
1305 {
1306   \prop_get:NnNTF \l__xparse_command_arg_specs_prop {#1}
1307   \ArgumentSpecification
1308   { \tl_show:N \ArgumentSpecification }
1309   {
1310     \__msg_kernel_error:nxx { xparse } { unknown-document-command }
1311     { \token_to_str:N #1 }
1312   }
1313 }
1314 \cs_new_protected:Npn \__xparse_show_arg_spec:n #1
1315 {
1316   \prop_get:NnNTF \l__xparse_environment_arg_specs_prop {#1}
1317   \ArgumentSpecification
1318   { \tl_show:N \ArgumentSpecification }
1319   {
1320     \__msg_kernel_error:nxx { xparse } { unknown-document-environment }
1321     { \tl_to_str:n {#1} }
1322   }
1323 }
(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.

```

1324 \group_begin:
1325 \char_set_lccode:nn { \Q } { \- }
1326 \char_set_lccode:nn { \F } { \F }
1327 \char_set_lccode:nn { \N } { \N }
1328 \char_set_lccode:nn { \T } { \T }
1329 \char_set_lccode:nn { \V } { \V }
1330 \tl_to_lowercase:n
1331 {
1332   \group_end:
1333   \prg_new_conditional:Npnn \__xparse_if_no_value:n #1 { T , F , TF }
1334   {

```

```

1335     \str_if_eq:onTF
1336     { \__xparse_if_value_aux:w ? #1 { } QNoValue- }
1337     { ? { } QNoValue- }
1338     { \prg_return_true: }
1339     { \prg_return_false: }
1340   }
1341   \cs_new:Npn \__xparse_if_value_aux:w #1 QNoValue- { #1 }
1342 }

```

(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.

```

1343 \__msg_kernel_new:nnnn { xparse } { bad-arg-spec }
1344 { Bad~argument~specification~'#1'. }
1345 {
1346   \c_msg_coding_error_text_tl
1347   The~argument~specification~provided~was~not~valid:~
1348   one~or~more~mandatory~pieces~of~information~were~missing. \\\
1349   LaTeX~will~ignore~this~entire~definition.
1350 }
1351 \__msg_kernel_new:nnnn { xparse } { command-already-defined }
1352 { Command~'#1'~already~defined! }
1353 {
1354   You~have~used~\NewDocumentCommand
1355   with~a~command~that~already~has~a~definition. \\\
1356   The~existing~definition~of~'#1'~will~be~overwritten.
1357 }
1358 \__msg_kernel_new:nnnn { xparse } { command-not-yet-defined }
1359 { Command ~'#1'~not~yet~defined! }
1360 {
1361   You~have~used~\RenewDocumentCommand
1362   with~a~command~that~was~never~defined.\\
1363   A~new~command~'#1'~will~be~created.
1364 }
1365 \__msg_kernel_new:nnnn { xparse } { environment-already-defined }
1366 { Environment~'#1'~already~defined! }
1367 {
1368   You~have~used~\NewDocumentEnvironment
1369   with~an~environment~that~already~has~a~definition.\\
1370   The~existing~definition~of~'#1'~will~be~overwritten.
1371 }
1372 \__msg_kernel_new:nnnn { xparse } { environment-mismatch }
1373 { Mismatch~between~start~and~end~of~environment. }
1374 {
1375   The~current~environment~is~called~'#1',~but~you~have~tried~to~
1376   end~one~called~'#2'.~Environments~have~to~be~properly~nested.

```

```

1377 }
1378 \_msg_kernel_new:nnnn { xparse } { environment-not-yet-defined }
1379 { Environment~'#1'~not-yet-defined! }
1380 {
1381   You~have~used~\RenewDocumentEnvironment
1382   with~an~environment~that~was~never~defined.\\
1383   A~new~environment~'#1'~will~be~created.
1384 }
1385 \_msg_kernel_new:nnnn { xparse } { environment-unknown }
1386 { Environment~'#1'~undefined. }
1387 {
1388   You~have~tried~to~start~an~environment~called~'#1',~
1389   but~this~has~never~been~defined.\\
1390   The~command~will~be~ignored.
1391 }
1392 \_msg_kernel_new:nnnn { xparse } { expandable-ending-optional }
1393 { Argument~specification~for~expandable~command~ends~with~optional~argument. }
1394 {
1395   \c_msg_coding_error_text_tl
1396   Expandable~commands~must~have~a~final~mandatory~argument~
1397   (or~no~arguments~at~all).~You~cannot~have~a~terminal~optional~
1398   argument~with~expandable~commands.
1399 }
1400 \_msg_kernel_new:nnnn { xparse } { inconsistent-long }
1401 { Inconsistent~long~arguments~for~expandable~command. }
1402 {
1403   \c_msg_coding_error_text_tl
1404   The~arguments~for~an~expandable~command~must~either~all~be~
1405   short~or~all~be~long.~You~have~tried~to~mix~the~two~types.
1406 }
1407 \_msg_kernel_new:nnnn { xparse } { invalid-expandable-argument-type }
1408 { Argument~type~'#1'~not~available~for~an~expandable~function. }
1409 {
1410   \c_msg_coding_error_text_tl
1411   The~letter~'#1'~does~not~specify~an~argument~type~which~can~be~used~
1412   in~an~expandable~function.
1413   \\ \\
1414   LaTeX~will~assume~you~want~a~standard~mandatory~argument~(type~'m').
1415 }
1416 \_msg_kernel_new:nnnn { xparse } { missing-required }
1417 { Failed~to~find~required~argument~starting~with~'#1'. }
1418 {
1419   There~is~supposed~to~be~an~argument~to~the~current~function~starting~with~
1420   '#1'.~LaTeX~did~not~find~it,~and~will~insert~'#2'~as~the~value~to~be~
1421   processed.
1422 }
1423 \_msg_kernel_new:nnnn { xparse } { not-single-token }
1424 { Argument~delimiter~should~be~a~single~token:~'#1'. }
1425 {
1426   \c_msg_coding_error_text_tl

```

```

1427 The~argument~specification~provided~was~not~valid:~
1428 in~a~place~where~a~single~token~is~required,~LaTeX~found~'#1'. \\ \\
1429 LaTeX~will~ignore~this~entire~definition.
1430 }
1431 \_msg_kernel_new:nnnn { xparse } { processor-in-expandable }
1432 { Argument~processors~cannot~be~used~with~expandable~functions. }
1433 {
1434   \c_msg_coding_error_text_tl
1435   The~argument~specification~for~'#1~contains~a~processor~function:~
1436   this~is~only~supported~for~standard~robust~functions.
1437 }
1438 \_msg_kernel_new:nnnn { xparse } { split-excess-tokens }
1439 { Too~many~'#1'~tokens~when~trying~to~split~argument. }
1440 {
1441   LaTeX~was~asked~to~split~the~input~'#3'~
1442   at~each~occurrence~of~the~token~'#1',~up~to~a~maximum~of~#2~parts.~
1443   There~were~too~many~'#1'~tokens.
1444 }
1445 \_msg_kernel_new:nnnn { xparse } { unknown-argument-type }
1446 { Unknown~argument~type~'#1'~replaced~by~'m'. }
1447 {
1448   \c_msg_coding_error_text_tl
1449   The~letter~'#1'~does~not~specify~a~known~argument~type.~
1450   LaTeX~will~assume~you~want~a~standard~mandatory~argument~(type~'m').
1451 }
1452 \_msg_kernel_new:nnnn { xparse } { unknown-document-command }
1453 { Unknown~document~command~'#1'. }
1454 {
1455   You~have~asked~for~the~argument~specification~for~a~command~'#1',~
1456   but~this~is~not~a~document~command.
1457 }
1458 \_msg_kernel_new:nnnn { xparse } { unknown-document-environment }
1459 { Unknown~document~environment~'#1'. }
1460 {
1461   You~have~asked~for~the~argument~specification~for~a~command~'#1',~
1462   but~this~is~not~a~document~environment.
1463 }
1464 \_msg_kernel_new:nnnn { xparse } { verbatim-newline }
1465 { Verbatim~argument~of~'#1~ended~by~end~of~line. }
1466 {
1467   The~verbatim~argument~of~'#1~cannot~contain~more~than~one~line,~but~the~end~
1468   of~the~current~line~has~been~reached.~You~have~probably~forgotten~the~
1469   closing~delimiter.
1470   \\ \\
1471   LaTeX~will~ignore~'#2'.
1472 }
1473 \_msg_kernel_new:nnnn { xparse } { verbatim-already-tokenized }
1474 { Verbatim~command~#1~illegal~in~command~argument. }
1475 {
1476   The~command~#1~takes~a~verbatim~argument.~It~may~not~appear~within~

```

```

1477 the~argument~of~another~function.
1478 \\ \\
1479 LaTeX~will~ignore~'#2'.
1480 }
    Intended more for information.
1481 \__msg_kernel_new:nnn { xparse } { define-command }
1482 {
1483   Defining~document~command~#1~
1484   with~arg.~spec.~'#2'~\msg_line_context:.
1485 }
1486 \__msg_kernel_new:nnn { xparse } { define-environment }
1487 {
1488   Defining~document~environment~'#1'~
1489   with~arg.~spec.~'#2'~\msg_line_context:.
1490 }
1491 \__msg_kernel_new:nnn { xparse } { redefine-command }
1492 {
1493   Redefining~document~command~#1~
1494   with~arg.~spec.~'#2'~\msg_line_context:.
1495 }
1496 \__msg_kernel_new:nnn { xparse } { redefine-environment }
1497 {
1498   Redefining~document~environment~'#1'~
1499   with~arg.~spec.~'#2'~\msg_line_context:.
1500 }

```

2.14 User functions

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

\BooleanFalse
\BooleanTrue

Design-space names for the Boolean values.

```

1501 \cs_new_eq:NN \BooleanFalse \c_false_bool
1502 \cs_new_eq:NN \BooleanTrue \c_true_bool

```

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

\DeclareDocumentCommand
\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand

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

```

1503 \cs_new_protected:Npn \DeclareDocumentCommand #1#2#3
1504 { \__xparse_declare_cmd:Nnn #1 {#2} {#3} }
1505 \cs_new_protected:Npn \NewDocumentCommand #1#2#3
1506 {
1507   \cs_if_exist:NTF #1
1508   {
1509     \__msg_kernel_error:nnx { xparse } { command-already-defined }
1510     { \token_to_str:N #1 }
1511   }
1512   { \__xparse_declare_cmd:Nnn #1 {#2} {#3} }
1513 }
1514 \cs_new_protected:Npn \RenewDocumentCommand #1#2#3

```

```

1515 {
1516   \cs_if_exist:NTF #1
1517   { \__xparse_declare_cmd:Nnn #1 {#2} {#3} }
1518   {
1519     \__msg_kernel_error:nxx { xparse } { command-not-yet-defined }
1520     { \token_to_str:N #1 }
1521   }
1522 }
1523 \cs_new_protected:Npn \ProvideDocumentCommand #1#2#3
1524 { \cs_if_exist:NF #1 { \__xparse_declare_cmd:Nnn #1 {#2} {#3} } }

```

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

\DeclareDocumentEnvironment
\NewDocumentEnvironment
\RenewDocumentEnvironment
\ProvideDocumentEnvironment

Very similar for environments.

```

1525 \cs_new_protected:Npn \DeclareDocumentEnvironment #1#2#3#4
1526 { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
1527 \cs_new_protected:Npn \NewDocumentEnvironment #1#2#3#4
1528 {
1529   \cs_if_exist:cTF {#1}
1530   { \__msg_kernel_error:nxx { xparse } { environment-already-defined } {#1} }
1531   { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
1532 }
1533 \cs_new_protected:Npn \RenewDocumentEnvironment #1#2#3#4
1534 {
1535   \cs_if_exist:cTF {#1}
1536   { \__xparse_declare_env:nnnn {#1} {#2} {#3} {#4} }
1537   { \__msg_kernel_error:nxx { xparse } { environment-not-yet-defined } {#1} }
1538 }
1539 \cs_new_protected:Npn \ProvideDocumentEnvironment #1#2#3#4
1540 { \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.

```

1541 \cs_new_protected:Npn \DeclareExpandableDocumentCommand #1#2#3
1542 { \__xparse_declare_expandable_cmd:Nnn #1 {#2} {#3} }

```

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

\IfBooleanTF

The logical $\langle true \rangle$ and $\langle false \rangle$ 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`.

```

1543 \cs_new_eq:NN \IfBooleanTF \bool_if:NTF
1544 \cs_new_eq:NN \IfBooleanT \bool_if:NT
1545 \cs_new_eq:NN \IfBooleanF \bool_if:NF

```

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

\IfNoValueTF

Simple re-naming.

```

1546 \cs_new_eq:NN \IfNoValueF \__xparse_if_no_value:nF
1547 \cs_new_eq:NN \IfNoValueT \__xparse_if_no_value:nT
1548 \cs_new_eq:NN \IfNoValueTF \__xparse_if_no_value:nTF

```

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

`\IfValueTF` Inverted logic.

```
1549 \cs_set:Npn \IfValueF { \__xparse_if_no_value:nT }
1550 \cs_set:Npn \IfValueT { \__xparse_if_no_value:nF }
1551 \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 7.)
```

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

```
1552 \tl_new:N \ProcessedArgument
(End definition for \ProcessedArgument. This function is documented on page 8.)
```

`\ReverseBoolean` Simple copies.

```
\SplitArgument 1553 \cs_new_eq:NN \ReverseBoolean \__xparse_bool_reverse:N
\SplitList      1554 \cs_new_eq:NN \SplitArgument \__xparse_split_argument:nnn
\TrimSpaces     1555 \cs_new_eq:NN \SplitList \__xparse_split_list:nn
               1556 \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`.

```
1557 \cs_new_eq:NN \ProcessList \tl_map_function:nN
(End definition for \ProcessList. This function is documented on page 9.)
```

`\GetDocumentCommandArgSpec` More simple mappings.

```
\GetDocumentEnvironmentArgSpec 1558 \cs_new_eq:NN \GetDocumentCommandArgSpec \__xparse_get_arg_spec:N
\ShowDocumentCommandArgSpec    1559 \cs_new_eq:NN \GetDocumentEnvironmentArgSpec \__xparse_get_arg_spec:n
\ShowDocumentEnvironmentArgSpec 1560 \cs_new_eq:NN \ShowDocumentCommandArgSpec \__xparse_show_arg_spec:N
                               1561 \cs_new_eq:NN \ShowDocumentEnvironmentArgSpec \__xparse_show_arg_spec:n
(End definition for \GetDocumentCommandArgSpec. This function is documented on page 11.)
```

2.15 Package options

`\l__xparse_options_clist` Key-value option to log information: done by hand to keep dependencies down.

```
\l__xparse_log_bool 1562 \clist_new:N \l__xparse_options_clist
1563 \DeclareOption* { \clist_put_right:NV \l__xparse_options_clist \CurrentOption }
1564 \ProcessOptions \relax
1565 \keys_define:nn { xparse }
1566 {
1567   log-declarations .bool_set:N = \l__xparse_log_bool ,
1568   log-declarations .initial:n = true
1569 }
1570 \keys_set:nV { xparse } \l__xparse_options_clist
1571 \bool_if:NF \l__xparse_log_bool
1572 {
1573   \msg_redirect_module:nnn { LaTeX / xparse } { info } { none }
1574   \msg_redirect_module:nnn { LaTeX / xparse } { warning } { none }
1575 }
(End definition for \l__xparse_options_clist. This function is documented on page ??.)
1576 \</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, 1016
\%	21
\-	6, 1325
\@	1226, 1231
\\	21, 1348, 1355, 1362, 1369, 1382, 1389, 1413, 1428, 1470, 1478
\{	21, 959
\}	21
\^	21, 1002
_	21
_msg_kernel_error:nn	435, 570, 578
_msg_kernel_error:nnx	268, 272, 307, 442, 496, 501, 506, 550, 555, 1290, 1299, 1310, 1320, 1509, 1519, 1530, 1537
_msg_kernel_error:nnxx	836, 1020, 1026
_msg_kernel_error:nnxxx	1262
_msg_kernel_expandable_error:nnn	1157, 1175
_msg_kernel_info:nnxx	63, 151
_msg_kernel_new:nnn	1481, 1486, 1491, 1496
_msg_kernel_new:nnnn	1343, 1351, 1358, 1365, 1372, 1378, 1385, 1392, 1400, 1407, 1416, 1423, 1431, 1438, 1445, 1452, 1458, 1464, 1473
_msg_kernel_warning:nnxx	59, 147
_xparse_add_arg:V	677, 1042
_xparse_add_arg:n	603, 703, 709, 721, 730, 739, 838, 859, 863, 877, 951, 1042, 1042, 1056
_xparse_add_arg:o	617, 622, 623, 1010, 1042
_xparse_add_arg_aux:V	1042, 1066
_xparse_add_arg_aux:n	1042, 1053, 1057, 1068
_xparse_add_expandable_grabber_mandatory	512, 558, 558
_xparse_add_expandable_grabber_optional	456, 460, 525, 529, 535, 558, 566
_xparse_add_expandable_long_check:	558, 560, 568, 575
_xparse_add_expandable_type_+ :w	428
_xparse_add_expandable_type_> :w	440
_xparse_add_expandable_type_D :w	447, 450, 452
_xparse_add_expandable_type_D_aux:NNn	447, 461, 464, 530
_xparse_add_expandable_type_D_aux:Nn	447, 457, 479, 526
_xparse_add_expandable_type_G :w	494, 499
_xparse_add_expandable_type_R :w	516, 519, 521
_xparse_add_expandable_type_d :w	447, 447
_xparse_add_expandable_type_g :w	494, 494
_xparse_add_expandable_type_l :w	504, 504
_xparse_add_expandable_type_m :w	310, 497, 502, 507, 509, 509, 551, 556
_xparse_add_expandable_type_r :w	516, 516
_xparse_add_expandable_type_t :w	533, 533
_xparse_add_expandable_type_u :w	548, 548
_xparse_add_expandable_type_v :w	553, 553
_xparse_add_grabber_mandatory:N	350, 358, 368, 382, 389, 403, 403
_xparse_add_grabber_optional:N	334, 343, 375, 403, 413
_xparse_add_type_+ :w	314
_xparse_add_type_> :w	321
_xparse_add_type_D :w	329, 330, 331
_xparse_add_type_G :w	339, 340, 340
_xparse_add_type_R :w	363, 364, 365
_xparse_add_type_d :w	329, 329
_xparse_add_type_g :w	338, 338
_xparse_add_type_l :w	347, 347
_xparse_add_type_m :w	311, 353, 353
_xparse_add_type_r :w	363, 363
_xparse_add_type_t :w	372, 372
_xparse_add_type_u :w	379, 379
_xparse_add_type_v :w	386, 386

__xparse_bad_arg_spec:wn . . .	201, 210, 217, 224, 236, 244, 251, 256, 271, 271	__xparse_expandable_grab_D:nnNNwN	1069, 1085, 1091
__xparse_bool_reverse:N	1199, 1199, 1553	__xparse_expandable_grab_D:w	1069, 1069
__xparse_break_point:n	82, 85, 85, 266, 271	__xparse_expandable_grab_D_alt:NNwNn	1113, 1114, 1115
__xparse_count_mandatory:N . . .	180, 180, 183, 186, 190, 202, 205, 211, 218, 221, 225, 230, 238, 246, 252, 258	__xparse_expandable_grab_D_alt:Nw	1113
__xparse_count_mandatory_aux:N	191, 193	__xparse_expandable_grab_D_alt:Nwn	1122, 1127, 1171
__xparse_count_type+:w	199	__xparse_expandable_grab_D_alt:w . .	1113, 1113
__xparse_count_type>:w	199	__xparse_expandable_grab_R:w	1144, 1144
__xparse_count_type_D:w	199, 213	__xparse_expandable_grab_R_alt:w . .	1162, 1162
__xparse_count_type_G:w	199, 222	__xparse_expandable_grab_R_alt_aux:NNwNn	1162, 1163, 1164
__xparse_count_type_R:w	240	__xparse_expandable_grab_R_aux:NNNwNn	1145, 1146
__xparse_count_type_d:w	199, 206	__xparse_expandable_grab_R_aux:Nwn	1144
__xparse_count_type_g:w	199, 220	__xparse_expandable_grab_m:w	1140, 1140
__xparse_count_type_m:w	196, 199, 227	__xparse_expandable_grab_m_aux:wNn	1140, 1141, 1142
__xparse_count_type_r:w	232	__xparse_expandable_grab_t:w	1180, 1180
__xparse_count_type_t:w	199, 248	__xparse_expandable_grab_t_aux:Nwn	1180, 1181, 1182
__xparse_count_type_u:w	199, 254	__xparse_flush_m_args:	105, 316, 324, 333, 342, 349, 357, 367, 374, 381, 388, 392, 392
__xparse_declare_cmd:Nnn	45, 45, 1504, 1512, 1517, 1524	__xparse_get_arg_spec:N	1285, 1285, 1558
__xparse_declare_cmd_all_m:Nn	79, 86, 86	__xparse_get_arg_spec:n	1285, 1294, 1559
__xparse_declare_cmd_aux:Nnn	45, 48, 53, 55	__xparse_grab_D:w	583, 583
__xparse_declare_cmd_internal:Nnn	45, 68, 70, 84	__xparse_grab_D_aux:NNnN	597, 599, 607, 832
__xparse_declare_cmd_internal:cnx	45, 161	__xparse_grab_D_aux:NNnnNn	585, 590, 594, 596, 597, 597
__xparse_declare_cmd_mixed:Nn	78, 81, 86, 97	__xparse_grab_D_long:w	583, 588
__xparse_declare_cmd_mixed_aux:Nn	86, 101, 103	__xparse_grab_D_long_trailing:w	583, 595
__xparse_declare_cmd_mixed_expandable:Nn	86, 100, 120	__xparse_grab_D_nested:NNnnN	614, 648, 651
__xparse_declare_env:nnnn	138, 138, 1526, 1531, 1536, 1540	__xparse_grab_D_nested:w	648, 666, 684
__xparse_declare_env_internal:nnnn	138, 157, 159	__xparse_grab_D_trailing:w	583, 593
__xparse_declare_expandable_cmd:Nnn	45, 50, 1542	__xparse_grab_G:w	686, 686
__xparse_expandable_grab_D:NNNwN	1069	__xparse_grab_G_aux:nnNn	686, 688, 693, 696, 698, 699
__xparse_expandable_grab_D:NNNwNn	1070, 1071	__xparse_grab_G_long:w	686, 691
__xparse_expandable_grab_D:NNNwNnn	1069, 1078, 1083, 1109, 1153	__xparse_grab_G_long_trailing:w	686, 697
__xparse_expandable_grab_D:Nw	1069, 1087, 1090	__xparse_grab_G_trailing:w	686, 695
		__xparse_grab_R:w	826, 826

_xparse_grab_R_aux:NNnnN	_xparse_grab_v_bgroup_loop:
..... 826, 827, 829, 830 954, 960, 962, 978, 986
_xparse_grab_R_long:w	_xparse_grab_v_bgroup_loop:N
..... 826, 828 954, 965, 968
_xparse_grab_arg:w 580, 580, 601, 609, 834	_xparse_grab_v_group_end:
_xparse_grab_arg_auxi:w 882, 912, 950, 1009
..... 580, 581	_xparse_grab_v_long:w
_xparse_grab_arg_auxii:w 882, 889
..... 580, 582	_xparse_grab_v_token_if_char:NTF ..
_xparse_grab_expandable_end:wN 922, 937, 970, 1040, 1040
..... 128, 1190, 1190	_xparse_if_no_value:n
_xparse_grab_l:w 1333
..... 713, 713	_xparse_if_no_value:nF
_xparse_grab_l_aux:nN 713, 714, 716, 717 1546, 1550
_xparse_grab_l_long:w	_xparse_if_no_value:nT
..... 713, 715 1547, 1549
_xparse_grab_m:w	_xparse_if_no_value:nTF
..... 726, 726 1048, 1324, 1548, 1551
_xparse_grab_m_1:w	_xparse_if_value_aux:w
..... 744 1336, 1341
_xparse_grab_m_2:w	_xparse_prepare_signature:N
..... 744 273, 281, 283, 292,
_xparse_grab_m_3:w 319, 336, 345, 351, 361, 370, 377,
..... 744 384, 390, 438, 445, 477, 492, 514, 546
_xparse_grab_m_4:w	_xparse_prepare_signature:n
..... 744 74, 273, 273
_xparse_grab_m_5:w	_xparse_prepare_signature_add:N ..
_xparse_grab_m_6:w 273, 295, 298
..... 744	_xparse_prepare_signature_bypass:N
_xparse_grab_m_7:w 273, 286, 288, 327
..... 744	_xparse_process_arg:n .. 326, 1191, 1191
_xparse_grab_m_8:w	_xparse_process_to_str:n .. 1197, 1197
..... 744	_xparse_put_arg_expandable:nw
_xparse_grab_m_long:w 1188, 1188, 1189
..... 726, 735	_xparse_put_arg_expandable:ow 1097,
_xparse_grab_t:w 1102, 1103, 1130, 1135, 1136, 1188
..... 842, 842	_xparse_show_arg_spec:N 1304, 1304, 1560
_xparse_grab_t_aux:NnNn	_xparse_show_arg_spec:n 1304, 1314, 1561
..... 842, 844, 848, 850, 852, 853	_xparse_single_token_check:n
_xparse_grab_t_long:w 208, 209, 215,
..... 842, 847 216, 234, 235, 242, 243, 250, 260, 260
_xparse_grab_t_long_trailing:w ...	_xparse_single_token_check_aux:nwn
..... 842, 851 260, 263, 265
_xparse_grab_t_trailing:w ... 842, 849	_xparse_split_argument:nnn
_xparse_grab_u:w 1240, 1240, 1554
..... 869, 869	_xparse_split_argument_aux:n
_xparse_grab_u_aux:nnN 1240, 1256, 1275
..... 869, 870, 872, 873	_xparse_split_argument_aux:nnnn ..
_xparse_grab_u_long:w 1240, 1243, 1247
..... 869, 871	_xparse_split_argument_aux:wn
_xparse_grab_v:w 1240, 1276, 1277
..... 882, 884	_xparse_split_list:nn
_xparse_grab_v_aux:w 882, 887, 892, 894 1205, 1207, 1242, 1555
_xparse_grab_v_aux_abort:	_xparse_split_list_multi:nV 1205, 1237
..... 909, 927, 933, 946, 966, 989, 991, 1007	
_xparse_grab_v_aux_abort:w	
..... 991, 1011, 1013	
_xparse_grab_v_aux_catcodes:	
..... 924, 957, 991, 991	
_xparse_grab_v_aux_loop:N	
..... 919, 925, 929, 943	
_xparse_grab_v_aux_loop:NN	
..... 919, 932, 935	
_xparse_grab_v_aux_loop_end:	
..... 919, 940, 948, 980	
_xparse_grab_v_aux_put:N	
..... 921, 942, 959, 977, 985, 1032, 1032	
_xparse_grab_v_aux_test:N 902, 919, 919	
_xparse_grab_v_bgroup: .. 907, 954, 955	

_xparse_split_list_multi:nn	\char_set_catcode_parameter:n	... 1005
.....	1205 , 1215 , 1217 , 1224	\char_set_lccode:nn
_xparse_split_list_single:Nn	6, 7, 8, 631 , 639 , 1016 ,
.....	1205 , 1214 , 1227	1231 , 1325 , 1326 , 1327 , 1328 , 1329
_xparse_trim_spaces:n	1283 , 1283 , 1556	\clist_new:N 1562
\~ 21	\clist_put_right:NV 1563
		\cs_generate_from_arg_count:cNnn	...
		106 , 122 , 172
_ 21	\cs_generate_from_arg_count:Ncnn	... 88
		\cs_generate_variant:Nn
		84 , 1056 , 1068 , 1189 , 1224
A		\cs_if_exist:cF 1540
\ArgumentSpecification	.. 1285 , 1288 ,	\cs_if_exist:cTF	... 141 , 144 , 1529 , 1535
	1297 , 1303 , 1307 , 1308 , 1317 , 1318	\cs_if_exist:NF 1524
		\cs_if_exist:NTF 57 , 1507 , 1516
B		\cs_if_exist_use:cF 300
\bool_if:NF 91 , 92 , 434 , 1545 , 1571	\cs_if_free:cTF 195
\bool_if:NT 303 , 408 , 420 , 1544	\cs_new:Npn 684 , 1069 , 1071 , 1083 ,
\bool_if:nT 577	1090 , 1091 , 1113 , 1115 , 1127 , 1140 ,
\bool_if:NTF 77 , 99 , 133 ,	1142 , 1144 , 1146 , 1162 , 1164 , 1180 ,
.....	309 , 466 , 481 , 536 , 1003 , 1201 , 1543	1182 , 1188 , 1190 , 1275 , 1277 , 1341
\bool_if:nTF 355 , 1209	\cs_new_eq:NN
\bool_new:N 23 , 27 , 29 , 38 , 41	85 , 1501 , 1502 , 1543 , 1544 , 1545 ,
\bool_set_false:N 47 ,	1546 , 1547 , 1548 , 1553 , 1554 , 1555 ,
.....	67 , 155 , 275 , 277 , 279 , 285 , 410 ,	1556 , 1557 , 1558 , 1559 , 1560 , 1561
.....	426 , 476 , 491 , 513 , 545 , 563 , 573 , 886	\cs_new_protected:cpn	.. 199 , 321 , 440 ,
\bool_set_true:N	744 , 753 , 763 , 773 , 783 , 793 , 804 , 815
.....	52 , 156 , 317 , 323 , 430 , 432 , 891	\cs_new_protected:Npn
\BooleanFalse	... 7 , 863 , 1186 , 1501 , 1501	55 , 70 , 86 , 97 , 103 , 120 ,
\BooleanTrue	... 7 , 859 , 1185 , 1501 , 1502	138 , 159 , 180 , 186 , 193 , 206 , 213 ,
		222 , 232 , 240 , 248 , 254 , 260 , 265 ,
C		271 , 273 , 288 , 298 , 329 , 331 , 340 ,
\c__xparse_no_value_tl 5 ,	363 , 365 , 372 , 379 , 447 , 452 , 464 ,
.....	12 , 330 , 339 , 364 , 450 , 519 , 1010 , 1270	479 , 516 , 521 , 580 , 581 , 582 , 583 ,
\c__xparse_shorthands_prop	588 , 593 , 595 , 597 , 607 , 651 , 686 ,
.....	14 , 14 , 15 , 16 , 17 , 189 , 291	691 , 695 , 697 , 699 , 713 , 715 , 717 ,
\c__xparse_special_chars_seq	726 , 735 , 826 , 828 , 830 , 842 , 847 ,
.....	18 , 19 , 20 , 995	849 , 851 , 853 , 869 , 871 , 873 , 894 ,
\c_false_bool 1202 , 1501	919 , 929 , 935 , 962 , 968 , 1013 , 1032 ,
\c_group_begin_token 706 , 904 , 983	1040 , 1042 , 1057 , 1191 , 1197 , 1199 ,
\c_group_end_token 972	1207 , 1227 , 1240 , 1247 , 1283 , 1285 ,
\c_msg_coding_error_text_tl	.. 1346 ,	1294 , 1304 , 1314 , 1503 , 1505 , 1514 ,
.....	1395 , 1403 , 1410 , 1426 , 1434 , 1448	1523 , 1525 , 1527 , 1533 , 1539 , 1541
\c_one	431 , 958 , 1249 , 1251 , 1257 , 1263 , 1269	\cs_new_protected_nopar:cpn	204 , 314 , 428
\c_space_tl	\cs_new_protected_nopar:Npn
.....	33 , 107 , 113 , 115 , 123 , 129 , 131 , 136	45 , 50 , 220 , 227 , 283 , 338 ,
\c_true_bool 1203 , 1502	347 , 353 , 386 , 392 , 403 , 413 , 494 ,
\c_zero	... 394 , 421 , 569 , 975 , 1044 , 1061	499 , 504 , 509 , 533 , 548 , 553 , 558 ,
\char_set_catcode_active:N 1226	566 , 575 , 884 , 889 , 912 , 948 , 991 , 1007
\char_set_catcode_other:N 996 , 999	\cs_new_protected_nopar:Npx 955
\char_set_catcode_other:n 1004		

L	
\l__xparse_all_long_bool .. 23 , 23 , 92 , 133 , 275 , 432 , 434 , 466 , 481 , 536 , 577	\l__xparse_options_clist 1562 , 1562 , 1563 , 1570
\l__xparse_args_tl 24 , 24 , 112 , 117 , 166 , 583 , 588 , 593 , 595 , 604 , 625 , 678 , 686 , 691 , 695 , 697 , 704 , 710 , 713 , 715 , 722 , 726 , 731 , 735 , 740 , 744 , 748 , 749 , 753 , 758 , 759 , 763 , 768 , 769 , 773 , 778 , 779 , 783 , 788 , 789 , 793 , 798 , 800 , 804 , 809 , 811 , 815 , 820 , 822 , 826 , 828 , 839 , 842 , 847 , 849 , 851 , 860 , 864 , 869 , 871 , 878 , 894 , 952 , 1011 , 1023 , 1029 , 1045 , 1051 , 1063	\l__xparse_processor_bool 41 , 41 , 279 , 285 , 323 , 355
\l__xparse_command_arg_specs_prop 25 , 25 , 66 , 1287 , 1306	\l__xparse_processor_int 42 , 42 , 111 , 1044 , 1050 , 1059 , 1060 , 1061 , 1193 , 1194
\l__xparse_current_arg_int 26 , 26 , 34 , 75 , 95 , 108 , 124 , 174 , 276 , 294 , 318 , 325 , 431 , 437 , 444	\l__xparse_signature_tl 43 , 43 , 116 , 127 , 280 , 326 , 335 , 344 , 369 , 376 , 383 , 396 , 405 , 415 , 471 , 486 , 540 , 561 , 571
\l__xparse_environment_arg_specs_prop 28 , 28 , 154 , 1296 , 1316	\l__xparse_split_list_seq 1205 , 1205 , 1219 , 1221
\l__xparse_environment_bool 27 , 27 , 67 , 77 , 156	\l__xparse_split_list_tl 1205 , 1206 , 1229 , 1235 , 1237
\l__xparse_expandable_aux_name_tl 30 , 30 , 31 , 469 , 473 , 484 , 488 , 539 , 542	\l__xparse_tmp_tl 44 , 44 , 189 , 190 , 291 , 292 , 669 , 672
\l__xparse_expandable_bool 29 , 29 , 47 , 52 , 91 , 99 , 155 , 303 , 309	\l__xparse_v_arg_tl 882 , 883 , 900 , 917 , 951 , 1022 , 1028 , 1034
\l__xparse_fn_tl 36 , 36 , 114 , 611 , 635 , 643 , 655 , 661 , 671 , 682 , 701 , 707 , 719 , 724 , 728 , 733 , 737 , 742 , 746 , 751 , 755 , 761 , 765 , 771 , 775 , 781 , 785 , 791 , 795 , 802 , 806 , 813 , 817 , 824 , 855 , 867 , 875 , 880	\l__xparse_v_nesting_int 954 , 954 , 958 , 974 , 975 , 984
\l__xparse_function_tl 33 , 37 , 37 , 72 , 107 , 113 , 115 , 123 , 129 , 131 , 136 , 443	\l__xparse_v_rest_of_signature_tl 882 , 882 , 896 , 952 , 1023 , 1029
\l__xparse_log_bool 1562 , 1567 , 1571	log-declarations (option) 11
\l__xparse_long_bool 38 , 38 , 277 , 317 , 355 , 408 , 410 , 420 , 426 , 430 , 476 , 491 , 513 , 545 , 563 , 573 , 577 , 886 , 891 , 1003	
\l__xparse_m_args_int 39 , 39 , 75 , 278 , 360 , 394 , 397 , 399 , 401 , 511	M
\l__xparse_mandatory_args_int 40 , 40 , 182 , 229 , 237 , 245 , 257 , 398 , 399 , 411 , 421 , 564 , 569	\msg_line_context: 1484 , 1489 , 1494 , 1499
\l__xparse_nesting_a_tl 648 , 648 , 653 , 657 , 665 , 675 , 677	\msg_redirect_module:nnn 1573 , 1574
\l__xparse_nesting_b_tl 648 , 649 , 654 , 658 , 667 , 669 , 670 , 676	N
	\N 7 , 1327
	\NewDocumentCommand .. 6 , 1354 , 1503 , 1505
	\NewDocumentEnvironment 6 , 1368 , 1525 , 1527
	O
	log-declarations 11
	P
	\peek_meaning_remove:NTF 904 , 1018
	\peek_meaning_remove_ignore_spaces:NTF 833
	\peek_N_type:TF 901 , 931 , 964
	\prg_new_conditional:Npnn 1333
	\prg_replicate:nn 1269 , 1276
	\prg_return_false: 1339
	\prg_return_true: 1338
	\ProcessedArgument 8 , 1047 , 1064 , 1066 , 1198 , 1202 , 1203 , 1220 , 1222 , 1244 , 1253 , 1259 , 1267 , 1284 , 1552 , 1552

\token_to_str:N	60,	\use_iii:nnnn	1101
	64, 195, 197, 304, 308, 837, 1021,	\use_none:n	85,
	1027, 1037, 1291, 1311, 1510, 1520		616, 617, 620, 623, 657, 685, 1133, 1136
\TrimSpaces	9, <u>1553</u> , 1556	\use_none:nn	1096, 1100, 1103, 1129
U			
\use:c	197, 600, 706, 857, 1059	\use_none:nnn	1094
\use_i:nn	658	\use_none_delimit_by_q_stop:w	
\use_ii:nn	622, 1086, 1135		1258, 1277, 1281
\use_ii:nnn	621, 1134	V	
\use_iii:nnn	1102	\V	8, 1329