The `easylist` package for numbered items

Paul Isambert
zappathustra@free.fr
http://paulisambert.free.fr

January 6, 2009

Abstract

This package is designed for typesetting lists of numbered items (like Wittgenstein's *Tractatus* or—more likely—the outline of a work yet to be done) with a single active character acting as the only command. Various options are available to achieve greater control over the general appearance of the list.

Contents

1 Choosing the symbol

Here's what the LaTeX code with `easylist` in its default usage looks like:

```
\begin{easylist}
§ First proposition.
§§ Interesting comment.
§§§ A note on the comment.
§§§ Another note.
§§§§ By the way...
§§§§§ This is a subsub...-proposition.
§ Let's start something new...
\end{easylist}
```

And it yields:

1. First proposition.
1.1. Interesting comment.
1.1.1. A note on the comment.
1.1.2. Another note.
1.1.2.1. By the way...
1.1.2.1.1. This is a subsub...-proposition.
2. Let's start something new...

(No, the pinky box *isn't* part of the package.)

Now, the section sign § might not be readily accessible on some (most?) keyboards (although it is accessible and useless on French ones). So you can choose another one instead by setting an option when calling the package (§ is default):

```
\usepackage[pilcrow]{easylist} to use ¶
\usepackage[at]{easylist} to use @
\usepackage[sharp]{easylist} to use #
\usepackage[ampersand]{easylist} to use &
```

The selected character is made active between `\begin{easylist}` and `\end{easylist}` and then returns to its initial value. So when using # for instance, just make sure to define new commands outside the `easylist`, or use the `\Activate` and `\Deactivate` commands in your list; as you might have guessed, they make the symbol respectively active and back to its original category. (Ending a list and then beginning a new one will not be noticeable in the final product, so you can interrupt your list to create a new command.) In what follows, I will be using §, but everything applies to other symbols too.[1]

2 Usage

First of all, you have to decide how many counters you need. By default, `easylist` creates 10 counters, but you might want more, so just specify a number as an option when calling the package: `\usepackage[50]{easylist}` will create 50 counters. Who knows? this might be useful. You can create as many counters as TeX allows you to. If you exceed its limit, it'll say it. You can also specify a number below 10, if you want less counters to be created.

The example above says it all, so let's repeat it:

[1] You might not be happy with the symbols and maybe you'd like to use another one, or simply have your favorite symbol as default to avoid remembering such a cumbersome name as 'pilcrow'. Here's a simple hack that does the job: select the entire code of the package, and replace all occurrences of § with your symbol. Make sure you won't use it in the list for other purposes, though.

```
\begin{easylist}
§ First proposition.
§§ Interesting comment.
§§§ A note on the comment.
§§§ Another note.
§§§§ By the way...
§§§§§ This is a subsub...-proposition.
§ Let's start something new...
\end{easylist}
```

Your list must be enclosed between `\begin{easylist}` and `\end{easylist}`, which is welcome since a character is made active and active characters are notoriously dangerous. As you might have guessed, one § creates a proposition of the first level, two §'s make a proposition of the second level, and so on down to the fifth level. If you concatenate more §'s than available, you'll get an error message telling you to ask for more and the problematic line will begin with !!! instead of numbers. Every sequence of §'s must terminate with a space, otherwise numbers won't be printed.

Two things are worth mentioning. First, note that § automatically creates a new line, so typing your propositions in a row will not affect the result. Hence

```
\begin{easylist} § First proposition.  §§ Interesting comment.  \end{easylist}
```

still yields

1. First proposition.
1.1. Interesting comment.

Second, when skipping one or more level(s), that(those) level(s) will be numbered 0 (except otherwise specified—see below), as in:

```
\begin{easylist}
§ First proposition.
§§§ A sub-comment to the first proposition.
\end{easylist}
```

which yields:

1. First proposition.
1.0.1. A sub-comment to the first proposition.

That's it. The following sections deal with labelling and referring and layout options, but for basic purpose you don't need more than that. Predefined style, available with `\begin{easylist}[<Style>]`, are treated in section 5.

3 Referring to an item

You can use LaTeX's `\label`, `\ref` and `\pageref` to refer to an item. The `\label` can appear anywhere in the text of the item (affecting only `\pageref`) but if you put it after the row of §'s, make sure there's a space in between, otherwise § won't fire, as mentionned above. The output of `\ref` is the counters of the item referred to with their original denotation and punctuation but not their original `Style` or `CtrCom` (see below), that is they adopt style where `\ref` is called. Only counters that have not been hidden will appear.

4 Options

Several parameters are available to achieve a finer control over the final output. The general command is `\ListProperties(key=value,key=value...)`, where `key` is a parameter. `ListProperties` affects all *subsequent* items and all subsequent lists, wherever it is issued. If you want to set the parameters back to

default, use \NewList, which can also have an argument (between parentheses like \ListProperties) and will then function as \ListProperties if you want to specify again a parameter. Note that you don't need to put all your parameters in the same \ListProperties: several \ListProperties(key=value...) have the same effect as a big \ListProperties with all parameters at once. By definition, this is not the case for \NewList—which, by the way, I'm using in all examples here, although I show \ListProperties just to make you believe in the illusion of a stand-alone list...

Parameter*n* affects the *n*th counter (e.g. 5 in 14.5.22 is the second counter) or the item of the *n*th level (e.g. 14.5.22 is a counter of the third level). I think there isn't much room for ambiguity, and at least the context will make clear what I'm talking about—and if you're still uncertain, well, just try the parameter!

If a value contains a comma or a closing parenthesis, you should enclose it between braces or easylist will take them to be delimiters. For instance, if you want a closing parenthesis to be FinalMark (see below), say \ListProperties(FinalMark={)}). And if you want to put a framebox anywhere, say for instance \ListProperties(Style2**={\framebox(5,5){}}).

4.1 Parameters affecting the *n*th counter in an item number

Start*n*
Start*n**

- Start*n*=<*Number*> makes the *n*th counter start at <*Number*>, if and only if it immediately precedes an item containing that counter, otherwise it is useless.
- Start*n**=<*Counter*> makes the *n*th counter dependent on <*Counter*>, where <*Counter*> is an external counter, like \thesection. The counter cannot be controlled anymore and stubbornly follows <*Counter*>. To make it back to normal, say Start*n**=NA, which is of course default.

```
\begin{easylist}
\ListProperties(Start1*=\thesection,Start2=17)
§ First proposition.
§§ Numbering doesn't work.
\ListProperties(Start2=17)
§§ This is better.
§ Hey, I can't move on!
§ I must be stuck to an external counter!
\ListProperties(Start1*=NA)
§ Okay, it works again.
\end{easylist}
```

4. First proposition.
4.1. Numbering doesn't work.
4.17. This is better.
4. Hey, I can't move on!
4. I must be stuck to an external counter!
5. Okay, it works again.

Mark
Mark*n*
FinalMark
FinalMark*n*

- Mark=<*Punctuation*> sets the punctuation of all counters to <*Punctuation*>.
- Mark*n*=<*Punctuation*> sets the punctuation of the *n*th counter to <*Punctuation*>.
- FinalMark=<*Punctation*> sets the punctuation of the final counter, if specified, for all items. To unspecify it without using \NewList, say FinalMark*n*=NA.
- FinalMark*n*=<*Punctuation*> sets the punctuation of the last (i.e. the *n*th) counter of the *n*th item. To unspecify it without using \NewList, say FinalMark*n*=NA.

The difference between Mark(*n*) and FinalMark(*n*) is that the latter supersedes the former at the end of the counters numbering an item. In case FinalMark(*n*) is set to NA, then Mark(*n*) is used in all positions. The default punctuation mark is a full stop, and FinalMark is unspecified (so all item numbers end with a full stop by default).

In the following example, note that = followed by nothing is understood as a normal value, albeit empty, and that specifying FinalMark3 *after* FinalMark makes the former win. If FinalMark3 had appeared *before* FinalMark, then its value would have been superseded. It works the same for all the parameters that follow.

There's a final caveat: if you want fixed spaces as marks, you shouldn't use LaTeX's \hspace but TeX's original \hskip. The former makes it all wrong, and I don't really know why (it doesn't stand being the replacement text of a macro defined with \edef or \xdef). You may not be used to \hskip, but it's very simple. Instead of \hspace{1cm}, for instance, say \hskip 1cm.

```
\ListProperties(Mark=.,FinalMark=,FinalMark3={})
```

1 The world is all that exists...
1.1 ... said Ludwig to Lizzie...
1.1.1) ... but she wasn't listening.

Numbers
Numbers*n*

- Numbers=*<Number denotation>* sets the way all counters are printed.
- Numbers*n*=*<Number denotation>* sets the way the *n*th counter is printed. Here are the admissible values:

`r` for lower case roman numerals;
`R` for upper case roman numerals;
`l` for lower case letters;
`L` for upper case letters;
`z` for Zapf's Dingbats;
`a` for arabic numbers, which are the default value.

Note that if you choose letters for a given counter, it should not exceed 26, or TEX will complain—and you'll have no number at all. Since I think that using letters with more than, say, 10 items, is ill-advised, I didn't overcome that limitation.

```
\ListProperties(Numbers2=R,Numbers3=L,Numbers4=r,Numbers5=l,Numbers6=z)
```

1. I...
1.I. ... really...
1.I.A. ... like...
1.I.A.i. ... numbers...
1.I.A.i.a. ... however...
1.I.A.i.a.∞. ... they are printed.

4.2 Parameters affecting numbers and items of the *n*th level

Hide
Hide*n*

- Hide=*<Number>* hides the first *<Number>* counters for all items. It isn't very useful unless you want all counters to be hidden for all items, that is you want lists of unnumbered items, in which case you may say `Hide=100` (or `Hide=10000` if you have more than 100 counters!).
- Hide*n*=*<Number>* hides the first *<Number>* counters for items of the *n*th level. This is more useful.

If *<Number>* > *n*, then nothing is printed. Note that if you refer to an item with hidden numbers, those will not appear when referring to the item with `\ref`. See the example below.

```
\ListProperties(Hide2=1,Hide3=2,Progressive*=.5cm,Numbers3=l,
Numbers4=r,FinalMark3={})
```

1. Now, what's the difference with `enumerate`? asks Leslie.
    1. Frankly, I don't know, Don answers.
        a) Should have made it proprietary.
            1.1.a.i. And suddenly all counters reappear... This is ugly! they both exclaim, but they cannot help noticing that referring to the previous item with `\ref` yields a).

Style
Style*n*
Style*
Style*n*
Style**
Style*n***

- Style=*<Format>* sets the style of counters and text for all items.
- Style*n*=*<Format>* sets the style of counters and text for items of the *n*th level.
- Style*=*<Format>* sets the style of all counters.
- Style*n**=*<Format>* sets the style of the counters of items belonging to the *n*th level.
- Style**=*<Format>* sets the style of all item texts.
- Style*n***=*<Format>* sets the style of the texts of items belonging to the *n*th level.

Since numbers and texts are enclosed in groups, *<Format>* can safely be a command such as `\bfseries` or `\itshape`, etc., or even `\color{green}` of the `xcolor` package. The no-star version interacts freely with the other two (up to (LA)TEX's font limitations).

Those parameters are actually placeholders in front of the counter or the text of the item (or both in case of `Style`), so you can use them to add nice symbols, for instance.

The default style is the style of your document.

```
\ListProperties(Style=\color{blue},Style1*=\bfseries,
Style2*=\itshape,Style1**=\scshape$\bullet$ ,
Style1**=$\diamond$ ,Style3**=$\Rightarrow$ ,Hide3=3)
```

1. • A FUNDAMENTAL PROPOSITION.
1.1. ◇ An essential comment.
⇒ First...
⇒ Then...
⇒ And we should not forget...

CtrCom
CtrComn

• CtrCom=*<Command>* makes *<Command>* into a command whose argument is the counters of all items.
• CtrComn=*<Command>* makes *<Command>* into a command whose argument is the counters of items of the *n*th level.

Note that *<Command>* can only be a command taking one token (i.e. possibly a group) as its last argument. Only the bare command must be issued when specifying the parameter. In the following example, `CtrCom=\colorbox{pink}` thus means `CtrCom=\colorbox{pink}{COUNTERS}`.

```
\ListProperties(CtrCom=\fbox,CtrCom3=\colorbox{pink})
```

1. A box is a box is a box...
1.1. ... said Gertie to Jim...
1.1.1. ... who was looking for Leopold.

Hang
Hangn

• Hang=*<Boolean>* lets all item texts hang from their counters if *<Boolean>* is `true`.
• Hangn=*<Boolean>* let all item texts of level *n* hang from their counters if *<Boolean>* is `true`.

If you don't want texts to hang, set *<Boolean>* to `false`, or `F`, or `RDNZL` or even `True` for that matter, i.e. anything but `true`. By default, no text hangs from its counter.

If you specify `Margin` or `Progressive` (see below), the counter will be at the specified distance and the text of the item will be at that distance plus the width of the counter. If you specify `Indent` (see below again), the first paragraph won't be indented.

```
\ListProperties(Margin1=1cm,Hang1=true,Indent1=.5cm)
```

1. I'm a nice item with a margin of 1cm for my counter and, well, 1cm plus something else for me.
   Here I have an indent because I'm a brave new paragraph.

```
\ListProperties(Hang1=false)
```

2. Hey! I'm not hanging anymore! I just have a 1cm margin! And I'm already indented!

Align
Alignn

• Align=*<keyword or dimension>* aligns all item texts at the same level for all levels.
• Alignn=*<keyword or dimension>* aligns all item texts of the *n*th level.

This needs an explanation. Item numbers don't have the same width, even if they belong to the same counter. For instance, the first item of the first level has `1` as its number, while the 33rd one of the same level has `33`, and obviously `33` is a wider expression than `1`. So the corresponding item texts won't start at the same horizontal distance, i.e. they wont be aligned. And if texts hang from their counters, the entire left margin will be unaligned, which might be pretty ugly.

This parameter is meant to avoid such a situation. If *<keyword>* is `move`, then the item number will be moved to the left if it is too large (as in the `enumerate` environment). If *<keyword>* is `fixed`, the `FinalSpace` (see below), i.e. the space between the item number and the item text, will be shrinked (or stretched) accordingly (as in a table of contents). If it is reduced to nothing and the counter is still too wide, then the counter will spread on the text and you'll get a warning; then you should increase `FinalSpace` so the *first* counter will be larger and the problematic one will have more room. In both cases, the reference width is the width of the first item counter (of the adequate level, of course) that `easylist` meets *after* the parameter is set. It is very likely to be `1`, thus a narrow counter. Finally, if you use a *<dimension>* instead of a keyword, it has the same effect as `fixed`, except that the reference width is not the width of the first counter of that level but the specified dimension. `FinalSpace` then becomes useless.

Set *<keyword>* to anything but `fixed` and `move` to turn off this parameter, which by default is not in use. Note that items belonging to different levels aren't aligned with one another.

```
\ListProperties(Hang=true,Margin=1cm,
FinalSpace=2em)
```

1. Here is the first item of my very uninteresting yet very convenient list.

1000. Here's the 1000th one which obviously isn't aligned with what precedes, although both are on the same level.

```
\ListProperties(Hang=true,Margin=1cm,
Align=fixed,FinalSpace=2em)
```

1. Here is the first item of my very uninteresting yet very convenient list.

1000. Here's the 1000th one which obviously *is* aligned with what precedes. Fortunately, `FinalSpace` was wide enough.

```
\ListProperties(Hang=true,Margin=1cm,
Align=move,FinalSpace=2em)
```

1. Here is the first item of my very uninteresting yet very convenient list.

1000. Here's the 1000th one which is also aligned with what precedes, but it would have run out of the box if `Margin` hadn't been taken care of.

```
\ListProperties(Hang=true,Margin=1cm,Align=2cm)
```

1. Hey, my counter is 2cm wide.

1000. Mine too.

A NOTE ON DIMENSIONS: the following parameters take *<Dimensions>* as values. However, those dimensions should be specified explicitely (e.g. `\ListProperties(Margin=2cm)`) or with a command (e.g. `\newcommand{\mydimension}{2cm}` and then `\ListProperties(Margin=\mydimension)`), but you should not directly use real TEX dimensions defined with `\newdimen` (or LATEX's `\newlength`), etc. In case you want such dimensions here, you have to prefix them with `\the`. So if you have defined a dimension `\mydimen`, `ListProperties(Margin=\the\mydimen)` is ok, but not `\ListProperties(Margin=\mydimen)`. This may sound paradoxical, and indeed it is, but `easylist` runs some tests on the parameters before TEX gets crazy (you want to avoid 10 error messages when you have 10 counters), and in order to test a dimension, it has to be readable, hence `\the`.

`Margin`
`Marginn`

- `Margin=`*`<Dimension>`* sets the distance from the left margin at which all items should start.
- `Marginn=`*`<Dimension>`* sets the distance from the left margin at which items of the nth level should start.

A negative value doesn't move the left margin to the left, but indents the right margin.

```
\ListProperties(Margin2=3ex,Margin3=6ex,Margin4=9ex,
Margin5=12ex)
```

1. First proposition.
   1.1. Interesting comment.
      1.1.1. A note on the comment.
      1.1.2. Another note.
      This is fascinating. We can start a new paragraph at any level and it remains where it should be.
         1.1.2.1. By the way...
            1.1.2.1.1. This is a subsub...-proposition.
2. Let's start something new...

`Progressive`
`Progressive*`

- `Progressive=`*`<Dimension>`* sets the margin of all items to a distance depending on their level. Namely, items of the nth level have a margin of $n \times$ *<Dimension>*.
- `Progressive*=`*`<Dimension>`* does the same, except that items of the nth level have a margin of $n \times$ *<Dimension>* $-$ *<Dimension>*. This means that items of the first level will be at the current margin and that the progressive indentation will start at the second item.

Thus the previous example could have been typeset with:

```
\ListProperties(Progressive*=3ex)
```
1. First proposition.
   1.1. Interesting comment.
      1.1.1. A note on the comment.
      1.1.2. Another note.
      This is fascinating. We can start a new paragraph at any level and it remains where it should be.
         1.1.2.1. By the way...
            1.1.2.1.1. This is a subsub...-proposition.
2. Let's start something new...

You can still specify `Margin` (*after* `Progressive`) if you want some items to be at a specified distance.

`Space` `Spacen` `Space*` `Spacen*`

- `Space=`*`<Dimension>`* sets the vertical space between items of a different level.
- `Spacen=`*`<Dimension>`* sets the vertical space between items of the nth level and previous items belonging to another level.
- `Space*=`*`<Dimension>`* sets the vertical space between items of the same level.
- `Spacen*=`*`<Dimension>`* sets the vertical space between items of the nth level and previous items belonging to the same level.

This space is added to the normal space between two lines. You can use a negative value if you want to compact your list.

```
\ListProperties(Space2=1cm,Space2*=.5cm)
```
1. Innocent item.

1.1. What a space! It must be `Space2`!

1.2. This one is less impressive. It must be `Space2*`.

`Indent` `Indentn`

- `Indent=`*`<Dimension>`* sets the indentation of all paragraphs.
- `Indentn=`*`<Dimension>`* sets the indentation of paragraphs belonging to items of the nth level.

```
\ListProperties(Indent2=2cm,Margin2=1cm)
```
1. Here is an unaffected item.
   1.1. Here is a paragraph. Its text is totally silly. But it exhibits a nice indentation.
   Here's a new paragraph, under the same number. Its text is still silly, but it still has a nice indentation too.

`FinalSpace` `FinalSpacen`

- `FinalSpace=`*`<Dimension>`* sets the distance between the item number (more precisely, the last mark) and the text for all items. Default is `.3em`.
- `FinalSpacen=`*`<Dimension>`* sets the distance between the item number (more precisely, the last mark) and the text for items of the nth level. This may not be very useful, except when adjusting it to meet the requirements of alignment.

In case you set the `Align` parameter to `fixed`, `FinalSpace` may be shrinked or stretched.

```
\ListProperties(FinalSpace2=1cm)
```
1. The following item wants to be seen.
1.1. Yes I do.

5 Predefined styles

You can load some predefined styles with `\begin{easylist}[`*`<Style>`*`]`. They simply issue `\NewList` with some special values. You can still add `\ListProperties` to modify them further, and you can still interrupt your list and start it again later, and you don't have to specify `\begin{easylist}[`*`<Style>`*`]`:

`\begin{easylist}` is enough, since it will inherit the properties of the previous one (if it hasn't been `\NewList`'ed, of course).

• `tractatus` is an imitation of Wittgenstein's *Tractatus Logico-Philosophicus*. It's quite simple: `Mark1` is turned to a dot, and all other marks are turned off.

1. The world is all that exists.
1.1 If something doesn't exist, it doesn't belong to the world.
1.11 Of something that doesn't exist, we should not speak.
2. I won't speak of things that don't exist.
2.1 Although they have some appeal to me.
2.10001 I really can't resist.

• `checklist` has no numbers, only check boxes, items are indented according to their levels, and the style of the first level is `\bfseries`.

Things to do

☐ **Finish the `easylist` package**
  ☐ Find new predefined styles
  ☐ Comment the code
    ☐ Avoid being verbose and making dumb jokes to hide weaknesses
☐ **Tidy up the room**

• `booktoc` roughly emulates a table of contents according to the `book` class (without page numbers, of course). Items of the first level are formatted like parts, and so on. Items of the seventh level and higher are not formatted at all.

I My part

1 My chapter
1.1 My wonderful section
1.1.1 My tremendous subsection
1.1.1.1 My very exciting subsubsection
1.1.1.1.1 My delicate paragraph
1.1.1.1.1.1 My secret subparagraph where everything is unveiled and which goes to the end of the line to show its hanging from the counter

2 My second chapter

• `articletoc` does the same for the `article` class. Items of the first level are sections, etc. Items of the fifth level and higher are not formatted.

1 A very interesting section
1.1 A subsection that will explain great theories
1.1.1 Here we find a proof
1.1.1.1 And a corollary here
1.1.1.1.1 This paragraph seems really cool

• `enumerate` mimicks the `enumerate` environment as defined in the LaTeX base and `article` class. Items of the fifth level and higher aren't formatted.

1. I have somehing to say...
   (a) But I don't know if I can...
      i. This is `enumerate` made easy...
         A. Oh yeah, you can spot some differences too, maybe in the separation between items...

(Two hours later)

200. Now that we've come thus far, let's study alignment of items in this emulated `enumerate` environment.

• `itemize` does the same for the `itemize` environment.

- Hey, this is a bullet.
  - And this is a dash.
    - Here we have an \ast (don't forget the $'s!).
      - And finally a \cdot, in case you didn't know.

6 Trouble with boxes

Active characters, as is well known, are nasty beings, and it's hard to keep them activated all the way down. That's why the `easylist` environment in its present state won't work in boxes. That is

```
\fbox{%
\begin{easylist}
§ First proposition.
\end{easylist}}
```

will print:

ğ First proposition.

where ğ is the normal value of § (in T1 font encoding). Lists in boxes aren't that common (except for this document), but such a limitation is always annoying. The solution is to make active the character you've chosen to create items just before the box and then turn it back to its initial value. For that purpose, `easylist` has the two commands `\Activate` and `\Deactivate`. So here's what you'll do:

```
\Activate
\fbox{%
\begin{easylist}
§ First proposition.
\end{easylist}}
\Deactivate
```

and you'll get:

1. First proposition.

Unfortunately, this has to be issued for every box. That is, the following code won't work (even if you surround your command definition with `\Activate` and `\Deactivate`—I told you that active characters were nasty):

```
\newcommand{\myfbox}[1]{\Activate\fbox{#1}\Deactivate}
\myfbox{%
\begin{easylist}
§ First proposition.
\end{easylist}}
```

Finally, for some boxes, as for instance the `\fbox` above, a `\NewList` should be issued at the beginning of the list. Don't ask me why. Instead, avoid `\fbox`es.

7 An example

Now, it so happened that French writer Jacques Roubaud published a book named **La Dissolution** just as I released the first version of this package (although the book was written in 2000). He uses MS word and has some troubles with the numbered items of which his text is made. I don't know if he knows TEX (he's a retired professional mathematician, so who knows?), but anyway here's an excerpt from the book as a tribute to that coincidence. It's in French.

```
\NewList(Style*=\footnotesize,Style3=\color{red},Style4=\color{blue!60!black})
\ListProperties(Style5=\color{green!60!black},Style6=\color{violet})
\ListProperties(Style7=\color{yellow!50!brown},Style8=\color{gray})
\ListProperties(Start1=70,Start2=6,Progressive*=2em,Mark= )
```

70 6 Après une heure environ de lecture de choses à lire et de lecture de paysage

70 6 1 m'amenant à découvrir que le Châlons traversé par le train ne se nommait plus, comme dans mon souvenir d'écolier, -sur Marne, mais -en Champagne, réforme onomastique récente qui m'avait jusqu'alors échappé, et avait vraisemblablement obligé les autorités municipales à un intense "lobbying"

70 6 1 1 il y aura bientôt un Villefranche-en-Beaujolais-nouveau, si Mâcon laisse faire

70 6 1 2 je vérifie, à tout hasard, que Reims n'est pas devenu Reims-en-Champagne

70 6 1 2 1 aucune ville X n'a choisi de se renommer "X en Champagne Pouilleuse"

70 6 1 2 1 1 laquelle a été baptisée, pendant que je ne regardais pas, "Champagne crayeuse"

70 6 1 2 1 1 1 rebaptême qui a vraisemblablement obligé les autorités de quelques groupements d'autorités municipales à un intense "lobbying"

70 6 1 2 1 1 1 1 les progrès du P.C. sont globaux comme l'économie du même adjectif

(© Éditions Nous)

8 Implementation

I'm not used to DocStrip so I input the code by hand with the `fancyvrb` package. Gaps in line numbering correspond to blank lines in the package. Besides, I'm pretty talkative, so I put the code in red. It will be easier to skip comments.

8.1 Declarations and options

After some commented-out information, here are the usual declarations:

```
21 \NeedsTeXFormat{LaTeX2e}
22 \ProvidesPackage{easylist}[2008/12/20 v.1.2 Numbered items with a single command.]
```

To process options, we already need conditionals and counts. The first four conditionals and the first count will be used in options, while the remaining conditional and counts are used in the following tests. We also define a convenient shorthand.

```
24 \makeatletter
25
26 \newif\ifPilcrow
27 \newif\ifAt
28 \newif\ifSharp
29 \newif\ifAmpersand
30 \newif\ifDubiousFigure
31
```

```
\newcount\el@CounterTotal
\el@CounterTotal10
\newcount\el@Scratch
\def\el@Advance#1{\advance#1 by 1\relax}
```

The `\el@NumberCheck` test is a basic token-by-token test that checks whether a sequence is made of numbers, which unfortunately have no category code of their own. The command takes the next character, compares it to 0, then 1, up to 9, etc. until it matches; if it doesn't the sequence is not a number and `\DubiousFiguretrue` is output.

First, `\el@NumberCheck` looks whether the next character is `?`, the terminator, in which case the control counter is set to 0, `\e@synext` (called at the end of the command to create recursion) is deactivated and the command comes to an end. So we can write `\el@NumberCheck 2563?` and `2563` will be tested.

```
\def\el@NumberCheck#1{%
\expandafter\if#1?%
  \el@Scratch0%
  \def\e@synext##1{\relax}%
```

Then, if the argument is not `?`, but the control counter has reached 10, then the argument is not a number (all figures have been exhausted). The counter is set to 0, the sequence is said doubtful and the rest of it is discarded:

```
\else%
  \ifnum\el@Scratch=10%
    \el@Scratch0%
    \def\e@synext##1?{\relax}%
    \DubiousFiguretrue%
```

If all figures have not been tried, the argument is compared to the present one, which is the value of the control counter. If it matches, the counter is reset and the command proceeds to the next character. If not, the counter is stepped and the command is called on the same argument:

```
  \else%
    \expandafter\if#1\the\el@Scratch%
      \el@Scratch0%
      \def\e@synext##1{\el@NumberCheck}%
    \else%
      \el@Advance\el@Scratch%
      \let\e@synext\el@NumberCheck%
    \fi%
  \fi
\fi\e@synext{#1}}
```

Now we can declare options. When it comes to choosing the symbol, they simply makes some conditional true. The conditional will be used at the end of the package, when we define the environment.

```
\DeclareOption{pilcrow}{\Pilcrowtrue}
\DeclareOption{at}{\Attrue}
\DeclareOption{sharp}{\Sharptrue}
\DeclareOption{ampersand}{\Ampersandtrue}
```

The remaining (undeclared) option is checked and if it's a valid number (if it passes the test above), then `\el@CounterTotal`, which tells the entire package how many counters, parameters, etc., must be created, is set to its value:

```
\DeclareOption*{%
\expandafter\el@NumberCheck\CurrentOption?%
\ifDubiousFigure%
  \PackageError{easylist}{%
  ^^J==> `\CurrentOption' is not a valid number (in package options).
  ^^J==> It is ignored and there are only 10 counters}{}%
\else%
  \el@CounterTotal\CurrentOption%
\fi\DubiousFigurefalse}
\ProcessOptions\relax
```

We need two more counters for the rest of the package (well, for now):

```
\newcount\el@ControlCounter
\el@ControlCounter1%
\newcount\el@CounterLevel
\el@CounterLevel1%
```

8.2 Basic recursive definitions

Since the number of counters depends on `\el@CounterTotal`, I can't create them in advance, so I designed recursive commands whose purpose is to create other commands in a given amount. The general trick is quite basic: simply use `\easynext` at the end of the command, which is `\let` to the command itself if the desired quantity (specified by `\el@CounterTotal`) has not been reached yet, and to `\relax` otherwise.

First, `\Generic@Counter` creates as many counters as needed. They are named `List1`, `List2`, `List3`, etc., and they are respectively the first, second, third, etc. counter of the final output. It's so simple that we don't need `\easynext` to create tail recursion. We simply `\expandafter` the command itself at the end of the conditional, so it jumps the `\fi`.

```
\def\el@GenericCounter{%
\ifnum\el@ControlCounter>\el@CounterTotal%
  \el@ControlCounter1%
\else%
  \newcounter{List\the\el@ControlCounter}%
  \el@Advance\el@ControlCounter%
  \expandafter\el@GenericCounter%
\fi}

\el@GenericCounter
```

`\Generic@Def` creates the commands that define parameters of the counters, that is `Margin`, `Style`, and so on, once again according to `\el@CounterTotal`.

Basically, `\Generic@Def[A]{B}{C}` will create commands `\B1A`, `\B2A`, `\B3A`, etc., with the definition `C`. The optional argument is needed to handle stars as in `\Start1*`, `\Style2*`, `\Style2**`, etc. (Yes, it would have been much simpler and somewhat more coherent to define and use `\Style**2` instead of `\Style2**`, for instance. But `\Style**2` does not please my eye.) Of course, `\Style2**` is not a valid TEX command, since it does not contain only category 11 characters (i.e. letters), but `\cscommand Style2**\endcsname` is and I use it throughout the package.

First, we check whether there still needs commands. If not, i.e. if the control counter is higher than the number of required levels, then the control counter is reset and the command does not reiterate:

```
\newcommand{\el@GenericDef}[3][]{%
\ifnum\el@ControlCounter>\el@CounterTotal%
  \def\easynext[##1]##2##3{\relax}%
  \el@ControlCounter1%
```

If there still needs commands, then one is created whose name depends on the value of the control counter. The `\def` is global (`\gdef`) because the `easylist` environment is made of concatenated groups; hence, if `\ListProperties` is called between `\begin{easylist}` and `\end{easylist}`, the parameters so defined would be stuck to their groups. Moreover, the `\gdef` is prefixed with `\expandafter`, otherwise it would try to define `\csname`, which would be a very bad idea. Globality and delay of `\def` are pervasive in this package.

```
\else%
  \expandafter\gdef\csname #2\the\el@ControlCounter#1\endcsname{#3}%
```

Once the command is created, the control counter is stepped and the command reiterates:

```
  \el@Advance\el@ControlCounter%
  \let\easynext\el@GenericDef%
\fi%
\easynext[#1]{#2}{#3}}
```

Now we create the default values of the parameters. These are important even if they're empty, because in what follows we'll look for invalid parameters as undefined commands. The command `\el@PreviousItem` is used for the `Space` parameter. It records the level of the previous item, and since the first list has no previous item, it is set to 0. `NA` is the value of a parameter which has to be discarded.

```
\def\el@PreviousItem{0}
\el@GenericDef{FinalMark}{NA}
\el@GenericDef{Mark}{.}
\el@GenericDef{Margin}{0cm}
\el@GenericDef{Numbers}{a}
\el@GenericDef{Style}{}
\el@GenericDef[*]{Style}{}
\el@GenericDef[**]{Style}{}
\el@GenericDef{Indent}{0cm}
\el@GenericDef{Start}{NA}
\el@GenericDef[*]{Start}{NA}
\el@GenericDef{CtrCom}{}
\el@GenericDef{Space}{0cm}
\el@GenericDef[*]{Space}{0cm}
\el@GenericDef{Hide}{0}
\el@GenericDef{Hang}{false}
\el@GenericDef{FinalSpace}{.3em}
\el@GenericDef{Align}{false}
```

The `Progressive(*)` parameter is different, since it creates a different definition for each command (namely the `Margin` command), so it can't use `\el@GenericDef`. It sets the `Margin` of each item to $n \times$ *<Dimension>* where *<Dimension>* is the argument of the command and n is the level of the item. The starred version simply substracts one argument (which is a dimension) to each `Margin`, which is thus set to $n \times$ *<Dimension>* $-$ *<Dimension>*, and so items of the first level stay at the current margin.

First we need a new conditional and a new dimension:

```
\newif\ifProgressiveStar
\newdimen\el@ProgressiveDimension
```

`Progressive` calls `\el@ProgressiveMargin` and `Progressive*` calls `\el@ProgressiveMargin*`. It's the same command, but with a different value for the conditional.

```
\def\el@ProgressiveMargin{%
\@ifstar%
  {\ProgressiveStartrue\el@ProgressiveM@rgin}%
  {\ProgressiveStarfalse\el@ProgressiveM@rgin}}
```

`\el@ProgressiveM@rgin` does the real job. First, as usual, it checks whether there are enough commands, in which case it resets and stops:

```
\def\el@ProgressiveM@rgin#1{%
\ifnum\el@ControlCounter>\el@CounterTotal%
  \def\easynext##1{\relax}%
  \el@ControlCounter1%
```

If a command must be created, it sets the dimension to the value of the argument and multiply it by the level of that command:

```
\else%
  \el@ProgressiveDimension#1%
  \multiply\el@ProgressiveDimension by \el@ControlCounter%
```

Then, if in the starred version, it substracts one argument to the dimension:

```
  \ifProgressiveStar%
    \advance\el@ProgressiveDimension by -#1%
  \fi%
```

The `Margin` of the level is set to the value of the dimension. The definition is global for the same reason as above and immediate (`\xdef`) because the value of the dimension is changed at each iteration of `\el@ProgressiveM@rgin`, so we want to store it beforehand. Finally, the dimension is stringed with `\the` because we want to be able to test it (dimensions are unanalyzable tokens). Actually, it has already been tested when the user called the `Progressive` parameter. But since it becomes the value of the `Margin`s, and since these are tested after each `\ListProperties`, we want it to be so.

```
  \expandafter\xdef\csname Margin\the\el@ControlCounter\endcsname{%
    \the\el@ProgressiveDimension}%
```

Finally, we step the control counter and reiterate:

```
  \el@Advance\el@ControlCounter%
  \let\easynext\el@ProgressiveM@rgin%
\fi%
\easynext{#1}}
```

8.3 Parameters

In the first (nonrecursive) version of this package, I used the `keyval` package to set the parameters. But I was not able to use `keyval` pairs into recursive definitions, so I had to create such pairs myself. This proved rather easy, albeit fastidious and ugly—because the package has to check parameters to process them. I'm nonetheless very happy with that because I was able to taylor more precise tests.

`\ListProperties(Arg)` calls `\el@ListProperties` on '`A=A,Arg,Z=Z,`' with '`A=A`' to avoid troubles in case of an empty `\ListProperties()` and '`Z=Z`' as a terminator. Once parameters have been set, it runs some tests on the testable arguments, described below. The values of `Style` or `Mark` cannot be tested, of course, because they're so diverse.

```
\def\ListProperties(#1){%
\el@ListProperties A=A,#1,Z=Z,%
\el@GenericNumberCheck{Hide}%
\el@GenericNumberCheck{Start}%
\el@GenericNumberCheck[*]{Start}%
\el@GenericLetterCheck%
\el@GenericUnitSearch{Margin}%
\el@GenericUnitSearch{Indent}%
\el@GenericUnitSearch{Space}%
\el@GenericUnitSearch[*]{Space}%
\el@GenericUnitSearch{FinalSpace}}
```

`\NewList` calls `\el@NewList` and then `\ListProperties` if the next character is an opening parenthesis.

```
\def\NewList{%
\@ifnextchar(%
  {\el@NewList\ListProperties}%
  {\el@NewList}}
```

`\el@NewList` simply sets all parameters back to default and reset all counters. `\el@ResetCounters` (which needs setting `\el@ControlCounter` to `0`) is described in section 8.6. Don't worry for `\el@ControlCounter` set to `0`, the tests below bring it back to `1`, its default value.

```
\def\el@NewList{%
\el@ControlCounter0%
\el@ResetCounters%
\gdef\el@PreviousItem{0}%
\el@GenericDef{FinalSpace}{.3em}
\el@GenericDef{FinalMark}{NA}%
\el@GenericDef{Mark}{.}%
\el@GenericDef{Margin}{0cm}%
\el@GenericDef{Numbers}{a}%
\el@GenericDef{Style}{}%
\el@GenericDef[*]{Style}{}%
\el@GenericDef[**]{Style}{}%
\el@GenericDef{Indent}{0cm}%
\el@GenericDef{Start}{NA}%
\el@GenericDef[*]{Start}{NA}%
\el@GenericDef{CtrCom}{}%
\el@GenericDef{Space}{0cm}
\el@GenericDef[*]{Space}{0cm}%
\el@GenericDef{Hide}{0}%
\el@GenericDef{Hang}{false}%
\el@GenericDef{Align}{false}}
```

Finally, we need keywords to identify parameters without numbers, like `Style`, and some values:

```
\def\el@MarginTest{Margin}
\def\el@MarkTest{Mark}
\def\el@FinalMarkTest{FinalMark}
\def\el@NumbersTest{Numbers}
\def\el@IndentTest{Indent}
\def\el@StyleTest{Style}
\def\el@CtrStyleTest{Style*}
\def\el@ParStyleTest{Style**}
\def\el@CounterCommandTest{CtrCom}
\def\el@ProgressiveTest{Progressive}
\def\el@ProgressiveStarTest{Progressive*}
\def\el@StartTest{Start}
\def\el@StartStarTest{Start*}
\def\el@SpaceTest{Space}
\def\el@SpaceStarTest{Space*}
\def\el@HideTest{Hide}
\def\el@HangTest{Hang}
\def\el@FinalSpaceTest{FinalSpace}
\def\el@AlignTest{Align}
\def\el@True{true}
\def\el@False{false}
\def\el@Fixed{fixed}
\def\el@AlreadyFixed{alreadyfixed}
\def\el@Move{move}
\def\el@AlreadyMoved{alreadymoved}
```

Here are some shorthands. `^^J` is the character creating a new line and `==>` is simply a useless decoration.

```
\newcommand{\el@Error}[4][]{%
\PackageError{easylist}{^^J==> `#3' is not a valid #4 (#2=#3). It is ignored#1}{}}
\def\el@DimenError#1#2{%
\el@Error[.^^J==> Note that true TeX dimensions should be prefixed with%
^^J==> \string\the\space in \string\ListProperties]{#1}{#2}{dimension}}
```

And here comes the ugly definition. Most of the conditionals aren't indented because they amount to exclusive cases, and it's ugly enough. The command takes two parameters delimited by `=` and a comma at the end (i.e. `Parameter=Value,`), does a lot of testing and if everything is ok assigns `Value` to `\Parameter`. Roughly speaking.

First we reset all conditionals and names issued by tests (see below):

```
\def\el@ListProperties#1=#2,{%
\DubiousFigurefalse%
\DubiousLetterfalse%
\DubiousNumberfalse%
\DubiousParameterfalse%
\Pointfalse%
\Signfalse%
\def\el@Parameter{}%
\def\el@ParameterNumber{}%
```

Then we set the default value of tail recursion and store the names of the parameter and the value:

```
\let\easynext@Properties\el@ListProperties%
\def\el@TempParameter{#1}%
\def\el@TempValue{#2}%
```

If the parameter is equal to `Z`, we redefine tail recursion as `\relax` so the process comes to an end. If the parameter is `A`, we simply go on, since `A` is just a placeholder.

```
\if#1Z%
  \let\easynext@Properties\relax%
\else\if#1A%
```

Now we use the keywords to catch `Margin`, `Style`, etc., which are not treated like `Margin2`, `Style5`, etc. If the parameter is `Margin`, for instance, we check whether the value is a correct dimension (tests are described in the next section), and in case it is, we launch `\el@GenericDef{Margin}{#2}`, which defines `\Margin1`, `\Margin2`, etc., with the value as their definition.

```
\else\ifx\el@TempParameter\el@MarginTest%
  \expandafter\el@UnitSearch#2?
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@GenericDef{Margin}{#2}%
  \fi%
```

`Progressive(*)`, `Indent`, `Space(*)` and `FinalSpace` behave like `Margin` and test dimensions:

```
\else\ifx\el@TempParameter\el@ProgressiveTest%
  \expandafter\el@UnitSearch#2?%
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@ProgressiveMargin{#2}%
  \fi%
\else\ifx\el@TempParameter\el@ProgressiveStarTest%
  \expandafter\el@UnitSearch#2?%
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@ProgressiveMargin*{#2}%
  \fi%
\else\ifx\el@TempParameter\el@IndentTest%
  \expandafter\el@UnitSearch#2?%
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@GenericDef{Indent}{#2}%
  \fi%
\else\ifx\el@TempParameter\el@SpaceTest%
  \expandafter\el@UnitSearch#2?%
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@GenericDef{Space}{#2}%
  \fi%
\else\ifx\el@TempParameter\el@SpaceStarTest%
  \expandafter\el@UnitSearch#2?%
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@GenericDef[*]{Space}{#2}%
  \fi%
\else\ifx\el@TempParameter\el@FinalSpaceTest%
  \expandafter\el@UnitSearch#2?%
  \ifDubiousFigure%
    \el@DimenError{#1}{#2}%
  \else%
    \el@GenericDef{FinalSpace}{#2}%
  \fi%
```

`Hide` tests whether its argument is a number. (If you don't understand `\el@Error`, see above, where it's been defined.)

```
\else\ifx\el@TempParameter\el@HideTest%
  \expandafter\el@NumberCheck#2?%
  \ifDubiousFigure%
```

```
    \el@Error{#1}{#2}{number}%
  \else%
    \el@GenericDef{Hide}{#2}%
  \fi%
```

For `Numbers`, we check that the value is a correct number denotation, namely `a`, `r`, `R`, `l`, `L` and `z` (which will be turned into commands later on).

```
\else\ifx\el@TempParameter\el@NumbersTest%
  \el@LetterCheck{#2}%
  \ifDubiousLetter%
    \el@Error%
      {#1}{#2}{number denotation}%
  \else%
    \el@GenericDef{Numbers}{#2}%
  \fi%
```

For `Align` and `Hang`, we test whether the value is some keyword, and in the case of `Align`, if it is not a keyword, we test whether it's a dimension.

```
\else\ifx\el@TempParameter\el@AlignTest%
  \ifx\el@TempValue\el@Fixed%
    \el@GenericDef{Align}{fixed}%
    \else\ifx\el@TempValue\el@Move%
      \el@GenericDef{Align}{move}%
    \else\ifx\el@TempValue\el@False%
    \else%
      \expandafter\el@UnitSearch\el@TempValue?%
      \ifDubiousFigure%
        \el@Error%
          [.^^J==> Admissible values are ‘false’, ‘fixed’, ‘move’ or a dimension]%
          {#1}{#2}{value for ‘Align’}%
        \DubiousFigurefalse%
      \else%
        \el@GenericDef{Align}{#2}%
      \fi%
    \fi\fi\fi%
\else\ifx\el@TempParameter\el@HangTest%
  \ifx\el@TempValue\el@True%
    \el@GenericDef{Hang}{true}%
  \else\ifx\el@TempValue\el@False%
      \el@GenericDef{Hang}{false}%
  \else%
  \el@Error%
    [.^^J==> Admissible values are ‘true’ or ‘false’]%
    {#1}{#2}{value for ‘Hang’}%
  \fi\fi%
```

`Start(*)` is not allowed without specifying a number, so an error is issued if it appears without one:

```
\else\ifx\el@TempParameter\el@StartTest%
  \PackageError{easylist}%
    {^^J==> ‘Start’ can’t be used without a number, so it is ignored}{}%
\else\ifx\el@TempParameter\el@StartStarTest%
  \PackageError{easylist}%
    {^^J==> ‘Start*’ can’t be used without a number, so it is ignored}{}%
```

Finally, with `(Final)Mark`, `Style(*(*))`, and `CtrCommand`, there's not much we can test, so we just launch a generic definition and hope.

```
\else\ifx\el@TempParameter\el@MarkTest%
  \el@GenericDef{Mark}{#2}%
\else\ifx\el@TempParameter\el@FinalMarkTest%
  \el@GenericDef{FinalMark}{#2}%
\else\ifx\el@TempParameter\el@StyleTest%
```

```
  \el@GenericDef{Style}{#2}%
\else\ifx\el@TempParameter\el@CtrStyleTest%
  \el@GenericDef[*]{Style}{#2}%
\else\ifx\el@TempParameter\el@ParStyleTest%
  \el@GenericDef[**]{Style}{#2}%
\else\ifx\el@TempParameter\el@CounterCommandTest%
  \el@GenericDef{CtrCom}{#2}%
```

Now, if the parameter is not a keyword, it is either something like `Style2` or an error from the user. The former is treated below. In the latter case, we put the parameter to the test and issue some error message accordingly: whether it has a wrong name, or a number higher than the amount of counters asked for, or both, or none. We can do that because we define and use parameters as commands with `\csname Parameter\endcsname`. Using `\csname Command\endcsname` is equivalent to `\relax` if no such command has been defined (unlike `\Command` which yields an error message). That's the reason why all parameters have to be defined, albeit sometimes vacuously, as have been done above.

So we see whether `\csname Parameter\endcsname` is equal to `\relax`, and if it is, we test it to see what's wrong:

```
\else\expandafter\ifx\csname #1\endcsname\relax%
  \el@DubiousParameter#1?%
```

If it has a wrong name and a wrong number, we say:

```
  \ifDubiousParameter%
    \ifDubiousNumber%
      \PackageError{easylist}{^^J==> `#1' is not a valid parameter. It is ignored.%
      ^^J==> Besides, you don't have \el@ParameterNumber\space counters}{}%
```

If only the name is wrong:

```
    \else%
      \PackageError{easylist}{^^J==> `#1' is not a valid parameter. It is ignored}{}%
    \fi%
```

If only the number is wrong:

```
  \else%
    \ifDubiousNumber%
      \PackageError{easylist}{^^J==> You don't have \el@ParameterNumber\space%
      counters, so `#1' is ignored.%
      ^^J==> Ask for more of them}{}%
```

And finally if none are wrong, as in the case of `Style**2`, which the test cannot detect:

```
    \else%
      \PackageError{easylist}{^^J==> Something is wrong with `#1' but I don't know what.%
      ^^J==> Maybe you put stars before numbers or you specified a number%
      ^^J==> to Progressive. Anyway, it is ignored}{}%
    \fi%
  \fi%
```

If nothing is wrong, we simply define the parameter as a command with the value as its definition:

```
\else%
  \expandafter\gdef\csname #1\endcsname{#2}%
```

And we uglily close that ugly definition and call `\el@CommaKiller`:

```
\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi%
\el@CommaKiller}
```

`\el@CommaKiller` simply gobbles stray commas to avoid trouble with `\ListProperties(Hide=5,)`, or `\ListProperties(Hide=5,,,,,,,,,Margin=2cm)` for that matter. If the next character is a comma, it calls `\el@Comm@Killer`, which takes an argument, namely the comma, and does nothing with it but calls `\el@CommaKiller` again, thus proceeding to the next character. If the latter is not a comma, `\easynext@Properties` is called, which has been defined as `\el@ListProperties` or as `\relax`, depending on remaining arguments in `\ListProperties`.

```
\def\el@Comm@Killer#1{\el@CommaKiller}
\def\el@CommaKiller{\@ifnextchar,{\el@Comm@Killer}{\easynext@Properties}}
```

8.4 Parametric tests

Here are the tests used on parameter names and values in `\ListProperties` and `\el@ListProperties` (hence the bad pun that titles this section). One of them, namely `\el@NumberCheck`, has been defined at the beginning of the package because we needed it to test options. First, we need some conditionals (`\ifDubiousFigure` was defined with `\el@NumberCheck`) and temporary names:

```
\newif\ifDubiousLetter
\newif\ifDubiousParameter
\newif\ifDubiousNumber
\newif\ifPoint
\newif\ifSign

\def\el@Parameter{}
\def\el@ParameterNumber{}
\def\el@Void{}
```

First, `\el@DubiousParameter` is called when `\el@ListProperties` finds an undefined parameter. I use it simply to give more detailed error messages; I could have let `easylist` always prompt the same error message, but using packages with this kind of parameters, I sometime get lost with their names. So I think details are welcome in error messages, even if it's not much. Anyway, `\el@DubiousParameter` scans strings token by token until it meets a `?`. A parameter name is made of letters, numbers, and stars. This test looks for them and lumps them together. For instance, when it meets a letter, it stores it in `\el@Parameter`, which is defined as `\el@P@rameter` plus the letter, where `\el@P@rameter` has been `\let` to `\el@Parameter` beforehand (watch the `@`'s!), i.e. with its previous value.

So first, we `\let` temporary names and numbers. In case this is the first iteration, they are just empty. We also reset a conditional.

```
\def\el@DubiousParameter#1{%
\let\el@P@rameter\el@Parameter%
\let\el@P@rameterNumber\el@ParameterNumber%
\DubiousFigurefalse%
```

In case we meet the terminator, we call the two tests that'll try the parameter and its number (described below). Otherwise, we define `\easynext` to be the command itself, so it will iterate until the question mark:

```
\if#1?%
  \def\easynext{\el@ParameterNumberTest\el@ParameterTest}%
\else%
  \let\easynext\el@DubiousParameter%
```

Now we look for letters, i.e. characters of category code 11. If we find one, we store it in `\el@Parameter`:

```
  \ifcat#1a%
    \edef\el@Parameter{\el@P@rameter#1}%
```

If the character is not a letter, we already test what we have collected as letters. We don't wait for the end of the sequence, otherwise `Sty7le`, for instance, would be analyzed as `Style` and `7` and we'd be forced to say that it is wrong but we don't know why. So it just makes error messages a little more efficient: `Sty` is analyzed and rejected, and then when `Sty` and `le` are concatenated, although they will pass the test, the wrong analysis cannot be undone.

```
  \else%
    \el@ParameterTest%
```

Now we look for stars and collect them into the parameter name, just like letters:

```
    \if#1*
      \edef\el@Parameter{\el@P@rameter#1}%
```

Finally, we analyse everyhing else, i.e. characters of category code 12 (or worse!) except stars. When we find such a character, we test whether it's a number:

```
    \else%
      \el@NumberCheck#1?
```

If it's not, we can already say the parameter name is wrong, since it should contain only letters, stars, and numbers.

```
      \ifDubiousFigure%
        \DubiousParametertrue%
```

If the character is a number, we collect it in `\el@ParameterNumber`, to be tested later.

```
      \else%
        \edef\el@ParameterNumber{\el@P@rameterNumber#1}%
```

And we close and call `\easynext`:

```
      \fi%
    \fi%
  \fi%
\fi\easynext}
```

Now we test what we have gathered. The following two tests output values to conditionals, which are then used at the end of `\el@ListProperties` (see above), namely whether the parameter has a bad name or a bad number or both or none.

The name of the parameter is simply compared to keywords with `\el@ParameterTest`, and if it matches with none, we fire the proper conditional. If the name matches `Progressive(*)`, we turn down bad numbers, otherwise `easylist` will see 'good name, bad number', and say 'you don't have so many counters', while actually the problem is that this parameter don't take a number. So `easylist` will see 'good name, good number', say 'something's wrong but I don't know what' and politely suggest that maybe you put a number to `Progressive`. (Sorry for the unindented conditionals.)

```
\def\el@ParameterTest{%
\ifx\el@Parameter\el@MarginTest%
\else\ifx\el@Parameter\el@MarkTest%
\else\ifx\el@Parameter\el@FinalMarkTest%
\else\ifx\el@Parameter\el@NumbersTest%
\else\ifx\el@Parameter\el@IndentTest%
\else\ifx\el@Parameter\el@StyleTest%
\else\ifx\el@Parameter\el@CtrStyleTest%
\else\ifx\el@Parameter\el@ParStyleTest%
\else\ifx\el@Parameter\el@CounterCommandTest%
\else\ifx\el@Parameter\el@ProgressiveTest%
  \DubiousNumberfalse%
\else\ifx\el@Parameter\el@ProgressiveStarTest%
  \DubiousNumberfalse%
\else\ifx\el@Parameter\el@StartTest%
\else\ifx\el@Parameter\el@StartStarTest%
\else\ifx\el@Parameter\el@HideTest%
\else\ifx\el@Parameter\el@SpaceTest%
\else\ifx\el@Parameter\el@SpaceStarTest%
\else\ifx\el@Parameter\el@HangTest%
\else\ifx\el@Parameter\el@FinalSpaceTest%
\else\ifx\el@Parameter\el@AlignTest%
\else\DubiousParametertrue%
\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi}
```

Now we test numbers. If there is none, everything's ok, but if there's one and it's larger than the total number of counters (stored in `el@CounterTotal`), then the conditional is launched.

```
\def\el@ParameterNumberTest{%
\ifx\el@ParameterNumber\el@Void%
\else%
  \ifnum\el@ParameterNumber>\el@CounterTotal%
    \DubiousNumbertrue%
  \fi%
\fi}
```

8.5 Non-parametric tests

Now we test the values of the parameters (hence, once again, the bad title) before TEX gets crazy. Indeed, if the user say, for instance, `Margin=2`, we want to disable `2` (which is not a dimension) before TEX tries to put it where it belongs and starts complaining that there lacks a legal unit of measure. Those tests are used in `\ListProperties` and `\el@ListProperties` above.

8.5.1 Dimensions

So, first, we devise a test for dimensions. It's a token-by-token test, and that's the reason why dimensions defined with TEX's `\newdimen` or LATEX's `\newlength` should be prefixed with `\the` (they're unreadable otherwise). A dimension is made of at most one plus or minus sign at the beginning, numbers separated by at most one dot or comma for decimals, and two letters which form the unit's name. That's the basis on which we can build our test.

First, we define default tail:

```
\def\el@UnitSearch#1{%
\let\easynext\el@UnitSearch%
```

Then we look for the plus or minus sign. It's ok if we have never encountered one before, in which case the dimension is wrong and we let the test gobble the rest of the sequence.

```
\if#1-%
  \ifSign%
    \DubiousFiguretrue%
    \def\easynext##1?{\relax}%
  \else%
    \Signtrue%
  \fi%
\else%
  \if#1+%
    \ifSign%
      \DubiousFiguretrue%
      \def\easynext##1?{\relax}%
    \else%
      \Signtrue%
    \fi%
```

If the character is not a plus minus sign, then none can appear later: the signs are always at the beginning of a dimension. Thus, as soon as we meet a character which is not a plus or minus sign, we do as if we had encountered one, i.e. with set the corresponding conditional to true.

```
  \else%
  \Signtrue%
```

Then, if we meet the terminator, it means that the dimension is wrong, since dimensions end with letters (the unit) and if letters are met, another test is launched which goes to the end of the sequence (and the terminator). So `\el@UnitSearch` should never see `?` with a valid dimension. (We use `\DubiousFiguretrue`, although we're testing dimensions, because that single conditional is enough. We don't need one for numbers, one for units, etc. As long as something is wrong, the dimension is wrong, and I think the user can find out what, contrary to the parametric tests above that were more detailed.)

```
    \if#1?%
      \DubiousFiguretrue%
      \let\easynext\relax%
```

Next, if we meet a point or a comma, it's ok unless we already met once, in which case the dimension is wrong, and we simply gobble the rest of the sequence:

```
    \else%
      \if#1.%
        \ifPoint%
          \DubiousFiguretrue%
          \def\easynext##1?{\relax}%
        \else%
          \Pointtrue%
```

```
463         \fi%
464       \else%
465         \if#1,%
466           \ifPoint
467             \DubiousFiguretrue%
468             \def\easynext##1?{\relax}%
469           \else%
470             \Pointtrue%
471           \fi%
```

If we find a letter, we quit `\el@UnitSearch` for another test.

```
472         \else%
473           \ifcat#1a%
474             \def\easynext{\el@UnitCheck#1}%
```

Now, if the character is none of the above, then its category code is 12. So we test whether it's a number:

```
475           \else%
476             \el@NumberCheck#1?%
```

If it's not, that doesn't mean that the dimension is wrong. Indeed, if it is a TEX or LATEX dimension as defined above, then its prefixing with `\the` turned it into a readable sequence of characters, but they all have category code 12, even letters. So the unit (which is always `pt`) can't be detected by `\ifcat#1a` above, for the `p` is not of the same category anymore. So we run a special test:

```
477             \ifDubiousFigure%
478               \def\easynext{\el@DimenUnitCheck#1}%
```

And we close. The rest is up to the tests that have been called.

```
479             \fi%
480           \fi%
481         \fi%
482       \fi%
483     \fi%
484   \fi%
485 \fi\easynext}
```

Before testing units, we need keywords, namely their names. I hope I didn't miss one[2]:

```
487 \def\el@Em{em}%
488 \def\el@Ex{ex}%
489 \def\el@Centimetre{cm}%
490 \def\el@Millimetre{mm}%
491 \def\el@Inch{in}%
492 \def\el@Pica{pc}%
493 \def\el@Point{pt}
494 \def\el@Didot{dd}%
495 \def\el@Cicero{cc}%
496 \def\el@BigPoint{bp}%
497 \def\el@ScaledPoint{sp}%
```

Now we simply compare what we've found to those keywords. `\el@UnitCheck` takes the rest of the sequence (from the first letter found with `\el@UnitSearch`) as its argument. In case nothing matches, then the dimension is wrong (when fed with two commands, `\ifx` tests whether they have the same expansion).

```
499 \def\el@UnitCheck#1?{%
500 \def\el@TempUnit{#1}%
501 \ifx\el@TempUnit\el@Em%
502 \else\ifx\el@TempUnit\el@Ex%
```

[2] Actually, I missed those same dimensions prefixed with `true` (see the *TEXbook*, chapter 10), but since there might be any number of spaces, including zero, between `true` and the unit, it's hard to test. And I think `true` is hardly used—and one can still define the appropriate dimension beforehand. What, me, reluctant?

```
\else\ifx\el@TempUnit\el@Centimetre%
\else\ifx\el@TempUnit\el@Millimetre%
\else\ifx\el@TempUnit\el@Inch%
\else\ifx\el@TempUnit\el@Pica%
\else\ifx\el@TempUnit\el@Point%
\else\ifx\el@TempUnit\el@Didot%
\else\ifx\el@TempUnit\el@Cicero%
\else\ifx\el@TempUnit\el@BigPoint%
\else\ifx\el@TempUnit\el@ScaledPoint%
\else\DubiousFiguretrue\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi}
```

Here's the little test for the case letters have turned to category 12. In that case, the unit is `pt` (dimensions always use `pt`, even if you define them with another unit). The conditional `\if` compares character codes, not category codes, so it works fine here. If the first character is `p` and the second is `t`, then we have a valid unit and we can turn off `\ifDubiousFigure` (which was turned on by `\el@NumberCheck` in `\el@UnitSearch` above).

```
\def\el@DimenUnitCheck#1#2?{%
\if#1p%
  \if#2t%
    \DubiousFigurefalse%
  \fi%
\fi}
```

Now, those tests work fine when we have something like `Margin=2cm`. `\el@ListProperties` identifies the keyword and runs the test on `2cm` accordingly. But if we have `Margin3=2cm`, we can't do that: `Margin3` is not identified, we simply know that it's been already defined and thus it's a valid parameter. But apart from that, we don't know anything—actually, `\el@ListProperties` doesn't even know it's a `Margin`; it just defines it because its name is ok. So we have to test those parameters after they've been redefined by `\el@ListProperties`. However, we don't know which parameters have been specified, so we must test them all. But we don't know in advance how many of them there are: it depends on the number of counters that the user has asked for. So we need a recursive test, just like we had a recursive definition. That's the meaning of all those `\el@Generic...` tests appended at the end of `\ListProperties` above.

Here's the generic version of `\el@UnitSearch`. First, we do some cleaning:

```
\newcommand{\el@GenericUnitSearch}[2][]{%
\Pointfalse%
\Signfalse%
```

Then, if the control counter is higher than `\el@CounterTotal`, then we have tested all parameters and we stop:

```
\ifnum\el@ControlCounter>\el@CounterTotal%
  \el@ControlCounter1%
  \def\easynext[##1]##2{\relax}%
```

Otherwise, we take the value of the parameter of the level we're in (according to the control counter). The name of the parameter is made out of the two arguments, the first of which is optional since it is used for stars. So `\el@GenericUnitSearch[*]{Space}` will test `Space1*`, `Space2*`, etc.:

```
\else%
  \edef\el@TempTestable{\csname #2\the\el@ControlCounter#1\endcsname}%
```

Then we launch `\el@UnitSearch`. In case we have a invalid dimension, we issue an error message and redefine the command to its default value (unfortunately not its previous value in case the user had already specified one), which is `.3em` for `FinalSpace` and `0cm` for anything else.

```
  \expandafter\el@UnitSearch\el@TempTestable?%
  \ifDubiousFigure%
    \def\el@Name{#2}%
    \ifx\el@Name\el@FinalSpace%
      \el@Error[^^J==> and #2\the\el@ControlCounter#1\space is set to .3em.%
      ^^J==> Note that true TeX dimensions should be prefixed with%
      ^^J==> \string\the\space in \string\ListProperties]%
      {#2\the\el@ControlCounter#1}{\el@TempTestable}{dimension}%
```

```
      \expandafter\gdef\csname #2\the\el@ControlCounter#1\endcsname{.3em}%
    \else%
      \el@Error[^^J==> and #2\the\el@ControlCounter#1\space is set to 0cm.%
      ^^J==> Note that true TeX dimensions should be prefixed with%
      ^^J==> \string\the\space in \string\ListProperties]%
      {#2\the\el@ControlCounter#1}{\el@TempTestable}{dimension}%
      \expandafter\gdef\csname #2\the\el@ControlCounter#1\endcsname{0cm}%
    \fi%
  \fi%
```

Finally, we set the stage for a new iteration:

```
  \DubiousFigurefalse%
  \el@Advance\el@ControlCounter%
  \let\easynext\el@GenericUnitSearch%
\fi\easynext[#1]{#2}}%
```

8.5.2 Number denotation

Here's a test to check that the value of `Numbers` is valid. Well, it's quite simple, it just compares letters:

```
\def\el@LetterCheck#1{%
\if#1a%
\else\if#1r%
\else\if#1R%
\else\if#1l%
\else\if#1L%
\else\if#1z%
\else\DubiousLettertrue%
\fi\fi\fi\fi\fi\fi}
```

The generic version is exactly the same thing as `\el@GenericUnitSearch` above, so I don't comment it. It's even simpler since we don't need an argument: only one parameter uses this test.

```
\def\el@GenericLetterCheck{%
\ifnum\el@ControlCounter>\el@CounterTotal%
  \el@ControlCounter1%
  \def\easynext{\relax}%
\else%
  \edef\el@TempTestable{\csname Numbers\the\el@ControlCounter\endcsname}%
  \expandafter\el@LetterCheck\el@TempTestable%
  \ifDubiousLetter%
    \PackageError{easylist}%
      {^^J==> `\el@TempTestable' is not a valid number denotation %
      (Numbers\the\el@ControlCounter=\el@TempTestable).%
      ^^J==> It is ignored and those numbers will be arabic numbers}{}%
    \expandafter\gdef\csname Numbers\the\el@ControlCounter\endcsname{a}%
  \fi%
  \DubiousLetterfalse%
  \el@Advance\el@ControlCounter%
  \let\easynext\el@GenericLetterCheck%
\fi\easynext}%
```

8.5.3 Numbers

Finally we devise a generic version of `\el@NumberCheck` (defined at the beginning of the package). It concerns `Hide` and `Start(*)`. The beginning is as usual:

```
\newcommand{\el@GenericNumberCheck}[2][]{%
\ifnum\el@ControlCounter>\el@CounterTotal%
  \el@ControlCounter1%
  \def\easynext[##1]##2{\relax}%
```

We store the value of the parameter, make sure it's not `NA` (the unspecified value of `Start(*)`), and run `\el@NumberCheck`:

```
\else%
  \edef\el@TempTestable{\csname #2\the\el@ControlCounter#1\endcsname}%
  \ifx\el@TempTestable\el@NA%
  \else%
    \expandafter\el@NumberCheck\el@TempTestable?%
```

If something's wrong, we issue the appropriate message depending on the name of the parameter and the presence or absence of a star, redefine the command, and close as usual:

```
    \ifDubiousFigure%
      \def\el@Name{#2}%
      \ifx\el@Name\el@HideTest%
        \el@Error[^^J==> and no counter will be hidden for items of level \the\el@ControlCounter]%
          {#2\the\el@ControlCounter#1}{\el@TempTestable}{number}%
        \expandafter\gdef\csname #2\the\el@ControlCounter#1\endcsname{0}%
      \else%
        \if#1*%
          \el@Error[^^J==> and this counter will continue its progression]%
            {#2\the\el@ControlCounter#1}{\el@TempTestable}{counter}%
        \else%
          \el@Error[^^J==> and this counter will continue its progression]%
            {#2\the\el@ControlCounter#1}{\el@TempTestable}{number}%
        \fi%
        \expandafter\gdef\csname #2\the\el@ControlCounter#1\endcsname{NA}%
      \fi%
    \fi%
  \fi%
  \DubiousFigurefalse%
  \el@Advance\el@ControlCounter%
  \let\easynext\el@GenericNumberCheck%
\fi\easynext[#1]{#2}}%
```

8.6 Creating items

Now we can start typesetting items. First of all, we want counters to be properly reset when higher ones are stepped. So `\el@ResetCounters` sets a scratch counter to the control counter, steps it (because we're interested in those counters that are below the current one) and launch the real job.

```
\def\el@ResetCounters{%
\el@Scratch\el@ControlCounter%
\el@Advance\el@Scratch%
\el@@ResetCounters}
```

First, if we have done all counters, we stop.

```
\def\el@@ResetCounters{%
\ifnum\el@Scratch>\el@CounterTotal%
  \let\easynext\relax%
  \el@Scratch0%
```

Otherwise, we set the counters under investigation to 0, conditionally reset its `Start` value to `NA` and reiterate. Resetting `Start` just forces the user to define this parameter before an item containing that counter, which is just security. But we don't want this to happen with an intermediate counter. Indeed, setting for instance the first counter to 10 just before an item of the *second* level needs to reset the second counter, except when that counter has also just been specified with `Start2`. Now, when building an item's number, `\el@CounterLevel` represents its level (in our example, it's 2) and `\el@ControlCounter` represents the counter we're currently working with (here, 1). So the conditional does the job: it resets the second counter but not its `Start` value, in case there is one.

```
\else%
  \setcounter{List\the\el@Scratch}{0}%
```

```
  \ifnum\el@ControlCounter=\el@CounterLevel%
    \expandafter\gdef\csname Start\the\el@Scratch\endcsname{NA}%
  \fi%
  \el@Advance\el@Scratch%
  \let\easynext\el@@ResetCounters%
\fi\easynext}
```

We need some simple definitions, including a font definition for Zapf's Dingbats.

```
\def\el@ItemCounter{}
\def\el@NA{NA}
\font\el@ZapfDingbats=pzdr%
```

Now, here's the command that handles numbers and their punctuation. It defines `\el@ItemCounter`, to be inserted in the final product. First, we turn user's input into commands for the `Numbers` parameter. We do this here because a command such as `\l` has to be local, since it's used to typeset some letter. The `\z` command actually changes the font.

```
\def\el@PrintCounters{%
\def\a{\arabic}%
\def\l{\alph}%
\def\L{\Alph}%
\def\r{\roman}%
\def\R{\Roman}%
\def\z{\el@ZapfDingbats\arabic}
```

The definition is twofold: it prints the first numbers, and then the last one, i.e. the nth one in an item of the nth level. There are few differences. So, in case we're one of the first counters, i.e. in case `\el@ControlCounter` is lower than `\el@CounterLevel` (which basically records how many §'s we've seen), and if `Start*` for this counter is unspecified (equal to `NA`) but `Start` is not, then we turn that counter to the value of `Start`, set it back to `NA` (because we don't want the counter to be stuck to that value), and reset lower counters. Indeed, if you say `\ListProperties(Start1=5)` and then type §§, chances are you want the second counter to start at 1, don't you?

```
\ifnum\el@ControlCounter<\el@CounterLevel%
  \expandafter\ifx\csname Start\the\el@ControlCounter*\endcsname\el@NA%
    \expandafter\ifx\csname Start\the\el@ControlCounter\endcsname\el@NA%
    \else%
      \setcounter{List\the\el@ControlCounter}{\csname Start\the\el@ControlCounter\endcsname}%
      \expandafter\gdef\csname Start\the\el@ControlCounter\endcsname{NA}%
      \el@ResetCounters%
    \fi%
```

And if `Start*` is *not* unspecified, first we check if the value has changed: indeed, since `Start*` is meant to follow an external counter, changing its value will affect lower counters. For instance, if the first counter follows sections, every new section increases it by one and we want the second, third, etc., counters to be reset as with any incrementing of a higher counter. So if the current value of a counter is not equal to the value of its `Start*` parameter fully expanded, we launch `\el@ResetCounters`. In any case, we set the counter to `Start*`.

```
  \else%
    \expandafter\ifnum\csname theList\the\el@ControlCounter\endcsname=%
      \csname Start\the\el@ControlCounter*\endcsname%
    \else%
      \el@ResetCounters%
    \fi
    \setcounter{List\the\el@ControlCounter}{\csname Start\the\el@ControlCounter*\endcsname}%
  \fi%
```

Now, if the current counter is higher than `Hide` for the current level, we print it or rather store it in `\el@ItemCounter`, which before all stores itself, i.e. previous values of itself, so when we reiterate we retrieve everything from the previous iteration.

```
  \ifnum\el@ControlCounter>\csname Hide\the\el@CounterLevel\endcsname%
    \xdef\el@ItemCounter{\el@ItemCounter%
```

We store the counter value in a group, because if its `Numbers` parameter is `z`, we change the font, and we don't want that to spread. This parameter is used in a somewhat cumbersome but efficient fashion; we retrieve its value with `\csname Numbers\the\el@ControlCounter\endcsname`, which yields, say, `a`, and since it's enclosed in an additional `\csname`/`\endcsname` pair, we get `\a`, which was defined at the beginning of the macro.

```
    \bgroup%
    \csname\csname Numbers\the\el@ControlCounter\endcsname\endcsname{List\the\el@ControlCounter}%
    \egroup%
```

Finally, we put the punctuation mark, close the `\el@ItemCounter` and the conditional and set the stage for tail recursion:

```
    \csname Mark\the\el@ControlCounter\endcsname}%
  \fi%
  \el@Advance\el@ControlCounter%
  \let\easynext\el@PrintCounters%
```

Now, if we're concerned with the last counter of the item, first we increment it and reset all counters below:

```
\else%
  \stepcounter{List\the\el@CounterLevel}%
  \el@ResetCounters%
```

Then, as before, we check `Start` and `Start*` and act accordingly, except that we don't have to check whether `Start*` has changed, because counters are reset anyway.

```
  \expandafter\ifx\csname Start\the\el@ControlCounter*\endcsname\el@NA%
    \expandafter\ifx\csname Start\the\el@ControlCounter\endcsname\el@NA%
    \else%
      \setcounter{List\the\el@ControlCounter}{\csname Start\the\el@ControlCounter\endcsname}%
      \expandafter\gdef\csname Start\the\el@ControlCounter\endcsname{NA}%
    \fi%
  \else%
    \setcounter{List\the\el@ControlCounter}{\csname Start\the\el@ControlCounter*\endcsname}%
  \fi%
```

We build `\el@ItemCounter` just the same except that if `FinalMark` is specified, we put it instead of `Mark`.

```
  \ifnum\el@ControlCounter>\csname Hide\the\el@CounterLevel\endcsname%
    \xdef\el@ItemCounter{\el@ItemCounter%
    \bgroup%
    \csname\csname Numbers\the\el@ControlCounter\endcsname\endcsname{List\the\el@ControlCounter}%
    \egroup%
    \expandafter\ifx\csname FinalMark\the\el@ControlCounter\endcsname\el@NA%
      \csname Mark\the\el@ControlCounter\endcsname%
    \else%
      \csname FinalMark\the\el@ControlCounter\endcsname%
    \fi}%
  \fi%
```

And we stop the iteration and close:

```
  \el@ControlCounter1%
  \let\easynext\relax%
\fi%
\easynext}
```

Now, for the final big definition, we need two boxes, two dimensions, and a definition which is an imitation of LaTeX's `\@sptoken` to define a space, except that I enclosed it in a group here (since `\:` already exists). It makes `\el@Space` a space.

```
\newbox\el@CounterBox
\newbox\el@ControlBox
\newdimen\el@TotalMargin
\newdimen\el@LeftMove
{\def\:{\global\let\el@Space= }\: }
```

First, we turn # into a normal character because we'll need it in the error message, then we check whether `\elNextToken`, which has been stored by § (see the definition of the `easylist` environment below), and which represents the next character after the current §, is a space, in which case we must proceed. We reset `\el@LeftMove` (see below) and then close a group. Indeed, a group is open after each item number so that the text can have `Style**` command like `\color{red}` without affecting the rest of the text. `\begin{easylist}` has a corresponding `\begingroup` and `\end{easylist}` a `\endgroup`.

```
\catcode`#=12
\def\elCreateItem{%
\ifx\elNextToken\el@Space%
  \global\el@LeftMove=0pt%
  \endgroup%
```

Now, if `\el@CounterLevel`, which is incremented each time a § is followed by another § and which records the level of the item we're building, is higher than the total number of counters asked for, we issue an error message (with the proper symbol, hence the ugly conditional) and replace the item number by boxed exclamation marks. (ğ is the output of § and ű is the output of ¶ in the current encoding. As you may know, verbatimizing is not really showing the bare input, and there's nothing I can do about it because, as far as I'm aware, § and ¶ as glyphs are only accessible through commands which, in a verbatim environment, woulnd't launch... So we'll have to live with it to the end of the package.)

```
  \ifnum\el@CounterLevel>\el@CounterTotal%
    \PackageError{easylist}{^^J==> Too many %
    \ifAmpersand&\else\ifAt @\else\ifPilcrowű\else\ifSharp#\elseğ\fi\fi\fi\fi's.%
    ^^J==> You can't use more than \el@CounterTotal\space%
    \ifAmpersand&\else\ifAt @\else\ifPilcrowű\else\ifSharp#\elseğ\fi\fi\fi\fi's%
    ^^J==> unless you specify it when calling the package}{}
    \par\noindent\fbox{!!!}\begingroup%
```

If everything's okay, we create a paragraph and add `Space` or `Space*` depending on the level of the previous item, which was stored in `\el@PreviousItem` (`0` means that there was no previous item).

```
  \else%
    \par%
    \expandafter\ifnum\el@PreviousItem=0%
    \else%
      \expandafter\ifnum\el@PreviousItem=\el@CounterLevel%
        \vskip\csname Space\the\el@CounterLevel*\endcsname%
      \else%
        \vskip\csname Space\the\el@CounterLevel\endcsname%
      \fi%
    \fi%
```

Then we launch `\el@PrintCounters` to create `\el@ItemCounter`, and set the working margin to the value of `Margin` for the current level.

```
    \el@PrintCounters%
    \el@TotalMargin\csname Margin\the\el@CounterLevel\endcsname%
```

Now we build a box with the counter in full regalia, i.e. nested in its `CtrCom`, accompanied by its `Style(*)` and with the `FinalSpace` added at the end of it if not all counters were hidden (since if there's no counter, you don't want a space). The inner pair of braces creates a group that will prevent `Style(*)` values like `\color{blue}` from spreading to the text item.

```
    \setbox\el@CounterBox=\hbox{{%
      \csname CtrCom\the\el@CounterLevel\endcsname{%
      \csname Style\the\el@CounterLevel\endcsname%
      \csname Style\the\el@CounterLevel*\endcsname%
      \el@ItemCounter}%
      \ifnum\el@CounterLevel>\csname Hide\the\el@CounterLevel\endcsname%
        \hskip\csname FinalSpace\the\el@CounterLevel\endcsname%
      \fi}}%
```

Now we have to checked the `Align` parameter. If it is `false`, we do nothing, of course. If it is set to `fixed`, we store the width of the box we've just built, and it will serve as our reference width for this level. We turn `Align` to `alreadyfixed` for items to come.

```
    \expandafter\ifx\csname Align\the\el@CounterLevel\endcsname\el@False%
    \else\expandafter\ifx\csname Align\the\el@CounterLevel\endcsname\el@Fixed%
      \expandafter\xdef\csname CounterBoxWidth\the\el@CounterLevel\endcsname{%
        \the\wd\el@CounterBox}%
      \expandafter\gdef\csname Align\the\el@CounterLevel\endcsname{alreadyfixed}%
```

If `Align` is `alreadyfixed`, we rebuild the box, this time setting its width to the reference width and replacing `FinalSpace` with a glue.

```
    \else\expandafter\ifx\csname Align\the\el@CounterLevel\endcsname\el@AlreadyFixed%
      \setbox\el@CounterBox=\hbox to \csname CounterBoxWidth\the\el@CounterLevel\endcsname{{%
        \csname CtrCom\the\el@CounterLevel\endcsname{%
        \csname Style\the\el@CounterLevel\endcsname%
        \csname Style\the\el@CounterLevel*\endcsname%
        \el@ItemCounter}%
        \hfil}}%
```

We copy that new box into `\el@ControlBox` to get the *real* width of the counter, compare it to the desired value, and if it is larger, we issue a warning (but use the box anyway):

```
      \setbox\el@ControlBox=\hbox{\unhcopy\el@CounterBox}%
      \expandafter\ifdim\wd\el@ControlBox>\csname CounterBoxWidth\the\el@CounterLevel\endcsname%
        \PackageWarning{easylist}{%
          ^^J==> This counter is to wide and will spread on%
          ^^J==> the item text. You should increase FinalSpace%
          ^^J==> if you use ‘fixed’ or increase the dimension%
          ^^J==> if you specified one.
          ^^J==>}%
      \fi%
```

If `Align` is `move`, we store the width of the box once again and set `Align` to `alreadymoved`.

```
      \else\expandafter\ifx\csname Align\the\el@CounterLevel\endcsname\el@Move%
        \expandafter\xdef\csname CounterBoxWidth\the\el@CounterLevel\endcsname{%
          \the\wd\el@CounterBox}%
        \expandafter\gdef\csname Align\the\el@CounterLevel\endcsname{alreadymoved}%
```

And if `Align` is `alreadymoved`, we set a dimension to the width of the current box minus the desired width, i.e. we determine the extra space, which we'll remove when we insert the box.

```
      \else\expandafter\ifx\csname Align\the\el@CounterLevel\endcsname\el@AlreadyMoved%
        \el@LeftMove=\wd\el@CounterBox%
        \advance\el@LeftMove by -\csname CounterBoxWidth\the\el@CounterLevel\endcsname%
```

Finally, if `Align` is none of the above, it should be a dimension. But we have to be sure of this, so we run `\el@UnitSearch`, and if it's ok, we rebuild the box with the dimension, which we store, and turn `Align` to `alreadyfixed`. We measure the width one again an issue an warning if the box is overfull.

```
      \else%
        \edef\el@TempTestable{\csname Align\the\el@CounterLevel\endcsname}%
        \DubiousFigurefalse%
        \Signfalse%
        \Pointfalse%
        \expandafter\el@UnitSearch\el@TempTestable?%
        \ifDubiousFigure%
          \el@Error%
            [.^^J==> Admissible values are ‘false’, ‘fixed’, ‘move’ or a dimension]%
            {Align\the\el@CounterLevel}{\csname Align\the\el@CounterLevel\endcsname}%
            {value for ‘Align’}%
            \expandafter\gdef\csname Align\the\el@CounterLevel\endcsname{false}%
        \else%
          \expandafter\xdef\csname CounterBoxWidth\the\el@CounterLevel\endcsname{%
            \csname Align\the\el@CounterLevel\endcsname}%
            \setbox\el@CounterBox=\hbox to \csname CounterBoxWidth\the\el@CounterLevel\endcsname{{%
              \csname CtrCom\the\el@CounterLevel\endcsname{%
```

```
777                \csname Style\the\el@CounterLevel\endcsname%
778                \csname Style\the\el@CounterLevel*\endcsname%
779                \el@ItemCounter}%
780                \hfil}}%
781            \expandafter\gdef\csname Align\the\el@CounterLevel\endcsname{alreadyfixed}%
782            \setbox\el@ControlBox=\hbox{\unhcopy\el@CounterBox}%
783            \expandafter\ifdim\wd\el@ControlBox>\csname CounterBoxWidth\the\el@CounterLevel\endcsname%
784            \PackageWarning{easylist}{%
785              ^^J==> This counter is to wide and will spread on%
786              ^^J==> the item text. You should increase FinalSpace%
787              ^^J==> if you use 'fixed' or increase the dimension%
788              ^^J==> if you specified one.
789              ^^J==>}%
790            \fi%
791          \fi%
792        \fi\fi\fi\fi\fi%
```

Now we have to deal with the `Hang` parameter. If it is on, in case we're in a `alreadymoved` situation, we add the reference width to the working margin and substract it from `\parindent`; in any other case, we do the same with the current width of the box (which is the same of the desired width if `Align` is `alreadyfixed`). The net effect of this is to leave room for the item number in the left margin and to move the first line (with `\parindent`) on the left by its width.

```
793      \expandafter\ifx\csname Hang\the\el@CounterLevel\endcsname\el@True%
794        \expandafter\ifx\csname Align\the\el@CounterLevel\endcsname\el@AlreadyMoved%
795          \advance\el@TotalMargin by \csname CounterBoxWidth\the\el@CounterLevel\endcsname%
796          \parindent=-\csname CounterBoxWidth\the\el@CounterLevel\endcsname%
797        \else%
798          \advance\el@TotalMargin by \wd\el@CounterBox%
799          \parindent=-\wd\el@CounterBox%
800        \fi%
```

If `Hang` is turned off, we simply set `\parindent` to value specified by the user:

```
801      \else%
802        \parindent=\csname Indent\the\el@CounterLevel\endcsname%
803      \fi%
```

Finally, we set the left margin, and with `\hangafter0` we tell TEX to start indenting from the first line on. Then we add the negative space (set to 0pt if `Align` is not `alreadymoved`) and release the box.

```
804      \hangafter0\hangindent\el@TotalMargin%
805      \hskip-\el@LeftMove\box\el@CounterBox%
```

We're done with the counter and we must now set the stage for the item text. First, we open a group for the reason above, set `\@currentlabel` (used by LATEX to know what should be `\label`'ed with `\label`) to `\el@ItemCounter` and clear the latter (since it recusively defines itself, see above).

```
806      \begingroup%
807      \edef\@currentlabel{\el@ItemCounter}%
808      \gdef\el@ItemCounter{}%
```

We shape the paragraphs to come by setting `\parindent` to the value issued by the user (since the first paragraph has already begun with the printing of the counter, hanging items won't be affected) and by repeating the margin with `\everypar` (since `\hangafter` and `\hangindent` are reset after each paragraph). And, of course, we issue the `Style(**)` of this level.

```
809      \parindent=\csname Indent\the\el@CounterLevel\endcsname%
810      \everypar{\hangafter0\hangindent\el@TotalMargin}%
811      \csname Style\the\el@CounterLevel\endcsname%
812      \csname Style\the\el@CounterLevel**\endcsname%
```

Finally, we set some values, simply increment the level counter if there was no space after all, close by ignoring spaces so that the distance from the counter to the text will be rigidly fixed by `FinalSpace`, and restore `#`.

```
  \fi%
  \xdef\el@PreviousItem{\the\el@CounterLevel}%
  \global\el@CounterLevel1%
\else%
  \global\el@Advance\el@CounterLevel%
\fi\ignorespaces}
\catcode`#=6
```

Here are the predefined styles. There's nothing much to comment. `\elPredefinedStyle` is called by `\begin{easylist}` and does nothing if nothing follows.

```
\def\el@Tractatus{tractatus}
\def\el@CheckList{checklist}
\def\el@BookToc{booktoc}
\def\el@ArticleToc{articletoc}
\def\el@Enumerate{enumerate}
\def\el@Itemize{itemize}

\def\elPredefinedStyle{\@ifnextchar[{\el@PredefinedStyle}{}}

\def\el@PredefinedStyle[#1]{%
\def\el@TempStyle{#1}%
\ifx\el@TempStyle\el@Tractatus%
  \NewList(Mark=,Mark1=.)%
\else\ifx\el@TempStyle\el@CheckList%
  \NewList(%
  Hide=1000,Progressive*=1em,Hang=true,%
  Style*={\framebox(7,7){}}\hskip.6em,
  Style1**=\bfseries)
\else\ifx\el@TempStyle\el@BookToc%
  \NewList(%
    Hang=true,FinalMark=,Hide=1,%
    Style1=\large\bfseries,Numbers1=R,Space1=2.25em,Space1*=2.25em,Hide1=0,Hang1=false,Align1=2em,%
    Style2=\bfseries,Space2=1em,Space2*=1em,Align2=1.5em,%
    Margin3=1.5em,Margin4=3.8em,Margin5=7em,Margin6=10em,Margin7=12em,%
    Align3=2.3em,Align4=3.2em,Align5=4.1em,Align6=5em,Align7=6em)%
\else\ifx\el@TempStyle\el@ArticleToc%
  \NewList(%
    Hang=true,FinalMark=,%
    Align1=1.5em,Style1=\bfseries,Space1=1em,Space1*=1em,%
    Margin2=1.5em,Margin3=3.8em,Margin4=7em,Margin5=10em,%
    Align2=2.3em,Align3=3.2em,Align4=4.1em,Align5=5em)%
\else\ifx\el@TempStyle\el@Enumerate%
  \NewList(%
    FinalSpace=.5em,Hang=true,Mark=.,Space=4pt,Space*=4pt,Align=move,%
    Margin1=1.2em,%
    Margin2=2.9em,Style2*={(},Mark2={)},Numbers2=l,Hide2=1,%
    Margin3=5.6em,Numbers3=r,Hide3=2,%
    Margin4=6.8em,Numbers4=L,Hide4=3)%
\else\ifx\el@TempStyle\el@Itemize%
  \NewList(%
    Hang=true,Space=4pt,Space*=4pt,Hide=1000,%
    Margin1=1.5em,Style1*=\textbullet\hskip .5em,%
    Margin2=3.7em,Style2*=--\hskip .5em,%
    Margin3=5.9em,Style3*=$\ast$\hskip .5em,%
    Margin4=7.8em,Style4*=$\cdot$\hskip .5em)%
\else%
  \PackageError{easylist}{^^J==> `\el@TempStyle' is not a valid predefined style}{}%
\fi\fi\fi\fi\fi\fi}
```

Finally, we define the environment. First we turn off `@` and store the category codes of the symbols to be made active. I could have simply said for instance `\catcode`#=6` but who knows, catcodes might have

been changed by a previous package. `\edef\AtCatcode{\number\catcode‘@\relax}` is useless since we just said `\makeatletter`, but it’s for the sake of symmetry.

```
\makeatother

\edef\SectionCatcode{\number\catcode‘ğ}%
\edef\PilcrowCatcode{\number\catcode‘ű}%
\edef\SharpCatcode{\number\catcode‘#}%
\edef\AtCatcode{\number\catcode‘@}%
\edef\AmpersandCatcode{\number\catcode‘&}%
```

Then we make them active to allow a definition that will make them, say, active.

```
\catcode‘ğ=\active
\catcode‘ű=\active
\catcode‘@=\active
\catcode‘#=\active
\catcode‘&=\active
```

Now, according to the option loaded with the package, we create the proper environment. For instance, if `@` was chosen, we make it active in the environment and defines it as follows: it stores the next item in `\elNextToken` and calls `\elCreateItem`, which will act accordingly.

```
\ifAt
  \def\easylist{%
  \catcode‘@=\active%
  \def@{\futurelet\elNextToken\elCreateItem}%
```

Next `\easylist` opens a group to match those created by items and launch `\elPredefinedStyles`, while `\endeasylist` closes a group and add a `\par` that will ensure that the last item of an `easylist` is properly formatted.

```
  \begingroup\elPredefinedStyle}
  \def\endeasylist{\endgroup\par}
```

Finally we define `\Activate` and `\Deactivate`, which are straightforward. `\AtCatcode` is `\xdef`’ed because we want the current value category code of `@`, and `\Activate` might be issued in a group, so we want `\AtCatcode` to be global.

```
  \gdef\Activate{%
  \xdef\AtCatcode{\number\catcode‘@}%
  \catcode‘@=\active}
  \gdef\Deactivate{\catcode‘@=\AtCatcode}
```

And we do the same for the other symbols:

```
\else
  \ifPilcrow
    \def\easylist{%
    \catcode‘ű=\active%
    \defű{\futurelet\elNextToken\elCreateItem}%
    \begingroup\elPredefinedStyle}
    \def\endeasylist{\endgroup\par}
    \gdef\Activate{%
    \xdef\PilcrowCatcode{\number\catcode‘ű}%
    \catcode‘ű=\active}
    \gdef\Deactivate{\catcode‘ű=\PilcrowCatcode}
  \else
    \ifSharp
      \def\easylist{%
      \catcode‘#=\active%
      \def#{\futurelet\elNextToken\elCreateItem}%
      \begingroup\elPredefinedStyle}
      \def\endeasylist{\endgroup\par}
      \gdef\Activate{%
      \xdef\SharpCatcode{\number\catcode‘#}%
```

```
      \catcode`#=\active}
      \gdef\Deactivate{\catcode`#=\SharpCatcode}
    \else
      \ifAmpersand
        \def\easylist{%
        \catcode`&=\active%
        \def&{\futurelet\elNextToken\elCreateItem}%
        \begingroup\elPredefinedStyle}
        \def\endeasylist{\endgroup\par}
        \gdef\Activate{%
        \xdef\AmpersandCatcode{\number\catcode`&}%
        \catcode`&=\active}
        \gdef\Deactivate{\catcode`&=\AmpersandCatcode}
      \else
        \def\easylist{%
        \catcode`ğ=\active%
        \defğ{\futurelet\elNextToken\elCreateItem}%
        \begingroup\elPredefinedStyle}%
        \def\endeasylist{\endgroup\par}
        \gdef\Activate{%
        \xdef\SectionCatcode{\number\catcode`ğ}%
        \catcode`ğ=\active}
        \gdef\Deactivate{\catcode`ğ=\SectionCatcode}
      \fi
    \fi
  \fi
\fi
```

At last, we restore the original catcodes and say goodbye.

```
\catcode`&=\AmpersandCatcode
\catcode`#=\SharpCatcode
\catcode`@=\AtCatcode
\catcode`ű=\PilcrowCatcode
\catcode`ğ=\SectionCatcode
```