The xparse package*
Generic document command parser

The LaTeX3 Project†

2009/11/06

1 Creating document commands

The xparse package provides a high-level interface for producing document-level commands. In that way, it is intended as a replacement for the LaTeX 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`
- `\IfNoValue(TF)` (the need for `\IfValue(TF)` is currently an item of active discussion)x

*This file has version number 1727, last revised 2009/11/06.

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

- `\IfBoolean(TF)`

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

1.1 Specifying arguments

Before introducing the functions used to create document commands, the method for specifying arguments with xparse will be illustrated. In order to allow each argument to be defined independently, xparse does not simply need to know the number of arguments for a function, but also the nature of each one. This is done by constructing an *argument specification*, which defines the number of arguments, the type of each argument and any additional information needed for xparse to read the user input and properly pass it through to internal functions.

The basic form of the argument specifier is a list of letters, where each letter defines a type of argument. As will be described below, some of the types need additional information, such as default values. The argument types can be divided into two, those which define arguments that are mandatory (potentially raising an error if not found) and those which define optional arguments. The mandatory types are:

`m` A standard mandatory argument, which can either be a single token alone or multiple tokens surrounded by curly braces. Regardless of the input, the argument will be passed to the internal code surrounded by a brace pair. This is the xparse type specifier for a normal TEX argument.

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

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

The types which define optional arguments are:

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

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

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

`D` As for `d`, but returns ⟨*default*⟩ if no value is given: `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`. Internally, the `o`, `d` and `O` types are short-cuts to an appropriated-constructed `D` type argument.

`s` An optional star, which will result in a value `\BooleanTrue` if a star is present and `\BooleanFalse` otherwise (as described later).

`t` An optional ⟨*token*⟩, which will result in a value `\BooleanTrue` if ⟨*token*⟩ is present and `\BooleanFalse` otherwise. Given as `t`⟨*token*⟩.

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

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

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

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

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

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

Note that after parsing the input there will be always exactly the same number of brace groups or tokens as the number of letters in the argument specifier.

Two more tokens have a special meaning when creating an argument specifier. First, `+` is used to make an argument long (to accept paragraph tokens). In contrast to LATEX 2_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 1.4.

1.2 Declaring commands and environments

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

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

\DeclareDocumentCommand
\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand

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

This family of commands are used to create a document-level ⟨*function*⟩. The argument specification for the function is given by ⟨*arg spec*⟩, and the function will execute ⟨*code*⟩.

As an example:

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

would be a way to define a `\chapter` command which would essentially behave like the current LaTeX 2ε command (except that it would accept an optional argument even when a `*` was parsed). The `\typesetnormalchapter` could test its first argument for being `\NoValue` to see if an optional argument was present.

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

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

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

\DeclareDocumentEnvironment
\NewDocumentEnvironment
\RenewDocumentEnvironment
\ProvideDocumentEnvironment

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

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

TEXhackers note: When loaded as part of a LaTeX3 format, these, these commands do not create a pair of macros `\`⟨*environment*⟩ and `\end`⟨*environment*⟩. Thus LaTeX3 environments have to be accessed using the `\begin` ... `\end` mechanism. When xparse is loaded as a LaTeX 2_ε package, `\`⟨*environment*⟩ and `\end`⟨*environment*⟩ are defined, as this is necessary to allow the new environment to work!

1.3 Testing special values

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

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

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

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

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

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

`\IfValue`*TF* ⋆ `\IfValueTF {`⟨*argument*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
The reverse form of the `\IfNoValue(TF)` tests are also available as `\IfValue(TF)`. The context will determine which logical form makes the most sense for a given code scenario.

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

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

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

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

1.4 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`. This is initialised as a toks before each processor is called, but may also be set equal to any other variable type:

```
\toks_set:Nn  \ProcessedArgument { <content> }
\toks_set:NV  \ProcessedArgument \LocalVariable
\cs_set_eq:NN \ProcessedArgument \LocalVariable
```

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

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

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

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

The `\xparse_process_comma_split:n` processor splits the grabbed argument at the first comma, returning the two parts of the result in braces. If no comma is found, the second part of the returned value `\NoValue`. This function is intended to aid the processing of co-ordinate pairs. For example, to create a co-ordinate argument which raises an error if not given:

```
\DeclareDocumentCommand \foo
  { >{\xparse_process_comma_split:n} d() } {
    \IfNoValueTF #1 {
      \ERROR
    }{
      \foo_internal:nn #1
    }
  }
```

For the input `\foo(1.1,2.2)`, `#1` here would equal `{1.1}{2.2}`, and so `\foo_internal:nn` receives exactly the correct number of arguments. A similar function which takes an optional co-ordinate pair, could be created as:

```
\DeclareDocumentCommand \foo
  { >{\xparse_process_comma_split:n} D(){0,0} } {
      \foo_internal:nn #1
  }
```

This illustrates that the processor function will be applied to the default value, which therefore includes a comma.

1.5 Separating interface and implementation

One *experimental* idea implemented in xparse is to separate out document command interfaces (the argument specification) from the implementation (code). This

is carried out using a pair of functions, `\DeclareDocumentCommandInterface` and `\DeclareDocumentCommandImplementation`

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

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

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

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

1.6 Fully-expandable document commands

There are *very rare* occasion when it may be useful to create functions using a fully-expandable argument grabber. To support this, xparse can create expandable functions as well as the usual robust ones. This imposes a number of restrictions on the nature of the arguments accepted by a function, and the code it implements. This facility should only be used when *absolutely necessary*; if you do not understand when this might be, *do not use these functions*!

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

This command is used to create a document-level ⟨*function*⟩, which will grab its arguments in a fully-expandable manner. The argument specification for the function is given by ⟨*arg spec*⟩, and the function will execute ⟨*code*⟩. In general, ⟨*code*⟩ will also be fully expandable, although it is possible that this will not be the case (for example, a function for use in a table might expand so that `\omit` is the first non-expandable token).

Parsing arguments expandably imposes a number of restrictions on both the type of arguments that can be read and the error checking available:

- The function must have at least one mandatory argument, and in particular the last argument must be one of the mandatory types (`l`, `m` or `u`).
- All arguments are either short or long: it is not possible to mix short and long argument types.
- The 'optional group' argument types `g` and `G` are not available.

- It is not possible to differentiate between, for example `\foo[` and `\foo{[}`: in both cases the `[` will be interpreted as the start of an optional argument. As a result result, checking for optional arguments is less robust than in the standard version.

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

1.7 Variables and constants

`\c_xparse_shorthands_prop` Shorthands and replacement text: set up at the start of the package, and not be be altered later!

`\l_xparse_arg_toks` Token register used as internal representation of `\ProcessedArgument`. Unlike the later, this register should not be used directly when creating new processors.

`\l_xparse_args_toks` Token register for arguments as they are picked up for passing on to user functions.

`\l_xparse_environment_args_toks` Token register to pass arguments to the end of an environment from the beginning.

`\l_xparse_environment_bool` When creating functions, a short cut can be taken if all of the arguments are of `m` type. The code for environments cannot do that, and so a flag is needed.

`\l_xparse_error_bool` For flagging up errors when making expandable commands.

`\l_xparse_function_tl` Needed to pass along the function name when creating in an expandable manner. This is needed as a series of functions have to be created when making expandable functions. (In contrast, standard robust functions need at most two functions.)

`\l_xparse_last_arg_tl` The last argument type added. As this must be mandatory when creating expandable commands, this variable is needed to enforce this behaviour.

`\l_xparse_long_bool` Flag used to indicate creation of `\long` arguments.

`\l_xparse_m_args_int` Used to enumerate the `m` arguments with no modifications (i.e., neither long nor processed after grabbing).

`\l_xparse_m_only_bool` Flag used to indicate that all arguments are of type `m`, with no no modifications.

`\l_xparse_mandatory_args_int` For counting up all mandatory arguments so that the code can tell when optional arguments come after the last mandatory one. Counts down again as mandatory arguments are added to the signature, so will be zero for any trailing optional arguments.

`\l_xparse_nested_int`
`\l_xparse_nested_toks` Nested optional (delimited) arguments have to be handled by hand: TEX will not count up the token-matching. So an integer is needed to count tokens, and a token register to build up the argument.

`\l_xparse_processor_bool` When converting an argument specification into a signature there is a need to know if there are any argument processors set up. This is used to tell if `m` arguments can simply be counted up or need handling on a one-off basis.

`\l_xparse_processor_int` Each time a processor is set up in the grabber routine, it is stored and the total number of processors is recorded here. Later, the variable is counted back down to use the processors in reverse order to the collection order.

`\l_xparse_signature_toks` For constructing the signature of the function defined. As xparse works through an argument specification, grabber functions are added to this toks for each argument.

`\l_xparse_tmp_tl` Scratch space, used for example to convert shorthand argument types into the full versions.

`\l_xparse_total_args_int` Used to enumerate the total number of arguments (i.e., the number of letters in the argument specification).

`\q_xparse_stop` A private delimiting quark: needed by the expandable function system.

1.8 Internal functions

`\xparse_add_arg:n`
`\xparse_add_arg:V` `\xparse_add_arg:n` ⟨*grabbed arg*⟩

Adds ⟨*grabbed arg*⟩ to the output xparse supplies to the defined ⟨*code*⟩, applying any post-processing that is needed.

`\xparse_add_grabber_mandatory:N`
`\xparse_add_grabber_optional:N` `\xparse_add_grabber_mandatory:N` ⟨*grabber type*⟩

Adds appropriate grabber for ⟨*grabber type*⟩ to the signature being constructed, making it long if necessary. The `optional` version includes a second check to see if space skipping should be on or off.

`\xparse_add_type_+:w`
`\xparse_add_type_>:w`
`\xparse_add_type_d:w`
`\xparse_add_type_D:w`
`\xparse_add_type_g:w`
`\xparse_add_type_G:w`
`\xparse_add_type_l:w`
`\xparse_add_type_m:w`
`\xparse_add_type_t:w`
`\xparse_add_type_u:w` `\xparse_add_type_u:w`

Carry out necessary processes to add given ⟨*type*⟩ of argument to the signature being constructed. Depending on the argument type being added, one or more arguments will be absorbed.

`\xparse_check_and_add:N` `\xparse_check_and_add:N` ⟨*arg spec*⟩

Ensures that ⟨*arg spec*⟩ is valid, and if so adds it to the signature being constructed.

`\xparse_count_mandatory:n`
`\xparse_count_mandatory:N` `\xparse_count_mandatory:N` ⟨*arg spec*⟩

Used to count how many mandatory arguments an argument specification contains. The `n` function carries out the set up, before handing of to the `N` function. This reads one token, and calls the appropriate counter function.

\xparse_count_type_>:w
\xparse_count_type_+:w
\xparse_count_type_d:w
\xparse_count_type_D:w
\xparse_count_type_g:w
\xparse_count_type_G:w
\xparse_count_type_l:w
\xparse_count_type_m:w
\xparse_count_type_t:w
\xparse_count_type_u:w

\xparse_count_type_D:w

Used to count up mandatory arguments: one function for each argument type so that a simple loop can be used. Only the functions for mandatory arguments do any more than call the loop again.

\xparse_declare_cmd:Nnn

\xparse_declare_cmd:Nnn ⟨*function*⟩ {⟨*signature*⟩} {⟨*code*⟩}

Declares ⟨*function*⟩ using ⟨*signature*⟩ for argument definition and ⟨*code*⟩ as expansion.

TEXhackers note: This is the internal name for \DeclareDocumentCommand.

\xparse_declare_cmd_interface:Nnn

\xparse_declare_cmd_interface:Nnn ⟨*function*⟩ {⟨*implementation*⟩} {⟨*signature*⟩}

Declares ⟨*function*⟩ using ⟨*signature*⟩, which should have code stored as ⟨*implementation*⟩.

TEXhackers note: This is the internal name for \DeclareDocumentCommandInterface.

\xparse_declare_cmd_implementation:nNn

\xparse_declare_cmd_implementation:nNn {⟨*implementation*⟩} ⟨*number*⟩ {⟨*code*⟩}

Declares ⟨*code*⟩ taking ⟨*number*⟩ arguments as an ⟨*implementation*⟩, to be accessed using an interface defined elsewhere.

TEXhackers note: This is the internal name for \DeclareDocumentCommandImplementation.

\xparse_declare_env:nnnn

\xparse_declare_env:nnnn {⟨*env*⟩} {⟨*arg spec*⟩} {⟨*start code*⟩} {⟨*end code*⟩}

Declares ⟨*env*⟩ as an environment taking ⟨*arg spec*⟩ arguments at \begin{⟨*env*⟩}. The

⟨*start code*⟩ is executed at the beginning of the environment, and the ⟨*end code*⟩ at the end. Both parts may use the arguments defined by ⟨*arg spec*⟩.

TEXhackers note: This is the internal name for `\DeclareDocumentEnvironment`.

`\xparse_flush_m_args:` `\xparse_flush_m_args:`

Adds an outstanding `m` arguments to the signature.

`\xparse_grab_arg:w` `\xparse_grab_arg:w` ⟨*args*⟩

Function re-defined each time an argument is grabbed to actually do the grabbing. It is this function which will raise an error if an argument runs away.

`\xparse_grab_D:w`
`\xparse_grab_D_long:w`
`\xparse_grab_D_trailing:w`
`\xparse_grab_D_long_trailing:w`
`\xparse_grab_G:w`
`\xparse_grab_G_long:w`
`\xparse_grab_G_trailing:w`
`\xparse_grab_G_long_trailing:w`
`\xparse_grab_l:w`
`\xparse_grab_l_long:w`
`\xparse_grab_m:w`
`\xparse_grab_m_long:w`
`\xparse_grab_m_1:w`
`\xparse_grab_m_2:w`
`\xparse_grab_m_3:w`
`\xparse_grab_m_4:w`
`\xparse_grab_m_5:w`
`\xparse_grab_m_6:w`
`\xparse_grab_m_7:w`
`\xparse_grab_m_8:w`
`\xparse_grab_t:w`
`\xparse_grab_t_long:w`
`\xparse_grab_t_trailing:w`
`\xparse_grab_t_long_trailing:w`
`\xparse_grab_u:w`
`\xparse_grab_u_long:w`
`\xparse_grab_D:w` ⟨*arg data*⟩ `\l_xparse_args_toks`

Argument grabbing functions, which re-arrange other ⟨*arg data*⟩ so that the argument is read correctly. The `trailing` versions do not skip spaces when searching for optional

arguments. For each argument type, the various versions feed the appropriate information to a common auxiliary function which then sets up `\xparse_grab_arg:w` to actually carry out the argument absorption.

`\xparse_if_no_value:nTF` ⋆ `\xparse_if_no_value:nTF {`⟨*arg*⟩`}`
`{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Executes ⟨*true code*⟩ if ⟨*arg*⟩ is equal to the special `\NoValue` marker and ⟨*false code*⟩ otherwise. Provided that the primitive `\(pdf)strcmp` is available, this function is expandable.

`\xparse_prepare_signature:n`
`\xparse_prepare_signature:N` `\xparse_prepare_signature:n {`⟨*arg specs*⟩`}`

Parse one or more ⟨*arg specs*⟩ and convert to an output ⟨*signature*⟩.

`\xparse_process_arg:n` `\xparse_process_arg:n {`⟨*processor*⟩`}`

Sets up code to apply ⟨*processor*⟩ to next grabbed argument.

1.9 Creating expandable commands

`\xparse_exp_add_type_d:w`
`\xparse_exp_add_type_D:w`
`\xparse_exp_add_type_l:w`
`\xparse_exp_add_type_m:w`
`\xparse_exp_add_type_t:w`
`\xparse_exp_add_type_u:w` `\xparse_exp_add_type_u:w {`⟨*delimiter*⟩`}`

Carry out necessary processes to add given ⟨*type*⟩ of argument for an expandable command. Depending on the argument type being added, one or more arguments will be absorbed.

`\xparse_exp_check_and_add:N` `\xparse_exp_check_and_add:N` ⟨*arg spec*⟩

Ensures that ⟨*arg spec*⟩ is valid, and if so adds it to expandable function being constructed.

`\xparse_exp_declare_cmd:Nnn` `\xparse_exp_declare_cmd:Nnn` ⟨*function*⟩ `{`⟨*signature*⟩`}`
`{`⟨*code*⟩`}`

Declares ⟨*function*⟩ using ⟨*signature*⟩ for argument definition and ⟨*code*⟩ as expansion, and creating an expandable command.

TEXhackers note: This is the internal name for `\DeclareExpandableDocumentCommand`.

`\xparse_exp_prepare_function:n`
`\xparse_exp_prepare_function:N` `\xparse_exp_prepare_function:n {⟨arg specs⟩}`

Parse one or more ⟨*arg specs*⟩ and convert to an expandable function.

`\xparse_exp_set:cpx` `\xparse_exp_set:cpx ⟨csname⟩ ⟨parameters⟩ {⟨code⟩}`
An alias for either `\cs_set:cpx` or `\cs_set_nopar:cpx`, depending on the `\long` status of the expandable function.

2 xparse implementation

The usual lead-off: only needed for the package, of course (one day we may have a LaTeX3 kernel).

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\RequirePackage{expl3}
⟨/package⟩
⟨*initex | package⟩
```

2.1 Variables and constants

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

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

`\l_xparse_arg_toks` Token registers for single grabbed argument when post-processing.

```
\toks_new:N \l_xparse_arg_toks
```

`\l_xparse_args_toks` Token registers for grabbed arguments.

```
\toks_new:N \l_xparse_args_toks
```

`\l_xparse_environment_args_toks` Used to pass arguments to the end of an environment.

```
\toks_new:N \l_xparse_environment_args_toks
```

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

```
\bool_new:N \l_xparse_environment_bool
```

`\l_xparse_error_bool` Used to signal an error when creating expandable functions.

```
\bool_new:N \l_xparse_error_bool
```

`\l_xparse_function_tl` When creating expandable functions, the current function name needs to be passed along.

```
\tl_new:N \l_xparse_function_tl
```

`\l_xparse_last_arg_tl` Used when creating expandable arguments.

```
\toks_new:N \l_xparse_last_arg_tl
```

`\l_xparse_long_bool` A flag for `\long` arguments.

```
\bool_new:N \l_xparse_long_bool
```

`\l_xparse_m_args_int` The number of simple `m` arguments is tracked so they can be dumped *en masse*.

```
\int_new:N \l_xparse_m_args_int
```

`\l_xparse_m_only_bool` A flag to indicate that only `m` arguments have been found.

```
\bool_new:N \l_xparse_m_only_bool
```

`\l_xparse_mandatory_args_int` So that trailing optional arguments can be picked up, a count has to be taken of all mandatory arguments. This is then decreased as mandatory arguments are added to the signature, so will be zero only if there are no more mandatory arguments to add.

```
\int_new:N \l_xparse_mandatory_args_int
```

`\l_xparse_nested_int`
`\l_xparse_nested_toks` To deal with nested delimited arguments, the code needs to do some token counting 'by hand'. That requires an integer, and also a token register to store the growing argument collected.

```
\int_new:N \l_xparse_nested_int
\toks_new:N \l_xparse_nested_toks
```

\l_xparse_processor_bool When reading through the argument specifier, a flag is needed to show that a processor has been found for the current argument. This is used when checking how to handle `m` arguments.

```
\bool_new:N \l_xparse_processor_bool
```

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

```
\int_new:N \l_xparse_processor_int
```

\l_xparse_signature_toks Token registers for constructing signatures.

```
\toks_new:N \l_xparse_signature_toks
```

\l_xparse_tmp_tl A general purpose token list variable.

```
\tl_new:N \l_xparse_tmp_tl
```

\l_xparse_total_args_int Thje total number of arguments is used to create the internal function which has a fixed number of arguments.

```
\int_new:N \l_xparse_total_args_int
```

\q_xparse_stop A private quark, used for delimiting arguments when making expandable functions.

```
\quark_new:N \q_xparse_stop
```

2.2 Turning the argument specifier into grabbers

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

```
\cs_new_nopar:Npn \xparse_add_grabber_mandatory:N #1 {
  \toks_put_right:Nx \l_xparse_signature_toks {
    \exp_not:c {
      xparse_grab_ #1 \bool_if:NT \l_xparse_long_bool { _long } :w
    }
  }
  \bool_set_false:N \l_xparse_long_bool
  \int_decr:N \l_xparse_mandatory_args_int
}
\cs_new_nopar:Npn \xparse_add_grabber_optional:N #1 {
```

```
  \toks_put_right:Nx \l_xparse_signature_toks {
    \exp_not:c {
      xparse_grab_ #1
      \bool_if:NT \l_xparse_long_bool { _long }
      \intexpr_compare:nF {
        \l_xparse_mandatory_args_int > \c_zero
      } { _trailing }
      :w
    }
  }
  \bool_set_false:N \l_xparse_long_bool
}
```

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

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

```
\cs_new_nopar:cpn { xparse_add_type_+:w } {
  \xparse_flush_m_args:
  \bool_set_true:N \l_xparse_long_bool
  \bool_set_false:N \l_xparse_m_only_bool
  \int_decr:N \l_xparse_total_args_int
  \xparse_prepare_signature:N
}
```

`\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_aux:N` rather than `\xparse_prepare_signature:N` so that the flag is not reset.

```
\cs_new:cpn { xparse_add_type_>:w } #1 {
  \bool_set_true:N \l_xparse_processor_bool
  \xparse_flush_m_args:
  \int_decr:N \l_xparse_total_args_int
  \toks_put_right:Nn \l_xparse_signature_toks {
    \xparse_process_arg:n {#1}
  }
  \xparse_prepare_signature_aux:N
}
```

`\xparse_add_type_d:w` To save on repeated code, `d` is actually turned into the same grabber as is used by `D`, by putting the `\NoValue` default in the correct place. So there is some simple argument re-arrangement to do. Remember that `#1` and `#2` should be single tokens.

```
\cs_new:Npn \xparse_add_type_d:w #1#2 {
  \xparse_add_type_D:w #1 #2 { \NoValue }
}
```

`\xparse_add_type_D:w` All of the optional delimited arguments are handled internally by the `D` type. At this stage, the two delimiters are stored along with the default value.

```
\cs_new:Npn \xparse_add_type_D:w #1#2#3 {
  \xparse_flush_m_args:
  \xparse_add_grabber_optional:N D
  \toks_put_right:Nn \l_xparse_signature_toks { #1 #2 {#3} }
  \xparse_prepare_signature:N
}
```

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

```
\cs_new_nopar:Npn \xparse_add_type_g:w {
  \xparse_add_type_G:w { \NoValue }
}
```

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

```
\cs_new:Npn \xparse_add_type_G:w #1 {
  \xparse_flush_m_args:
  \xparse_add_grabber_optional:N G
  \toks_put_right:Nn \l_xparse_signature_toks { {#1} }
  \xparse_prepare_signature:N
}
```

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

```
\cs_new_nopar:Npn \xparse_add_type_l:w {
  \xparse_flush_m_args:
  \xparse_add_grabber_mandatory:N l
  \xparse_prepare_signature:N
}
```

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

```
\cs_new_nopar:Npn \xparse_add_type_m:w {
```

```
  \bool_if:nTF {
    \l_xparse_long_bool || \l_xparse_processor_bool
  } {
    \xparse_flush_m_args:
    \xparse_add_grabber_mandatory:N m
  }{
    \int_incr:N \l_xparse_m_args_int
  }
  \xparse_prepare_signature:N
}
```

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

```
\cs_new:Npn \xparse_add_type_t:w #1 {
  \xparse_flush_m_args:
  \xparse_add_grabber_optional:N t
  \toks_put_right:Nn \l_xparse_signature_toks { #1 }
  \xparse_prepare_signature:N
}
```

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

```
\cs_new:Npn \xparse_add_type_u:w #1 {
  \xparse_flush_m_args:
  \xparse_add_grabber_mandatory:N u
  \toks_put_right:Nn \l_xparse_signature_toks { {#1} }
  \xparse_prepare_signature:N
}
```

`\xparse_check_and_add:N` This function checks if the argument type actually exists and gives an error if it doesn't.

```
\cs_new_nopar:Npn \xparse_check_and_add:N  #1 {
  \cs_if_free:cTF { xparse_add_type_ #1 :w } {
    \msg_kernel_error:nnx { xparse } { unknown-argument-type } {#1}
    \xparse_add_type_m:w
  }{
    \use:c { xparse_add_type_ #1 :w }
  }
}
```

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

```
\cs_new:Npn \xparse_count_mandatory:n #1 {
  \int_zero:N \l_xparse_mandatory_args_int
  \xparse_count_mandatory:N #1 \q_nil
}
\cs_new:Npn \xparse_count_mandatory:N #1 {
  \quark_if_nil:NF #1 {
    \prop_if_in:NnTF \c_xparse_shorthands_prop {#1} {
      \prop_get:NnN \c_xparse_shorthands_prop {#1} \l_xparse_tmp_tl
      \exp_last_unbraced:NV \xparse_count_mandatory:N \l_xparse_tmp_tl
    }{
      \xparse_count_mandatory_aux:N #1
    }
  }
}
\cs_new:Npn \xparse_count_mandatory_aux:N #1 {
  \cs_if_free:cTF { xparse_count_type_ #1 :w } {
    \xparse_count_type_m:w
  }{
    \use:c { xparse_count_type_ #1 :w }
  }
}
```

```
\xparse_count_type_>:w
\xparse_count_type_+:w
\xparse_count_type_d:w
\xparse_count_type_D:w
\xparse_count_type_g:w
\xparse_count_type_G:w
\xparse_count_type_l:w
\xparse_count_type_m:w
\xparse_count_type_t:w
\xparse_count_type_u:w
```

For counting the mandatory arguments, a function is provided for each argument type that will mop any extra arguments and call the loop function. Only the counting functions for mandatory arguments actually do anything: the rest are simply there to ensure the loop continues correctly.

```
\cs_new:cpn { xparse_count_type_>:w } #1 {
  \xparse_count_mandatory:N
}
\cs_new_nopar:cpn { xparse_count_type_+:w } {
  \xparse_count_mandatory:N
}
\cs_new:Npn \xparse_count_type_d:w #1#2 {
  \xparse_count_mandatory:N
}
\cs_new:Npn \xparse_count_type_D:w #1#2#3 {
  \xparse_count_mandatory:N
}
\cs_new_nopar:Npn \xparse_count_type_g:w {
  \xparse_count_mandatory:N
}
\cs_new:Npn \xparse_count_type_G:w #1 {
  \xparse_count_mandatory:N
}
\cs_new_nopar:Npn \xparse_count_type_l:w {
  \int_incr:N \l_xparse_mandatory_args_int
  \xparse_count_mandatory:N
}
\cs_new_nopar:Npn \xparse_count_type_m:w {
```

```
  \int_incr:N \l_xparse_mandatory_args_int
  \xparse_count_mandatory:N
}
\cs_new:Npn \xparse_count_type_t:w #1 {
  \xparse_count_mandatory:N
}
\cs_new:Npn \xparse_count_type_u:w #1 {
  \int_incr:N \l_xparse_mandatory_args_int
  \xparse_count_mandatory:N
}
```

\xparse_declare_cmd:Nnn
\xparse_declare_cmd_aux:Nnn
\xparse_declare_cmd_aux:cnn
\xparse_declare_cmd_all_m:Nn
\xparse_declare_cmd_mixed:Nn

First, the signature is set up from the argument specification. There is then a check: if only `m` arguments are needed (which includes functions with no arguments at all) then the definition is simple. On the other hand, if the signature is more complex then an internal function actually contains the code with the user function as a simple wrapper.

```
\cs_new:Npn \xparse_declare_cmd:Nnn #1#2 {
  \cs_if_exist:NTF #1
    {
      \msg_kernel_warning:nnxx { xparse } { redefine-command }
        { \exp_not:N #1 } { \exp_not:n {#2} }
    }
    {
      \msg_kernel_info:nnxx { xparse } { define-command }
        { \exp_not:N #1 } { \exp_not:n {#2} }
    }
  \xparse_declare_cmd_aux:Nnn #1 {#2}
}
\cs_new:Npn \xparse_declare_cmd_aux:Nnn #1#2#3 {
  \xparse_count_mandatory:n {#2}
  \xparse_prepare_signature:n {#2}
  \bool_if:NTF \l_xparse_m_only_bool {
    \xparse_declare_cmd_all_m:Nn #1 {#3}
  }{
    \xparse_declare_cmd_mixed:Nn #1 {#3}
  }
}
\cs_generate_variant:Nn \xparse_declare_cmd_aux:Nnn { cnn }
\cs_new:Npn \xparse_declare_cmd_all_m:Nn #1#2 {
  \cs_generate_from_arg_count:NNnn
    #1 \cs_set_protected_nopar:Npn \l_xparse_total_args_int {#2}
}
\cs_new:Npn \xparse_declare_cmd_mixed:Nn #1#2 {
  \cs_set_protected_nopar:Npx #1 {
    \exp_not:n {
      \int_zero:N \l_xparse_processor_int
      \toks_set:Nn \l_xparse_args_toks
    } { \exp_not:c { \token_to_str:N #1 } }
    \toks_use:N \l_xparse_signature_toks
    \exp_not:n{ \toks_use:N \l_xparse_args_toks }
```

```
  }
  \cs_generate_from_arg_count:cNnn
    { \token_to_str:N #1 } \cs_set:Npn \l_xparse_total_args_int {#2}
}
```

`_declare_cmd_implementation:nNn` Creating a stand-alone implementation using the 'two-part' mechanism is quite easy as this is just a wrapper for `\cs_generate_from_arg_count:cNnn`.

```
\cs_new:Npn \xparse_declare_cmd_implementation:nNn #1#2#3 {
  \cs_generate_from_arg_count:cNnn { implementation_ #1 :w }
    \cs_set:Npn {#2} {#3}
}
```

`parse_declare_cmd_interface:Nnn`
`_declare_cmd_interface_all_m:Nn`
`_declare_cmd_interface_mixed:Nn`

As with the basic function `\xparse_declare_cmd:Nnn`, there are three things to do here. First, generate a signature from the argument specification. Then use that to create a function which will call the implementation part. Finally, a holder implementation is created. As before, there is a short-cut for functions which only have `m` type arguments.

```
\cs_new:Npn \xparse_declare_cmd_interface:Nnn #1#2#3 {
  \xparse_prepare_signature:n {#3}
  \bool_if:NTF \l_xparse_m_only_bool {
    \xparse_declare_cmd_interface_all_m:Nn #1 {#2}
  }{
    \xparse_declare_cmd_interface_mixed:Nn #1 {#2}
  }
  \cs_generate_from_arg_count:cNnn { implementation_ #2 :w }
    \cs_set:Npn \l_xparse_total_args_int  { ``#2'' }
}
\cs_new:Npn \xparse_declare_cmd_interface_all_m:Nn #1#2 {
  \cs_generate_from_arg_count:NNnn
    #1 \cs_set_protected_nopar:Npn \l_xparse_total_args_int
    { \use:c { implementation_ #2 :w } }
}
\cs_new:Npn \xparse_declare_cmd_interface_mixed:Nn #1#2 {
  \cs_set_protected_nopar:Npx #1 {
    \exp_not:n {
      \int_zero:N \l_xparse_processor_int
      \toks_set:Nn \l_xparse_args_toks
    } { \exp_not:c { \token_to_str:N #1 } }
    \toks_use:N \l_xparse_signature_toks
    \exp_not:n{ \toks_use:N \l_xparse_args_toks }
  }
  \cs_generate_from_arg_count:cNnn
    { \token_to_str:N #1 } \cs_set:Npn \l_xparse_total_args_int
    { \use:c { implementation_ #2 :w } }
}
```

`\xparse_declare_env:nnnn` The idea here is to make sure that the end of the environment has the same arguments available as the beginning.

```
\cs_new:Npn \xparse_declare_env:nnnn #1#2#3#4 {
  \bool_set_true:N \l_xparse_environment_bool
⟨/initex | package⟩
⟨*initex⟩
  \cs_if_exist:cTF { environment_begin_ #1 :w }
⟨/initex⟩
⟨*package⟩
  \cs_if_exist:cTF {#1}
⟨/package⟩
⟨*initex | package⟩
    {
      \msg_kernel_warning:nnxx { xparse } { redefine-environment }
        {#1} { \exp_not:n {#2} }
    }
    {
      \msg_kernel_info:nnxx { xparse } { define-environment }
        {#1} { \exp_not:n {#2} }
    }
  \xparse_declare_cmd_aux:cnn { environment_begin_ #1 :w } {#2} {
    \group_begin:
      \toks_set_eq:NN \l_xparse_environment_args_toks
        \l_xparse_args_toks
      #3
  }
  \bool_set_false:N \l_xparse_environment_bool
  \cs_set_nopar:cpx { environment_end_ #1 :w } {
      \exp_not:N \exp_last_unbraced:NV
      \exp_not:c { environment_end_ #1 _aux:N }
      \exp_not:N \l_xparse_environment_args_toks
    \exp_not:N \group_end:
  }
  \cs_set_nopar:cpx { environment_end_ #1 _aux:N } ##1 {
    \exp_not:c { environment_end_ #1 _aux :w }
  }
  \cs_generate_from_arg_count:cNnn
    { environment_end_ #1 _aux :w } \cs_set:Npn
    \l_xparse_total_args_int {#4}
⟨/initex | package⟩
⟨*package⟩
  \cs_set_eq:cc {#1} { environment_begin_ #1 :w }
  \cs_set_eq:cc { end #1 } { environment_end_ #1 :w }
⟨/package⟩
⟨*initex | package⟩
}
```

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

```
\cs_new_nopar:Npn \xparse_flush_m_args: {
  \cs_if_exist:cT {
    xparse_grab_m_ \int_use:N \l_xparse_m_args_int :w
  } {
    \toks_put_right:Nx \l_xparse_signature_toks {
       \exp_not:c { xparse_grab_m_ \int_use:N \l_xparse_m_args_int :w }
    }
    \int_set:Nn \l_xparse_mandatory_args_int {
      \l_xparse_mandatory_args_int - \l_xparse_m_args_int
    }
  }
  \int_zero:N \l_xparse_m_args_int
  \bool_set_false:N \l_xparse_m_only_bool
}
```

`\xparse_if_no_value:n`*TF* Tests for `\NoValue`.

```
\prg_new_conditional:Nnn \xparse_if_no_value:n { TF,T,F } {
  \tl_if_eq:nnTF {#1} { \NoValue } {
    \prg_return_true:
  }{
    \prg_return_false:
  }
}
```

`\xparse_prepare_signature:n` Creating the signature is a case of working through the input and turning into the output in `\l_xparse_signature_toks`. A track is also kept of the total number of arguments. This function sets everything up then hands off to the parser.

```
\cs_new:Npn \xparse_prepare_signature:n #1 {
  \bool_set_false:N \l_xparse_long_bool
  \int_zero:N \l_xparse_m_args_int
  \bool_if:NTF \l_xparse_environment_bool {
    \bool_set_false:N \l_xparse_m_only_bool
  }{
    \bool_set_true:N \l_xparse_m_only_bool
  }
  \bool_set_false:N \l_xparse_processor_bool
  \toks_clear:N \l_xparse_signature_toks
  \int_zero:N \l_xparse_total_args_int
  \xparse_prepare_signature:N #1 \q_nil
}
```

`\xparse_prepare_signature:N` `\xparse_prepare_signature_aux:N` The main signature-preparation loop is in two parts, to keep the code a little clearer. Most of the checks here is pretty clear, with a key point to watch what is next on the stack so that the loop continues correctly.

```
\cs_new:Npn \xparse_prepare_signature:N #1 {
  \bool_set_false:N \l_xparse_processor_bool
```

```
  \xparse_prepare_signature_aux:N #1
}
\cs_new:Npn \xparse_prepare_signature_aux:N #1 {
  \quark_if_nil:NTF #1 {
    \bool_if:NF \l_xparse_m_only_bool {
      \xparse_flush_m_args:
    }
  }{
    \prop_if_in:NnTF \c_xparse_shorthands_prop {#1} {
      \prop_get:NnN \c_xparse_shorthands_prop {#1} \l_xparse_tmp_tl
      \exp_last_unbraced:NV \xparse_prepare_signature:N \l_xparse_tmp_tl
    }{
      \int_incr:N \l_xparse_total_args_int
      \xparse_check_and_add:N #1
    }
  }
}
```

`\xparse_process_arg:n` Processors are saved for use later during the grabbing process.

```
\cs_new:Npn \xparse_process_arg:n #1 {
  \int_incr:N \l_xparse_processor_int
  \cs_set:cpn {
    xparse_processor_ \int_use:N \l_xparse_processor_int :n
  } ##1
    { #1 {##1} }
}
```

2.3 Grabbing arguments

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

```
\cs_new:Npn \xparse_add_arg:n #1 {
  \intexpr_compare:nTF { \l_xparse_processor_int = \c_zero } {
    \toks_put_right:Nn \l_xparse_args_toks { {#1} }
  }{
    \xparse_if_no_value:nTF {#1} {
      \int_zero:N \l_xparse_processor_int
      \toks_put_right:Nn \l_xparse_args_toks { {#1} }
    }{
      \xparse_add_arg_aux:n {#1}
    }
  }
}
```

```
\cs_generate_variant:Nn \xparse_add_arg:n { V }
\cs_new:Npn \xparse_add_arg_aux:n #1 {
  \cs_set_eq:NN \ProcessedArgument \l_xparse_arg_toks
  \use:c { xparse_processor_ \int_use:N \l_xparse_processor_int :n }
    {#1}
  \int_decr:N \l_xparse_processor_int
  \intexpr_compare:nTF { \l_xparse_processor_int = \c_zero } {
    \toks_put_right:Nx \l_xparse_args_toks {
      { \exp_not:V \ProcessedArgument }
    }
  }{
    \xparse_add_arg_aux:V \ProcessedArgument
  }
}
\cs_generate_variant:Nn \xparse_add_arg_aux:n { V }
```

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

`\xparse_grab_arg:w` `\xparse_grab_arg_aux_i:w` `\xparse_grab_arg_aux_ii:w`

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

```
\cs_new:Npn \xparse_grab_arg:w { }
\cs_new:Npn \xparse_grab_arg_aux_i:w { }
\cs_new:Npn \xparse_grab_arg_aux_ii:w { }
```

`\xparse_grab_D:w` `\xparse_grab_D_long:w` `\xparse_grab_D_trailing:w` `\xparse_grab_D_long_trailing:w`

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

```
\cs_new:Npn \xparse_grab_D:w #1#2#3#4 \l_xparse_args_toks {
  \xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_nopar:Npn
    { _ignore_spaces }
}
\cs_new:Npn \xparse_grab_D_long:w #1#2#3#4 \l_xparse_args_toks {
  \xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set:Npn
    { _ignore_spaces }
}
\cs_new:Npn \xparse_grab_D_trailing:w #1#2#3#4 \l_xparse_args_toks {
  \xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set_nopar:Npn { }
}
\cs_new:Npn \xparse_grab_D_long_trailing:w #1#2#3#4
  \l_xparse_args_toks {
```

```
  \xparse_grab_D_aux:NNnnNn #1 #2 {#3} {#4} \cs_set:Npn { }
}
```

`\xparse_grab_D_aux:NNnnNn`

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. First, the opening token is removed, then a check is made for a group. If it looks like the entire argument is a group, then an extra set of braces are added back in. The closing token is then used to collect everything else. There is then a test to see if there is nesting, by looking for a 'spare' open-argument token. If that is found, things hand off to a loop to deal with that.

```
\cs_new:Npn \xparse_grab_D_aux:NNnnNn #1#2#3#4#5#6 {
  #5 \xparse_grab_arg:w #1 {
    \peek_meaning:NTF \c_group_begin_token {
      \xparse_grab_arg_aux_i:w
    }{
      \xparse_grab_arg_aux_ii:w
    }
  }
  #5 \xparse_grab_arg_aux_i:w ##1 {
    \peek_charcode:NTF #2 {
      \xparse_grab_arg_aux_ii:w { {##1} }
    }{
      \xparse_grab_arg_aux_ii:w {##1}
    }
  }
  #5 \xparse_grab_arg_aux_ii:w ##1 #2 {
    \tl_if_in:nnTF {##1} {#1} {
      \xparse_grab_D_nested:NNnNn #1 #2 {#4} #5 {##1}
    }{
      \xparse_add_arg:n {##1}
      #4 \l_xparse_args_toks
    }
  }
  \use:c { peek_charcode #6 :NTF } #1 {
    \xparse_grab_arg:w
  }{
    \xparse_add_arg:n {#3}
    #4 \l_xparse_args_toks
  }
}
```

`\xparse_grab_D_nested:NNnNn`
`\xparse_grab_D_nested_aux:n`

Catching nested optional arguments means more work. As TEX does not help here, the brackets have to be counted by hand. The code then keeps looking for closing tokens until all of the opening ones are matched.

```
\cs_new:Npn \xparse_grab_D_nested:NNnNn #1#2#3#4#5 {
  \int_zero:N \l_xparse_nested_int
```

```
  \toks_set:Nn \l_xparse_nested_toks { #5 #2 }
  \cs_set:Npn \xparse_grab_D_nested_aux:n ##1 {
    \tl_if_eq:nnT {#1} {##1} {
      \int_incr:N \l_xparse_nested_int
    }
  }
  \tl_map_function:nN {#5} \xparse_grab_D_nested_aux:n
  #4 \xparse_grab_arg:w ##1 #2 {
    \int_decr:N \l_xparse_nested_int
    \tl_map_function:nN {##1} \xparse_grab_D_nested_aux:n
    \intexpr_compare:nTF { \l_xparse_nested_int = \c_zero } {
      \toks_put_right:Nn \l_xparse_nested_toks {##1}
      \xparse_add_arg:V \l_xparse_nested_toks
      #3 \l_xparse_args_toks
    }{
      \toks_put_right:Nn \l_xparse_nested_toks { ##1 #2 }
      \xparse_grab_arg:w
    }
  }
  \xparse_grab_arg:w
}
\cs_new:Npn \xparse_grab_D_nested_aux:n #1 { }
```

\xparse_grab_G:w
\xparse_grab_G_long:w
\xparse_grab_G_trailing:w
\xparse_grab_G_long_trailing:w
\xparse_grab_G_aux:nnNn

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

```
\cs_new:Npn \xparse_grab_G:w #1#2 \l_xparse_args_toks {
  \xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_nopar:Npn { _ignore_spaces }
}
\cs_new:Npn \xparse_grab_G_long:w #1#2 \l_xparse_args_toks {
  \xparse_grab_G_aux:nnNn {#1} {#2} \cs_set:Npn { _ignore_spaces }
}
\cs_new:Npn \xparse_grab_G_trailing:w #1#2 \l_xparse_args_toks {
  \xparse_grab_G_aux:nnNn {#1} {#2} \cs_set_nopar:Npn { }
}
\cs_new:Npn \xparse_grab_G_long_trailing:w #1#2 \l_xparse_args_toks {
  \xparse_grab_G_aux:nnNn {#1} {#2} \cs_set:Npn { }
}
\cs_set:Npn \xparse_grab_G_aux:nnNn #1#2#3#4 {
  #3 \xparse_grab_arg:w ##1 {
    \xparse_add_arg:n {##1}
    #2 \l_xparse_args_toks
  }
  \use:c { peek_meaning #4 :NTF } \c_group_begin_token {
    \xparse_grab_arg:w
  }{
    \xparse_add_arg:n {#1}
    #2 \l_xparse_args_toks
  }
}
```

\xparse_grab_l:w
\xparse_grab_l_long:w
\xparse_grab_l_aux:nN

Argument grabbers for mandatory TEX arguments are pretty simple.

```
\cs_new:Npn \xparse_grab_l:w #1 \l_xparse_args_toks {
  \xparse_grab_l_aux:nN {#1} \cs_set_nopar:Npn
}
\cs_new:Npn \xparse_grab_l_long:w #1 \l_xparse_args_toks {
  \xparse_grab_l_aux:nN {#1} \cs_set:Npn
}
\cs_new:Npn \xparse_grab_l_aux:nN #1#2 {
  #2 \xparse_grab_arg:w ##1## {
    \xparse_add_arg:n {##1}
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
```

\xparse_grab_m:w
\xparse_grab_m_long:w

Collecting a single mandatory argument is quite easy.

```
\cs_new:Npn \xparse_grab_m:w #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1 {
    \xparse_add_arg:n {##1}
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:Npn \xparse_grab_m_long:w #1 \l_xparse_args_toks {
  \cs_set:Npn \xparse_grab_arg:w ##1 {
    \xparse_add_arg:n {##1}
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
```

\xparse_grab_m_1:w
\xparse_grab_m_2:w
\xparse_grab_m_3:w
\xparse_grab_m_4:w
\xparse_grab_m_5:w
\xparse_grab_m_6:w
\xparse_grab_m_7:w
\xparse_grab_m_8:w

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

```
\cs_new:cpn { xparse_grab_m_1:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1 {
    \toks_put_right:Nn \l_xparse_args_toks { {##1} }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:cpn { xparse_grab_m_2:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2 {
    \toks_put_right:Nn \l_xparse_args_toks { {##1} {##2} }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
```

```
}
\cs_new:cpn { xparse_grab_m_3:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2##3 {
    \toks_put_right:Nn \l_xparse_args_toks { {##1} {##2} {##3} }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:cpn { xparse_grab_m_4:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2##3##4 {
    \toks_put_right:Nn \l_xparse_args_toks { {##1} {##2} {##3} {##4} }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:cpn { xparse_grab_m_5:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2##3##4##5 {
    \toks_put_right:Nn \l_xparse_args_toks {
      {##1} {##2} {##3} {##4} {##5}
    }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:cpn { xparse_grab_m_6:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2##3##4##5##6 {
    \toks_put_right:Nn \l_xparse_args_toks {
      {##1} {##2} {##3} {##4} {##5} {##6}
    }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:cpn { xparse_grab_m_7:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2##3##4##5##6##7 {
    \toks_put_right:Nn \l_xparse_args_toks {
      {##1} {##2} {##3} {##4} {##5} {##6} {##7}
    }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
\cs_new:cpn { xparse_grab_m_8:w } #1 \l_xparse_args_toks {
  \cs_set_nopar:Npn \xparse_grab_arg:w ##1##2##3##4##5##6##7##8 {
    \toks_put_right:Nn \l_xparse_args_toks {
      {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8}
    }
    #1 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
```

```
}
```

\xparse_grab_t:w
\xparse_grab_t_long:w
\xparse_grab_t_trailing:w
\xparse_grab_t_long_trailing:w
\xparse_grab_t_aux:NnNn

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

```
\cs_new:Npn \xparse_grab_t:w #1#2 \l_xparse_args_toks {
  \xparse_grab_t_aux:NnNn #1 {#2} \cs_set_nopar:Npn { _ignore_spaces }
}
\cs_new:Npn \xparse_grab_t_long:w #1#2 \l_xparse_args_toks {
  \xparse_grab_t_aux:NnNn #1 {#2} \cs_set:Npn { _ignore_spaces }
}
\cs_new:Npn \xparse_grab_t_trailing:w #1#2 \l_xparse_args_toks {
  \xparse_grab_t_aux:NnNn #1 {#2} \cs_set_nopar:Npn { }
}
\cs_new:Npn \xparse_grab_t_long_trailing:w #1#2 \l_xparse_args_toks {
  \xparse_grab_t_aux:NnNn #1 {#2} \cs_set:Npn { }
}
\cs_new:Npn \xparse_grab_t_aux:NnNn #1#2#3#4 {
  #3 \xparse_grab_arg:w {
    \use:c { peek_charcode_remove #4 :NTF } #1 {
      \xparse_add_arg:n { \BooleanTrue }
      #2 \l_xparse_args_toks
    }{
      \xparse_add_arg:n { \BooleanFalse }
      #2 \l_xparse_args_toks
    }
  }
  \xparse_grab_arg:w
}
```

\xparse_grab_u:w
\xparse_grab_u_long:w
\xparse_grab_u_aux:NnN

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

```
\cs_new:Npn \xparse_grab_u:w #1#2 \l_xparse_args_toks {
  \xparse_grab_u_aux:NnN {#1} {#2} \cs_set_nopar:Npn
}
\cs_new:Npn \xparse_grab_u_long:w #1#2 \l_xparse_args_toks {
  \xparse_grab_u_aux:NnN {#1} {#2} \cs_set:Npn
}
\cs_new:Npn \xparse_grab_u_aux:NnN #1#2#3 {
  #3 \xparse_grab_arg:w ##1 #1 {
    \xparse_add_arg:n {##1}
    #2 \l_xparse_args_toks
  }
  \xparse_grab_arg:w
}
```

2.4 Argument processors

\xparse_process_to_str:n

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

```
\cs_new:Npn \xparse_process_to_str:n #1 {
  \toks_set:Nx \ProcessedArgument {
    \tl_to_str:n {#1}
  }
}
```

\xparse_process_comma_split:n
\xparse_process_comma_split_aux:w

Turns a co-ordinate pair into two separate values.

```
\cs_new:Npn \xparse_process_comma_split:n #1 {
  \tl_if_in:nnTF {#1} { , } {
    \xparse_process_comma_split_aux:w #1 \q_stop
  }{
    \toks_set:Nn \ProcessedArgument { {#1} { \NoValue } }
  }
}
\cs_new:Npn \xparse_process_comma_split_aux:w #1 , #2 \q_stop {
  \toks_set:Nn \ProcessedArgument { {#1} {#2} }
}
```

2.5 Creating expandable functions

The trick here is to pass each grabbed argument along a chain of auxiliary functions. Each one ultimately calls the next in the chain, so that all of the arguments are passed along delimited using `\q_xparse_stop`. At the end of the chain, the marker is removed so that the user-supplied code can be passed the correct number of arguments. All of this is done by expansion!

\xparse_exp_add_type_d:w

As in the standard case, the trick here is to slot in the default and treat as type `D`.

```
\cs_new:Npn \xparse_exp_add_type_d:w #1#2 {
  \xparse_exp_add_type_D:w #1 #2 { \NoValue }
}
```

\xparse_exp_add_type_D:w

The most complex argument to grab in an expandable manner is the general delimited one. First, a short-cut is set up in `\l_xparse_tmpa_tl` for the name of the current grabber function. This is then created to grab one argument and test if it is equal to the opening delimiter. If the test fails, the code adds the default value and closing delimiter before 'recycling' the argument. In either case, the second auxiliary function is called. It finds the closing delimiter and so the optional argument (if any). The function then calls the next one in the chain, passing along the argument(s) grabbed thus-far using `\q_xparse_stop` as a marker.

```
\cs_new:Npn \xparse_exp_add_type_D:w #1#2#3 {
  \tl_set:Nx \l_xparse_tmpa_tl {
    \exp_after:wN \token_to_str:N \l_xparse_function_tl
    \int_use:N \l_xparse_total_args_int
  }
```

```
  \xparse_exp_set:cpx { \l_xparse_tmpa_tl } ##1 \q_xparse_stop ##2 {
    \exp_not:N \tl_if_head_eq_charcode:nNTF {##2} #1 {
      \exp_not:c { \l_xparse_tmpa_tl aux }
        ##1 \exp_not:N \q_xparse_stop
    }{
      \exp_not:c { \l_xparse_tmpa_tl aux }
        ##1 \exp_not:N \q_xparse_stop  #3 #2 {##2}
    }
  }
  \xparse_exp_set:cpx { \l_xparse_tmpa_tl aux}
    ##1 \q_xparse_stop ##2 #2 {
    \exp_not:c {
      \exp_after:wN \token_to_str:N \l_xparse_function_tl
      \intexpr_eval:n { \l_xparse_total_args_int + 1 }
    } ##1 {##2} \exp_not:N \q_xparse_stop
  }
  \xparse_exp_prepare_function:N
}
```

`\xparse_exp_add_type_l:w`
`\xparse_exp_add_type_m:w`

Gathering `l` and `m` arguments is almost the same. The grabber for the current argument is created to simply get the necessary argument and pass it along with any others through to the next function in the chain.

```
\cs_new_nopar:Npn \xparse_exp_add_type_l:w {
  \xparse_exp_set:cpx {
    \exp_after:wN \token_to_str:N \l_xparse_function_tl
    \int_use:N \l_xparse_total_args_int
  } ##1 \q_xparse_stop ##2## {
    \exp_not:c {
      \exp_after:wN \token_to_str:N \l_xparse_function_tl
      \intexpr_eval:n { \l_xparse_total_args_int + 1 }
    }
    ##1 {##2} \exp_not:N \q_xparse_stop
  }
  \xparse_exp_prepare_function:N
}
\cs_new_nopar:Npn \xparse_exp_add_type_m:w {
  \int_incr:N \l_xparse_m_args_int
  \xparse_exp_set:cpx {
    \exp_after:wN \token_to_str:N \l_xparse_function_tl
    \int_use:N \l_xparse_total_args_int
  } ##1 \q_xparse_stop ##2 {
    \exp_not:c {
      \exp_after:wN \token_to_str:N \l_xparse_function_tl
      \intexpr_eval:n { \l_xparse_total_args_int + 1 }
    }
    ##1 {##2} \exp_not:N \q_xparse_stop
  }
  \xparse_exp_prepare_function:N
}
```

`\xparse_exp_add_type_t:w` Looking for a single token is a simpler version of the `D` code. The same idea of picking up one argument is used, but there is no need for a second function as there is no closing token to find. So either `\BooleanTrue` or `\BooleanFalse` are added to the list of arguments. In the later case, the grabber argument must be 'recycled'.

```
\cs_new:Npn \xparse_exp_add_type_t:w #1 {
  \tl_set:Nx \l_xparse_tmpa_tl {
    \exp_after:wN \token_to_str:N \l_xparse_function_tl
    \intexpr_eval:n { \l_xparse_total_args_int + 1 }
  }
  \xparse_exp_set:cpx {
    \exp_after:wN \token_to_str:N \l_xparse_function_tl
    \int_use:N \l_xparse_total_args_int
  } ##1 \q_xparse_stop ##2 {
    \exp_not:N \tl_if_head_eq_charcode:nNTF {##2} #1 {
      \exp_not:c { \l_xparse_tmpa_tl }
      ##1 \exp_not:n { { \BooleanTrue } \q_xparse_stop }
    }{
      \exp_not:c { \l_xparse_tmpa_tl }
      ##1 \exp_not:n { { \BooleanFalse } \q_xparse_stop {##2} }
    }
  }
  \xparse_exp_prepare_function:N
}
```

`\xparse_exp_add_type_u:w` Setting up for a `u` argument is a case of defining the grabber for the current argument in a delimited fashion. The rest of the process is as the other grabbers: add to the chain and call the next function.

```
\cs_new:Npn \xparse_exp_add_type_u:w #1 {
  \xparse_exp_set:cpx {
    \exp_after:wN \token_to_str:N \l_xparse_function_tl
    \int_use:N \l_xparse_total_args_int
  } ##1 \q_xparse_stop ##2 #1 {
    \exp_not:c {
      \exp_after:wN \token_to_str:N \l_xparse_function_tl
      \intexpr_eval:n { \l_xparse_total_args_int + 1 }
    }
    ##1 {##2} \exp_not:N \q_xparse_stop
  }
  \xparse_exp_prepare_function:N
}
```

`\xparse_exp_check_and_add:N` Virtually identical to the normal version, except calling the expandable `add` functions rather than the standard versions.

```
\cs_new_nopar:Npn \xparse_exp_check_and_add:N  #1 {
  \cs_if_free:cTF { xparse_exp_add_type_ #1 :w } {
    \msg_kernel_error:nnx { xparse } { unknown-argument-type } {#1}
```

```
    \tl_set:Nn \l_xparse_last_arg_tl { m }
    \xparse_exp_add_type_m:w
  }{
    \tl_set:Nn \l_xparse_last_arg_tl {#1}
    \use:c { xparse_exp_add_type_ #1 :w }
  }
}
```

\xparse_exp_declare_cmd:Nnn
\xparse_exp_declare_cmd_all_m:Nn
\xparse_exp_declare_cmd_mixed:Nn
\xparse_exp_declare_cmd_mixed_aux:Nn

The overall scheme here is very different from the standard method. For each argument, an internal function is created to grab an argument and pass along previous ones. Each 'daisy chains' to call the next one in the sequence. Thus at the end of the chain, an extra 'argument' function is included to unwind the chain and pass data to the the internal function containing the actual code. If all of the arguments are type `m`, then the same tick is used as in the standard version. The `x` in the lead-off and mop-up functions makes sure that the braces around the first argument are not lost.

```
\cs_new:Npn \xparse_exp_declare_cmd:Nnn #1#2#3 {
  \cs_if_exist:NTF #1
    {
      \msg_kernel_warning:nnxx { xparse } { redefine-command }
        { \exp_not:N #1 } { \exp_not:n {#2} }
    }
    {
      \msg_kernel_info:nnxx { xparse } { define-command }
        { \exp_not:N #1 } { \exp_not:n {#2} }
    }
  \tl_set:Nn \l_xparse_function_tl {#1}
  \xparse_exp_prepare_function:n {#2}
  \intexpr_compare:nTF {
    \l_xparse_total_args_int = \l_xparse_m_args_int
  } {
    \xparse_exp_declare_cmd_all_m:Nn #1 {#3}
  }{
    \xparse_exp_declare_cmd_mixed:Nn #1 {#3}
  }
}
\cs_new:Npn \xparse_exp_declare_cmd_all_m:Nn #1#2 {
  \bool_if:NTF \l_xparse_long_bool {
    \cs_generate_from_arg_count:NNnn
      #1 \cs_set:Npn \l_xparse_total_args_int {#2}
  }{
    \cs_generate_from_arg_count:NNnn
      #1 \cs_set_nopar:Npn \l_xparse_total_args_int {#2}
  }
}
\cs_new:Npn \xparse_exp_declare_cmd_mixed:Nn #1#2 {
  \exp_args:NnV \tl_if_in:nnTF { l m u } \l_xparse_last_arg_tl {
    \xparse_exp_declare_cmd_mixed_aux:Nn #1 {#2}
  }{
```

```
    \msg_kernel_error:nn { xparse } { expandable-ending-optional }
  }
}
\cs_new:Npn \xparse_exp_declare_cmd_mixed_aux:Nn #1#2 {
  \cs_set_nopar:Npx #1 {
    \exp_not:c { \token_to_str:N #1 1 } x \exp_not:N \q_xparse_stop
  }
  \cs_set_nopar:cpx {
    \token_to_str:N #1 \intexpr_eval:n { \l_xparse_total_args_int + 1  }
  } x ##1 \q_xparse_stop {
    \exp_not:c { \token_to_str:N #1 } ##1
  }
  \cs_generate_from_arg_count:cNnn
    { \token_to_str:N #1 } \cs_set:Npn \l_xparse_total_args_int {#2}
}
```

\xparse_exp_prepare_function:n
\xparse_exp_prepare_function_aux:n

A couple of early validation tests. Processors are forbidden, as are `g`, `l` and `u` arguments (the later more for ease than any technical reason).

```
\cs_new:Npn \xparse_exp_prepare_function:n  #1 {
  \bool_set_false:N \l_xparse_error_bool
  \tl_if_in:nnT {#1} { > } {
    \msg_kernel_error:nnx { xparse } { processor-in-expandable } {#1}
    \bool_set_true:N \l_xparse_error_bool
  }
  \tl_if_in:nnT {#1} { g } {
    \msg_kernel_error:nnx { xparse } { grouped-in-expandable }
      { g } {#1}
    \bool_set_true:N \l_xparse_error_bool
  }
  \tl_if_in:nnT {#1} { G } {
    \msg_kernel_error:nnx { xparse } { grouped-in-expandable }
      { G } {#1}
    \bool_set_true:N \l_xparse_error_bool
  }
  \bool_if:NF \l_xparse_error_bool {
    \xparse_exp_prepare_function_aux:n {#1}
  }
}
\cs_new:Npn \xparse_exp_prepare_function_aux:n  #1 {
  \cs_set_eq:NN \xparse_prepare_next:w \xparse_exp_prepare_function:N
  \cs_set_eq:NN \xparse_exp_set:cpx \cs_set_nopar:cpx
  \bool_set_false:N \l_xparse_long_bool
  \int_zero:N \l_xparse_m_args_int
  \int_zero:N \l_xparse_total_args_int
  \tl_if_in:nnT {#1} { + } {
    \bool_set_true:N \l_xparse_long_bool
    \cs_set_eq:NN \xparse_exp_set:cpx \cs_set:cpx
  }
  \xparse_exp_prepare_function:N #1 \q_nil
```

```
}
```

\xparse_exp_prepare_function:N
\xparse_exp_prepare_function_long:N
\xparse_exp_prepare_function_short:N

Preparing functions is a case of reading the signature, as in the normal case. However, everything has to be either short or long, and so there is an extra step to make sure that once one + has been seen everything has one. That detour then takes us back to a standard looping concept.

```
\cs_new:Npn \xparse_exp_prepare_function:N #1 {
  \bool_if:NTF \l_xparse_long_bool {
    \xparse_exp_prepare_function_long:N #1
  }{
    \xparse_exp_prepare_function_short:N #1
  }
}
\cs_new:Npn \xparse_exp_prepare_function_long:N #1 {
  \quark_if_nil:NF #1 {
    \tl_if_eq:nnTF {#1} { + } {
      \xparse_exp_prepare_function_short:N
    }{
      \msg_kernel_error:nn { xparse } { expandable-inconsistent-long }
      \xparse_exp_prepare_function_short:N #1
    }
  }
}
\cs_new:Npn \xparse_exp_prepare_function_short:N #1 {
  \quark_if_nil:NF #1 {
    \prop_if_in:NnTF \c_xparse_shorthands_prop {#1} {
      \prop_get:NnN \c_xparse_shorthands_prop {#1} \l_xparse_tmp_tl
      \bool_if:NT \l_xparse_long_bool {
        \tl_put_left:Nn \l_xparse_tmp_tl { + }
      }
      \exp_last_unbraced:NV \xparse_exp_prepare_function:N
        \l_xparse_tmp_tl
    }{
      \int_incr:N \l_xparse_total_args_int
      \xparse_exp_check_and_add:N #1
    }
  }
}
```

\xparse_exp_set:cpx

A short-cut to save constantly re-testing \l_xparse_long_bool.

```
\cs_new_eq:NN \xparse_exp_set:cpx \cs_set_nopar:cpx
```

2.6 Messages

Some messages intended as errors.

```
\msg_kernel_new:nnnn { xparse } { command-already-defined }
  {Command ‘#1’ already defined!}
  {%
    You have used \token_to_str:N \NewDocumentCommand\\%
    with a command that already has a definition.\\%
    Perhaps you meant \token_to_str:N \RenewDocumentCommand.%
  }
\msg_kernel_new:nnnn { xparse } { command-not-yet-defined }
  {Command ‘#1’ not yet defined!}
  {%
    You have used \token_to_str:N \RenewDocumentCommand\\%
    with a command that was never defined.\\%
    Perhaps you meant \token_to_str:N \NewDocumentCommand.%
  }
\msg_kernel_new:nnnn { xparse } { environment-already-defined }
  {Environment ‘#1’ already defined!}
  {%
    You have used \token_to_str:N \NewDocumentEnvironment\\%
    with a command that already has a definition.\\%
    Perhaps you meant \token_to_str:N \RenewDocumentEnvironment.%
  }
\msg_kernel_new:nnnn { xparse } { environment-not-yet-defined }
  {Environment ‘#1’ not yet defined!}
  {%
    You have used \token_to_str:N \RenewDocumentEnvironment\\%
    with a command that was never defined.\\%
    Perhaps you meant \token_to_str:N \NewDocumentEnvironment.%
  }
\msg_kernel_new:nnnn { xparse } { expandable-ending-optional }
  {%
    Signature for expandable command ends with \\%
    optional argument \msg_line_context:.%
  }
  {%
    Expandable commands must have a final mandatory argument\\%
    (or no arguments at all). You cannot have a terminal optional\\%
    argument with expandable commands.%
  }
\msg_kernel_new:nnnn { xparse } { expandable-inconsistent-long }
  {%
    Inconsistent handling of long arguments for\\%
    expandable command \msg_line_context:.%
  }
  {%
    The arguments for an expandable command must either all be\\%
    short or all be long. You have tried to mix the two types.%
  }
\msg_kernel_new:nnnn { xparse } { grouped-in-expandable }
  {%
    Argument specifier ‘#1’ forbidden in expandable commands
```

```
    \msg_line_context:.%
  }
  {%
    Argument specification ‘#2’ contains the optional grouped
    argument ‘#1’:\\%
    this is only supported for standard robust functions.%
  }
\msg_kernel_new:nnnn { xparse } { processor-in-expandable }
  {%
    Argument processors cannot be used\\%
    with expandable functions \msg_line_context:.%
  }
  {%
    Argument specification ‘#1’ contains a processor function:\\%
    this is only supported for standard robust functions.%
  }
\msg_kernel_new:nnnn { xparse } { unknown-argument-type }
  {Unknown argument type ‘#1’ replaced by ‘m’. Fingers crossed ...}
  {%
    The letter ‘#1’ does not specify a known argument type.\\%
    I’m assuming you want a standard mandatory argument (type ‘m’).%
  }
```

Intended more for information.

```
\msg_kernel_new:nnn { xparse } { define-command }
  {%
    Defining document command #1\\%
    with arg. spec. ‘#2’ \msg_line_context:.%
  }
\msg_kernel_new:nnn { xparse } { define-environment }
  {%
    Defining document environment ‘#1’\\%
    with arg. spec. ‘#2’ \msg_line_context:.%
  }
\msg_kernel_new:nnn { xparse } { redefine-command }
  {%
    Redefining document command #1\\%
    with arg. spec. ‘#2’ \msg_line_context:.%
  }
\msg_kernel_new:nnn { xparse } { redefine-environment }
  {%
    Redefining document environment ‘#1’\\%
    with arg. spec. ‘#2’ \msg_line_context:.%
  }
```

2.7 User functions

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

\BooleanFalse
\BooleanTrue

Design-space names for the Boolean values.

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

\DeclareDocumentCommand
\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand

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

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

reDocumentCommandImplementation
DeclareDocumentCommandInterface

The separate implementation/interface system is again pretty simple to create at the outer layer.

```
\cs_new_protected:Npn \DeclareDocumentCommandImplementation #1#2#3 {
  \xparse_declare_cmd_implementation:nNn {#1} #2 {#3}
}
\cs_new_protected:Npn \DeclareDocumentCommandInterface #1#2#3 {
  \xparse_declare_cmd_interface:Nnn #1 {#2} {#3}
}
```

\DeclareDocumentEnvironment
\NewDocumentEnvironment
\RenewDocumentEnvironment
\ProvideDocumentEnvironment

Very similar for environments.

```
\cs_new_protected:Npn \DeclareDocumentEnvironment #1#2#3#4 {
  \xparse_declare_env:nnnn {#1} {#2} {#3} {#4}
}
\cs_new_protected:Npn \NewDocumentEnvironment #1#2#3#4 {
⟨/initex | package⟩
```

```
⟨*initex⟩
  \cs_if_exist:cTF { environment_begin_ #1 :w } {
⟨/initex⟩
⟨*package⟩
  \cs_if_exist:cTF {#1} {
⟨/package⟩
⟨*initex | package⟩
    \msg_kernel_error:nnx { xparse }
      { environment-already-defined } {#1}
  }{
    \xparse_declare_env:nnnn {#1} {#2} {#3} {#4}
  }
}
\cs_new_protected:Npn \RenewDocumentEnvironment #1#2#3#4 {
⟨/initex | package⟩
⟨*initex⟩
  \cs_if_exist:cTF { environment_begin_ #1 :w } {
⟨/initex⟩
⟨*package⟩
  \cs_if_exist:cTF {#1} {
⟨/package⟩
⟨*initex | package⟩
    \xparse_declare_env:nnnn {#1} {#2} {#3} {#4}
  }{
    \msg_kernel_error:nnx { xparse }
      { environment-not-yet-defined } {#1}
  }
}
\cs_new_protected:Npn \ProvideDocumentEnvironment #1#2#3#4 {
⟨/initex | package⟩
⟨*initex⟩
  \cs_if_exist:cF { environment_begin_ #1 :w } {
⟨/initex⟩
⟨*package⟩
  \cs_if_exist:cF { #1 } {
⟨/package⟩
⟨*initex | package⟩
    \xparse_declare_env:nnnn {#1} {#2} {#3} {#4}
  }
}
```

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

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

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

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

`\IfNoValue`*TF* Simple re-naming.

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

`\IfValue`*TF* Inverted logic.

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

`\NoValue` The marker for no value being give: this can be typeset safely. This is coded by hand as making it `\protected` ensures that it will not turn into anything else by accident.

```
\cs_new_protected:Npn \NoValue { -NoValue- }
```

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

```
\cs_new:Npn \ProcessedArgument { }
```

⟨/initex | 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.

D

E

F

G

I

L

M

N

P

Q