

The `concepts` package^{*}

Michiel Helvensteijn
`mhelvans+latex@gmail.com`

December 8, 2012

1 Introduction and Motivation

Documents with a lot of formal notation (such as papers about mathematics or theoretical computer science) can introduce a number of concepts that need to be managed. They'll have *names*, *descriptions* and associated *symbols* that need to be typeset, as well as *relations* between them.

I'm now writing my PhD thesis in the field of Theoretical Computer Science. It will be especially heavy with definitions. I need to make sure that every symbol is associated with no more than one concept and that all names and symbols are consistently used. I'll also need to generate a glossary with this information. But I don't want to manually keep track of all that. It's error-prone and it takes time. I'd rather focus on the theory.

There are already techniques and packages that help lighten the load:

- Rather than use any names or symbols directly in the text, declare a macro for each one. If you ever need to change one, you'll only have to do it in one place. You'll also be far less likely to introduce any typos.
- There are packages out there to keep track of and output a glossary¹.

But I'm not aware of any technique or package to ensure I'm not using a name or symbol inconsistently, thereby potentially confusing the reader. And even if there was, this is all still a lot of overhead.

I wrote the `concepts` package to automate all this stuff for me. Every time I introduce a new concept in my thesis I 'declare' it once. The package then defines all necessary macros for me and checks that I'm using them properly.

In `concepts` 0.0.4, most of the above already works but it cannot generate a glossary yet. In future versions, it will interface with the `glossaries` package to accomplish this.

I'm also planning to implement a rudimentary typesystem, to catch even more kinds of mistakes in symbol usage. There will also be additional ways of typesetting concept-related stuff, such as useful sentence fragments.

^{*}This document corresponds to `concepts` 0.0.4, dated 2012/12/08. Development of this package is organized at <http://latex-concepts.googlecode.com>. I am happy to receive feedback there.

¹It is sometimes called a nomenclature. The distinction is subtle.

2 Usage

In short: every concept first has to be declared using the `\NewConcept` macro. Afterwards, its name and associated symbols can be typeset using other macros.

`\NewConcept` $\{\langle concept\ key\rangle\}$ [$\langle options\rangle$]

Every concept needs a unique $\langle concept\ key\rangle$, by which it will be identified for the rest of the document. This key can also be used to automatically derive the name of the concept as well as the macro used to typeset the name.

Usually, you'll also want to add $\langle options\rangle$. This optional argument takes a comma-separated list of **key=value** pairs. The following is a list of available options. Note that the option names are case-sensitive:

name the name of the concept
— default: $\langle concept\ key\rangle$

Name the capitalized name of the concept, for use in the beginning of a sentence
— default: $\langle name\rangle$ with the first letter capitalized

plural the plural form of the name
— default: $\langle name\rangle$ s

Plural the capitalized, pluralized name of the concept
— default: $\langle plural\rangle$ with the first letter capitalized

namecmd the 'short' command that may be used to typeset the name of this concept; this option **has** to be specified for any command to be defined, but the **=value** part may be omitted to get the default
— default: $\backslash\langle concept\ key\rangle$

symbols a comma-separated list of symbols that may represent instances of this concept, delimited by curly brackets
— default: $\{\}$ (the empty list)

symbolcmd the 'short' command that may be used to typeset a specific symbol of this concept, given a numeric index

Here are a few examples which will also be used to illustrate the other commands:

```
\NewConcept{swproduct}[
  name      = software product,    % options 'plural', 'Plural'
  Name      = Software Product,    % are implicitly defined
  namecmd   = \product,            % defines \product
  symbols   = {p, q, r},           % p, q, r represent products
  symbolcmd = \p                   % defines \p
]

\let\delta\relax                % I won't be using the greek letter
\let\d\relax                     % or the 'dot under a letter' macro

\NewConcept{delta}[
  namecmd,                        % defines \delta
  symbols   = {x, y, z},          % x, y and z represent deltas
  symbolcmd = \d                  % defines \d
]
```

There are certain restrictions on new concept declarations. You may not use the same $\langle concept\ key \rangle$ more than once. You may not use the same symbol for more than one concept (this is a feature; the package will report an error if you do). Also, both $\langle namecmd \rangle$ and $\langle symbolcmd \rangle$ are subject to the rules governing $\backslash newcommand$. They may not be reused, or you will see a standard L^AT_EX error. Finally, any value you supply must behave properly in an expansion-only context, e.g. be robust.

$\backslash ConceptOption$ $\{ \langle concept\ key \rangle \} \{ \langle option\ key \rangle \}$

This command can be used to get back any option value given a specific $\langle concept\ key \rangle$ and $\langle option\ key \rangle$. For example:

```
 $\backslash ConceptOption\{delta\}\{Plural\}$ 
```

Deltas

```
 $\edef\prd{\backslash ConceptOption\{swproduct\}\{namecmd\}}
\{\ttfamily\expandafter\detokenize\expandafter\{\backslash prd\}\}%
\expands\ to\ '\backslash prd'$ 
```

$\backslash product$ expands to “software product”

$\backslash ConceptOption$ is ‘fully expandable’, meaning that it can expand at least down to the value that was given to the option. This is not guaranteed for the $\backslash ConceptName$ and $\backslash ConceptSymbol$ commands, for instance.

As you can observe from the $\backslash product$ example above, options that expect a command sequence are stored with an accompanying $\backslash noexpand$. That means that in an $\backslash edef$ context, $\backslash ConceptOption$ expands down to the stored command but no further. After that you can expand it further if you wish.

$\backslash ConceptName$ $[\wedge] [*] \{ \langle concept\ key \rangle \}$

With this command you can typeset the name of the concept with $\langle concept\ key \rangle$ in any of four forms. The \wedge modifier gives you the capitalized version. The $*$ modifier gives you the plural version. The combination gives you both. The order between \wedge and $*$ doesn’t matter.

```
 $\backslash ConceptName^*\{delta\}$  can transform a  $\backslash ConceptName\{swproduct\}$ .
```

Deltas can transform a software product.

$\langle namecmd \rangle$ $[\wedge] [*]$

This is the ‘short’ version of $\backslash ConceptName$, specific for each concept that was declared with the `namecmd` option. It also supports the same modifiers.

```
 $\backslash delta^*$  can easily transform a  $\backslash product$ .
```

Deltas can easily transform a software product.

`\ConceptSymbol` { $\langle concept\ key \rangle$ } [$\langle symbol\ index \rangle$]

This command typesets a specific symbol associated with a given concept. Specify the concept by providing the $\langle concept\ key \rangle$ and the symbol by giving the $\langle symbol\ index \rangle$.

The index is 1-based and points to a place in the symbol list provided with the concept options. If a concept has only one allocated symbol the index may be omitted. If there is more than one symbol, the index is mandatory.

```
$\ConceptSymbol{delta}[2] \cdot \ConceptSymbol{delta}[1] =
\ConceptSymbol{delta}[1] \cdot \ConceptSymbol{delta}[2]$
```

$$y \cdot x = x \cdot y$$

$\langle symbolcmd \rangle$ [$\langle symbol\ index \rangle$]

This is the ‘short’ version of `\ConceptSymbol`, specific for each concept that was declared with the `symbolcmd` option. The optional index is given directly following the command itself (optionally with whitespace inbetween). To be clear: the index doesn’t need any delimiters.

```
$(\d2 \cdot \d1)(\p1) = \d2(\d1(\p1)) = \d2(\p1') = \p1''$
```

$$(y \cdot x)(p) = y(x(p)) = y(p') = p''$$

As you can see, this short construct requires a lot less space than the full `\ConceptSymbol` command, so its use is recommended for readability.

3 Future Work

Everything up to this version of the package has been a bit of an experiment for me. A way to get me started. I may still fix one or two issues for the 0.0.x series, but I will soon start from scratch with all I’ve learned.

There will be two major changes starting from version 0.1.0. First of all, the package will be built on top of the `glossaries` package, which already does much of the work I’m now doing manually. This was always the plan, as we’ll want to typeset a glossary with our concepts, and I don’t want to reinvent the wheel. The `glossaries` package is actively developed and has a great amount of features we can take advantage of. Secondly, I will program the 0.1.0 series using L^AT_EX₃.

Here is an incomplete list of the features I am planning to implement:

- full integration with the `glossaries` package
- commands to *modify* existing concepts
- typesetting a *summary* of the concepts introduced in each chapter / section
- typesetting *sentence fragments* such as $\boxed{x, y, z}$ using only the code `\d[1,2,3]` or $\boxed{\deltas x, y, z}$ using only `\delta[1,2,3]`

- management of *tuples* and *sets* of concept instances
- management of *subconcepts* plus a rudimentary *typesystem* that ensures concept instances are not used where a different concept is expected

4 Implementation

We now show and explain the entire implementation in `concepts.sty`.

4.1 Package Info

```
1 \NeedsTeXFormat{LaTeX2e}
2 \ProvidesPackage{concepts}[2012/12/08 0.0.4
3   managing names and symbols of document specific formal concepts]
```

4.2 Packages

These are the packages we'll need.

```
4 \RequirePackage{etextools}
5 \RequirePackage{nth}
6 \RequirePackage{xspace}
7 \RequirePackage{xparse} % 1
8 \RequirePackage{ltxkeys}[2012/11/17] % 2
9 \RequirePackage{xstring}
```

We need a very recent version of `ltxkeys` in order to properly handle list-values. Note that `xparse` needs to be loaded before `ltxkeys` or things go wrong somehow.

4.3 Facilitating Easy Data Access

`\cnc@d` $\{\langle identifier \rangle\}$

`\cnc@g` $\{\langle identifier \rangle\}$

This package needs to store and retrieve a lot of data. To make the rest of the code more readable, we define the following commands. They allow a more freeform description of the data.

`\cnc@d` returns a control sequence name that resolves to a specific piece of data in `\csname` context. We can get access to the data itself by using `\cnc@g`.

```
10 \newcommand*\cnc@d[1]{\cnc@data@#1}
11 \newcommand*\cnc@g[1]{\csuse{\cnc@d{#1}}}
```

Both take an identifier of one the following shapes:

- `concepts`
- `concept(\name).option(\name)`
- `concept(\name).option(\name).given`

- `concept(\langle name \rangle).option(\langle name \rangle).count`
- `concept(\langle name \rangle).option(\langle name \rangle).index(\langle number \rangle)`
- `symbol(\langle name \rangle).concept`

4.4 Private General Purpose Macros and Toggles

`\cnc@upper` `{\langle string \rangle}`

We’re going to need a command that capitalizes the first letter of a string which fully expands its argument. So here it is.

```
12 \newcommand*\cnc@upper}[1]{\ExpandAftercmds\MakeUppercase{#1}}
```

`\cnc@grabnumber` `{\langle token sequence containing #1 \rangle} [\langle number \rangle]`

This is a command we’re going to use for the automatically defined short symbol macros later. It has one ‘real’ mandatory argument and then it captures all numerals (0...9) that follow it. These numerals are then substituted for all occurrences of #1 in the mandatory argument which is then ‘returned’.

```
13 \newcommand{\cnc@grabnumber}[1]{%
14   \def\cnc@dowithnum##1{#1}%
15   \futuredef [0123456789]{\cnc@n}{\cnc@dowithnum{\cnc@n}}%
16 }
```

4.5 Private Specific-purpose Macros

`\cnc@conceptname` `{\langle plural \rangle} {\langle capitalized \rangle} {\langle plural \rangle} {\langle concept key \rangle}`

This typesets the name of a specific concept in one of four forms. It can be capitalized or not; and it can be singular or plural. We define this private macro because there will be two public macros with this functionality and we want to define the behavior in only one place.

The first three arguments are `xparse` style booleans. The first and third are the same because the public macros allow both orders between the `*` and `^` modifiers and we want to have a simple one-to-one mapping between their arguments and the arguments of this private macro.

```
17 \newcommand{\cnc@conceptname}[4]{%
```

We test if *both* the first and third arguments are true, meaning that the public command has two `*` modifiers. If it does, we give a package error.

```
18   \ifboolexpr{ test{\IfBooleanTF{#1}} and test{\IfBooleanTF{#3}} }{%
19     \PackageError{concepts}%
20       {You used the * modifier twice; once is enough}%
21       {I will pretend you just used one *}%
22   }{}
```

And then we simply typeset the correct value from our datastore.

```

23   \ifboolexpr{ test{\IfBooleanTF{#1}} or test{\IfBooleanTF{#3}} }{%
24       \IfBooleanTF{#2}%
25       {\cnc@g{concept{#4}.option(Plural)}}%
26       {\cnc@g{concept{#4}.option(plural)}}%
27   }{%
28       \IfBooleanTF{#2}%
29       {\cnc@g{concept{#4}.option(Name)}}%
30       {\cnc@g{concept{#4}.option(name)}}%
31   }%

```

We end with an `\xspace` because the public command may need one.

```

32   \xspace%
33 }

```

`\cnc@conceptsymb`ol $\{\langle concept\ key\rangle\}\{\langle index\rangle\}$

This is the private macro which takes a concept key and an index and returns the corresponding symbol from our data-store. We use it in the public macros that offer this functionality.

```

34 \newcommand*{\cnc@conceptsymb}[2]{%

```

We check if an actual numerical value was passed.

```

35   \IfInteger{#2}{%

```

If we got one, we check whether it is larger than the number of symbols allocated to the concept.

```

36       \edef\cnc@symbolcount{\cnc@g{concept{#1}.option(symbols).count}}%
37       \ifnumcomp{#2}{>}{\cnc@symbolcount}{%

```

If it is, we report an ‘index out of bounds’ error. We first prepare an appropriate sentence fragment so the error message becomes more readable.

```

38       \edef\cnc@nth{#2\mathstrut\delimiterspace}%
39       \ExpandNext\IfStrEq{\cnc@symbolcount}{0}{%
40           \edef\cnc@so many{no symbols}%
41       }{\ExpandNext\IfStrEq{\cnc@symbolcount}{1}{%
42           \edef\cnc@so many{only 1 symbol}%
43       }{%
44           \edef\cnc@so many{only \cnc@symbolcount\space symbols}%
45       }}%
46       \PackageError{concepts}%
47           {You asked for the \cnc@nth\space ‘#1’ symbol,
48             but\MessageBreak the ‘#1’ concept has
49             \cnc@so many\space allocated}%
50           {I will pretend you didn’t ask for a symbol here.}%

```

```
51      }%
```

But if the number is within bounds, great! We just return the stored symbol.

```
52      {%
53          \cnc@g{concept(#1).option(symbols).index(#2)}%
54      }%

55  }%
```

Now follows the ‘else’ branch: we didn’t get a numerical index.

```
56  {%
```

If there is only one symbol allocated to this concept, we don’t care and return that symbol.

```
57      \edef\cnc@symbolcount{\cnc@g{concept(#1).option(symbols).count}}%
58      \ifnumcomp{\cnc@symbolcount}{=}{1}{}%
59          \cnc@g{concept(#1).option(symbols).index(1)}%
60      }%
```

If there are multiple symbols, the lack of an index is ambiguous and we report a package error.

```
61      {%
62          \PackageError{concepts}%
63              {You didn't specify a number, but the '#1'
64              \MessageBreak concept has more than one symbol
65              allocated; please\MessageBreak specify a number
66              to typeset a specific symbol}%
67              {I will pretend you didn't ask for a symbol here.}%
68      }%

69  }%
70 }
```

4.6 Public Macros

We now implement the macros that will be used directly by package users.

\NewConcept $\{\langle concept\ key\rangle\} [\langle options\rangle]$

The `\NewConcept` macro is defined using `xparse`, since we want the optional argument after the mandatory one (it’s just nicer).

```
71 \NewDocumentCommand{\NewConcept}{m O{}}{%
```

Is this concept key already defined? If so, we report a package error.


```

72   \xifinlist{#1}{\cnc@g{concepts}}{%
73     \PackageError{concepts}%
74       {The concept key '#1' is already taken}%
75       {I will pretend that this '\protect\newconcept'%
76         didn't happen.}%
77   }%

```

Otherwise, we start the actual processing of this new concept.

```

78   {%

```

We add the concept to the concepts list in our datastore.

```

79     \listcsxadd{\cnc@d{concepts}}{#1}%

```

We then check which options were explicitly specified by the user. This results in a set of toggles in our datastore, which may be used by other code.

```

80     \DeclareRobustCommand*\cnc@registertoggle}[1]{%
81       \newtoggle{\cnc@d{concept(#1).option(##1).given}}%
82       \togglefalse{\cnc@d{concept(#1).option(##1).given}}%
83       \ltxkeys@newordkey[cnc@toggle]{#1}{##1}[]%
84       {\toggletrue{\cnc@d{concept(#1).option(##1).given}}}%
85     }
86     \cnc@registertoggle{name}%
87     \cnc@registertoggle{Name}%
88     \cnc@registertoggle{plural}%
89     \cnc@registertoggle{Plural}%
90     \cnc@registertoggle{namecmd}%
91     \cnc@registertoggle{symbols}%
92     \cnc@registertoggle{symbolcmd}%
93     \ltxkeys@setkeys*cnc@toggle{#1}{#2}%

```

We now register the concept name options `name`, `Name`, `plural` and `Plural`. This is also where we set their default values.

```

94     \ltxkeys@newordkey[cnc]{#1}{name}%
95     [#1]%
96     {\csdef{\cnc@d{concept(#1).option(name)}}{##1}}%
97     \ltxkeys@newordkey[cnc]{#1}{Name}%
98     [\cnc@upper{\cnc@g{concept(#1).option(name)}}]%
99     {\csdef{\cnc@d{concept(#1).option(Name)}}{##1}}%
100    \ltxkeys@newordkey[cnc]{#1}{plural}%
101    [\cnc@g{concept(#1).option(name)}s]%
102    {\csdef{\cnc@d{concept(#1).option(plural)}}{##1}}%
103    \ltxkeys@newordkey[cnc]{#1}{Plural}%
104    [\cnc@upper{\cnc@g{concept(#1).option(plural)}}]%
105    {\csdef{\cnc@d{concept(#1).option(Plural)}}{##1}}%

```

We next register the `namecmd` option. Its default value is the concept key with a `\` in front of it. Unlike most other options, though, we require the option name to be explicitly given by the user to actually define the macro. The following code also contains the test.

```

106      \expandaftercnds{\ltxkeys@newordkey[cnc]{#1}{namecmd}%
107      []{\csname#1\endcsname}}{%
108      \iftoggle{\cnc@d{concept(#1).option(namecmd).given}}{%

```

We register the option value as given.

```

109      \csdef{\cnc@d{concept(#1).option(namecmd)}}{\noexpand#1}

```

$\langle namecmd \rangle$ `[*] [^] [*]`

If the option is processed, we define the concept-specific name command with the following code:

```

110      \NewDocumentCommand{##1}{t* t^ t*}{%
111      \cnc@conceptname{####1}{####2}{####3}{#1}%
112      }%
113      }{}%
114      }%

```

We now register the `symbols` option. This option expects a list. The callback of the following code processes it one symbol at a time. And we first initialize the symbol counter to 0 in case the list is empty

```

115      \csdef{\cnc@d{concept(#1).option(symbols).count}}{0}%
116      \ltxkeys@newlistkey[cnc]{#1}{symbols}[]{%

```

We check if this particular symbol is already defined. We don't want a symbol allocated to different concepts. Or allocated twice to the same concept, for that matter. If all is fine, we update the symbol counter for this concept, we add the symbol itself and we update the reverse map we use for checking duplicates.

```

117      \ifcsundef{\cnc@d{symbol(\detokenize{##1}).concept}}{%
118      \csdef{\cnc@d{concept(#1)%
119      .option(symbols)%
120      .count}}{\ltxkeys@listcount}%
121      \csdef{\cnc@d{concept(#1)%
122      .option(symbols)%
123      .index(\ltxkeys@listcount)}}{##1}%
124      \csdef{\cnc@d{symbol(\detokenize{##1}).concept}}{##1}%
125      }%

```

If the symbol is already in use we report a package error.

```

126      {%
127      \PackageError{concepts}%
128      {The symbol '\detokenize{##1}' is already allocated
129      to the '\cnc@g{symbol(\detokenize{##1}).concept}'
130      concept}%
131      {I will pretend that you did not
132      try to add this symbol.}%
133      }%

```

```
134      }%
```

We register the `symbolcmd` option. It does not really have a default, but we give an empty default so we can test for the empty string inside.

```
135      \ltxkeys@newwordkey[cnc]{#1}{symbolcmd}[]{%
136          \ifblank{##1}{-}{%}
```

We register the option value as given.

```
137          \csdef{\cnc@d{concept(#1).option(symbolcmd)}}{\noexpand##1}
```

`\symbolcmd` [`\index`]

If the option is processed, we now define the concept-specific ‘short’ command used to typeset one of the allocated symbols. It doesn’t have a conventional argument, but it grabs all numerals following it and uses that as an index to the symbol.

```
138          \newrobustcmd*{##1}{%
139              \cnc@grabnumber{%
140                  \cnc@conceptsymbols{#1}{#####1}%
141              }%
142          }%

143      }%
144  }%
```

Finally, we issue the command to parse and process all options.

```
145      \ltxkeys@launchkeys[cnc]{#1}{#2}%

146  }%
147 }
```

`\ConceptOption` {`\concept key`} {`\option key`}

The point of this command is that it can retrieve any option value in an expandable way. That means we can’t use `xparse`, but we don’t need it.

```
148 \newcommand*\ConceptOption}[2]{%
149     \cnc@g{concept(#1).option(#2)}%
150 }
```

Unfortunately, as of writing this, not all options are stored in a fully expandable way yet. But they will be in a later version.

`\ConceptName` [*] [ˆ] [*] {`\concept key`}

This implementation simply calls our private macro for retrieving the name in one of four forms.

```

151 \NewDocumentCommand{\ConceptName}{t* t^ t* m}{%
152   \cnc@conceptname{#1}{#2}{#3}{#4}%
153 }

```

`\ConceptSymbol` $\{\langle concept\ key\rangle\} [\langle index\rangle]$

This implementation simply calls our private macro for retrieving the symbol with the given index. The index is optional (as it is in the ‘short’ version) and defaults to 1.

```

154 \NewDocumentCommand{\ConceptSymbol}{m O{1}}{%
155   \cnc@conceptsymbols{#1}{#2}%
156 }

```

Change History

0.0.1	General: initial version 1	0.0.3	General: separated the .dtx file from the .sty file 1
0.0.2	General: put the package into a .dtx file 1	0.0.4	General: finished the documentation and made a few fixes 1

Index

Numbers written in *italic* refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in *roman* refer to the code lines where the entry is used.

C	<code>\cnc@somany</code> 40, 42, 44, 49	<code>\detokenize</code>
<code>\cnc@n</code> 15	<code>\cnc@symbolcount</code> 117, 124, 128, 129	
<code>\cnc@conceptname</code> 17, 111, 152	36, 37, 39, 41, 44, 57, 58	E
<code>\cnc@conceptsymbols</code> 34, 140, 155	<code>\cnc@upper</code> . 12, 98, 104	<code>\edef</code> 36, 38, 40, 42, 44, 57
<code>\cnc@d</code> . . . 10, 79, 81, 82, 84, 96, 99, 102, 105, 108, 109, 115, 117, 118, 121, 124, 137	<code>\ConceptName</code> . . . 3, 151	<code>\endcsname</code> 107
<code>\cnc@dowithnum</code> . . 14, 15	<code>\ConceptOption</code> . . 3, 148	<code>\ExpandAftercmds</code> . . 12
<code>\cnc@g</code> . . . 10, 25, 26, 29, 30, 36, 53, 57, 59, 72, 98, 101, 104, 129, 149	<code>\ConceptSymbol</code> . . 4, 154	<code>\expandaftercmds</code> . . 106
	<code>\csdef</code> 96, 99, 102, 105, 109, 115, 121, 137	<code>\ExpandNext</code> 39, 41
	<code>\csedef</code> 118, 124	F
	<code>\csname</code> 107	<code>\futuredef</code> 15
	<code>\csuse</code> 11	I
D		<code>\ifblank</code> 136
<code>\cnc@grabnumber</code> 13, 139	<code>\DeclareRobustCommand</code> 80	<code>\IfBooleanTF</code> 18, 23, 24, 28
<code>\cnc@nth</code> 38, 47	<code>\def</code> 14	<code>\ifboolexpr</code> 18, 23
<code>\cnc@registertoggle</code> 80, 86–92	<code>\delimiter</code> 38	<code>\ifcsundef</code> 117
		<code>\IfInteger</code> 35

\ifnumcomp	37, 58	\MessageBreak	48, 64, 65	19, 46, 62, 73, 127
\IfStrEq	39, 41			\protect 75
\iftoggle	108	N		\ProvidesPackage . . . 2
L		\NeedsTeXFormat	1	R
\listcsxadd	79	\newcommand		\RequirePackage . . . 4–9
\ltxkeys@launchkeys	145	. 10–13, 17, 34, 148		
\ltxkeys@listcount .		\NewConcept	2, <u>71</u>	S
.	120, 123	\newconcept	75	\space 44, 47, 49
\ltxkeys@newlistkey	116	\NewDocumentCommand		
\ltxkeys@newordkey .		. 71, 110, 151, 154		T
. . . 83, 94, 97,		\newrobustcmd	138	\togglefalse 82
100, 103, 106, 135		\newtoggle	81	\toggletrue 84
\ltxkeys@setkeys . .	93	\noexpand	109, 137	
		\nthSuff	38	X
M		P		\xifinlist 72
\MakeUppercase	12	\PackageError		\xspace 32