The xparse package*

FMi, CAR, DPC, MH

Abstract

The interfaces described in this document are not meant to be final but only as a basis for discussion. Building productive applications using the current code is discouraged.

1 Interface

This package implements high-level interface commands for class file writers which allows the separation of formatting commands (typically instances of so-called 'templates') and their arguments from the signature of document-level commands.

This works by declaration commands that provide a general specification method for the typical LaTeX syntax, e.g., star-form, optional arguments, and mandatory arguments. A command (or environment) declared in this way parses the input according to its spec and presents its findings in a normalized way for further processing.

1.1 Argument spec

An argument specification is a list of letters each representing a type of argument, i.e., `m` is a mandatory argument (surrounded by braces), `o` an optional argument (surrounded by brackets if present), and `s` represents a star (which might be present or not). Thus the argument spec for headings as implemented by `\@startsection` in standard LaTeX would be represented by the three letters `som`. Other argument types are available and can be added at will. For a complete list of built-in argument types see the next section.

There is one important argument specifier worth taking note of. The `P` does not behave as a normal argument specifier but allows the next argument to take an argument containing the `\par` token which is not allowed by default.

*This file has version number 748, last revised 2008/08/03.

1.2 Parsing results

To normalise the result of parsing the input according to an argument specification it is important to uniquely identify all arguments found. For this reason each parsing operation initiated by one of the argument spec letters will result in an identifiable output as follows:

m will return the parsed argument surrounded by a brace pair, i.e., will normally be the identity;

o will return the parsed argument surrounded by a brace pair if present. Otherwise it will return the token `\NoValue`;

O{default} will return the parsed argument surrounded by a brace pair if present. Otherwise it will return the `{default}` as specified above.

S{⟨*symbol*⟩} will return either the token `\c_true` or `\c_false` depending on whether or not the next token is ⟨*symbol*⟩.

s will return either the token `\c_true` or `\c_false` depending on whether or not a star was parsed. This is just a shorthand for `S{*}`.

c will parse the syntax `(`⟨*x*⟩`,`⟨*y*⟩`)`, i.e., a coordinate pair and will return `{{`⟨*x*⟩`}{`⟨*y*⟩`}}` as the result. If no open parentheses is scanned an error is signalled.

C{{x-default}{y-default}} behaves like `c`, i.e., parses a coordinate pair if present. If the coordinate pair is missing it returns the default values instead.

l Reads everything up to the first left brace as the argument.

For example, given the spec `soomO{default}` the input `*[Foo]{Bar}` would be parsed as `\c_true{Foo}\NoValue{Bar}{default}`. In other words there will be always exactly the same number of brace groups or tokens as the number of letters in the argument spec.

There are a few exceptions to this rule as the rôle of following letters is to affect how the next argument is read.

> takes a mandatory argument like `{P}` or `{PW}` and inserts these extra argument specifiers so that they act on the next regular argument type. See below.

P will cause the next argument to allow `\par` in it's argument. Thus the specification `m>{P}mm` will cause the command to read three mandatory arguments but only the second can contain the `\par` token.

W will cause the next argument to *not* ignore a space when trying to scan ahead for a special symbol such as a `*` or an optional argument. This is mostly useful when the last argument(s) of a document command are optional (see example later on).

1.3 Applying the parsing results

Since the result of the parsing is a well defined number of tokens or brace groups it is easy to pass them on in any order to any processing function. To this end the tokens or brace groups are associated with the standard argument specifiers in TEX macros, i.e., `#1`, `#2`, and so forth. This limits the argument specification to a maximum number of 9 letters, but for practical applications this should be sufficient.

1.4 The class designer interface

This package provides commands for declaring commands and environments to be used within the document body.

`\DeclareDocumentCommand` The `\DeclareDocumentCommand` declaration takes three arguments. The first argument is the name of the command to be declared, the second is the argument specification in the syntax described above, and the third is the action to be carried out once the arguments are parsed. Within the third argument `#1`, `#2`, etc. denote the result of the parsing, e.g.,

```
\DeclareDocumentCommand{\chapter}{som}
  { \IfBooleanTF {#1}
      { \typesetnormalchapter {#2}{#3} }
      { \typesetstarchapter   {#3} }
  }
```

would be a way to define a `\chapter` command which would essentially behave like the current LATEX 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.

Of course something like the `\IfNoValueTF` test could also be placed inside a function that would process all three arguments, thus using the templates and their instances as provided by the `template` package such a declaration would probably look more like the following example:

```
\DeclareDocumentCommand{\chapter}{som}
  { \UseInstance {head} {A-head-main} {#1} {#2} {#3} }
```

Using the `\DeclareDocumentCommand` interface it is easy to modify the document-level syntax while still applying the same layout-generating functions, e.g., a class that would not support optional arguments or star forms for the heading commands could define `\chapter` like this:

```
\DeclareDocumentCommand{\chapter}{m}
  { \UseInstance {head} {A-head-main} \BooleanFalse \NoValue {#1} }
```

while a class that would allow for an additional optional argument (for whatever reason) could define it like that:

```
\DeclareDocumentCommand{\chapter}{somo}
  { \doSomethingWithTheExtraOptionalArg {#4}
    \UseInstance {head} {A-head-main} {#1} {#2} {#3} }
```

\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand

If \DeclareDocumentCommand is used on an already defined command, it'll overwrite it. The commands \NewDocumentCommand, \RenewDocumentCommand and \ProvideDocumentCommand have identical syntax to that of \DeclareDocumentCommand with the exception that \New... will only define the command if it is undefined, \Renew... will only redefine an existing command and \Provide... will only define the command if it is already undefined.

\DeclareDocumentEnvironment
\NewDocumentEnvironment
\RenewDocumentEnvironment
\ProvideDocumentEnvironment

The \DeclareDocumentEnvironment declaration is similar to \DeclareDocumentCommand except that it takes four arguments: the first being the environment name (without a backslash), the second again the argument-spec, and the third and forth are the actions taken at start aqnd end of the environment. The parsed arguments are available to both the start and the finish as #1, #2, etc.

All commands and environments defined with the above commands are automatically robust.

1.5 A different interface (for class design?)

Combining signature and top-level definition of a command \foo as in

```
\DeclareDocumentCommand \foo { m c m }
{ \typeout{1:#1}
  \typeout{2:#2}
  \typeout{3:#3}
}
```

is fine in certain cases, e.g., if the user wants to declare a few commands this way in the preamble of a document. However in a class file it probably better to completely separate signature (i.e., argument #1 and #2) from top-level implementation (i.e., argument #3).

```
\DeclareDocumentCommandInterface \foo {bar} { m c m }
```

The above now declare only the signature of the command \foo and states that the implementation is to be found under the label bar.

A bunch of such statements would then for the first time clearly define a document class (i.e., what a document class instance needs to define to be compliant).

An instance would then consist of a lot of declarations of the type

```
\DeclareDocumentCommandImplementation {bar} {3}
{ \typeout{1:#1}
  \typeout{2:#2}
  \typeout{3:#3}
}
```

i.e., defining how to format things being referenced as part of the signature.

The functions `\DeclareDocumentCommandInterface` and `\DeclareDocumentCommandImplementat` are implemented for now but it is not sure if they will stay. If a document command is defined with `\DeclareDocumentCommandInterface` but there is no matching interface it prints the label in quotes as ``bar''.

1.6 Checking for a value or boolean

It would make perfect sense to make `\NoValue` a LaTeX3 quark but as there is a real danger of it getting executed by accident it is best not to as it would result in an infinite loop.

`\NoValue` For this reason `\NoValue` is defined to expand to the string `-NoValue-` which would get typeset if ever executed thus clearly indicating the type of error the writer made.

However this makes testing for this token slightly complicated as in that case the test

```
\tlp_set:Nn \l_seen_tlp {#1}
\tlp_set:Nn \l_hidden_novalue_tlp {\NoValue}
\tlp_if_eq:NNTF \l_seen_tlp \l_hidden_novalue_tlp
```

will be true if `#1` was `\NoValue` but false if `#1` itself contains a macro which contains `\NoValue`; a case that happens unfortunately very often in practice. Using an unguarded `x` type expansion to define `\l_seen_tlp` is out of the question as `#1` is typically either `\NoValue` or arbitrary user input for which we can't properly guard it unless we are sure people only use robust commands. When running a pure LaTeX3 format where all document commands are robust this is perhaps something to be looked at again but when running on top of LaTeX 2_ε we have to be careful.

Therefore we use a somewhat different procedure here, which goes like this:

1. Store `#1` in a temporary variable and check if it equals a variable containing `\NoValue`. If true, execute the ⟨*true*⟩ code, otherwise go to 2.

2. Peek at the first token in `#1`. If this is a macro taking no arguments expand `#1` once and go to 1. Otherwise execute the ⟨*false*⟩ code.

The reason for the careful peeking ahead is that `#1` may very well be a macro taking arguments and it is not certain that these arguments are present! Hence this could lead to the well-known

```
Argument of \XXX has an extra }
```

and we wouldn't want that... Goes without saying that this procedure is quite tedious but usually it will exit after the first time `#1` has been inspected. One additional test could be added, namely that `#1` should also contain exactly one token but I don't think that is going to matter much.

\IfNoValueTF
\IfNoValueT
\IfNoValueF
\IfValueTF
\IfValueT
\IfValueF

These macros are used for checking if an optional argument was present as in the example below.

```
\DeclareDocumentCommand\testcmd{om}{
  \IfNoValueTF{#1}{``#2''}{#1,#2}
}
```

\IfBooleanTF
\IfBooleanT
\IfBooleanF

As mentioned earlier, the parsing result for a symbol argument like `s` or `S{+}` is represented as the one of the tokens `\c_true` or `\c_false` respectively. To test for these values the macro `\IfBooleanTF` can be used. It expects as its first argument either `\c_true` or `\c_false` and executes its second or third argument depending on this value. `\IfBooleanT` and `\IfBooleanF` are obvious shortcuts.

At one point in time I thought that one can represent everything using `\NoValue`, e.g., for the star case either return `*` or `\NoValue`. However, this slows down processing of commands like `\\*` considerably since they would then have to use a slower internal test instead of a fast two-way switch.

1.7 New argument types and shorthands

\DeclareArgumentType

New argument types can be added at in a fairly straight forward manner with the command `\DeclareArgumentType`. This command takes seven arguments: `#1` is a symbol denoting the argument type, `#2` is the token the scanner should look for, `#3` is one of the words `meaning`, `charcode` or `catcode` which is handed down to the underlying `\peek_meaning:NTF` or `\peek_charcode:NTF` etc. functions. For instance you can use the `charcode` version if you want to pick up a literal `*` and not just something that has the same *meaning* as a regular `*`. `#4` is for executing a special action (like an error message) if no argument is found, `#5` is the default value in case of a missing argument, `#6` is what the scanner will encounter, and finally `#7` is what the scanner grabs from the argument type. This is perhaps clearer with an example... This is how the `o` type is implemented:

```
\DeclareArgumentType o [ {meaning} {} {\NoValue} {#1[#2]} {#2}
```

In a similar fashion a `b` type expecting its argument inside `<` and `>` would be defined as

```
\DeclareArgumentType b < {meaning} {} {\NoValue} {#1<#2>} {#2}
```

\DeclareArgumentTypeWithDefault

`\DeclareArgumentTypeWithDefault` is even simpler as this type expects the default value to be input at the time of argument specification. Thus arguments `#3` and `#4` from above are unnecessary and are simply absent. Thus the `O` type is implemented[1] as

```
\DeclareArgumentTypeWithDefault O [ {meaning}{#1[#2]} {#2}
```

[1] I would say it is at best wishful thinking that the interface could be something like `\DeclareArgumentTypeWithDefault C{2}{meaning}{(#1,#2)}{{#1}{#2}}` as it would be quite troublesome implementation-wise and the current syntax isn't that difficult to use.

and again a similar B type could be implemented as

```
\DeclareArgumentTypeWithDefault B < {meaning}{#1<#2>} {#2}
```

\DeclareArgumentTypeShorthand

There is also a possibility to define a shorthand for a specific sequence of arguments with the command \DeclareArgumentTypeShorthand. As an example the s argument type is just a different way of saying S{*} so it's simply implemented as

```
\DeclareArgumentTypeShorthand s {S{*}}
```

Similarly one could add a shorthand M for inserting five mandatory arguments where the latter is allowed to take \par in its argument:

```
\DeclareArgumentTypeShorthand M {mmmm>{P}m}
```

\DeclareSymbolArgument

Finally there is also the command \DeclareSymbolArgument which defines the low level interface to looking ahead for specific symbols and removing them. For instance the S type argument is declared as

```
\DeclareSymbolArgument S{meaning}
```

So symbols picked up by this type will compare the meaning of the tokens. One could do a

```
\DeclareSymbolArgument A{charcode}
```

and then the A type would compare character codes instead. For instance a command which expects a + symbol would return ⟨*true*⟩ if a + of any catcode is detected, not just catcode 12 as is usual.

1.8 Some comments on the need for the O specifier

With \newcommand there is the possibility of specifying a default for an optional argument which is stored away in a more or less efficient manner. For example below is the old definition of \linebreak as can be found in the LaTeX2e kernel:

```
\def\linebreak{\@testopt{\@no@lnbk-}4}
\def\@no@lnbk #1[#2]{%
  \ifvmode
    \@nolnerr
  \else
    \@tempskipa\lastskip
    \unskip
    \penalty #1\@getpen{#2}%
    \ifdim\@tempskipa>\z@
      \hskip\@tempskipa
```

```
      \ignorespaces
    \fi
  \fi}
```

Ignoring for the moment that the above is slightly optimised an expansion of this code under `\tracingall` will result in about 90 lines of tracing output. If we reimplement this using `\DeclarDocumentCommand\linebreak{o}` we have to use `\IfNoValueTF` to find out if an argument was present which (because of the careful expansion we do internally) results in about twice as much of tracing lines. In contrast using `O{4}` as below we end up with 110 lines, which seems roughly the price we have to pay for the extra generality available (though this could perhaps even be reduced by a better implementation of the parsing machine as originally done by David, before i talked him into adding support for arguments in the end code of an environment).[2]

```
\DeclareDocumentCommand\linebreak { O{4} }
 {
  \ifvmode
    \@nolnerr
  \else
    \@tempskipa\lastskip
    \unskip
 %    \IfNoValueTF{#1}
 %       {\break}
 %       {\penalty -\@getpen{#1}}
    \penalty -\@getpen{#1}
    \ifdim\@tempskipa>\z@
      \hskip\@tempskipa
      \ignorespaces
    \fi
  \fi
 }
```

1.9 A somewhat complicated example

This example reimplements the `\makebox` and `\framebox` interface of LaTeX both of which are unfortunately quite overloaded syntactically. For this reason the example isn't meant to show good coding practice but to show the power of the interface even though applied in a somewhat bad way.

`\makebox` and `\framebox` support the following different document syntax forms:

- `\makebox{A}`: only a single mandatory argument.
- `\makebox[20pt]{B}`: one optional argument specify the box width.
- `\makebox[30pt][r]{C}`: two optional arguments the second specifying the text position within the box being made (l,c,r being allowed with c being the default).

[2] These comments were made before the extensive changes in expl3 and xparse in 2005.

- `\makebox(20,30){D}`: within picture mode the width is specified not as an optional argument in brackets but as a coordinate pair.
- `\makebox(0,0)[lt]{E}`: in that case an optional argument following coordinate pair denotes the placement within the box which can have different values compared to case C above.

To cater for this overloaded structure we can define `\makebox` to be something like the following:

```
\DeclareDocumentCommand \makebox { C{\NoValue} o O{c} m}
 { \IfNoValueTF{#1}
     { \ltx@maketextbox{#2}{#3}{#4} }
     { \ltx@makepicbox #1  {#2}{#4} } }
```

In other words: if we do not see a coordinate pair first (i.e., first argument is `\NoValue` then we expect up to two optional arguments (the width of the box or `\NoValue` if not given and the placement specifier with a default of `c` if not given) and one mandatory one which is the text. In that case we pass argument 2 to 4 to an internal function `\ltx@maketextbox` which builds the text box.

Otherwise, if the first argument is a coordinate pair we parse an optional argument denoting the placement specifier. Since `xparse` doesn't support variant syntax we actually parse for another optional argument (which has no meaning in that case and is in fact ignored if present) followed by a mandatory one containing the box text. In that case we pass the coordinate pair (`#1`), the specifier (`#2`), and the text (`#4`) to the function `\ltx@makepicbox` which builds the picture box. Note the special handling of the coordinatesD: which are passed without surrounding braces to `\ltx@makepicbox`: since the coordinate argument looks like `{x-val}{y-val}` the receiving function `\ltx@makepicbox` gets the `x-val` as argument one and the `y-val` as argument two.

A definition for `\framebox` would look more or less identical except that we would need to pass the arguments to slightly different internal functions. The alternative is to give the internal functions an extra argument that controls whether or not a frame is bying built.

`\ltx@makepicbox`

So here is a possible implementation of `\ltx@makepicbox` that builds a picture box with or without frame. It takes the following mandatory arguments:

1. x-part of coordinate
2. y-part of coordinate
3. placement specifier, e.g., `lt` or `\NoValue`
4. text of box
5. the token `\frame` (if a frame should surround the box) or the token `\@firstofone` — not pretty i agree

```
\def\ltx@makepicbox#1#2#3#4#5
  {
   #5{
     \vbox to#2\unitlength
      {\let\mb@b\vss \let\mb@l\hss\let\mb@r\hss
       \let\mb@t\vss
       \IfNoValueF{#3}
        {
         \@tfor\reserved@a :=#3\do{
           \if s\reserved@a
             \let\mb@l\relax\let\mb@r\relax
           \else
             \expandafter\let\csname mb@\reserved@a\endcsname\relax
           \fi}%
        }
       \mb@t
       \hb@xt@ #1\unitlength{\mb@l #4\mb@r}
       \mb@b
       \kern\z@}
   }
  }
```

`\ltx@maketextbox` For the text case the internal function could take the following mandatory arguments:

1. width of the box or `\NoValue` (denoting to build the box at natural width)
2. placement specifier, e.g., `l`. (No test for `\NoValue` being undertaken)
3. text of box
4. the token `\fbox` (if a frame should surround the box) or the token `\mbox`

The actual code is taken straight from the current LaTeX kernel and looks kind of scary.

```
\def\ltx@maketextbox#1#2#3#4
 {
  \IfNoValueTF{#1}
     {#4{#3}}
     {
      \leavevmode
      \@begin@tempboxa\hbox{#3}
       \setlength\@tempdima{#1}
       \ifx#4\fbox
         \setbox\@tempboxa\hb@xt@\@tempdima
              {\kern\fboxsep\csname bm@#2\endcsname\kern\fboxsep}
         \@frameb@x{\kern-\fboxrule}
       \else
         \hb@xt@\@tempdima{\csname bm@#2\endcsname}
       \fi
```

```
      \@end@tempboxa
    }
}
```

\makebox
\framebox

Given the above internal functions the declarations of \makebox and \framebox would then look like this:

```
\DeclareDocumentCommand \makebox { C{\NoValue} o O{c} m}
 { \IfNoValueTF{#1}
     { \ltx@maketextbox{#2}{#3}{#4}\mbox }
     { \ltx@makepicbox #1  {#2}{#4}\@firstofone } }

\DeclareDocumentCommand \framebox { C{\NoValue} o O{c} m}
 { \IfNoValueTF{#1}
     { \ltx@maketextbox{#2}{#3}{#4}\fbox }
     { \ltx@makepicbox #1  {#2}{#4}\frame } }
```

Here's another example which shows how one can use the W specifier. The amsmath environments use their own definition of \\ which goes under the name \math@cr. As there is a good chance the next line of math begins with either an asterisk or something in square brackets this often leads to errors. The following example definition works (I *did* try it).

```
\ExplSyntaxOn
\makeatletter
\DeclareDocumentCommand\math@cr{>{W}s >{W}O{\z@}}{
  \IfBooleanTF{#1}
  { \global\@eqpen\@M }{
    \global\@eqpen
      \ifnum\dspbrk@lvl <\z@
        \interdisplaylinepenalty
      \else
        -\@getpen\dspbrk@lvl
      \fi
  }
  \math@cr@@@
  \noalign{\vskip#2\relax}
}
\makeatother
\ExplSyntaxOff
```

Note that this definition automatically avoids problems within the alignments even if it is called as \\&.

1.10 Open issues

In this section unresolved issues or ideas to think about and perhaps implement are collected. There is no particular order to them.

- Support for parsing verbatim type of arguments was considered (and actually implemented at one stage. E.g., `g{`⟨*prepare-parsing*⟩`}` where `g` first run ⟨*prepare-parsing*⟩ (which might involve `catcode` changes)[3], then look at the next token: if that would be a `{` it would scan a brace delimited argument (reverting catcodes of `{` and `}` if needed) otherwise would scan for an argument delimited by that token so that something like `\verb+%\+` would scan `%\` as its argument assuming that the ⟨*prepare-parsing*⟩ turned `%` and \ into non-letters.

  All kind of nasty problems lurking especially no proper error checks possible and of course as we all know such commands would then not work inside arguments of other commands. At least they would have some restrictions.

  Also no way to make the parsing smart by not accepting newlines as part of the code (`\verb` does this right now and this is really helpful as it catches runaways nicely).

- xparse should probably go into the LaTeX3 kernel in which case it should also provide the definitions of `\begin` and `\end` and this raises a few questions: Will environments be defined as control sequences `\foo` and `\endfoo`? If they are do we then want to include support for a separate command `\foo` and if so what about using the environment forms in special scanner mechanisms where we *can't* use `\begin{foo}`? In those cases one could use a special command like what the fancyvrb package provides with the command `\VerbatimEnvironment` and all packages that define such special arguments should then also make sure that the `\foo` command issued on its own should cause an error. Another solution would be to simply define environments as `\beginfoo` and `\endfoo` and then let `\begin` check if it was used to call an environment or a command. If the latter is the case then insert grouping as that could be handled by `\beginfoo` itself. When `\end` is reached check if `\endfoo` exists. If it does then use that, otherwise we were probably using a normal command which should just close the group.

1.11 Rejected ideas—and why

Let's say the document command `\testmo` has signature `mo`. If we call it as `\testmo{arg} [opt]` a following space will be obeyed. However calling it with the mandatory argument only as in `\testmo{arg}` will gobble a following space because the peek ahead in the `o` type ignores spaces. We can record that a space has been gobbled when peeking ahead and we can also put it back in if needed but I have only ever seen one request for this and in that case the better solution was to use a signature `mWo` so that no spaces are allowed between the mandatory and the optional argument. One could even argue that this is the most appropriate thing to do if the user is really that sensitive to spaces! In any case it is not implemented here as it creates an unnecessary overhead for something that is never used and a simple solution exists.

[3] urg horror!

1.12 Experimental features

Experimental features in an experimental packages... Needless to say that you can't rely on them staying!

One new feature is what I decided to call "pseudo arguments". Strictly speaking these aren't real arguments but instead read by removing a begin group token and then using `\tex_aftergroup:D` to regain control. The definition of `\footnote` in plain TEX uses this so that people can use `\verb` and the like inside the footnotes.

`\DeclarePseudoArgument` The command `\DeclarePseudoArgument`takes four arguments: `#1` is a name for the pseudo argument, `#2` is the number of arguments it takes, `#3` is the action to be taken right before the left brace is removed and `#4` is what to be done when the right brace has been read and we regain control. Note that both `#3` and `#4` may use arguments! Here is a silly example:

```
\DeclarePseudoArgument{boxtest}{1}
{Before:~`#1',\hbox_set_inline_begin:N \l_tmpa_box }
{\hbox_set_inline_end: \text_put_sp:  the~ box:~
  \hbox_unpack_and_clear:N\l_tmpa_box ,~
  After:~`#1'}
```

`\UsePseudoArgument` Now when you want to use it, you simply call it with the macro `\UsePseudoArgument`, which as its first argument takes a name and the remaining depend on how many arguments the pseudo argument was defined to have. Hence we could use our `boxtest` definition like this:

```
\DeclareDocumentCommand\sillyboxtest{m}{
  Testing~#1:~\UsePseudoArgument{boxtest}{#1}
}
```

Then calling `\sillyboxtest{AB}{a\verb*+% $%&\+b}` produces

> Testing AB: Before: 'AB', the box: a%␣$%&\b, After: 'AB'

`\UsePseudoArgument` should always come last in the document command definition.

2 Implementation

Versions prior to and including 1.19 contained several different implementations. These are gone now but can be looked at by getting an older version of the package from the CVS repository.

First the required packages.

```
1 ⟨*package⟩
2 \ProvidesExplPackage
3   {\filename}{\filedate}{\fileversion}{\filedescription}
4 \RequirePackage{l3tlp,l3num,l3toks,l3prg,l3int,l3seq,l3token}
```

2.1 Error and warning messages

Here we define the error message we're going to use in this package.

rse_already_defined_error_msg:N
rse_not_yet_defined_error_msg:N
rse_begins_with_end_error_msg:N
se_unknown_arg_type_error_msg:N
umber_of_arguments_error_msg:Nn
ommand_implementation_warning:n

The names should give the function's purpose away.

```
\def_new:NNn \xparse_already_defined_error_msg:N 1 {
  \xparse_error:x {
    Command~name~`\token_to_string:N #1'~ already~defined!
  }
}
\def_new:NNn \xparse_not_yet_defined_error_msg:N 1 {
  \xparse_error:x {
    Command~`\token_to_string:N #1'~ not~ yet~defined!
  }
}
\def_new:NNn \xparse_begins_with_end_error_msg:N 1 {
  \xparse_error:x {
    Command~`\token_to_string:N #1'~begins~with~
    `\token_to_string:N \end'!
  }
}
```

This one is for when we've encountered an unknown argument type in the argument specification. We can try to recover by putting in an `m` type instead and hope for the best.

```
\def_new:NNn \xparse_unknown_arg_type_error_msg:N 1{
  \xparse_error:x {
    Unknown~ argument~ type~ `#1'~
    I'll~ substitute~ it~ with~ `m'~ for~ now.~ Fingers~ crossed...
  }
}
```

A message for people using the wrong number of arguments.

```
\def_new:NNn \xparse_no_command_implementation_warning:n 1 {
  \xparse_warning:x {No~ implementation~ for~ `#1'~ defined}
}
```

\xparse_error:x
\xparse_warning:x

Here's how we produce the error messages and warnings currently. This awaits a proper error message module!

```
\def_new:NNn \xparse_error:x 1{\tex_errmessage:D {xparse~error:~#1}}
\def_new:NNn \xparse_warning:x 1{\io_put_term:x{xparse~warning:~#1}}
```

2.2 Checking for valid command names

\xparse_if_definable:NT
\xparse_if_definable:cT

A definable command is either `\c_undefined` or `\scan_stop:` and furthermore we won't allow it to start with `\end`. If the command is definable we do what was asked for otherwise we give an appropriate error message.

```
\def_new:NNn \xparse_if_definable:NTF 1 {
```

First check if the control sequence is free.

```
  \cs_free:NTF #1
```

If free check if it begins with \end.

```
  {
    \xparse_begins_with_end:NTF #1
    { \xparse_begins_with_end_error_msg:N #1 \use_arg_ii:nn }
    \use_arg_i:nn
  }
```

If not free give an error message.

```
  { \xparse_already_defined_error_msg:N #1 \use_arg_ii:nn }
}
\def_new:NNn \xparse_if_definable:cTF 0 {
  \exp_args:Nc \xparse_if_definable:NTF
}
```

\xparse_if_redefinable:NTF
\xparse_if_redefinable:cTF

A re-definable command is almost the same as above but here we demand that it already exists.

```
\def_new:NNn \xparse_if_redefinable:NTF 1 {
  \cs_free:NTF #1
  { \xparse_not_yet_defined_error_msg:N #1 \use_arg_ii:nn }
  {
    \xparse_begins_with_end:NTF #1
    { \xparse_begins_with_end_error_msg:N #1 \use_arg_ii:nn }
    \use_arg_i:nn
  }
}
\def_new:NNn \xparse_if_redefinable:cTF  0 {
  \exp_args:Nc \xparse_if_redefinable:NTF
}
```

\xparse_begins_with_end:NTF
\xparse_begins_with_end_aux:N

Finally we need the magic function for checking if the control sequence begins with \end. We wish to determine if the control sequence in question begins with \end which is forbidden. The end goal is to get a control sequence consisting of the first three letters to test against \end with \cs_if_eq_name_p:NN. The function \cs_to_str:N will remove the \ from a command name and return the rest of the characters with category code 12. Thus using an f type expansion to get the first three characters is perfectly safe as it'll stop right there. Then we turn these three letters into a control sequence with a simple expansion which is then tested with \xparse_begins_with_end_aux:N.

```
\def_new:NNn \xparse_begins_with_end:NTF 1 {
```

We do it all in a group as there is no need to fill up the hash table with weird three letter control sequences.

```
  \group_begin:
    \exp_args:Nc \xparse_begins_with_end_aux:N {
      \tlist_head_iii:f { \cs_to_str:N #1 ??}
    }
}
```

After the above expansion tricks we now have a real control sequence to test against \end. We end the group again.

```
\def_new:NNn \xparse_begins_with_end_aux:N 1 {
  \if:w \cs_if_eq_name_p:NN #1 \end
    \group_end:
    \exp_after:NN \use_arg_i:nn
  \else:
    \group_end:
    \exp_after:NN \use_arg_ii:nn
  \fi:
}
```

Note that these definitions make no use of temporary variables.

2.3 The low level definitions

First some helper functions

`\l_xparse_grabbed_args_toks`
`parse_end_environment_args_toks`
`\l_xparse_mandatory_args_int`
`\l_xparse_total_args_int`

We need some ⟨*token*⟩ registers to keep track of things. We also allocate an ⟨*int*⟩ register to keep track of the number of arguments.

```
\toks_new:N \l_xparse_grabbed_args_toks
\toks_new:N \l_xparse_end_environment_args_toks
\int_new:N \l_xparse_mandatory_args_int
\int_new:N \l_xparse_total_args_int
```

`se_declare_document_command:Nnn`
`se_declare_document_command:cnn`

`\xparse_declare_document_command:Nnn` is a two step procedure. The user level command \⟨*cmd*⟩ contains the argument grabbers while the internal command \\⟨*cmd*⟩ holds the actual definition of what to do with the arguments. Naturally this calls for every such user level command to be robust which is done using the ε-TEX protection feature.

```
\def_long:NNn \xparse_declare_document_command:Nnn 3{
```

First we prepare the signature.

```
  \xparse_prepare_signature:n {#2}
```

Now for the document command. Make it robust and make sure it starts with `\toks_set:Nn \l_xparse_grabbed_args_toks {\\`⟨*cmd*⟩`}`.

```
  \def_protected:Npx #1 {
    \exp_not:n { \toks_set:Nn \l_xparse_grabbed_args_toks }
    {\exp_not:c {\token_to_string:N #1}}
```

Then add the argument grabbers which contain code gathered when preparing the signature.

```
    \toks_use:N \l_xparse_grabbed_args_toks
```

Finally execute `\toks_use:N \l_xparse_grabbed_args_toks` which now holds the grabbed arguments enclosed in braces and starts with the internal command \\⟨*cmd*⟩.

```
    \exp_not:n{ \toks_use:N \l_xparse_grabbed_args_toks}
  }
```

Define the internal command by making it a `\long` macro with number of arguments equal to `\l_xparse_total_args_int` and definition as given by `#3`. Even

though this internal macro allows \par in its argument the decision is left to the individual argument grabbers.

```
  \def_long:cNn {\token_to_string:N #1}\l_xparse_total_args_int{#3}
}
\def_new:Npn \xparse_declare_document_command:cnn {
  \exp_args:Nc \xparse_declare_document_command:Nnn
}
```

clare_document_environment:nnnn

\xparse_declare_document_environment:nnnn is almost the same. In order to use the grabbed arguments we take them from \l_xparse_grabbed_args_toks just before they are executed. We also insert a \group_begin: as it will otherwise fail miserably in case a user tries to use it without \begin ... \end.

```
\def_long_new:NNn \xparse_declare_document_environment:nnnn 4 {
  \xparse_declare_document_command:cnn {#1}{#2}
  { \group_begin:
      \toks_set_eq:NN \l_xparse_end_environment_args_toks
                      \l_xparse_grabbed_args_toks
      #3
  }
```

We let \end⟨*envir*⟩ equal to a standard version which will do all the work.

```
  \let:cN {end #1} \xparse_parsed_end_environment:
```

Now define \end\\⟨*envir*⟩.

```
  \def_long:cNn {end \token_to_string:N \\ #1}
    \l_xparse_total_args_int{#4}
}
```

\xparse_parsed_end_environment:
se_parsed_end_environment_aux:N

\end⟨*envir*⟩ is set equal to \xparse_parsed_end_environment: which then executes \end\\⟨*envir*⟩. The \\⟨*envir*⟩ comes from the argument grabbing. The remainder of the token list is the grabbed arguments. We better make this robust as well.

```
\def_protected_new:NNn \xparse_parsed_end_environment: 0{
  \exp_after:NN \xparse_parsed_end_environment_aux:N
    \toks_use:N \l_xparse_end_environment_args_toks
  \group_end:
}
\def_new:NNn \xparse_parsed_end_environment_aux:N 1{
  \cs_use:c {end \token_to_string:N #1 }
}
```

2.4 The high level commands

Here we set up the functions for defining and redefining document commands and environments. It's all rather straight forward here so I've saved on the comments.

\DeclareDocumentCommand
\NewDocumentCommand
\RenewDocumentCommand

In case we can't do what the user wanted we gobble the next arguments from the input stream. For commands there are two arguments waiting.

```
\let_new:NN \DeclareDocumentCommand  \xparse_declare_document_command:Nnn
```

```
\def_new:NNn \NewDocumentCommand 1 {
  \xparse_if_definable:NTF #1
  { \xparse_declare_document_command:Nnn #1 }
  \use_none:nn
}
\def_new:NNn \RenewDocumentCommand 1 {
  \xparse_if_redefinable:NTF #1
  { \xparse_declare_document_command:Nnn #1 }
  \use_none:nn
}
```

\DeclareDocumentEnvironment
\NewDocumentEnvironment
\RenewDocumentEnvironment

Three arguments for environments.

```
\let_new:NN \DeclareDocumentEnvironment
              \xparse_declare_document_environment:nnnn
\def_new:NNn \NewDocumentEnvironment 1 {
  \xparse_if_definable:cTF {#1}
  { \xparse_declare_document_environment:nnnn {#1} }
  \use_none:nnn
}
\def_new:NNn \RenewDocumentEnvironment 1 {
  \xparse_if_redefinable:cTF {#1}
  { \xparse_declare_document_environment:nnnn {#1} }
  \use_none:nnn
}
```

\ProvideDocumentCommand
\ProvideDocumentEnvironment

When providing a command we just check if the control sequence is free. If it is let `\DeclareDocument...` do the rest, otherwise gobble the next arguments.

```
\def_new:NNn \ProvideDocumentCommand 1{
  \cs_free:NTF #1
  { \DeclareDocumentCommand #1}
  \use_none:nn
}
\def_new:NNn \ProvideDocumentEnvironment 1{
  \cs_free:cTF {#1}
  { \DeclareDocumentEnvironment {#1} }
  \use_none:nnn
}
```

2.4.1 A class designer interface

Below is the implementation of `\DeclareDocumentCommandInterface` and the close cousin `\DeclareDocumentCommandImplementation`. It is an open issue whether this concept should be supported.

DeclareDocumentCommandInterface

With this macro we define the document command `#1` to have signature `#3` which does the argument grabbing. The arguments are passed on to the internal command `\impl-#2`.

```
\def_long_new:NNn \DeclareDocumentCommandInterface 3{
  \xparse_prepare_signature:n {#3}
```

```
  \def_protected:Npx #1 {
    \exp_not:n { \toks_set:Nn \l_xparse_grabbed_args_toks }
    {\exp_not:c {impl-#2}}
    \toks_use:N\l_xparse_grabbed_args_toks
    \exp_not:n{ \toks_use:N \l_xparse_grabbed_args_toks}
  }
```

Define the internal command by making it a `\long` macro as we did for `\DeclareDocumentCommand` but give it just a default definition instead (usually an error message).

```
  \def_long:cNn {impl-#2} \l_xparse_total_args_int
    {\xparse_undefined_command_implementation:n{#2}}
}
```

efined_command_implementation:n What to do if no implementation is made and the command is called. Let's just typeset the name in quotes indicating that something is wrong.

```
\def_new:NNn \xparse_undefined_command_implementation:n 1{
  ``#1''
%  \ensuremath{\langle\textit{#1}\rangle}
  \xparse_no_command_implementation_warning:n {#1}
}
```

reDocumentCommandImplementation Defines the internal command `\impl-#1` with `#2` arguments and definition `#3`.

```
\def_long_new:NNn \DeclareDocumentCommandImplementation 3{
  \def_long:cNn {impl-#1}#2{#3}
}
```

2.5 Creating the signature

\xparse_prepare_signature:n The actual preparation of the signature is always the same but we have different ways of handling it afterwards, so we make it a separate function.

```
\def_new:NNn \xparse_prepare_signature:n 1 {
```

Initialize the counter taking care of the number of arguments. In case we reach more than nine we will give an error message later on.

```
  \int_zero:N \l_xparse_total_args_int
```

Clear the token register we use when building the signature and the number of "normal" mandatory arguments. Also clear out the special markers and initialize the two booleans to ⟨*false*⟩.

```
  \toks_clear:N \l_xparse_grabbed_args_toks
  \int_zero:N \l_xparse_mandatory_args_int
  \bool_gset_false:N \g_xparse_insert_marker_bool
  \bool_gset_false:N \g_xparse_allow_par_bool
  \tlp_gset_eq:NN \g_xparse_ignore_marker_tlp
                  \g_xparse_ignore_spaces_marker_tlp
```

Then call `\xparse_parse_signature:n`.

```
  \xparse_parse_signature:n #1 \q_nil
}
```

2.6 The argument types

All argument types must be denoted by single letters.[4] We have three categories of letters: Special markers, basic argument types, and shorthands.

For certain argument types where we peek ahead in the token stream we have a multitude of choices for how the macro should work: Should it ignore spaces or other tokens? Which test should it use for comparing tokens: meaning, catcode or charcode? Because of these difficulties every argument that require peeking ahead must be declared as one of the three test types.

2.6.1 Special markers

Since there are different markers that may be used at the same time, we use the a special "insert" marker `>` to do this.

`\g_xparse_ignore_marker_tlp` The current ignore marker is contained in this string.

```
\tlp_new:Nn \g_xparse_ignore_marker_tlp {}
```

`\g_xparse_ignore_marker_seq` We start out by declaring a new sequence to contain all the different types of ignore functions. Each element should be the missing part of a `\peek_meaning`⟨*text*⟩`:NTF` function.

```
\seq_new:N \g_xparse_ignore_marker_seq
```

`\g_xparse_insert_marker_bool` A boolean to check if we just saw an insert marker.

```
\bool_new:N \g_xparse_insert_marker_bool
```

`\xparse_add_arg_type_>:` The insert marker is called when this is seen. Is has a funny definition but it works. `#1` is some test in the loop function and `#2` is the list of markers.

```
\def_new:cpn {xparse_add_arg_type_>:} #1
  \xparse_read_arg_type_or_grab_default:n #2{
```

Insert pending `m` args first.

```
  \xparse_add_remaining_m_args:
```

Then we set the boolean true and subtract one from the argument count.

```
  \bool_gset_true:N \g_xparse_insert_marker_bool
  \int_decr:N \l_xparse_total_args_int
```

Then go through the list and call each subtype if it exists.

```
  \tlist_map_inline:nn{#2}{
    \xparse_check_and_add_argument_type:N ##1
  }
```

Finally call the loop again.

```
  \xparse_parse_signature:n
}
```

[4] You can use a multi-letter sequence to denote a shorthand but it requires you to put two brace groups around it in the signature and it's not officially supported.

\xparse_add_ignore_marker:Nnn A small function for adding a new ignore type marker.

```
\def_new:NNn \xparse_add_ignore_marker:Nnn 3{
  \tlp_new:cn {g_xparse #2 _marker_tlp}{#3}
  \seq_gpush:NC \g_xparse_ignore_marker_seq {g_xparse #2 _marker_tlp}
  \def_new:cpn {xparse_add_arg_type_#1:}{
    \tlp_gset_eq:Nc \g_xparse_ignore_marker_tlp {g_xparse #2 _marker_tlp}
  }
}
```

Here we add some ignore markers.

```
\xparse_add_ignore_marker:Nnn W{_ignore_nothing}{}
\xparse_add_ignore_marker:Nnn i{_ignore_spaces}{_ignore_spaces}
\xparse_add_ignore_marker:Nnn I{_ignore_pars}{_ignore_pars}
```

\xparse_add_arg_type_P: This is simple, just set a boolean.

```
\def_new:NNn \xparse_add_arg_type_P: 0{
  \bool_gset_true:N \g_xparse_allow_par_bool
}
```

\g_xparse_allow_par_bool The boolean for allowing the \par token.

```
\bool_new:N \g_xparse_allow_par_bool
```

\xparse_parse_signature:n \xparse_parse_signature:n reads the signature one token/brace group at a time and builds a list of argument grabbers which is stored in \l_xparse_grabbed_args_toks. The function is recursive which is why we use a quark to delimit it.

```
\def_new:NNn \xparse_parse_signature:n 1{
```

Check if we reached the end of the signature.

```
  \quark_if_nil:NTF #1
```

If we did we may have pending m type arguments so we clear them out here. Also check if the insert marker is true because if it is, something is wrong. Otherwise we just continue.

```
  {
    \xparse_add_remaining_m_args:
  }
```

Then proceed with adding arguments.

```
  {
    \int_incr:N \l_xparse_total_args_int
    \xparse_check_and_add_argument_type:N #1
```

If the insert marker is true at this point we reset it again and also disallow \par in arguments and make all \peek functions use the _ignore_spaces versions.

```
    \bool_if:NT \g_xparse_insert_marker_bool
    {
      \bool_gset_false:N \g_xparse_insert_marker_bool
      \bool_gset_false:N \g_xparse_allow_par_bool
      \tlp_gset_eq:NN \g_xparse_ignore_marker_tlp
                      \g_xparse_ignore_spaces_marker_tlp
    }
```

After adding arguments we better set the boolean false in case we don't have an insert marker.

First we add the remaining `m` arguments.

Then we run the loop again.

```
    \xparse_read_arg_type_or_grab_default:n
  }
}
```

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

```
\def_new:NNn \xparse_check_and_add_argument_type:N 1 {
  \cs_free:cTF {xparse_add_arg_type_#1:}
  { \xparse_unknown_arg_type_error_msg:N #1
    \int_incr:N \l_xparse_mandatory_args_int
  }
```

Otherwise we just add it with its dedicated function.

```
  { \cs_use:c {xparse_add_arg_type_#1:} }
}
```

`read_arg_type_or_grab_default:n`
`\xparse_grab_default_arg:n`
`se_grab_default_arg_allow_par:n` The function for grabbing a default value. The default setting is just to tun the loop again. However, some argument types need to add a default argument so we add these as well.

```
\let_new:NN \xparse_read_arg_type_or_grab_default:n
            \xparse_parse_signature:n
\def_new:NNn \xparse_grab_default_arg:n 1{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks {{#1}}
  \let:NN \xparse_read_arg_type_or_grab_default:n
          \xparse_parse_signature:n
  \xparse_parse_signature:n
}
\def_long_new:NNn \xparse_grab_default_arg_allow_par:n 1{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks {{#1}}
  \let:NN \xparse_read_arg_type_or_grab_default:n
          \xparse_parse_signature:n
  \xparse_parse_signature:n
}
```

2.6.2 The default argument types

When scanning the signature we need to add the argument type which is done by a function `\xparse_add_arg_type_`$\langle X\rangle$`:`, where $\langle X\rangle$ is the letter denoting the argument type. If not found an error message is issued ny `\xparse_unknown_arg_type_error_msg:N`.

By default the package implements the argument types `m`, `S`, `c`, `C`, `o`, and `O` argument types as low level ones. The `s` type is a shorthand as we shall see later.

As we don't make every single `m` argument a separate grabber all other types must clear out any pending `m`s.

Mandatory arguments Below follows the implementation of mandatory arguments.

`\xparse_add_arg_type_m:` Check if the user asked for an `m` argument allowing `\par` tokens and insert it. Otherwise increment the number of non-`\long` argument grabbers.

```
\def_new:Npn \xparse_add_arg_type_m: {
  \bool_if:NTF \g_xparse_allow_par_bool
  {
    \toks_put_right:Nn \l_xparse_grabbed_args_toks {\xparse_allow_par_m:w}
  }
  { \int_incr:N \l_xparse_mandatory_args_int }
}
```

`\xparse_add_remaining_m_args:` And here is the function for clearing out normal pending `m` arguments. There is little point in adding any arguments if none are required.

```
\def_new:Npn \xparse_add_remaining_m_args: {
  \int_compare:nNnF \l_xparse_mandatory_args_int = \c_zero
  {
    \toks_put_right:Nx \l_xparse_grabbed_args_toks {
      \exp_not:c{xparse_m
        \int_use:N \l_xparse_mandatory_args_int
        :w }
    }
    \int_zero:N \l_xparse_mandatory_args_int
  }
}
```

`\xparse_m1:w` `\xparse_m2:w` `\xparse_m3:w` `\xparse_m4:w` `\xparse_m5:w` `\xparse_m6:w` `\xparse_m7:w` `\xparse_m8:w` `\xparse_m9:w` Grabbing 1–9 mandatory arguments. The one grabbing nine arguments will automatically end with `\toks_use:N \l_xparse_grabbed_args_toks` anyway so we avoid problems with that one.

```
\def_new:cpn {xparse_m1:w} #1 \l_xparse_grabbed_args_toks#2{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m2:w} #1 \l_xparse_grabbed_args_toks #2#3{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}{#3}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m3:w} #1 \l_xparse_grabbed_args_toks #2#3#4{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}{#3}{#4}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m4:w} #1 \l_xparse_grabbed_args_toks #2#3#4#5{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}{#3}{#4}{#5}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m5:w} #1 \l_xparse_grabbed_args_toks #2#3#4#5#6{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}{#3}{#4}{#5}{#6}}
  #1 \l_xparse_grabbed_args_toks
```

```
}
\def_new:cpn {xparse_m6:w} #1 \l_xparse_grabbed_args_toks #2#3#4#5#6#7{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks
                     {{#2}{#3}{#4}{#5}{#6}{#7}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m7:w} #1 \l_xparse_grabbed_args_toks#2#3#4#5#6#7#8{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks
                     {{#2}{#3}{#4}{#5}{#6}{#7}{#8}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m8:w} #1\l_xparse_grabbed_args_toks
                                                    #2#3#4#5#6#7#8#9{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks
                     {{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}
  #1 \l_xparse_grabbed_args_toks
}
\def_new:cpn {xparse_m9:w} \toks_use:N \l_xparse_grabbed_args_toks
                                                  #1#2#3#4#5#6#7#8#9{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks
                     {{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}
  \toks_use:N \l_xparse_grabbed_args_toks
}
```

`\xparse_allow_par_m:w` The `m` type allowing `\par` tokens. We only define one of them as two in a row would be quite rare.

```
\def_long_new:Npn \xparse_allow_par_m:w #1 \l_xparse_grabbed_args_toks#2{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}}
  #1 \l_xparse_grabbed_args_toks
}
```

Arguments delimited by a left brace

`\xparse_add_arg_type_l:` `\xparse_l:w` `\xparse_allow_par_l:w` This is almost exactly the same as the `m` type except we read everything up to the first left brace.

```
\def_new:Npn \xparse_add_arg_type_l: {
  \xparse_add_remaining_m_args:
  \bool_if:NTF \g_xparse_allow_par_bool
  {
    \toks_put_right:Nn \l_xparse_grabbed_args_toks {\xparse_allow_par_l:w}
  }
  { \toks_put_right:Nn \l_xparse_grabbed_args_toks {\xparse_l:w} }
}
\def_new:Npn \xparse_l:w #1 \l_xparse_grabbed_args_toks#2#{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}}
  #1 \l_xparse_grabbed_args_toks
}
\def_long_new:Npn \xparse_allow_par_l:w #1 \l_xparse_grabbed_args_toks#2#{
  \toks_put_right:Nn \l_xparse_grabbed_args_toks{{#2}}
```

```
  #1 \l_xparse_grabbed_args_toks
}
```

Adding a symbol argument type

\DeclareSymbolArgument

The actual grabber functions just issue a `\peek_meaning_remove:NTF` or similar on the symbol chosen. If found, we put a `\c_true` on the argument stack and `\c_false` otherwise.

```
\def_new:NNn \DeclareSymbolArgument 2{
  \seq_map_variable:NNn \g_xparse_ignore_marker_seq \l_tmpa_tlp {
    \def:cpx {xparse \l_tmpa_tlp _#1:w}##1##2\l_xparse_grabbed_args_toks{
      \exp_not:c{peek_ #2 _remove \l_tmpa_tlp :NTF} ##1
      {
        \exp_not:N \toks_put_right:Nn \exp_not:N
        \l_xparse_grabbed_args_toks \exp_not:N \c_true
        ##2 \exp_not:N\l_xparse_grabbed_args_toks
      }
      {
        \exp_not:N \toks_put_right:Nn \exp_not:N
        \l_xparse_grabbed_args_toks \exp_not:N \c_false
        ##2 \exp_not:N\l_xparse_grabbed_args_toks
      }
    }
  }
```

Implementing a symbol type is fairly straight forward. It needs to grab a default argument. First we must clear out any pending `m` arguments.

```
  \def:cpn {xparse_add_arg_type_#1:} {
    \xparse_add_remaining_m_args:
    \toks_put_right:Nx \l_xparse_grabbed_args_toks
    { \exp_not:c {xparse \g_xparse_ignore_marker_tlp _#1:w } }
```

Nothing special when building the signature: We always grab a single default argument.

```
    \let:NN \xparse_read_arg_type_or_grab_default:n \xparse_grab_default_arg:n
  }
}
```

It is this easy now:

```
\DeclareSymbolArgument S{meaning}
```

Adding delimited arguments

\DeclareArgumentType

Argument types that read their argument delimited as the `o` type can be defined with this function. `#1` is the letter associated with the type, `#3` is the test type we should use, i.e., meaning, charcode or catcode. `#2` is the left delimiter LaTeX will look for, `#4` can be used for an error message or such in case of a missing value, and `#5` the value inserted in case the optional argument is missing. The last two arguments control how the arguments are picked up and parsed on to the inner parsing.

```
\def_new:NNn \DeclareArgumentType 7{
```

For instance, we want `\DeclareArgumentType a{charcode}[...` to define the function `\xparse_ignore_spaces_a:w` with meaning `\peek_charcode_ignore_spaces:NTF [{...}{`. We want to define a series of functions for each of these delimited argument types. One for each of the `ignore` types plus an identical version of each of these allowing the `\par` token in the argument. All the different ignore types are stored in the sequence `\g_xparse_ignore_marker_seq` so we go through that when doing the definitions.

```
  \seq_map_variable:NNn \g_xparse_ignore_marker_seq \l_tmpa_tlp {
    \def:cpx {xparse \l_tmpa_tlp _#1:w}##1\l_xparse_grabbed_args_toks{
      \exp_not:c{peek_#3 \l_tmpa_tlp :NTF} \exp_not:N #2
      { \exp_not:c{xparse_#1_#3_help:nw}{##1} }
      {
        \exp_not:n {
          #4 \toks_put_right:Nn \l_xparse_grabbed_args_toks {#5}
        }
        ##1 \exp_not:N\l_xparse_grabbed_args_toks
      }
    }
```

And now an almost identical version which allows the `\par` token.

```
    \def_long:cpx {xparse_allow_par \l_tmpa_tlp _#1:w}##1
    \l_xparse_grabbed_args_toks{
      \exp_not:c{peek_#3 \l_tmpa_tlp :NTF} \exp_not:N #2
      { \exp_not:c{xparse_allow_par_#1_#3_help:nw}{##1} }
      {
        \exp_not:n {
          #4 \toks_put_right:Nn \l_xparse_grabbed_args_toks {#5}
        }
        ##1 \exp_not:N\l_xparse_grabbed_args_toks
      }
    }
  }
```

Here is the function for building the argument grabbing. We clear out the remaining `m` arguments first.

```
  \def:cpn {xparse_add_arg_type_#1:} {
    \xparse_add_remaining_m_args:
    \toks_put_right:Nx \l_xparse_grabbed_args_toks {
      \exp_not:c {xparse
```

Insert `_allow_par` in the csname if needed. This also applies to the argument grabber (although this should probably just be the default).

```
        \bool_if:NT \g_xparse_allow_par_bool {_allow_par}
        \g_xparse_ignore_marker_tlp
        _#1:w
      }
    }
    \let:NN \xparse_read_arg_type_or_grab_default:n \xparse_parse_signature:n
  }
```

Now all we need to do is to define the helper functions.

```
  \xparse_define_helper:Nnnn #1{#3}{#6}{#7}
}
```

\DeclareArgumentTypeDefaultValue The same as above but expects a default value in the signature. Therefore it doesn't have the arguments of the default value and what to do if it is missing like the above.

```
\def_new:NNn \DeclareArgumentTypeDefaultValue 5{
  \seq_map_variable:NNn \g_xparse_ignore_marker_seq \l_tmpa_tlp {
    \def:cpx {xparse \l_tmpa_tlp _#1:w}##1##2\l_xparse_grabbed_args_toks{
      \exp_not:c{peek_#3 \l_tmpa_tlp :NTF} \exp_not:N #2
      { \exp_not:c{xparse_#1_#3_help:nw}{##2} }
      {
        \exp_not:N \toks_put_right:Nn
        \exp_not:N \l_xparse_grabbed_args_toks {{##1}}
        ##2 \exp_not:N \l_xparse_grabbed_args_toks
      }
    }
```

And now an almost identical version which allows the \par token.

```
    \def_long:cpx {xparse_allow_par \l_tmpa_tlp _#1:w}##1##2
    \l_xparse_grabbed_args_toks{
      \exp_not:c{peek_#3 \l_tmpa_tlp :NTF} \exp_not:N #2
      { \exp_not:c{xparse_allow_par_#1_#3_help:nw}{##2} }
      {
        \exp_not:N \toks_put_right:Nn
        \exp_not:N \l_xparse_grabbed_args_toks {{##1}}
        ##2 \exp_not:N \l_xparse_grabbed_args_toks
      }
    }
  }
```

Here is the function for building the argument grabbing. We clear out the remaining `m` arguments first.

```
  \def:cpn {xparse_add_arg_type_#1:} {
    \xparse_add_remaining_m_args:
    \toks_put_right:Nx \l_xparse_grabbed_args_toks {
      \exp_not:c {xparse
```

Insert `_allow_par` in the csname if needed. This also applies to the argument grabber (although this should probably just be the default).

```
        \bool_if:NT \g_xparse_allow_par_bool {_allow_par}
        \g_xparse_ignore_marker_tlp
        _#1:w
      }
    }
    \let:Nc \xparse_read_arg_type_or_grab_default:n
    {xparse_grab_default_arg
      \bool_if:NT \g_xparse_allow_par_bool {_allow_par}
      :n
```

```
    }
  }
```

Now all we need to do is to define the helper functions.

```
  \xparse_define_helper:Nnnn #1{#3}{#4}{#5}
}
```

`\xparse_define_helper:Nnnn`

All we need now is the helper functions. We define a generic interface that should be fairly easy to use.

```
\def_new:NNn \xparse_define_helper:Nnnn 4{
  \toks_set:Nn \l_tmpa_toks
  {
    #3
    {
```

Remember to add an extra set of braces here.

```
      \toks_put_right:Nn \l_xparse_grabbed_args_toks {{#4}}
      ##1 \l_xparse_grabbed_args_toks
    }
  }
```

The next bit looks a little ugly but that's life.

```
  \toks_set:Nx \l_tmpa_toks {
    \exp_not:n {\def:cpn{xparse_#1_#2_help:nw}}
    \toks_use:N \l_tmpa_toks
    \exp_not:n {\def:cpn{xparse_allow_par_#1_#2_help:nw}}
    \toks_use:N\l_tmpa_toks
  }
  \toks_use:N \l_tmpa_toks
}
```

All the above may seem like an awful lot of trouble but it is general and it allows the following simple definitions:

```
\DeclareArgumentType o[{meaning}{}{\NoValue}{#1[#2]}{#2}
```

For the `c` type we add give an error message.

```
\DeclareArgumentType c({meaning}{
  \xparse_error:x{
    Missing~ coordinate~ argument.~ A~ value~ of~ (0,0)~ is~ assumed}
  }
  {{00}}
  {#1(#2,#3)}{{#2}{#3}}
\DeclareArgumentTypeDefaultValue O[{meaning}{#1[#2]}{#2}
\DeclareArgumentTypeDefaultValue C({meaning}{#1(#2,#3)}{{#2}{#3}}
```

One could also do

```
\DeclareArgumentTypeDefaultValue O[{charcode}{#1#2#3]}{#3}
```

where argument `#2` is the bracket but not requiring it to have catcode 12. However we can't use the same trick for reading the end of the argument.

Another interesting one could be

```
\DeclareArgumentType a\c_group_begin_token{catcode}{}{\NoValue}{#1#2}{#2}
```

which would basically be an optional argument in braces.

2.6.3 Argument shorthands

\DeclareArgumentTypeShorthand

Letting a letter or symbol be a shorthand for a more complicated structure is rather easy actually. All we need to do is to gobble the remainder of the main loop in the signature creation until `\xparse_parse_signature:n`, decrement the argument counter, put in the replacement argument specifiers and start the loop again. The disadvantage of this scheme is that we have to cheat and give it the wrong name as it should have a `w` argument specification but I hope you can forgive this little white lie.

```
\def_new:NNn \DeclareArgumentTypeShorthand 2{
  \def_new:cpn {xparse_add_arg_type_#1:}
    ##1 \xparse_read_arg_type_or_grab_default:n {
    \int_decr:N \l_xparse_total_args_int
    \xparse_read_arg_type_or_grab_default:n #2
  }
}
```

\xparse_add_arg_type_s:

Defining the `s` type is easy now since it really means `S{*}`.

```
\DeclareArgumentTypeShorthand s {S{*}}
```

2.7 Checking for optional values

\IfBooleanTF
\IfBooleanT
\IfBooleanF

The logical ⟨*true*⟩ and ⟨*false*⟩ statements are just our normal `\c_true` and `\c_false` so testing for them is done with our usual `\bool_if:NTF` functions from l3prg.

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

\NoValue
\c_xparse_hidden_no_value_tlp

`\NoValue` is just a text string and we define it as a token list pointer. However we will actually test for token list pointers with *meaning* `\NoValue`. Hence if an argument is really `-NoValue-` it will not be detected but that shouldn't happen.

```
\tlp_new:Nn \NoValue {-NoValue-}
\tlp_new:Nn \c_xparse_hidden_no_value_tlp {\NoValue}
```

\xparse_if_no_value:nTF

The test start by making a token list pointer out of the first argument and then check if it is equal to `\c_xparse_hidden_no_value_tlp` which contains `\NoValue`.

```
\def_long_new:Npn \xparse_if_no_value:nTF #1{
  \tlp_set:Nx \l_tmpa_tlp{\exp_not:n{#1}}
  \tlp_if_eq:NNTF \l_tmpa_tlp \c_xparse_hidden_no_value_tlp
```

If they are equal we just exit and execute the ⟨*true*⟩ code.

```
  { \use_arg_i:nn }
```

If not, take a closer look at #1. Peek ahead at the first token in #1 and then call the function \xparse_if_no_value_aux: but only if #1 is not empty or a blank space: in that case we can execute the ⟨*false*⟩ code immediately. We must use this test at some point otherwise the macros for checking tokens will break since the argument may be empty.

```
  { \tlist_if_blank:nTF {#1}
    { \use_arg_ii:nn }
    {\peek_after:NN \xparse_if_no_value_aux: #1 \q_nil {#1} }
  }
}
```

\xparse_if_no_value_aux: Now we simply check the argument specification of the token. If it is not a macro the function \token_get_arg_spec:N returns \scan_stop:.

```
\def_long_new:Npn \xparse_if_no_value_aux: {
  \tlp_set:Nx \l_tmpa_tlp{\token_get_arg_spec:N \l_peek_token }
```

Then check if \l_tmpa_tlp is empty, because this means we have a macro taking zero arguments and we can expand it once safely. Otherwise we just exit and execute the false code. Remember that we have the sequence {#1} waiting after the \q_nil in \xparse_if_no_value:nTF above.

```
  \tlp_if_empty:NTF \l_tmpa_tlp
  {\use_arg_i_delimit_by_q_nil:nw {\exp_args:No\xparse_if_no_value:nTF}}
  {\use_arg_i_delimit_by_q_nil:nw {\use_arg_iii:nnn}}
}
```

\IfNoValueTF \IfNoValueTF and its varieties are implemented as subcases of \xparse_if_no_value:nTF.
\IfNoValueT
\IfNoValueF

```
\let_new:NN \IfNoValueTF \xparse_if_no_value:nTF
\def_long_new:NNn \IfNoValueT 2 {\xparse_if_no_value:nTF{#1}{#2}{}}
\def_long_new:NNn \IfNoValueF 1 {\xparse_if_no_value:nTF {#1}{}}
```

\IfValueTF
\IfValueT
\IfValueF

For \IfValueTF we just reverse the arguments.

```
\def_long_new:NNn \IfValueTF 3{\xparse_if_no_value:nTF {#1}{#3}{#2}}
\let_new:NN \IfValueT \IfNoValueF
\let_new:NN \IfValueF \IfNoValueT
```

2.8 Pseudo arguments

\l_xparse_pseudo_post_arg_tlp A temporary token list pointer to store the post-arguments.

```
\tlp_new:Nn \l_xparse_pseudo_post_arg_tlp {}
```

\DeclarePseudoArgument Must be called at the end of the replacement text. Can use the parameters picked up so far. #1 is a name for it, #2 is the number of arguments, #3 is the pre-argument definition, #4 is the post-argument definition. The idea here is to define two functions: One to execute before the special pseudo argument is found and one to to it afterwards. Hence we define such two functions with #2 arguments and definitions #3 and #4 respectively.

```
\def_long_new:NNn \DeclarePseudoArgument 4{
  \def_long:cNn {xparse_pseudo_pre_arg_#1:\prg_replicate:nn{#2}{n}}#2{#3}
  \def_long:cNn {xparse_pseudo_post_arg_#1:\prg_replicate:nn{#2}{n}}#2{#4}
```

In case the user doesn't put the argument in braces, it must be a single token which we just grab and then execute the post-argument. Surely it won't be a `\par` token!

```
   \def:cNn {xparse_pseudo_nobrace_arg_#1:N} 1 {
    ##1 \l_xparse_pseudo_post_arg_tlp
  }
```

The regular version of the command is then this:

1. Execute the pre-code at level n.
2. Store the post-code in a token list pointer at level n.
3. Look for a begin-group token.

4a If found, remove it, start a group (level $n+1$) and make sure the token list pointer from 2) is executed after the group has ended (at level n).

4b If not found, call the `nobrace_arg` version which also executes the token list pointer from 2) at level n.

This scheme enables these commands to be nested since they all appear at different grouping levels.

Next we define the function. Also make sure that it gets all the arguments carried over to the post-argument.

```
  \def_long:cNx {xparse_pseudo_arg_#1:w} #2
  {
    \exp_not:c {
      xparse_pseudo_pre_arg_#1: \prg_replicate:nn{#2}{n}
    }
    \cs_use:c{def_aux_use_\int_use:N \int_eval:n{#2}_parameter:}
    \exp_not:n {\tlp_set:Nn \l_xparse_pseudo_post_arg_tlp}
    {
      \exp_not:c {xparse_pseudo_post_arg_#1:\prg_replicate:nn{#2}{n}}
      \cs_use:c{def_aux_use_\int_use:N \int_eval:n{#2}_parameter:}
    }
    \exp_not:n{ \peek_catcode_remove_ignore_spaces:NTF \c_group_begin_token }
    {
```

Here we insert the left brace token which was removed removed. In case you're wondering then yes, I know that `\c_group_begin_token` and `\group_execute_after:N` are unexpandable but I prefer this method because I don't want to remember which commands are expandable and which are not.

```
     \exp_not:n {
        \c_group_begin_token
        \group_execute_after:N \l_xparse_pseudo_post_arg_tlp
      }
```

```
    }
    {
      \exp_not:c{xparse_pseudo_nobrace_arg_#1:N}
    }
  }
}
```

`\UsePseudoArgument` Just a nice wrapper for calling such an argument.

```
\def_new:NNn \UsePseudoArgument 1{\cs_use:c{xparse_pseudo_arg_#1:w}}
```