Setting up \futurelet characters, securing catcodes, and parsing options

The pcatcode package functionality would work best if it were built into the LaTeX2ε kernel, but it cannot be usefully added to the kernel now without adversely affecting document compatibility across different systems. This package therefore modifies one or two of the low-level package-loading functions defined by the kernel. Theoretically speaking, the pcatcode package itself has to guard against the kind of catcode problems that it is intended to circumvent. If you would like a nice little TeXnecian's exercise, try your hand, before looking at the code of the pcatcode package, at the task that I set for myself: find the minimal set of catcode assumptions that one has to make before attempting to establish normalcy, where normalcy is defined as the state at the end of the LaTeX kernel, just before the last \makeatother. This is the state that may normally be expected at the beginning of a \documentclass file, if the LaTeX format file does not have any extensions (e. g., babel) compiled in. *Michael J. Downes, 1958–2003*

The catoptions Package[☆,★]

Version 0.2.6

Ahmed Musa ✉
Preston, Lancashire, UK

11th September 2011

Summary The catoptions package provides several extensions to the pcatcode package. Apart from the tools related to setting up, preserving and restoring category codes, it includes many (LA)TeX programming tools and even new list and options processing interfaces. It modifies the LaTeX kernel's options parsing mechanism to forestall premature expansion of options and values (in the manner of the xkvltxp and kvoptions-patch packages), so that the catoptions package may be loaded even before \documentclass. In fact, the package is meant to be loaded on top of other packages, so as to exploit its catcode preserving scheme. Among other reasons, this necessitated the development of the options parsing scheme of this package. Only the catcode and options parsing facilities are treated in this manual; the application programming interfaces will be covered in the documentation of the ltxtools package. The machinery of the catoptions package adds no cost to the simple syntax of LaTeX's native options parser. Users who are already familiar with LaTeX's legacy options processing don't necessarily have to invest the time that is essential in learning the extensive machinery of existing key-value and option parsers. Existing packages don't have to be modified to use the features of the catoptions package. The catoptions package, while maintaining simplicity, does not strip off even one level of outer braces in parsing package options and in list processing. It robustly normalizes key-values and options prior to parsing. The options parsing scheme of the catoptions package has been tried as a replacement parser on the article class and on many packages, including hyperref, cleveref and natbib packages without difficulties. Packages that redefine LaTeX's native options processing internals may not work properly with catoptions package. I know that xcolor package redefines \@declareoption and the catoptions package has taken that fact into account, but there may be other packages that modify LaTeX's options processing internals that I am not aware of.

☆ The package is available at http://www.ctan.org/tex-archive/macros/latex/contrib/catoptions/.

★ This manual doesn't, as yet, explain all the currently available features and commands of the catoptions package. This manual continues to evolve, but many of the available functions will appear in the user guide of the forthcoming ltxtools package.

Contents

1 Motivation

My motivation for turning to the pcatcode package was to save myself the trouble of declaring category codes at the beginning of my packages. After spending some time on the pcatcode package, I discovered I had learnt enough to make changes and additions to some of its macros and functionality. After completing the catcode stuff in the catoptions package, I wanted to pass options to the package. If the package is to be loaded on top of other packages, as intended, then its options parsing scheme should be independent of other packages. But realizing the trouble with passing expandable option values to packages and classes via the LaTeX kernel's scheme, I decided to implement modifications to the kernel's options parser. Apart from the problem of premature expansion of options and values by the LaTeX kernel, outer curly braces in option values are lost during parsing. Indeed, the kernel's option parsing scheme doesn't recognize option values indicated with the equality sign.

2 Loading the package

The catoptions package can be loaded in class and style files and in documents (before or after \documentclass) with the options described in section 3. If the package is loaded before \documentclass, it will, by default, use the options parser of the package in place of LaTeX's options parsing scheme (namely, the more robust commands \XDeclareOption, \XExecuteOptions and \XProcessOptions in place of LaTeX's native commands \DeclareOption, \ExecuteOptions and \ProcessOptions)*. In that case, the legacy commands \DeclareOption, \ExecuteOptions and \ProcessOptions are aliased to \XDeclareOption, \XExecuteOptions and \XProcessOptions, respectively. The latter set of commands do immediately recognize that they have to deal with option functions that are based on the kernel's syntax and semantics. This allows expandable options and option values (together with options with spaces in their names) to be passed via \documentclass—if catoptions is loaded before \documentclass. This also implies that existing packages can use the options parsing scheme of catoptions package without modifying the packages.

However, if after loading catoptions package before \documentclass the user still wants to use the legacy commands \DeclareOption, \ExecuteOptions and \ProcessOptions, the package option usepox (see section 3) can be toggled to `false`. In this case, option values can't be passed

*The user interfaces of \XDeclareOption, \XExecuteOptions and \XProcessOptions are similar to those of \DeclareOption, \ExecuteOptions and \ProcessOptions but they aren't the same. See section 7.

via \documentclass. More precisely, option values passed via \documentclass are stripped off internally by catoptions so that they may be processed by \ProcessOptions.

3 Package options

The catoptions package has the options shown in Table 1. The boolean option verbose instructs the package to enter information onto the log file when some events happen (e. g., when commands are redefined by means of, say, the command \robust@redef, which has the same syntax with TEX's legacy \def; and when an option from the same family is multiply submitted to, say, \XExecuteOptions). This option may be used in the future to provide debugging features for the package.

Table 1: Package options

Option	Default	Meaning
verbose	false	The global boolean switch that determines if information should be logged for some tasks in the package.
usepox	true	The boolean switch that determines if the options parser of catoptions should be used for all options processing of packages loaded after \documentclass even if those packages are based on LATEX2ε's native options processing scheme.

As noted in section 2, if the catoptions package is loaded before \documentclass it will invariably use the options processing mechanism of the package. In that case (of the package being loaded before \documentclass), the option usepox directs the package to use the catoptions's options processing scheme for all the packages loaded after \documentclass, instead of LATEX's native options parsing procedures. This allows catoptions's options parsing scheme to be used for existing packages loaded after \documentclass without modifying the packages.

Assuming that the options 'textstyle' and 'name' belong to an existing package (say, 'mypackage') to be loaded later, then the following example demonstrates one feature of the catoptions package, in the case in which the package is loaded before \documentclass:

Example: Loading catoptions before \documentclass

```
\RequirePackage[verbose, usepox]{catoptions}
\documentclass[textstyle=\ttfamily,name={Mr X\"avier Sanchez},a4paper,
  draft,10pt]{article}
\usepackage{mypackage}
```

If the catoptions package wasn't loaded before \documentclass then the above example would fail, because LATEX's option parser can't handle options with values (much less values with expandable tokens).

4 Saving and restituting category codes

There are the following user commands for saving and returning category codes of 'other' characters to their previous states:

New macros

```
\cptnormalcatcodes
\cptpushcatcodes
\cptpopcatcodes
\UseNormalCatcodes
\GetCurrentCatcodeSubset
\cptfutureletsetup
```

The command \cptnormalcatcodes simply resets the category codes of all 'other' characters together with those of the space character and \^^I and \^^J to their standard values. The \endlinechar is also reset to its normal value by this command. The command \cptpushcatcodes pushes the current category codes for restitution later with \cptpopcatcodes.

The command \UseNormalCatcodes works only in packages and does more than one thing: it calls \cptpushcatcodes (to save the prevailing category codes) and then \cptnormalcatcodes (to enforce the standard category codes). At the end of the class file or package, it automatically issues \cptpopcatcodes to recover all the category codes earlier pushed. The command \UseNormalCatcodes can conveniently be issued at the start of the package and the developer can be assured of access to the standard category codes of all 'other' characters together with those of the space character and \^^I and \^^J. It should be called only once in a package: subsequent calls within the same package will have no effect.

After issuing the command \GetCurrentCatcodeSubset, you can do

Example: \currentcatcodesubset

```
\show\currentcatcodesubset
```

to see the current catcode setup. The command \cptfutureletsetup is described in section 5.

Example: \UseNormalCatcodes

```
\ProvidesPackage{mypackage}[2011/01/16 v0.01]
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\RequirePackage[verbose, usepox]{catoptions}
\UseNormalCatcodes
% At the end of 'mypackage' all the category codes pushed by
% \UseNormalCatcodes will automatically be popped.
```

5 Future-letting of 'other' characters

The command \cptfutureletsetup defines canonical control sequences to represent the following characters:

space	␣	exclam	!	dblquote	"	hash	#	dollar	$	ampersand	&
lquote	‘	rquote	’	lparen	(	rparen	)	star	*	plus	+
comma	,	hyphen	-	period	.	slash	/	colon	:	semicolon	;
less	<	greater	>	equal	=	question	?	lbracket	[	rbracket	]
hat	^	underscore	_	bslash	\	lbrace	{	rbrace	}	vert	|
tilde	~										

However, for efficiency reasons, the canonical control sequences will be defined for only the characters (or their names) appearing in the user-supplied list \declarefutureletset, whose syntax is

New macro: \declarefutureletset

```
18 \declarefutureletset[⟨stub⟩]{⟨set⟩}
```

Here, ⟨set⟩ is a comma-separated list of names from the 'other' characters listed above. The defined commands are prefixed with an optional ⟨stub⟩, whose default value is fl@. The defined commands have the syntaxes

Futurelet characters

```
19 \⟨stub⟩⟨char⟩
20 \if⟨stub⟩⟨char⟩
```

For efficiency gains, a call to \cptfutureletsetup automatically undefines all the canonical control sequences previously defined with a call to \cptfutureletsetup—before the new canonical control sequences are defined.

For instance, with the choices

Example: \declarefutureletset

```
21 \declarefutureletset[fl@]{space, star, lbracket}
22 \cptfutureletsetup
```

we have the following commands on hand for testing after \futurelet:

Examples: Futurelet commands

```
23 \fl@space      \fl@star      \fl@lbracket
24 \iffl@space    \iffl@star    \iffl@lbracket
```

If any of the commands emanating from concatenating ⟨stub⟩ with the name of the character is already defined, an error message is flagged. It should be noted that these commands are available only after issuing the command \cptfutureletsetup, which isn't called automatically anywhere by the catoptions package. Normally, at \AtBeginDocument the catoptions package calls the command \cptrestorecatcodes, without calling \cptfutureletsetup. Calling \declarefutureletset and \cptfutureletsetup is the user's duty.

The commands \declarefutureletset and \cptfutureletsetup are decoupled to allow the user to call \declarefutureletset only once, and perhaps much earlier, before calling the command \cptfutureletsetup as often as desired.

An inefficient call of the form

Example: \declarefutureletset

```
25 \declarefutureletset[fl@]{all}
```

where 'all' means that control sequences should be defined for all the available 'other' characters, will define all the following control sequences whenever \cptfutureletsetup is called:

Examples: Futurelet commands

```
\fl@space       \fl@exclam       \fl@dblquote     \fl@hash
\fl@dollar      \fl@ampersand    \fl@lrquote      \fl@lparen
\fl@rparen      \fl@star         \fl@plus         \fl@comma
\fl@hyphen      \fl@period       \fl@slash        \fl@colon
\fl@semicolon   \fl@less         \fl@equal        \fl@greater
\fl@question    \fl@lbracket     \fl@rbracket     \fl@hat
\fl@underscore  \fl@lquote       \fl@lbrace       \fl@vert
\fl@rbrace      \fl@tilde        \fl@bslash

\iffl@space     \iffl@exclam     \iffl@dblquote   \iffl@hash
\iffl@dollar    \iffl@ampersand  \iffl@lrquote    \iffl@lparen
\iffl@rparen    \iffl@star       \iffl@plus       \iffl@comma
\iffl@hyphen    \iffl@period     \iffl@slash      \iffl@colon
\iffl@semicolon \iffl@less       \iffl@equal      \iffl@greater
\iffl@question  \iffl@lbracket   \iffl@rbracket   \iffl@hat
\iffl@underscore\iffl@lquote     \iffl@lbrace     \iffl@vert
\iffl@rbrace    \iffl@tilde      \iffl@bslash
```

This will yield control sequences that may never be needed. While such a facility does exist, using it will be inefficient despite the large capacities of many modern TEX engines.

The use syntaxes for the commands \⟨stub⟩@⟨char⟩ and \if⟨stub⟩@⟨char⟩ are as follows:

Example: Futurelet characters

```
\futurelet\next\cmd *
\def\cmd{\ifx\fl@star\next ‘I saw star’\else ‘I didn’t see star’\fi}
\def\cmd{\iffl@star\next{‘I saw star’}{‘I didn’t see star’}}
```

Notice that \ifx\fl@comma\next is a conventional TEX test, while \iffl@... expects two LATEX branches (\@firstoftwo and \@secondoftwo). The commands are meant to be easy to recall and use. If, for example, you are testing for the presence of \tilde, you simply do \iffl@tilde\next{...}{...}, assuming a ⟨stub⟩ of fl@.

After setting up \futurelet characters with \cptfutureletsetup, you can reset active characters to catcode 13 by \futureletresetactives.

6 Setting up package preliminaries

Every package normally requires some preliminary declarations, such as seen below. The commands \StyleFilePurpose, \StyleFileRCSInfo, \StyleFileInfo and \SetStyleFileMessages are defined by the catoptions package. All of them have intuitive syntaxes, except the command \SetStyleFileMessages, whose syntax is explained below.

Example: Package preliminaries

```
\StyleFilePurpose{A collection of useful commands}
\StyleFileRCSInfo
$Id: mypackage.sty,v 0.1 2011/01/01 09:00:00 My Name Exp $
\ProvidesPackage{mypackage}[\StyleFileInfo]
\NeedsTeXFormat{LaTeX2e}[1996/12/01]
\SetStyleFileMessages[mypack@]{err}{warn}{info}
```

New macro: \SetStyleFileMessages

```
51 \SetStyleFileMessages[⟨stub⟩]{⟨err⟩}{⟨warn⟩}{⟨info⟩}
```

Here, ⟨stub⟩ is an optional prefix for the three package messages described below. The default value of ⟨stub⟩ is the first three characters of the package or class name (\@currname) concatenated with the 'at' sign '@'. The mandatory arguments ⟨err⟩, ⟨warn⟩, and ⟨info⟩ are the suffices for the package error, warning and information messages, respectively.

For instance, with the declaration

Example: \SetStyleFileMessages

```
52 \SetStyleFileMessages[mypack@]{error}{warning}{info}
```

the following commands are automatically defined by the catoptions package:

Example: \SetStyleFileMessages

```
53 \mypack@error    →   Error message of two parameters
54 \mypack@warning  →   Warning message of one parameter
55 \mypack@info     →   Information message of one parameter
```

The commands \mypack@error and \mypack@info can be used as follows:

Example: \SetStyleFileMessages

```
56 \ifcptonetokenTF{#1}{%
57   \mypack@info{Correct single argument '\detokenize{#1}': accepted}%
58 }{%
59   \mypack@error{Multiple arguments '\detokenize{#1}'}%
60     {Invalid multiple arguments '\detokenize{#1}' rejected}%
61 }
```

7 Options parsing

The interfaces of the options processing commands are as follows, which, apart from the optional family ⟨fam⟩ and default value ⟨default⟩, are syntactically similar to the corresponding native LaTeX commands. The optional family name is useful for defining unique options that stand only a remote chance of being mixed up with options of other families. And the optional default value is handy when the user doesn't supply a value for an option: no errors are produced in this case. Existing packages can be processed with these commands without any modifications to those packages. In fact, we have run many existing packages on the bases of these commands (by letting LaTeX's \DeclareOption, \ExecuteOptions and \ProcessOptions to these commands), without encountering difficulties.

New macro: \XDeclareOption

```
62 \XDeclareOption<⟨fam⟩>{⟨opt⟩}[⟨default⟩]{⟨fn1⟩}
63 \XDeclareOption*<⟨fam⟩>{⟨fn2⟩}
```

The unstarred variant of the macro \XDeclareOption declares an option that can be used as a package or class option and executed by \XExecuteOptions or \XProcessOptions. This macro is similar to the standard LaTeX macro \DeclareOption, but with this command the user can pass a value to the option as well. That value can be accessed by using #1 or \cpt@currval in ⟨fn1⟩.

This will contain ⟨default⟩ when no value has been specified for the option. The default value of the optional argument ⟨default⟩ is empty. This implies that when the user does not assign a value to the option ⟨opt⟩ and when no default value has been defined, no error message will ensue. The optional argument ⟨fam⟩ can be used to specify a unique family for the option. When the argument is not used, the macro will insert the default family name (namely, \@currname.\@currext).

Example: \XDeclareOption

```
64 \ProvidesPackage{mypackage}[\StyleFileInfo]
65 \cptnewbool{mybool}
66 \XDeclareOption{mybool}[true]{\@nameuse{mybool#1}}
67 \XDeclareOption{leftmargin}[.5\hsize]{\setlength\leftmargin{#1}}
```

The user would have noticed the use \cptnewbool in the above example†. The options mybool and leftmargin could then be called via any of the following statements:

Example: \XDeclareOption

```
68 % Inside class or package files:
69 \RequirePackage[mybool=true,leftmargin=20pt]{mypackage}
70 % Inside document file:
71 \usepackage[mybool=false,leftmargin=20pt]{mypackage}
72 % Via document class:
73 \documentclass[mybool,leftmargin=20pt]{myclass}
```

The starred (⋆) variant of the macro \XDeclareOption can be used to process unknown options. It is similar to LaTeX's \DeclareOption⋆, but it has additional features. You can deploy \CurrentOption within the ⟨fn⟩ of this macro to access the option name and value for which the option is unknown. These values (possibly including an option) could, for example, be passed on to another class or package or could be used as an extra class or package option that, for instance, specifies a style that should be loaded. Moreover, you can use #1, #2 and #3 within ⟨fn⟩ to refer to the current family, current option (without its value), and current value, respectively.

Note 7.1 This note refers to macro lines 62 and 63. In \XDeclareOption, the current option value can be accessed by using #1 in ⟨fn1⟩. However, in \XDeclareOption⋆ the current family name is accessed as #1, the option name is available in #2, and the current option value can be accessed with #3 (all in ⟨fn2⟩). These differences are important.

New macro: \XUnknownOptionHandler

```
74 \XUnknownOptionHandler<⟨fams⟩>[⟨na⟩]{⟨handler⟩}
```

The command \XUnknownOptionHandler can be used in place of \XDeclareOption⋆ to specify default action(s) for options undefined in any family in ⟨fams⟩‡. ⟨handler⟩ refers to the default function that should be executed if undeclared options are called from families ⟨fams⟩ by the user. \CurrentOption can be deployed within ⟨handler⟩ to access the option name (coupled with its current value) for which the option name is unknown. As mentioned above in the case of \XDeclareOption⋆, you can use #1, #2 and #3 within ⟨handler⟩ to access the current family, current option (uncoupled from its current value), and current value, respectively. ⟨na⟩ is the list

†The commands \XDeclareBooleanOption, \XDeclareBooleanOptions and \XDeclareBiBooleanOptions allow the user to define new boolean options without the need to first initialize the booleans with \newif or \cptnewbool as seen here.

‡Declaring a default handler for each unknown option is, of course, inefficient. We have, therefore, provided provisions for declaring such handers for only families of options.

of options for which ⟨handler⟩ shouldn't apply; for these options, the kernel's unknown option handler (i. e., \default@ds) would apply.

If the ⟨handler⟩ contains the special literal form define/.code= as in

Example: Unknown option handler

```
75 ⟨fn1⟩ define/.code=⟨callback⟩/.end ⟨fn2⟩
```

then each unknown option in families ⟨fams⟩ will be momentarily defined and executed with the option function ⟨callback⟩. The family of the new option will be \cpt@currfam (the prevailing family name); its default value will be \cpt@currval (the value specified by the user for the unknown option); and its macro function will be the above-indicated ⟨callback⟩. Again, you can use #1, #2 and #3 in ⟨handler⟩ to refer to the current option family, option name, and option value, respectively.

Note 7.2 Functions ⟨fn1⟩ and ⟨fn2⟩ on macro line 75 will be executed for each unknown function in families ⟨fams⟩, but (unlike ⟨callback⟩) will not be used in defining unknown options of ⟨fams⟩.

If the value given for the unknown option is true or false, the new (i. e., undeclared) option will be defined as a boolean option with macro prefix '\@currname @'; otherwise it will be defined as a command option with macro prefix 'cmd\@currname @'. (See the descriptions of the commands \XDeclareCommandOption and \XDeclareBooleanOption for the meaning of 'macro prefix'.) It should be noted that the token 'define' in the above syntax is not escaped, and that the token /.end is mandatory. If ⟨callback⟩ contains these literal forms, they have to be enclosed in curly braces.

The following example, among other things, produces a warning when the user supplies an option that was not previously declared:

Example: \XDeclareOption*

```
76 \XDeclareOption*<mypackage>{%
77   % Unknown options will not be defined automatically here:
78   \PackageWarningNoLine{mypackage}{Unknown option `\CurrentOption' ignored}%
79   % Current option family:
80   \def\currfamily{#1}%
81   % Current option without the given value:
82   \def\curroption{#2}%
83   % User-supplied value of current option:
84   \def\currvalue{#3}%
85   % User defined function:
86   \def\mymacro##1{\do@a{#2=#3}\do@b{##1}%
87   % `\CurrentOption' isn't equivalent to `#2=#3' because non-active
88   % spaces around `#3' would have been removed.
89 }
```

And the following example passes undeclared options to article class:

Example: \XDeclareOption*

```
90 \XDeclareOption*{\PassOptionsToClass{#2=#3}{article}}
```

The following example instructs \XProcessOptions to define all unknown keys on the fly with the callback shown.

Example: \XUnknownOptionHandler

```
\XUnknownOptionHandler<mypackage>{%
  % Unknown options will be defined automatically here with the
  % indicated code:
  define/.code=
  \def\x##1{##1\relax ‘option #2 = value #3’ in family ‘#1’}%
  /.end
  % The following will be executed for each unknown option of the
  % given family, but will not be a ‘callback’ for unknown options:
  \PackageWarningNoLine{mypackage}{Unknown option ‘#2’ of family
    ‘#1’ defined on the fly}%
}
```

If no default family handler has been provided by the user (through the macro \XDeclareOption* or \XUnknownOptionHandler) for the family of an undeclared option, and the option doesn't appear in the list \XExternalOptions, then that option will be logged (i. e., entered in the transcript file) as undefined at the end of the document. By default, \XExternalOptions contains all article, report and memoir class options. The user can update the \XExternalOptions list by means of the following commands:

New macros: \AddToExternalOptions, \RemoveFromExternalOptions

```
\AddToExternalOptions{⟨opts⟩}
\RemoveFromExternalOptions{⟨opts⟩}
```

where ⟨opts⟩ are the comma-separated option names to be added to, or removed from, the list.

Note 7.3 Unknown options that appear in \XExternalOptions list will not have their ⟨fn1⟩, ⟨fn2⟩ and ⟨callback⟩ of the default family handler (see macro line 75) executed unless they contain one of the commands \PassOptionsToClass and \PassOptionsToPackage.

New macro: \XDeclareOptions

```
\XDeclareOptions<⟨fam⟩>{⟨opts⟩}[⟨default⟩]{⟨fn⟩}
```

The command \XDeclareOptions is similar to \XDeclareOption but, instead of declaring just one option, it declares all the options in the comma-separated list ⟨opts⟩. Each option in the list ⟨opts⟩ is defined with the same family ⟨fam⟩, default value ⟨default⟩, and function ⟨fn⟩. The command \XDeclareOptions can be used to save tokens when it is required to declare a set of options with identical attributes.

New macros: \XDeclareInvalidOption, \XDeclareInvalidOptions

```
\XDeclareInvalidOption<⟨fams⟩>{⟨opt⟩}
\XDeclareInvalidOptions<⟨fams⟩>{⟨opts⟩}
```

The commands \XDeclareInvalidOption and \XDeclareInvalidOptions declare the given list ⟨opts⟩ as invalid or inadmissible within the specified families ⟨fams⟩. A user submitting the invalid option will receive the message the option author has specified for the given family via the command \XInvalidOptionHandler. Package and class authors don't necessarily have to provide invalid-option handler for each or any family (of their options) via the macro \XInvalidOptionHandler: the package provides a default error message for invoked invalid options when the author of the option hasn't defined an invalid-option handler.

New macro: \XInvalidOptionHandler

```
107 \XInvalidOptionHandler<⟨fams⟩>{⟨handler⟩}
```

The command \XInvalidOptionHandler defines, for each family ⟨fam⟩ in ⟨fams⟩, a one-parameter function, which, when specified, is used to replace the macros of options in the family ⟨fam⟩ when the options appear in ⟨opts⟩ list of \XDeclareInvalidOptions.

Example: \XInvalidOptionHandler

```
108 \XInvalidOptionHandler<fama,famb>{%
109   \@latex@error{Invalid or null option `\CurrentOption'}%
110     {I have encountered an invalid option `\CurrentOption':
111     Your package author has set this option as inadmissible.}%
112 }
113 \XDeclareInvalidOptions<fama,famb>{opta,optb}
```

New macro: \XDeclareCommandOption

```
114 \XDeclareCommandOption<⟨fam⟩>{⟨opt⟩}[⟨default⟩](⟨pref⟩){⟨fn⟩}
```

The command \XDeclareCommandOption will, apart from declaring the option ⟨opt⟩, also create a macro \⟨pref⟩@⟨opt⟩ to hold the user-supplied value of the option. The macro so created can be used in ⟨fn⟩ or in any other place. The default value of the optional ⟨pref⟩ is '\@currname @' prefixed with the letters cmd.

New macro: \XDeclareCommandOptions

```
115 \XDeclareCommandOptions<⟨fam⟩>{⟨opts⟩}[⟨default⟩](⟨pref⟩){⟨fn⟩}
```

The command \XDeclareCommandOptions is similar to \XDeclareCommandOption but, instead of declaring just one option, it declares all the options in the comma-separated list ⟨opts⟩. Each option in the list ⟨opts⟩ is defined with the same family ⟨fam⟩, default value ⟨default⟩, and function ⟨fn⟩.

New macro: \XDeclareBooleanOption

```
116 \XDeclareBooleanOption<⟨fam⟩>{⟨opt⟩}[⟨default⟩](⟨pref⟩){⟨fn⟩}
```

The command \XDeclareBooleanOption will, apart from declaring the option ⟨opt⟩, also create a boolean \if⟨pref⟩@⟨opt⟩. It will automatically toggle this boolean (to true or false) when the option is set and the input is valid, depending on the user-supplied value of the option. The macros so created can be used in ⟨fn⟩ or in any other place. The default value of the optional ⟨pref⟩ is '\@currname @' with no additional prefix. Only true or false may be submitted as the value of a boolean option.

New macro: \XDeclareBooleanOptions

```
117 \XDeclareBooleanOptions<⟨fam⟩>{⟨opts⟩}[⟨default⟩](⟨pref⟩){⟨fn⟩}
```

The command \XDeclareBooleanOptions is similar to \XDeclareBooleanOption but, instead of declaring just one option, it declares all the options in the comma-separated list ⟨opts⟩. Again, each option in the list ⟨opts⟩ is defined with the same family ⟨fam⟩, default value ⟨default⟩, and function ⟨fn⟩.

New macro: \XDeclareBiBooleanOptions

```
118 \XDeclareBiBooleanOptions<⟨fam⟩>{⟨opt1⟩,⟨opt2⟩}[⟨default⟩](⟨pref⟩){⟨fn1⟩}{⟨fn2⟩}
```

The command \XDeclareBiBooleanOptions declares the two options ⟨opt1⟩ and ⟨opt2⟩ in the comma-separated list of options. Again, each of the two declared options is defined with the same family ⟨fam⟩ and default value ⟨default⟩, but separate functions ⟨fn1⟩ and ⟨fn2⟩. A distinguishing characteristic of bi-boolean options is that when one option is true, the partner option is automatically turned false, and vice versa.

New macro: \XDeclareSwitchOption

```
119 \XDeclareSwitchOption<⟨fam⟩>{⟨opt⟩}[⟨default⟩](⟨pref⟩){⟨fn⟩}
```

In the parlance of the catoptions and ltxtools packages, a switch can assume only one of the two possible states: \cpttrue (which is defined as 00) or \cptfalse (which is defined as 01). A switch \swa can be tested with TEX's \if conditional. For example, we can do \if\swa...\fi. The catoptions package also provides the commands \ifswitchTF and \ifswitchFT for testing switches, which can be used as follows:

New macros: \ifswitchTF, \ifswitchFT

```
120 \ifswitchTF{⟨switch⟩}{⟨true text⟩}{⟨false text⟩}
121 \ifswitchFT{⟨switch⟩}{⟨false text⟩}{⟨true text⟩}
```

Note that here ⟨switch⟩ has no escape character, unless it evaluates to a switch name. So, after defining \def\swa{00} we could then do \ifswitchTF{swa}{true}{false}. It is somewhat risky to introduce new switches with \def or \let[§]. It is advisable to always use \newswitch instead. It has the syntax

New macro: \newswitch

```
122 \newswitch{⟨switch⟩}[⟨state⟩]
```

Again, note that here ⟨switch⟩ has no escape character, unless it evaluates to a switch name. ⟨state⟩ can be either true or false.

Switches are cheaper than native booleans since each switch swa has one and only one command \swa. Each native boolean boola, on the other hand, has up to three commands (namely, \ifboola, \boolatrue and \boolafalse. If you need to declare a large number of native booleans, it is advisable to consider using switches instead.

It is possible to toggle the state of a switch by simply using the commands:

New macros: \setswitchtrue, \setswitchfalse

```
123 \setswitchtrue{⟨switch⟩}
124   →   if switch ⟨switch⟩ exists, do \let\switch\cpttrue
125 \setswitchfalse{⟨switch⟩}
126   →   if switch ⟨switch⟩ exists, do \let\switch\cptfalse
```

The command \XDeclareSwitchOption will, apart from declaring the option ⟨opt⟩, also create a switch \⟨pref⟩@⟨opt⟩. It will automatically toggle this switch to true (equivalent to 00) or false

[§]Because of the need to do, e. g., \let\swa\cpttrue and \if\swa for any given switch \swa, switches, unlike toggles, don't have their own separate namespace and it is all too easy to redefine an existing switch.

(equivalent to 01) when the option is set and the input is valid, depending on the user-supplied value of the option. The macros so created can be used in ⟨fn⟩ or in any other place. The default value of the optional ⟨pref⟩ is again '\@currname @' with no additional prefix. Only true or false may be submitted as the value of a switch option.

New macro: \XDeclareSwitchOptions

```
127 \XDeclareSwitchOptions<⟨fam⟩>{⟨opts⟩}[⟨default⟩](⟨pref⟩){⟨fn⟩}
```

The command \XDeclareSwitchOptions is similar to \XDeclareSwitchOption but, instead of declaring just one option, it declares all the options in the comma-separated list ⟨opts⟩. Again, each option in the list ⟨opts⟩ is defined with the same family ⟨fam⟩, default value ⟨default⟩, and function ⟨fn⟩.

New macro: \XExecuteOptions

```
128 \XExecuteOptions<⟨fams⟩>{⟨opts⟩}[⟨na⟩]
```

The re-entrant \XExecuteOptions macro sets options created by \XDeclareOption and is basically a means of setting up the default values of the options. The optional argument ⟨fams⟩ can be used to specify a list of *families* that define the options. When the argument is not used, the macro will insert the default family name (\@currname.\@currext). The set ⟨na⟩ is the list of keys to be ignored (i. e., not executed if they appear in ⟨opts⟩).

This macro will not use the declaration done by \XDeclareOption* when undeclared options appear in its argument. Instead, in this case the macro will issue a warning and ignore the option. This differs from the behavior of LaTeX's \ExecuteOptions.

Example: \XExecuteOptions

```
129 \XExecuteOptions{leftmargin=0pt}
```

This initializes the option \leftmargin of macro line 67 to 0pt.

New macro: \XProcessOptions

```
130 \XProcessOptions<⟨fams⟩>[⟨na⟩]
131 \XProcessOptions*<⟨fams⟩>[⟨na⟩]
```

The re-entrant \XProcessOptions macro processes the options and values passed by the user to the class or package. The optional argument ⟨fams⟩ can be used to specify the *families* that have been used to define the options. The optional argument ⟨na⟩ can be used to specify options that should be ignored, i. e., not processed. When used in a class file, this macro will ignore unknown options. This allows the user to use global options in the \documentclass command which could be claimed by packages loaded later.

The starred (*) variant of \XProcessOptions works like the unstarred variant except that the former also copies user input from the \documentclass command and processes the options in the order specified by the \documentclass. When the user specifies an option in the \documentclass which also exists in the local family (or families) of the package calling \XProcessOptions*, the local option will be set as well. In this case, #1 in \XDeclareOption macro will contain the user-value entered in the \documentclass (or \usepackage or \RequirePackage) command for this option. First the global options from \documentclass will set local options and afterwards the local options (specified via \usepackage, \RequirePackage and \LoadClass or similar commands) will set local options, which could overwrite the global options set earlier, depending on how the options sections are organized. The macro \XProcessOptions* reduces to \XProcessOptions only

when issued from the class which forms the document class for the file at hand (to avoid setting the same options twice), but not for classes loaded later using, for instance, \LoadClass. Global options that do not exist in the local families of the package or class calling \XProcessOptions* will be simply ignored or highlighted.

The implementation here differs significantly from the LaTeX kernel's scheme of carrying out \ProcessOptions and \ProcessOptions*. It also deviates from the implementations by other options processing packages. The differences lie mainly in how the local and global options are distinguished and in the order of processing those options. Among other issues, the family structure introduced by the catoptions package (though lightweight) makes the independence between local and global options possible, even if the options from the two categories share the same namespace and are mixed in, say, \documentclass command. Also, document classes loaded by \LoadClass don't have the same primacy as the first document class. When using LaTeX kernel's \ProcessOptions or \ProcessOptions*, a class file can't copy document class options, even if the class file is loaded by \LoadClass. This is not the case with the catoptions package.

Examples: \XDeclareOption, \XExecuteOptions, \XProcessOptions

```
% This is a sample class file. We specify a family for the options,
% instead of using the default family (testclass.sty).
\ProvidesClass{testclass}[2011/01/15 v1.0 A test class]
\NeedsTeXFormat{LaTeX2e}
% The following loading of 'catoptions' may need to be commented out
% to avoid option clash with another loading of the package in the document.
% \RequirePackage{catoptions}
\UseNormalCatcodes
\newif\ifboola
\XDeclareOption<testclass>{boola}[true]{%
  \@nameuse{boola#1}%
  \ifboola\let\eat\@gobble\fi
}
\XDeclareOptions<testclass>{opta,optb}[true]{%
  \def\alloptsfunc##1{\def\tempa{##1}}%
}
% No need for \newif when declaring boolean options:
\XDeclareBooleanOption<testclass>{boolb}[true](test@){%
  \iftest@boolb
    \AtEndOfPackage{\gdef\tex{\TeX\xspace}}%
  \fi
}
\XDeclareBiBooleanOptions{draft,final}[true]test@{}{}
\XDeclareCommandOption<testclass>{color}[blue](test@){%
  Load 'color' package
}
% Disable option 'color' at \AtBeginDocument:
\XDisableOption*<testclass>{color}
\XDeclareOption<testclass>{align}[left]{%
  \ifstrcmpTF{#1}{left}{\let\align\raggedright}{%
    \ifstrcmpTF{#1}{right}{\let\align\raggedleft}{%
      \ifstrcmpTF{#1}{center}{\let\align\centering}{%
        \@latex@error{Invalid value '#1' for align}{%
          You have issued an illegal value '#1' for the variable 'align'.
        }%
      }%
```

```
    }%
  }%
}
\XDeclareInvalidOption<testclass>{deadoption}
\XDeclareOption*<testclass>{\PassOptionsToClass\CurrentOption{article}}
\XExecuteOptions<testclass>{boola,boolb}
\XProcessOptions*<testclass>\relax
\LoadClass{article}
\cptloadpackages{%
  ⟨pkg name⟩|⟨pkg opts⟩|⟨pkg date⟩;...;⟨more packages⟩
}
\endinput

% This is a sample document:
\RequirePackage[usepox]{catoptions}
\documentclass[
  align     = right,
  boola     = false,
  boolb     = true,
  name      = {Mr J\"avier Claudioos},
  a4paper,
  draft,
  10pt
]{testclass}
% You can call \usepackage{catoptions}, instead of
% \RequirePackage{catoptions}, after \documentclass, but then the
% \documentclass option 'name={Mr J\"avier Claudioos}' can't be processed.
\usepackage{cleveref}

\begin{document}
 Blackberry bush ... blackberry-lily.
\end{document}
```

New macro: \ifoptdefTF

```
\ifoptdefTF<⟨fams⟩>{⟨opt⟩}{⟨true⟩}{⟨false⟩}
\ifoptdefFT<⟨fams⟩>{⟨opt⟩}{⟨false⟩}{⟨true⟩}
```

The macro \ifoptdefTF returns ⟨true⟩ if option ⟨opt⟩ is undefined in one or more members of ⟨fams⟩, and ⟨false⟩ otherwise. The command \ifoptdefFT reverses the logic of \ifoptdefTF.

New macro: \XDisableOptions

```
\XDisableOptions<⟨fams⟩>{⟨opts⟩}
\XDisableOptions*<⟨fams⟩>{⟨opts⟩}
```

The command \XDisableOptions disables all the options in the list ⟨opts⟩ that can be found in the families ⟨fams⟩, i. e., it makes the options invalid thereafter. If any of the options in the list ⟨opts⟩ can't be found in the families ⟨fams⟩, it is simply ignored without warning: since families can be mixed in ⟨fams⟩, it wouldn't be meaningful issuing several warnings in this case.

The starred (*) variant of \XDisableOptions delays the invalidation of the ⟨opts⟩ until the invocation of \begin{document}. This may be used to bar users of the options from committing the options after the start of document, but not before.

New macro: \ifoptdisabled

```
\ifoptdisabledTF<⟨fams⟩>{⟨opt⟩}{⟨true⟩}{⟨false⟩}
\ifoptdisabledFT<⟨fams⟩>{⟨opt⟩}{⟨false⟩}{⟨true⟩}
```

The macro \ifoptdisabledTF returns ⟨true⟩ if option ⟨opt⟩ has been disabled from one or more members of ⟨fams⟩, and ⟨false⟩ otherwise. The command \ifoptdisabledFT reverses the logic of \ifoptdisabledTF.

New macro: \XLogDisabledOptions

```
\XLogDisabledOptions<⟨fams⟩>
```

The command \XLogDisabledOptions, which can be called before and after \begin{document}, writes in the current transcript file the options of the families ⟨fams⟩ that have been disabled so far. If no options have been disabled from any of the given families or if the given family does not exist, a message is logged to that effect.

8 Normalizing csv and kv lists

Any arbitrary parser-separated-values (csv) list can be normalized by means of the package command \csv@@normalize before processing the list.

New macro: \csv@@normalize

```
\csv@@normalize[⟨parser⟩]{⟨list⟩}
\csv@@normalize*[⟨parser⟩]⟨listcmd⟩
```

Here, ⟨list⟩, which is populated by parser-separated elements, is not expanded before normalization; ⟨listcmd⟩, on the other hand, is expanded once before normalization. The default value of the optional ⟨parser⟩ is comma ‘,’. ‘Normalization’ implies changing the category codes of all the active parsers to their standard values, as well as trimming leading and trailing spaces around the elements of the list and removing consecutive multiple parsers. Thus empty entries that are not enforced by curly braces are removed. The result (i. e., normalized list) is available in the macro \normalized@list (in the unstarred variant case) or ⟨listcmd⟩ (in the starred (⋆) variant case).

Example: \csv@@normalize

```
\begingroup
\catcode‘\;=\active
\gdef\x{x ;  ;  {y}; ;  z}
\endgroup

\csv@@normalize*[;]\x
% \show\x
```

New macro: \kv@@normalize

```
\kv@@normalize[⟨parser⟩]{⟨list⟩}
\kv@@normalize*[⟨parser⟩]⟨listcmd⟩
```

The command \kv@@normalize normalizes a list of key-value pairs, returning the result in the macro \normalized@list (in the unstarred variant case) or ⟨listcmd⟩ (in the starred (⋆) variant case). Besides dealing with multiple commas and the spaces between entries, in this case the spaces

between keys and the equality sign are removed and multiple equality signs are made only one. Moreover, the category codes of the arbitrary parser and the equality sign is made normal/other throughout the list. The command \kv@@normalize is meant for options or key-value parsing; it is used in the options processing scheme of catoptions package.

Example: \csv@@normalize

```
\begingroup
\catcode`\;\string=\active
\catcode`\=\string=\active
\gdef\x{x=A ; ;  y==={B} ; ;  z=C}
\endgroup

\kv@@normalize*[;]\x
% \show\x
```

9 Parsing csv and kv lists

New macros: \csv@@loop, \csv@@parse, \kv@@parse

```
\csv@@loop[⟨parser⟩][⟨list⟩]
\csv@@loop*[⟨parser⟩][⟨listcmd⟩]
\csv@@parse[⟨parser⟩][⟨list⟩]
\csv@@parse*[⟨parser⟩][⟨listcmd⟩]
\kv@@loop[⟨parser⟩][⟨list⟩]
\kv@@loop*[⟨parser⟩][⟨listcmd⟩]
\kv@@parse[⟨parser⟩][⟨list⟩]
\kv@@parse*[⟨parser⟩][⟨listcmd⟩]
```

The macros \csv@@parse and \kv@@parse—and their starred (⋆) variants—call \csv@@normalize and \kv@@normalize, respectively. On the other hand, the macros \csv@@loop and \kv@@loop and their starred (⋆) variants don't call \csv@@normalize (since not every list will require normalization). This is the only difference between \csv@@loop and \csv@@parse, and between \kv@@loop and \kv@@parse. The macros \csv@@loop and \csv@@parse are meant for general csv-list processing with an arbitrary parser, while the command \kv@@loop and \kv@@parse are designed for processing key-value lists. These macros loop over a given ⟨parser⟩-separated ⟨list⟩ and execute the user-defined, one-parameter, commands \csv@do and \kv@do, respectively, for every item in the list, passing the item as an argument and preserving outer braces. The default value of ⟨parser⟩ is comma ',' . The starred (⋆) variants of these commands expand ⟨listcmd⟩ once before commencing the loop.

Here are some points to note about these list processors:

a) The commands \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse aren't expandable.
b) If an item contains ⟨parser⟩, it must be wrapped in curly braces when using \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse, otherwise the elements may be mixed up during parsing. The braces will persist thereafter, but will of course be removed during printing (if the items are printed).
c) White spaces before and after the list separator are always ignored by the normalizer called by \csv@@parse and \kv@@parse. If an item contains ⟨parser⟩ or starts with a space, it must, therefore, be wrapped in curly braces before commencing these loops.
d) Since \csv@@loop and \kv@@loop don't call the normalizer, they preserve outer/surrounding spaces in the entries. Empty entries in ⟨list⟩ or ⟨listcmd⟩ will be processed by \csv@@loop

and \kv@@loop if the boolean cpt@useempty is true. You may thus issue the command \UseEmptyEntry or \DiscardEmptyEntry, which are based on the boolean cpt@useempty, before commencing the iteration. If empty entries are important to the task at hand, then issuing \UseEmptyEntry or \DiscardEmptyEntry prior to the commencement of the loop is recommended, because a previous call to either \csv@@loop or \kv@@loop (perhaps by another package) could have set cpt@useempty to a state that is no longer valid or desired.

e) The commands \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse will execute at least once for empty ⟨list⟩ or ⟨listcmd⟩.

f) The commands \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse can be nested to any level and can be mixed.

g) In the commands \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse, it is always possible to break out of the loop prematurely at any level of nesting, simply by issuing the command \loopbreak (see the example below). Breaking an inner loop doesn't affect the continuation of the outer loop, and vice versa; that is, loop breaks are nest-level-dependent.

h) The user can insert \csvbreak as an element in ⟨list⟩ for any of the commands \csv@@loop, \csv@@parse, \kv@@loop, \kv@@parse and \dofunclist with the hope of automatically breaking out of the list processing prematurely (i. e., before the list is exhausted). The tokens \listbreak and \breaklist are not defined or used by the catoptions package, to avoid name clashes with other packages. The catoptions package instead uses \csvbreak, which is an unexpandable token; if it were to be expandable, then experience has shown that a chaotic infinite loop could arise in an expansion context. Breaking out of the loop prematurely on the current nest level doesn't affect the continuation of the loop on the other levels.

Example: \csv@@parse

```
\begingroup
\catcode`\;=\active
\gdef\x{a ;  ;  {b}; ;  c}
\endgroup

\@tempcnta\z@
\def\csv@do#1{%
  \advance\@tempcnta\@ne
  \@namedef{x@\romannumeral\@tempcnta}{#1}%
}
\csv@@parse*[;]\x
% \show\x@ii

\def\xa{a , {b} , c}
\def\xb{x ; y ; {z}}
\def\csv@do#1{%
  \pushnumber\nra
  \csn@edef{arg@\romannumeral\nra}{#1}%
  \chardef\nrb\z@
  \def\csv@do##1{%
    \pushnumber\nrb
    % Breaking the inner loop here doesn't affect the outer loop:
    \ifnum\nrb>\@ne\loopbreak\fi
    \csn@def{arg@\romannumeral\nra @\romannumeral\nrb}{#1,##1}%
  }%
  \csv@@parse*[;]\xb
}
```

```
\csv@@parse*\xa
```

Examples: Using \csvbreak

```
\newvariables{count}{m@,n@}
\def\csv@do#1{%
  \advance\m@\@ne\n@\z@
  \def\csv@do##1{%
    \advance\n@\@ne
    \csn@def{w@\romannumeral\m@ @\romannumeral\n@}{#1,##1}%
  }%
  \csv@@parse[;]{ {x} ; {y} ; \csvbreak ; z  }%
}
\csv@@parse{ a , {b}, {c} }
```

The following is a pseudocode that depicts the use of \kv@@parse:

Example: \kv@@parse

```
\def\kv@do#1{%
  \def\CurrentOption{#1}%
  if \CurrentOption is not empty then
    Split \CurrentOption into option and value;
    Search if option exists in \@declaredoptions
    if option is found then
      Execute the option's function
    else
      Report option as unknown
    fi
  fi
}
% \kv@@parse will normalize \@classoptions before parsing it:
if there are declared options then
  \kv@@parse*\@classoptions
fi
```

New macro: \dofunclist

```
\def\do⟨params⟩{⟨fn⟩}
\dofunclist[⟨parser⟩]{⟨list⟩}
\dofunclist*[⟨parser⟩]{⟨listcmd⟩}
```

The \dofunclist command can be used to iterate over a ⟨parser⟩-separated ⟨list⟩ or ⟨listcmd⟩ and execute the auxiliary command \do for every item in the list, passing the item as an argument and preserving outer braces. While the user-defined commands \csv@do and \kv@do (required by \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse) must be one-parametered, the command \do (required by \dofunclist) can be multi-parametered. The command \dofunclist isn't expandable. ⟨params⟩ are the parameter texts for the command \do. White spaces before and after the list separator are always ignored, because the list is first normalized before parsing by \dofunclist. If an item contains ⟨parser⟩ or starts with a space, it must be wrapped in curly braces (to preserve the parser or space). The braces may persist thereafter, but will of course be removed during printing (if the items are printed). The default value of ⟨parser⟩ is comma ',' .

Empty entries in ⟨list⟩ will be ignored by the normalizer called by \dofunclist. Such empty entries, if needed later, would need to be enclosed in curly braces before commencing the iteration. The \dofunclist command, like all the looping macros of this section, can be nested to any level.

Examples: \dofunclist

```
\def\do#1{\item #1}
\begin{itemize}
  % The following parser (,) is superfluous since the default is comma:
  \dofunclist[,]{aaaa, bbbb, {cccc, dddd}, eeee}
\end{itemize}

% Let us load many packages compactly here. This is just an example of
% \dofunclist; you can instead use the more appropriate command
% \cptloadpackages to load packages see macro line 176:
\def\do#1.#2[#3]{%
  \@ifpackageloaded{#1}{%
    \@ifpackagelater{#1}{#2}{}{%
      \cpt@warn{Older version of package `#1' loaded earlier}%
    }%
  }{%
    \cptexpandsecond\usepackage{[\cpttrimspaces{#3}]}{#1}[#2]%
  }%
}
% The default parser (,) is implied below:
\dofunclist{%
  yfonts.2003/01/08[],
  pifont.2005/04/12[],
  helvet.2005/04/12[scaled=0.9],
  zref.2008/10/01[{user,lastpage}],
  xcolor.2007/01/21[{table,hyperref}]
}

% An example of nested loops follows. Here outer braces in the
% elements are preserved:
\chardef\m@\z@
\def\do#1{%
  \pushnumber\m@
  \chardef\n@\z@
  \def\do##1{%
    \pushnumber\n@
    \ifnum\n@>\@ne\loopbreak\fi
    \csn@def{w@\romannumeral\m@ @\romannumeral\n@}{#1,##1}%
  }%
  \dofunclist[;]{x;{y};z}%
}
\dofunclist[,]{a,{b},c,d,e}
```

While \dofunclist allows the user to define and call multi-parametered \do functions, outer curly braces in the *individual* arguments of \do may, in some rare cases, be lost in parsing (but only if the number of arguments exceeds one). This possibility depends on how the parameters and arguments of the user-supplied command \do are arranged. If preserving braces is essential to the user's need, then he/she might consider using the more robust commands \csv@@loop, \csv@@parse,

\kv@@loop and \kv@@parse. It is possible to robustly build a multi-parametered callback in the user-defined commands \csv@do and \kv@do required by \csv@@loop, \csv@@parse, \kv@@loop and \kv@@parse. In fact, this is what \dofunclist does internally. Admittedly, only experienced TEXnecians may be able to do that.

9.1 Looking ahead in csv lists

Imagine an instance in which, while processing a csv list, you need not only the current item of the list but also the next item. Moreover, you need to know when the last item of the list has been reached, so that, for example, you can do something peculiar with the last item. The \indrisloop list processor provides facilities to accomplish these tasks.

New macros: \indrisloop

```
\indrisloop[⟨parser⟩]{⟨list⟩}⟨fn⟩
\indrisloop*[⟨parser⟩]⟨listcmd⟩⟨fn⟩
```

The \indrisloop command can be used to iterate over a ⟨parser⟩-separated ⟨list⟩ or ⟨listcmd⟩ and execute the user-supplied one-parameter command ⟨fn⟩ for every item in the list, passing the item as an argument and preserving outer braces.

The \indrisloop iterator is of particular interest. The loop provides the macros \indrisdepth, \currindris, \currindris@i, \nextindris, \nextindris@i, \indrisnr, \iflastindris and \ifloopbreak, which mean, respectively, the current nested depth/level, the current item on the current level, the current item on an arbitrary level 'i', the next item on the current level, the next item on an arbitrary level 'i', the numerical order of the current item on the current level, the boolean that indicates that the last item of the list has been reached on the current level of nesting, and the boolean that can be used to break the loop prematurely (before the list is exhausted) on any level.

You can do \iflastindris...\fi on any level. On any given level, it is possible to break out of the current loop by simply issuing \loopbreak. This would not affect the progress of the loops on other levels. When \iflastindris is true on a given level 'i', then \nextindris and \nextindris@i are empty on that level, but not necessarily the macros \currindris and \currindris@i.

The following provides a simple application of \indrisloop, which I posted on comp.text.tex in July 2011 in response to a posted question.

Example: \indrisloop

```
\makeatletter
\robust@def*\and@or@comma{%
  \ifnumcmpTF\indrisnr>\@ne{%
    \ifboolTF{lastindris}{%
      \xifstrcmpTF\sav@lastrefsep{and}{ and }{, }%
    }{, }%
  }{}%
}
\robust@def*\lastrefsep#1{%
  \ifstrcmpTF{#1}{and}{}{%
    \ifstrcmpTF{#1}{comma}{}{%
      \ifstrcmpTF{#1}{,}{}{%
        \@ltx@err{Invalid argument for \string\lastrefsep}\@ehc
      }%
    }%
  }%
```

```
342   \gdef\sav@lastrefsep{#1}%
343 }
344 \robust@def*\secref{\secref@i{\S}{\S\S}}
345 \new@def*\secref@i#1#2#3{%
346   \begingroup
347   \def\reserved{#3}%
348   \csv@@normalize*[,]\reserved
349   \def\do##1{\and@or@comma\ref{##1}}%
350   \def\reserved@a##1,##2,##3\@nil{%
351     \ifblankTF{##1}{}{%
352       \ifblankTF{##2}{%
353         #1~\ref{##1}%
354       }{%
355         #2~\indrisloop*\reserved\do
356       }%
357     }%
358   }%
359   \expandafter\reserved@a\reserved,,\@nil
360   \endgroup
361 }
362 \begin{document}
363 \lastrefsep{and} % or \lastrefsep{comma}
364 \section{aaa}\label{sec:aaa}
365 \section{bbb}\label{sec:bbb}
366 \section{ccc}\label{sec:ccc}
367 \section{ddd}
368 \noindent\secref{sec:aaa,sec:bbb,sec:ccc}.
369 \par\noindent\secref{sec:aaa}
370 \end{document}
```

10 Parsing 'tsv' lists

New macros: \tsv@@parse, \tsv@@loop

```
371 \tsv@@loop{⟨list⟩}
372 \tsv@@loop*⟨listcmd⟩
373 \tsv@@parse{⟨list⟩}
374 \tsv@@parse*⟨listcmd⟩
```

The macros \tsv@@loop and \tsv@@parse loop over 'non-separated' tokens (tsv-list), picking each token as an argument and applying the one-parameter, user-supplied, function \tsv@do to it while preserving outer braces in the item. The macros \tsv@@loop and \tsv@@parse are meant for general tsv-list processing.

The macros \tsv@@loop and \tsv@@parse (and their starred (*) variants) of course don't call \csv@@normalize or \kv@@normalize, but instead they trim leading and trailing spaces of individual tokens/entries. The only difference between \tsv@@loop and \tsv@@parse is that the former calls \cpttrimspace, while the latter calls \cpttrimspaces. The expandable function \cpttrimspaces trims all leading and trailing spaces (including chains of implicit spaces) around its argument, while \cpttrimspace trims only one leading and one trailing space around its argument. Both \cpttrimspace and \cpttrimspaces preserve outer braces around their arguments,

are expandable, and leave the trimmed token unchanged from its original form. The latter requirement is a condition of Michael Downes.

The starred variants of \tsv@@loop and \tsv@@parse expand ⟨listcmd⟩ once before commencing the loop.

Empty entries in ⟨list⟩ or ⟨listcmd⟩ will be processed if the boolean cpt@useempty is true. You may thus issue the command \UseEmptyEntry or \DiscardEmptyEntry before commencing the iteration. Both commands \tsv@@loop and \tsv@@parse can be nested to any level and can be mixed.

In the commands \tsv@loop and \tsv@@parse, it is always possible to break out of the loop prematurely at any level of nesting by issuing the command \loopbreak. Again, breaking an inner loop doesn't affect the continuation of the outer loop, and vice versa. The user can insert \tsvbreak as an element in ⟨list⟩ for any of the commands \tsv@@loop and \tsv@@parse so as to automatically break out of the list processing prematurely (i. e., before the list is exhausted).

Examples: \tsv@@loop, \tsv@@parse

```
\newcount\m
\newcount\n
\def\tsv@do#1{%
  \advance\m\@ne\n\z@
  \def\tsv@do##1{%
    \advance\n\@ne
    \csn@edef{w@\romannumeral\m @\romannumeral\n}{#1,##1}%
  }%
  \tsv@@loop{ x {y} \tsvbreak z  }%
}
\tsv@@loop{ a {b} {c} }
```

11 Version history

The following change history highlights significant changes that affect user utilities and interfaces; mutations of technical nature are not documented in this section. The numbers on the right-hand side of the following lists are section numbers; the star sign (⋆) means the subject features in the package but is not reflected anywhere in this user guide.

Version 0.2.6 [2011/09/10]

Bug fix in the command \UseNormalCatcodes . . . . . section 4

Version 0.2.5 [2011/07/20]

Enlarged the command \newvariables to include \newwrite and \newswitch . . . . . ⋆

Version 0.2.4 [2011/07/04]

Changes to command \cptonlypreamble . . . . . ⋆

Version 0.2c [2011/06/08]

Modified syntax for the handler of unknown options . . . . . section 7

Version 0.2b [2011/04/02]

The following commands were introduced . . . . . section 7

\XDeclareInvalidOptions	\XUnknownOptionHandler	\XDisableOptions
\XLogDisabledOptions	\ifoptdisabledTF	\ifoptdefTF

Version 0.2a [2011/02/15]

For efficiency reasons, canonical control sequences for futurelet characters are no longer defined automatically. The user is now responsible for specifying the canonical control sequences that should be defined . . . . . section 5

The following plural-form commands were introduced . . . . . section 7

\XDeclareOptions	\XDeclareCommandOptions
\XDeclareBooleanOptions	\XDeclareBiBooleanOptions

Version 0.1 [2011/01/25]

First public release.

Index

Index numbers refer to page numbers.