The concepts package*

Michiel Helvensteijn
mhelvens+latex@gmail.com

January 1, 2013

Development of this package is organized at latex-concepts.googlecode.com.
I am happy to receive feedback there!

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 glossary[1].

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, using all these techniques at the same time is 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 0.0.6, 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, and more.

I'm also planning to implement a rudimentary typesystem, to catch even more kinds of mistakes in symbol usage. Also, I may want to integrate the `ligature` option from the semantic package, which allows you to choose arbitrary characters to typeset your symbols in math mode (with some restrictions).

*This document corresponds to concepts 0.0.6, dated 2012/12/29.
[1]It is sometimes called a nomenclature. The distinction is subtle.

2 Usage

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

`\NewConcept` `{`⟨*concept key*⟩`}` `{`⟨*options*⟩`}`

Every concept needs a unique ⟨*concept key*⟩, 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.

Then you'll want to add ⟨*options*⟩. This 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: ⟨*concept key*⟩

Name the capitalized name of the concept, for use in the beginning of a sentence
— default: ⟨*name*⟩ with the first letter capitalized

plural the plural form of the name
— default: ⟨*name*⟩`s`

Plural the capitalized, pluralized name of the concept
— default: ⟨*plural*⟩ 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: `\`⟨*concept key*⟩

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

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},               % p represents a product
  symbolcmd = \p                 % defines \p
}
```

```
\let\delta\relax \let\d\relax  % I won't be using these
\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 ⟨*concept key*⟩ 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 ⟨*namecmd*⟩ and ⟨*symbolcmd*⟩ are subject to the rules governing `\newcommand`. They may not be reused, or you will see a standard LaTeX error. Finally, any value you supply must behave properly in an expansion-only context, e.g. be robust.

`\ConceptOption` `{`⟨*concept key*⟩`}` `{`⟨*option key*⟩`}`

This command can be used to get back any option value given a specific ⟨*concept key*⟩ and ⟨*option key*⟩. For example:

```
\ConceptOption{delta}{Plural}
```

Deltas

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

`\product` expands to “software product”.

`\ConceptOption` is ‘fully expandable’, meaning that it can expand at least down to the value that was given to the option. (This is not (yet) guaranteed for the other commands.)

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

`\ConceptName` [`^`] [`*`] `{`⟨*concept key*⟩`}`

With this command you can typeset the name of the concept with ⟨*concept key*⟩ in any of four forms. The `^` modifier gives you the capitalized version. The `*` modifier gives you the plural version. The combination gives you both. The order between `^` and `*` doesn't matter.

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

Deltas can transform a software product.

⟨*namecmd*⟩ [`^`] [`*`]

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

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

Deltas can easily transform a software product.

`\ConceptSymbol` `{`⟨*concept key*⟩`}` `[`⟨*symbol index*⟩`]`

This command typesets a specific symbol associated with a a given concept. It is always typeset in math mode. Specify the concept with the ⟨*concept key*⟩ and the symbol with the ⟨*symbol index*⟩.

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.

```
\product* have symbols like \ConceptSymbol{swproduct}.
```

Software products have symbols like p.

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

$y \cdot x = x \cdot y$

⟨*symbolcmd*⟩ [⟨*symbol index*⟩]

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. It doesn't need any delimiters. However, you are allowed to use square brackets. See the list-variation of this short command below.

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

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

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

`\ConceptSymbols` `{`⟨*concept key*⟩`}` `[`⟨*separator*⟩`]` `[`⟨*last separator*⟩`]` `{`⟨*symbol indices*⟩`}`

This command typesets a ⟨*separator*⟩ separated list of symbols associated with the given concept, optionally with a different ⟨*last separator*⟩. Specify the concept with the ⟨*concept key*⟩ and the symbol-list with the ⟨*symbol indices*⟩. ⟨*separator*⟩ defaults to `,` and ⟨*last separator*⟩ defaults to ⟨*separator*⟩. The whole list is typeset in math mode, so if you'd like a non-math delimiter, you need to use `$` tokens.

The indices are 1-based and point to a place in the symbol list provided with the concept options. The index-list is mandatory, but can be empty.

```
The symbols \ConceptSymbols{delta}{1,2,3} represent \delta*.
```

The symbols x, y, z represent deltas.

```
$\{\ConceptSymbols{delta}[;]{1,1,1}\}$ contains only \d1.
```

$\{x; x; x\}$ contains only x.

```
What about \ConceptSymbols{delta}[,][$ and $]{3, 2, 1}?
```
What about z, y and x?

```
xx\ConceptSymbols{delta}{}xx
```
xxxx

If you need any symbol in the resulting list to have some decoration (like a prime, subscript or superscript) you can decorate the corresponding index accordingly. This currently only works for decorations that would be specified *after* the symbol. Each element of ⟨*symbol indices*⟩ still needs to start with the index itself:

```
\ldots such as the symbols in
  $(\ConceptSymbols{delta}{1_1, 2'', 3^{\d1(\p)}})$.
```
...such as the symbols in $(x_1, y'', z^{x(p)})$.

⟨*symbolcmd*⟩ `[`⟨*symbol indices*⟩`]`

This is the 'short' version of `\ConceptSymbols`, specific for each concept that was declared with the `symbolcmd` option. The ⟨*symbol indices*⟩ list needs to be delimited by square brackets as shown below.

```
$\forall \d[1,2] \in D:\ \d1 \,|\, \d2\ \Rightarrow\ {}
  \exists \d[3'] \in D:\ \d1 \prec \d3' \land \d2 \prec \d3'$
```
$\forall x, y \in D:\ x \| y\ \Rightarrow\ \exists z' \in D:\ x \prec z' \land y \prec z'$

```
xx\d[]xx\p[]xx
```
xxxxxx

`\ConceptNameAndSymbols` `[ ^ ]` `{`⟨*concept key*⟩`}` `[`⟨*separator*⟩`]` `[`⟨*last separator*⟩`]` `{`⟨*symbol indices*⟩`}`

This command is the 'hybrid' version of `\ConceptName` and `\ConceptSymbols`. It typesets the name of the concept followed by a ⟨*separator*⟩ separated list of symbols associated with the given concept, optionally with a different ⟨*last separator*⟩. Specify the concept with the ⟨*concept key*⟩ and the symbol-list with ⟨*symbol indices*⟩. ⟨*separator*⟩ defaults to `,` and ⟨*last separator*⟩ defaults to `$ and $`. The name is typeset in text mode and the list is typeset in math mode.

The indices are 1-based and point to a place in the symbol list provided with the concept options. The index-list is mandatory and cannot be empty.

You can still supply the `^` modifier to capitalize the name but the choice between singular and plural form is determined by the number of ⟨*symbol indices*⟩.

```
We use \ConceptNameAndSymbols{delta}[,][$ and also $]{1,2,3}.
```
We use deltas x, y and also z.

For some other neat tricks, read the documentation of `\ConceptSymbols` above.

⟨*namecmd*⟩ [^] [⟨*symbol indices*⟩]

This is a 'short' version of `\ConceptNameAndSymbols`, specific for each concept that was declared with the `namecmd` option. The ⟨*symbol indices*⟩ list needs to be delimited by square brackets as shown below. The list is comma-delimited and the last (or only) delimiter is the word 'and'.

```
\delta^[1,2] come before \delta[3].
```

Deltas x and y come before delta z.

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 LaTeX3.

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

- full integration with the glossaries package
- typesetting a *summary* of the concepts introduced in each chapter / section
- 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 from `concepts.sty`.

4.1 Package Info

```
1 \NeedsTeXFormat{LaTeX2e}
2 \ProvidesPackage{concepts}[2012/12/29 0.0.6
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` {⟨*identifier*⟩}

`\cnc@g` {⟨*identifier*⟩}

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(`⟨*name*⟩`).option(`⟨*name*⟩`).count`
- `concept(`⟨*name*⟩`).option(`⟨*name*⟩`).index(`⟨*number*⟩`)`
- `symbol(`⟨*name*⟩`).concept`

4.4 Private General Purpose Macros and Toggles

`\cnc@upper` `{`⟨*string*⟩`}`

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

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

`\cnc@grabnumber` `{`⟨*token sequence containing #1*⟩`}` [⟨*number*⟩]

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'.

```
\newcommand{\cnc@grabnumber}[1]{%
  \def\cnc@dowithnum##1{#1}%
  \futuredef[0123456789]{\cnc@@n}%
    {\expandafter\cnc@dowithnum\expandafter{\cnc@@n}}%
}
```

`\cnc@csvlistsize` `{`⟨*csvlist*⟩`}` `{`⟨*command sequence*⟩`}`

This macro takes a comma-separated list of... anything, and stores its size in the given macro as a simple decimal string.

```
\newcounter{cnc@listsize}
\newrobustcmd{\cnc@csvlistsize}[2]{%
  \setcounter{cnc@listsize}{0}%
  \def\do##1{\stepcounter{cnc@listsize}}\docsvlist{#1}%
  \edef#2{\arabic{cnc@listsize}}%
}
```

4.5 Private Specific-purpose Macros

`\cnc@conceptname` `{`⟨*plural*⟩`}` `{`⟨*capitalized*⟩`}` `{`⟨*plural*⟩`}` `{`⟨*concept key*⟩`}`

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.

```
\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.

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

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

```
  \ifboolexpr{ test{\IfBooleanTF{#1}} or test{\IfBooleanTF{#3}} }{%
    \IfBooleanTF{#2}%
      {\cnc@g{concept(#4).option(Plural)}}%
      {\cnc@g{concept(#4).option(plural)}}%
  }{%
    \IfBooleanTF{#2}%
      {\cnc@g{concept(#4).option(Name)}}%
      {\cnc@g{concept(#4).option(name)}}%
  }%
```

```
}
```

`\cnc@conceptsymbol` `{`⟨*concept key*⟩`}` `{`⟨*index*⟩`}`

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.

The first thing we do is grab the prefix of the second argument that consists of numerals. The rest of the argument is simply left in the input stream afterwards.

```
\newcommand*{\cnc@conceptsymbol}[2]{%
  \cnc@grabnumber{%
```

We check if an actual numerical value was passed.

```
    \ifstrempty{##1}{%
```

No, we didn't get a numerical index. If there is only one symbol allocated to this concept, we don't care and return that symbol.

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

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

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

```
    }%
```

Now follows the ‘else’ branch: we did get a numerical index!

```
    {%
```

We check whether it is larger than the number of symbols allocated to the concept.

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

```
        \edef\cnc@nth{##1\nthSuff0##1\delimiter}%
        \ExpandNext\IfStrEq{\cnc@symbolcount}{0}{%
          \edef\cnc@somany{no symbols}%
        }{\ExpandNext\IfStrEq{\cnc@symbolcount}{1}{%
          \edef\cnc@somany{only 1 symbol}%
        }{%
          \edef\cnc@somany{only \cnc@symbolcount\space symbols}%
        }}%
        \PackageError{concepts}%
          {You asked for the \cnc@nth\space '#1' symbol,
           but\MessageBreak the '#1' concept has
           \cnc@somany\space allocated}%
          {I will pretend you didn't ask for a symbol here.}%
```

```
      }%
```

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

```
      {%
        \ensuremath{\cnc@g{concept(#1).option(symbols).index(##1)}}%
      }%
```

```
    }%
```

Now ends our `\cnc@grabnumber` command, and we supply the second argument that may contain the numbers. Just in case it’s empty, we make sure we don’t grab anything that comes after the second argument by adding a `\relax`.

```
  }#2\relax%
```

```
}
```

`\cnc@conceptsymbols` `{`⟨*concept key*⟩`}` `{`⟨*separator*⟩`}` `{`⟨*last separator*⟩`}` `[`⟨*indices*⟩`]`

This is the private macro which takes a concept key and a comma-separated list of symbol-indices and returns a string-separated list of corresponding concept symbols from our data-store, possibly with a different token as the last separator. We use it in the public macros that offer this functionality. The last argument is optional to make it easier to define our ‘short’ symbol-list command later.

```
\newcounter{cnc@separatorcount}%
```

```
\NewDocumentCommand{\cnc@conceptsymbols}{m m m O{1}}{%
  \def\cnc@result{}%
```

We loop through the list of indices and produce the symbols one-by-one. We use a rather ugly trick to possibly have a different ⟨*last separator*⟩. Each separator is stored in a separate macro, and we simply redefine the last one after the loop. This causes one macro to be defined for every single separator in the document, but we don't care, since most of this will be rewritten when we switch to LaTeX3.

```
  \ifblank{#4}{}{%
    \def\do##1{%
      \stepcounter{cnc@separatorcount}%
      \edef\cnc@sepcsname{cnc@separator\arabic{cnc@separatorcount}}%
      \csdef{\cnc@sepcsname}{#2}%
      \expandafter\expandafter\expandafter\def\expandafter%
        \expandafter\expandafter\cnc@result\expandafter%
        \expandafter\expandafter{\expandafter\cnc@result%
        \csname\cnc@sepcsname\endcsname%
        \cnc@conceptsymbol{#1}{##1}}%
    }\docsvlist{#4}%
    \edef\cnc@sepcsname{cnc@separator\arabic{cnc@separatorcount}}%
    \csdef{\cnc@sepcsname}{#3}%
    \ensuremath{\expandafter\@gobble\cnc@result}%
  }%
```

```
}
```

`\cnc@nameandsymbols` `{`⟨*concept key*⟩`}` `{`⟨*star*⟩`}` `{`⟨*capitalized*⟩`}` `{`⟨*star*⟩`}`
`{`⟨*separator*⟩`}` `{`⟨*last separator*⟩`}` `{`⟨*indices*⟩`}`

This is the private macro which takes a concept key, a few modifiers, a comma-separated list of symbol-indices and custom separators and returns a separated list of the corresponding concept symbols from our data-store. We use it in the public macros that offer this functionality.

```
\newrobustcmd{\cnc@nameandsymbols}[7]{%
```

We first check if any `*` modifiers were given and, if so, generate an error.

```
  \ifboolexpr{ test{\IfBooleanTF{#2}} or test{\IfBooleanTF{#4}} }{%
    \PackageError{concepts}%
    {You used the * modifier, but pluralization\MessageBreak
     will be decided by the size of the index list}%
    {I will pretend you didn't use the * modifier.}%
  }{}%
```

We then typeset the name of the concept. We check pluralization and pass along the capitalization.

```
  \cnc@csvlistsize{#7}{\cnc@symbollistsize}%
  \ifnumcomp{\cnc@symbollistsize}{=}{1}{%
    \cnc@conceptname{\BooleanFalse}{#3}{\BooleanFalse}{#1}%
  }{%
    \cnc@conceptname{\BooleanTrue}{#3}{\BooleanFalse}{#1}%
```

```
  }%
```

Finally, we print the symbol list. We take away any whitespace at the end of the name (possible if the name is itself defined in terms of a public concept-name command that introduced an `\xspace`) and introduce a single space of our own.

```
  \unskip{} \cnc@conceptsymbols{#1}{#5}{#6}[#7]%
```

```
}
```

4.6 Public Macros

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

`\NewConcept` `{`⟨*concept key*⟩`}` `{`⟨*options*⟩`}`

We now define the `\NewConcept` command. It should (obviously) be robust.

```
\newrobustcmd*{\NewConcept}[2]{%
```

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

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

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

```
  {%
```

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

```
    \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.

```
    \DeclareRobustCommand*{\cnc@registertoggle}[1]{%
      \newtoggle{\cnc@d{concept(#1).option(##1).given}}%
      \togglefalse{\cnc@d{concept(#1).option(##1).given}}%
      \ltxkeys@newordkey[cnc@toggle]{#1}{##1}[]%
        {\toggletrue{\cnc@d{concept(#1).option(##1).given}}}%
    }%
    \cnc@registertoggle{name}%
    \cnc@registertoggle{Name}%
    \cnc@registertoggle{plural}%
    \cnc@registertoggle{Plural}%
    \cnc@registertoggle{namecmd}%
    \cnc@registertoggle{symbols}%
    \cnc@registertoggle{symbolcmd}%
    \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.

```
    \ltxkeys@newordkey[cnc]{#1}{name}%
      [#1]%
      {\csdef{\cnc@d{concept(#1).option(name)}}{##1}}%
    \ltxkeys@newordkey[cnc]{#1}{Name}%
      [\cnc@upper{\cnc@g{concept(#1).option(name)}}]%
      {\csdef{\cnc@d{concept(#1).option(Name)}}{##1}}%
    \ltxkeys@newordkey[cnc]{#1}{plural}%
      [\cnc@g{concept(#1).option(name)}\unskip s]%
      {\csdef{\cnc@d{concept(#1).option(plural)}}{##1}}%
    \ltxkeys@newordkey[cnc]{#1}{Plural}%
      [\cnc@upper{\cnc@g{concept(#1).option(plural)}}]%
      {\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.

```
    \expandaftercmds{\ltxkeys@newordkey[cnc]{#1}{namecmd}%
      []{\csname#1\endcsname}]{%
      \iftoggle{\cnc@d{concept(#1).option(namecmd).given}}{%
```

We register the option value as given.

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

⟨*namecmd*⟩ [*] [^] [*] [⟨*indices*⟩]

If the option is processed, we define the concept-specific name command. We distinguish between two cases: whether an index-list was provided or not.

```
        \NewDocumentCommand{##1}{t* t^ t* +o}{%
          \IfValueTF{####4}{%
            \cnc@nameandsymbols{#1}{####1}{####2}{####3}%
              {,}{$ and $}{####4}%
          }{%
            \cnc@conceptname{####1}{####2}{####3}{#1}\unskip\xspace%
          }%
        }%
      }{}%
    }%
```

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

```
    \csdef{\cnc@d{concept(#1).option(symbols).count}}{0}%
    \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.

```
      \ifcsundef{\cnc@d{symbol(\detokenize{##1}).concept}}{%
        \csedef{\cnc@d{concept(#1)%
                       .option(symbols)%
                       .count}}{\ltxkeys@listcount}%
        \csdef{\cnc@d{concept(#1)%
                      .option(symbols)%
                      .index(\ltxkeys@listcount)}}{##1}%
        \csedef{\cnc@d{symbol(\detokenize{##1}).concept}}{#1}%
      }%
```

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

```
      {%
        \PackageError{concepts}%
          {The symbol '\detokenize{##1}' is already allocated
           to the '\cnc@g{symbol(\detokenize{##1}).concept}'
           concept}%
          {I will pretend that you did not
           try to add this symbol.}%
      }%
```

```
    }%
```

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.

```
    \ltxkeys@newordkey[cnc]{#1}{symbolcmd}[]{%
      \ifblank{##1}{}{%
```

We register the option value as given.

```
        \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 or more 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. If a number was not detected, we pass control to the `\cnc@conceptsymbols` command, which is still able to grab a square bracket delimited list of indices.

```
        \newrobustcmd*{##1}{%
          \cnc@grabnumber{%
            \IfInteger{########1}{%
              \cnc@conceptsymbol{#1}{########1}%
            }{% TODO: check for square bracket (we still may want to report an error)
              \cnc@conceptsymbols{#1}{,}{,}%
            }%
          }%
        }%
```

```
      }%
    }%
```

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

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

`\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.

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

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` `[ * ]` `[ ^ ]` `[ * ]` `{`⟨*concept key*⟩`}`

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

```
\NewDocumentCommand{\ConceptName}{t* t^ t* m}{%
  \cnc@conceptname{#1}{#2}{#3}{#4}\unskip\xspace%
}
```

`\ConceptSymbol` `{`⟨*concept key*⟩`}` `[`⟨*index*⟩`]`

This implementation simply calls our private macro for retrieving the symbol with the given index. The index is optional and defaults to 1.

```
\NewDocumentCommand{\ConceptSymbol}{m O{1}}{%
  \cnc@conceptsymbol{#1}{#2}%
}
```

`\ConceptSymbols` `{`⟨*concept key*⟩`}` `[`⟨*separator*⟩`]` `[`⟨*last separator*⟩`]` `{`⟨*indices*⟩`}`

This implementation simply calls our private macro for retrieving the symbol list with the given indices. The index-list is mandatory but can be empty.

```
\NewDocumentCommand{\ConceptSymbols}{m +O{,} +o m}{%
    \IfValueTF{#3}{%
      \cnc@conceptsymbols{#1}{#2}{#3}[#4]%
    }{%
      \cnc@conceptsymbols{#1}{#2}{#2}[#4]%
    }%
}
```

`\ConceptNameAndSymbols` `{`⟨*concept key*⟩`}` `{`⟨*indices*⟩`}`

This implementation simply calls our private macro for typesetting the concept name and the symbol list with the given indices. The index-list is mandatory and cannot be empty.