

beamer named overlay specification with beanoves

Jérôme Laurens

v1.0 2022/10/28

Abstract

This package allows the management of multiple slide lists in **beamer** documents. Slide lists are very handy both during edition and to manage complex and variable **beamer** overlay specifications.

Contents

1	Minimal example	2
2	Named slide lists	2
2.1	Presentation	2
2.2	Defining named slide lists	2
3	Named overlay specifications	3
3.1	Named slide ranges	3
3.2	Named slide lists	5
4	?(...) query expressions	5
5	Implementation	5
5.1	Package declarations	5
5.2	Local variables	6
5.3	Overlay specification	6
5.3.1	In slide range definitions	6
5.3.2	Regular expressions	8
5.3.3	beamer.cls interface	11
5.3.4	Defining named slide ranges	11
5.3.5	Scanning named overlay specifications	15
5.3.6	Evaluation bricks	19
5.3.7	Evaluation	35
5.3.8	Reseting slide ranges	46

1 Minimal example

The document below is a contrived example to show how the **beamer** overlay specifications have been extended.

```
1 \documentclass {beamer}
2 \RequirePackage {beanoves}
3 \begin{document}
4 \Beanoves {
5     A = 1:2,
6     B = A.next:3,
7     C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \insertframenumber}
11 {\Large Slide \insertslidenumber}
12 \visible<?(A.1)> {Only on slide 1}\\
13 \visible<?(B.1)-?(B.last)> {Only on slide 3 to 5}\\
14 \visible<?(C.1)> {Only on slide 6}\\
15 \visible<?(A.2)> {Only on slide 2}\\
16 \visible<?(B.2::B.last)> {Only on slide 4 to 5}\\
17 \visible<?(C.2)> {Only on slide 7}\\
18 \visible<?(A.3)-> {From slide 3}\\
19 \visible<?(B.3::B.last)> {Only on slide 5}\\
20 \visible<?(C.3)> {Only on slide 8}\\
21 \end{frame}
22 \end{document}
```

On line 4, we use the `\Beanoves` command to declare named slide ranges. On line 5, we declare a slide range named ‘A’, starting at slide 1 and with length 2. On line 12, the extended *named overlay specification* `?(A.1)` stands for 1, on line 15, `?(A.2)` stands for 2 whereas on line 18, `?(A.3)` stands for 3. On line 6, we declare a second slide range named ‘B’, starting after the 2 slides of ‘A’ namely 3. Its length is 3 meaning that its last slide number is 5, thus each `?(B.last)` is replaced by 5. The next slide number after slide range ‘B’ is 6 which is also the start of the third slide range due to line 7.

2 Named slide lists

2.1 Presentation

Within a **beamer** frame, there are different slides that appear in turn. The main slide list is a range on integers covering all the slide numbers, from one to the total amount of slides. In general, a slide list is a range of positive integers identified by a unique name. The main practical interest is that such lists may be defined relative to one another, we can even have lists of slide ranges. Finally, we can use these lists to organize **beamer** overlay specifications logically.

2.2 Defining named slide lists

In order to define named slide lists, we can either use the `\Beanoves` command below before a **beamer** frame environment, or use the `beanoves` option of this environment. The

value of the `beanoves` option is similar to the argument of the `\Beanoves` commands, but the latter takes precedence on the former. This behaviour may be useful to input the very same source code into different frames and have different combinations of slides.

```

beanoves = {
  <name1>=<spec1>,
  <name2>=<spec2>,
  ...,
  <namen>=<specn>,
}

```

```

\Beanoves{
  <name1>=<spec1>,
  <name2>=<spec2>,
  ...,
  <namen>=<specn>,
}

```

The keys $\langle name_i \rangle$ are the slide lists names, they are case sensitive and must contain no spaces nor `'/'` character. In order to avoid name conflicts with floating point functions, it is suggested to let them contain an uppercase letter or an underscore. When the same key is used multiple times, only the last one is taken into account. Possible values for $\langle spec_i \rangle$ are the *slide range specifiers* $\langle first \rangle$, $\langle first \rangle:\langle length \rangle$, $\langle first \rangle::\langle last \rangle$, $:\langle length \rangle::\langle last \rangle$ where $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$ are algebraic expression involving any integer valued named overlay specifications defined below.

Also possible values are *slide list specifiers* which are comma separated list of *slide range specifiers* and *slide list specifier* between square brackets. The definition

$$\langle name \rangle = [\langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_n \rangle],$$

is a convenient shortcut for

$$\begin{aligned} \langle name \rangle.1 &= \langle spec_1 \rangle, \\ \langle name \rangle.2 &= \langle spec_2 \rangle, \\ &\dots, \\ \langle name \rangle.n &= \langle spec_n \rangle. \end{aligned}$$

The rules above can apply individually to each

$$\langle name \rangle.i = \langle spec_i \rangle.$$

Moreover we can go deeper: the definition

$$\langle name \rangle = [[\langle spec_{1.1} \rangle, \langle spec_{1.2} \rangle], [[\langle spec_{2.1} \rangle, \langle spec_{2.2} \rangle]]$$

is a convenient shortcut for

$$\begin{aligned} \langle name \rangle.1.1 &= \langle spec_{1.1} \rangle, \\ \langle name \rangle.1.2 &= \langle spec_{1.2} \rangle, \\ \langle name \rangle.2.1 &= \langle spec_{2.1} \rangle, \\ \langle name \rangle.2.2 &= \langle spec_{2.2} \rangle \end{aligned}$$

and so on.

3 Named overlay specifications

3.1 Named slide ranges

When *slide range specifications* are used, the named overlay specifications are detailed in the tables below together with their replacement meaning value as `beamer` standard

overlay specification.

$\langle name \rangle == [i, i + 1, i + 2, \dots]$	
syntax	meaning
$\langle name \rangle.1$	i
$\langle name \rangle.2$	$i + 1$
$\langle name \rangle.\langle integer \rangle$	$i + \langle integer \rangle - 1$

In the frame example below, we use the `\BeanovesEval` command for the demonstration. It is mainly used for debugging and testing purposes.

```

1 \Beanoves {
2   A = 3:6,
3 }
4 \begin{frame} {Frame \insertframenum} {Slide \insertslidenumber}
5 \ttfamily
6 \BeanovesEval(A.1) ==3,
7 \BeanovesEval(A.2) ==4,
8 \BeanovesEval(A.-1)==1,
9 \end{frame}

```

When the slide range has been given a length or an end, like in the frame example below, we also have

$\langle name \rangle == [i, i + 1, \dots, j]$			
syntax	meaning	example	output
$\langle name \rangle.length$	$j - i + 1$	A.length	6
$\langle name \rangle.last$	j	A.last	8
$\langle name \rangle.next$	$j + 1$	A.next	9
$\langle name \rangle.range$	$i \text{ '-' } j$	A.range	3-8

```

1 \Beanoves {
2   A = 3:6,
3 }
4 \begin{frame} {Frame \insertframenum} {Slide \insertslidenumber}
5 \ttfamily
6 \BeanovesEval(A.length) == 6,
7 \BeanovesEval(A.1)      == 3,
8 \BeanovesEval(A.2)      == 4,
9 \BeanovesEval(A.-1)     == 1,
10 \end{frame}

```

Using these specification on unfinite named slide ranges is unsupported. Finally each named slide range has a dedicated counter $\langle name \rangle.n$ which is some kind of variable that can be used and incremented¹.

$\langle name \rangle.n$: use the position of the counter

$\langle name \rangle.n+=\langle integer \rangle$: advance the counter by $\langle integer \rangle$ and use the new position

$++\langle name \rangle.n$: advance the counter by 1 and use the new position

Notice that “.n” can generally be omitted.

¹This is actually an experimental feature.

3.2 Named slide lists

After the definition

$$\langle name \rangle = [\langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_n \rangle]$$

the rules of the previous section apply recursively to each individual declaration

$$\langle name \rangle . i = \langle spec_i \rangle.$$

4 ?(...) query expressions

This is the key feature of the `beanoves` package, extending `beamer overlay specifications` included between pointed brackets. Before the *overlay specifications* are processed by the `beamer` class, the `beanoves` package scans them for any occurrence of ‘ $\langle ?(\langle queries \rangle) \rangle$ ’. Each one is then evaluated and replaced by its static counterpart. The overall result is finally forwarded to the `beamer` class.

The $\langle queries \rangle$ argument is a comma separated list of individual $\langle query \rangle$ ’s of next table. Sometimes, using $\langle name \rangle . range$ is not allowed as it would lead to an algebraic difference instead of a range.

query	static value	limitation
:	–	
::	–	
$\langle first\ expr \rangle$	$\langle first \rangle$	
$\langle first\ expr \rangle :$	$\langle first \rangle -$	no $\langle name \rangle . range$
$\langle first\ expr \rangle ::$	$\langle first \rangle -$	no $\langle name \rangle . range$
$\langle first\ expr \rangle : \langle length\ expr \rangle$	$\langle first \rangle - \langle last \rangle$	no $\langle name \rangle . range$
$\langle first\ expr \rangle :: \langle end\ expr \rangle$	$\langle first \rangle - \langle last \rangle$	no $\langle name \rangle . range$

Here $\langle first\ expr \rangle$, $\langle length\ expr \rangle$ and $\langle end\ expr \rangle$ both denote algebraic expressions possibly involving named overlay specifications and counters. As integers, they respectively evaluate to $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$.

For example both $\langle ?(A.next) \rangle$, $\langle ?(A.last+1) \rangle$, $\langle ?(A.1+A.length) \rangle$ give the same result as soon as the slide range named ‘A’ has been properly defined with a length.

Notice that nesting $\langle ?(\dots) \rangle$ expressions is not supported.

```
1 \*package
```

5 Implementation

Identify the internal prefix (L^AT_EX3 DocStrip convention).

```
2 @@=beanoves
```

5.1 Package declarations

```
3 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
4 \ProvidesExplPackage
5   {beanoves}
6   {2022/10/28}
7   {1.0}
8   {Named overlay specifications for beamer}
9   \cs_new:Npn \__beanoves_DEBUG_on: {
10    \cs_set:Npn \__beanoves_DEBUG:n ##1 {
```

```

11 \msg_term:nnn { beanoves } { :n } { ##1 }
12 }
13 }
14 \cs_new:Npn \__beanoves_DEBUG_off: {
15 \cs_set_eq:NN \__beanoves_DEBUG:n \use_none:n
16 }
17 \__beanoves_DEBUG_off:
18 \cs_generate_variant:Nn \__beanoves_DEBUG:n { x, V }
19 \int_zero_new:N \l__beanoves_group_int
20 \cs_set:Npn \__beanoves_group_begin: {
21 \group_begin:
22 \int_incr:N \l__beanoves_group_int
23 \__beanoves_DEBUG:x {GROUP~DOWN:~\int_use:N \l__beanoves_group_int}
24 }
25 \cs_set:Npn \__beanoves_group_end: {
26 \group_end:
27 \__beanoves_DEBUG:x {GROUP~UP:~\int_use:N \l__beanoves_group_int}
28 }

```

5.2 Local variables

We make heavy use of local variables and function scopes. Many functions are executed within a \TeX group, which ensures no name collision with the caller stack. In that case, variables need not follow exactly the \LaTeX 3 naming convention: we do not specialize with the module name. On execution, next initialization instructions declare the variables as side effect.

```

29 \int_zero_new:N \l__beanoves_split_int
30 \int_zero_new:N \l__beanoves_depth_int
31 \int_zero_new:N \g__beanoves_append_int
32 \bool_new:N \l__beanoves_no_counter_bool
33 \bool_new:N \l__beanoves_no_range_bool
34 \bool_new:N \l__beanoves_continue_bool

```

5.3 Overlay specification

5.3.1 In slide range definitions

$\backslash\text{g_beanoves_prop}$ $\langle\text{key}\rangle\text{--}\langle\text{value}\rangle$ property list to store the named slide lists. The basic keys are, assuming $\langle\text{name}\rangle$ is a slide list identifier,

$\langle\text{name}\rangle/\text{A}$ for the first index

$\langle\text{name}\rangle/\text{L}$ for the length when provided

$\langle\text{name}\rangle/\text{Z}$ for the last index when provided

$\langle\text{name}\rangle/\text{C}$ for the counter value, when used

$\langle\text{name}\rangle/\text{C0}$ for initial value of the counter (when reset)

Other keys are eventually used to cache results when some attributes are defined from other slide ranges. They are characterized by a $\text{'//'}.$

$\langle\text{name}\rangle//\text{A}$ for the cached static value of the first index

$\langle\text{name}\rangle//\text{Z}$ for the cached static value of the last index

$\langle name \rangle // L$ for the cached static value of the length

$\langle name \rangle // N$ for the cached static value of the next index

The implementation is private, in particular, keys may change in future versions.

```
35 \prop_new:N \g__beanoves_prop
```

(End definition for $\backslash g_beanoves_prop$.)

```
\__beanoves_gput:nn
\__beanoves_gput:nV
\__beanoves_gprovide:nn
\__beanoves_gprovide:nV
\__beanoves_item:n
\__beanoves_get:nN
\__beanoves_gremove:n
\__beanoves_gclear:n
\__beanoves_gclear_cache:n
\__beanoves_gclear:
```

```
\__beanoves_gput:nn {\langle key \rangle} {\langle value \rangle}
\__beanoves_gprovide:nn {\langle key \rangle} {\langle value \rangle}
\__beanoves_item:n {\langle key \rangle}
\__beanoves_get:n {\langle key \rangle} \langle tl variable \rangle
\__beanoves_gremove:n {\langle key \rangle}
\__beanoves_gclear:n {\langle key \rangle}
\__beanoves_gclear:
```

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L^AT_EX3 eponym functions, except $\backslash _beanoves_gprovide:nn$ which meaning is straightforward.

```
36 \cs_new:Npn \__beanoves_gput:nn {
37   \prop_gput:Nnn \g__beanoves_prop
38 }
39 \cs_new:Npn \__beanoves_gprovide:nn #1 #2 {
40   \prop_if_in:NnF \g__beanoves_prop { #1 } {
41     \prop_gput:Nnn \g__beanoves_prop { #1 } { #2 }
42   }
43 }
44 \cs_new:Npn \__beanoves_item:n {
45   \prop_item:Nn \g__beanoves_prop
46 }
47 \cs_new:Npn \__beanoves_get:nN {
48   \prop_get:NnN \g__beanoves_prop
49 }
50 \cs_new:Npn \__beanoves_gremove:n {
51   \prop_gremove:Nn \g__beanoves_prop
52 }
53 \cs_new:Npn \__beanoves_gclear:n #1 {
54   \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
55     \__beanoves_gremove:n { #1 / ##1 }
56   }
57 }
58 \cs_new:Npn \__beanoves_gclear_cache:n #1 {
59   \clist_map_inline:nn { /A, /L, /Z, /N } {
60     \__beanoves_gremove:n { #1 / ##1 }
61   }
62 }
63 \cs_new:Npn \__beanoves_gclear: {
64   \prop_gclear:N \g__beanoves_prop
65 }
66 \cs_generate_variant:Nn \__beanoves_gput:nn { nV }
67 \cs_generate_variant:Nn \__beanoves_gprovide:nn { nV }
```

<code>__beanoves_if_in_p:n *</code> <code>__beanoves_if_in_p:V *</code> <code>__beanoves_if_in:nTF *</code> <code>__beanoves_if_in:VTF *</code>	<code>__beanoves_if_in_p:n {<key>}</code> <code>__beanoves_if_in:nTF {<key>} {<true code>} {<false code>}</code> <p>Convenient shortcuts to test for the existence of some key, it makes the code more concise and readable.</p>
--	--

```

68 \prg_new_conditional:Npnn \__beanoves_if_in:n #1 { p, T, F, TF } {
69   \prop_if_in:NnTF \g__beanoves_prop { #1 } {
70     \prg_return_true:
71   } {
72     \prg_return_false:
73   }
74 }
75 \prg_generate_conditional_variant:Nnn \__beanoves_if_in:n {V} { p, T, F, TF }

```

<code>__beanoves_get:nNTF</code>	<code>__beanoves_get:nNTF {<key>} {<tl variable>} {<true code>} {<false code>}</code>
-----------------------------------	--

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute *<true code>* when the item is found, *<false code>* otherwise. In the latter case, the content of the *<tl variable>* is undefined. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```

76 \prg_new_conditional:Npnn \__beanoves_get:nN #1 #2 { T, F, TF } {
77   \prop_get:NnNTF \g__beanoves_prop { #1 } #2 {
78     \prg_return_true:
79   } {
80     \prg_return_false:
81   }
82 }

```

Utility message.

```

83 \msg_new:nnn { beanoves } { :n } { #1 }

```

5.3.2 Regular expressions

<code>\c__beanoves_name_regex</code>	<p>The name of a slide range consists of a non void list of alphanumerical characters and underscore, but with no leading digit.</p>
--------------------------------------	--

```

84 \regex_const:Nn \c__beanoves_name_regex {
85   [[:alpha:]]_ [[:alnum:]]_*
86 }

```

(End definition for `\c__beanoves_name_regex`.)

<code>\c__beanoves_path_regex</code>	<p>A sequence of <i><positive integer></i> items representing a path.</p>
--------------------------------------	---

```

87 \regex_const:Nn \c__beanoves_path_regex {
88   (?: \. \d+ ) *
89 }

```

(End definition for `\c__beanoves_path_regex`.)

<code>\c__beanoves_key_regex</code> <code>\c__beanoves_A_key_Z_regex</code>	<p>A key is the name of a slide range possibly followed by positive integer attributes using a dot syntax. The 'A_key_Z' variant matches the whole string.</p>
--	--

```

90 \regex_const:Nn \c__beanoves_key_regex {
91   \ur{c__beanoves_name_regex} \ur{c__beanoves_path_regex}
92 }

```



```

93 \regex_const:Nn \c__beanoves_A_key_Z_regex {
94   \A \ur{c__beanoves_key_regex} \Z
95 }

```

(End definition for `\c__beanoves_key_regex` and `\c__beanoves_A_key_Z_regex`.)

`\c__beanoves_dotted_regex` A specifier is the name of a slide range possibly followed by attributes using a dot syntax. This is a poor man version to save computations, a dedicated parser would help in error management.

```

96 \regex_const:Nn \c__beanoves_dotted_regex {
97   \A \ur{c__beanoves_name_regex} (?: \. [\^.] + ) * \Z
98 }

```

(End definition for `\c__beanoves_dotted_regex`.)

`\c__beanoves_colons_regex` For ranges defined by a colon syntax.

```

99 \regex_const:Nn \c__beanoves_colons_regex { :(:+)? }

```

(End definition for `\c__beanoves_colons_regex`.)

`\c__beanoves_int_regex` A decimal integer with an eventual leading sign next to the first digit.

```

100 \regex_const:Nn \c__beanoves_int_regex {
101   (?: [-+])? \d+
102 }

```

(End definition for `\c__beanoves_int_regex`.)

`\c__beanoves_list_regex` A comma separated list between square brackets.

```

103 \regex_const:Nn \c__beanoves_list_regex {
104   \A \[ \s*

```

Capture groups:

- 2: the content between the brackets, outer spaces trimmed out

```

105   ( [\^\\] %[---
106   ] *? )
107   \s* \] \Z
108 }

```

(End definition for `\c__beanoves_list_regex`.)

`\c__beanoves_split_regex` Used to parse slide list overlay specifications in queries. Next are the 10 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

```

109 \regex_const:Nn \c__beanoves_split_regex {
110   \s* ( ? :

```

We start with ‘++’ instrussions ².

- 1: $\langle id \rangle$ of a slide range
- 2: $\langle name \rangle$ of a slide range

```

111   \+ \+ (?: ( \ur{c__beanoves_name_regex} ) ! )? ( \ur{c__beanoves_name_regex} )

```

²At the same time an instruction and an expression... this is a synonym of expreccion

- 3: optionally followed by an integer path

112 (\ur{c__beanoves_path_regex}) (?: \. n)?

We continue with other expressions

- 4: $\langle id \rangle$ of a slide range
- 5: $\langle name \rangle$ of a slide range

113 | (?: (\ur{c__beanoves_name_regex}) !)? (\ur{c__beanoves_name_regex})

- 6: optionally followed by an integer path

114 (\ur{c__beanoves_path_regex})

Next comes another branching

115 (?:

- 7: the $\langle length \rangle$ attribute

116 \. l(e)ngth

- 8: the $\langle last \rangle$ attribute

117 | \. l(a)st

- 9: the $\langle next \rangle$ attribute

118 | \. ne(x)t

- 10: the $\langle range \rangle$ attribute

119 | \. (r)ange

- 11: the $\langle n \rangle$ attribute

120 | \. (n)

• 12: the poor man integer expression after ‘+=’. When it contains no parenthesis, it is an algebraic expression involving integers and $\langle key \rangle$ ’s. Otherwise it starts with a parenthesis and ends with the first parenthesis followed by a white space or the end of the text. This tricky definition allows quite any algebraic expression involving parenthesis. The problems may arise when dealing with nested expressions.

121 (?: \s* \+= \s*
122 ((?: \ur{c__beanoves_int_regex} | \ur{c__beanoves_key_regex})
123 (?: [+ \- */] (?: \d+ | \ur{c__beanoves_key_regex})) *
124 | \ (.*? \) (?: \Z | \s+)
125)
126)?

- 13: a trailing ‘- $\langle integer \rangle$ ’.

127 | \. (- \ur{c__beanoves_int_regex})

128)?

129) \s*

130 }

(End definition for \c__beanoves_split_regex.)

5.3.3 beamer.cls interface

```

131 \RequirePackage{keyval}
132 \define@key{beamerframe}{beanoves-id}[]{}
133 \tl_set:Nx \l__beanoves_id_tl { #1 }
134 \__beanoves_DEBUG_on:
135 \__beanoves_DEBUG:x {*****~THIS_IS_KEY}
136 \__beanoves_DEBUG_off:
137 }
138 \AddToHook{env/beamer@frameslide/before}{
139 \__beanoves_DEBUG_on:
140 \__beanoves_DEBUG:x {*****~THIS_IS_BEFORE}
141 \__beanoves_DEBUG_off:
142 }
143 \AddToHook{cmd/frame/before}{
144 \tl_clear:N \l__beanoves_id_tl
145 \__beanoves_DEBUG_on:
146 \__beanoves_DEBUG:x {*****~THIS_IS_FRAME}
147 \__beanoves_DEBUG_off:
148 }

```

5.3.4 Defining named slide ranges

<code>__beanoves_parse:Nnn</code>	<code>__beanoves_parse:nn <command> {<key>} {<definition>}</code>
------------------------------------	--

Auxiliary function called within a group. *<key>* is the slide key, including eventually a dotted integer path, *<definition>* is the corresponding definition. *<command>* is `__beanoves_range:nVVV` at runtime.

`\l_match_seq` Local storage for the match result.

(End definition for `\l_match_seq`. This variable is documented on page ??.)

```

\__beanoves_range:nnnn
\__beanoves_range:nVVV
\__beanoves_range_alt:nnnn
\__beanoves_range_alt:nVVV

```

```

\__beanoves_range:nnnn {<key>} {<first>} {<length>} {<last>}
\__beanoves_range_alt:nnnn {<key>} {<first>} {<length>} {<last>}

```

Auxiliary function called within a group. Setup the model to define a range. The alt variant does not override an already existing value.

```

149 \cs_new:Npn \__beanoves_range:Nnnnn #1 #2 #3 #4 #5 {
150   \tl_if_empty:nTF { #3 } {
151     \tl_if_empty:nTF { #4 } {
152       \tl_if_empty:nTF { #5 } {
153         \msg_error:nnn { beanoves } { :n } { Not~a~range::~~#2 }
154       } {
155         #1 { #2/Z } { #5 }
156       }
157     } {
158       #1 { #2/L } { #4 }
159       \tl_if_empty:nF { #5 } {
160         #1 { #2/Z } { #5 }
161         #1 { #2/A } { #2.last - (#2.length) + 1 }
162       }
163     }
164   } {
165     #1 { #2/A } { #3 }
166     \tl_if_empty:nTF { #4 } {
167       \tl_if_empty:nF { #5 } {
168         #1 { #2/Z } { #5 }
169         #1 { #2/L } { #2.last - (#2.1) + 1 }
170       }
171     } {
172       #1 { #2/L } { #4 }
173       #1 { #2/Z } { #2.1 + #2.length - 1 }
174     }
175   }
176 }
177 \cs_new:Npn \__beanoves_range:nnnn #1 {
178   \__beanoves_gclear:n { #1 }
179   \__beanoves_range:Nnnnn \__beanoves_gput:nn { #1 }
180 }
181 \cs_generate_variant:Nn \__beanoves_range:nnnn { nVVV }
182 \cs_new:Npn \__beanoves_range_alt:nnnn #1 {
183   \__beanoves_gclear_cache:n { #1 }
184   \__beanoves_range:Nnnnn \__beanoves_gprovide:nn { #1 }
185 }
186 \cs_generate_variant:Nn \__beanoves_range_alt:nnnn { nVVV }

```

```

\__beanoves_parse:Nn \__beanoves_parse:nn <command> {<key>}

```

Define a hidden range, for which slides are never shown. This is useful to conditionally show or hide a sequence of slides.

```

187 \cs_new:Npn \__beanoves_parse:Nn #1 #2 {
188   \__beanoves_gput:nn { #1/ } { }
189 }
190 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
191 \cs_new:Npn \__beanoves_do_parse:Nnn #1 #2 #3 {

```

The first argument has signature `nVVV`. This is not a list.

```

192 \tl_clear:N \l_a_tl
193 \tl_clear:N \l_b_tl
194 \tl_clear:N \l_c_tl
195 \regex_split:NnN \c__beanoves_colons_regex { #3 } \l_split_seq
196 \seq_pop_left:NNT \l_split_seq \l_a_tl {

```

`\l_a_tl` may contain the $\langle start \rangle$.

```

197 \seq_pop_left:NNT \l_split_seq \l_b_tl {
198 \tl_if_empty:NTF \l_b_tl {

```

This is a one colon range.

```

199 \seq_pop_left:NN \l_split_seq \l_b_tl

```

`\l_b_tl` may contain the $\langle length \rangle$.

```

200 \seq_pop_left:NNT \l_split_seq \l_c_tl {
201 \tl_if_empty:NTF \l_c_tl {

```

A :: was expected:

```

202 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(1):~#3 }
203 } {
204 \int_compare:nNnT { \tl_count:N \l_c_tl } > { 1 } {
205 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(2):~#3 }
206 }
207 \seq_pop_left:NN \l_split_seq \l_c_tl

```

`\l_c_tl` may contain the $\langle end \rangle$.

```

208 \seq_if_empty:NF \l_split_seq {
209 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(3):~#3 }
210 }
211 }
212 }
213 } {

```

This is a two colon range.

```

214 \int_compare:nNnT { \tl_count:N \l_b_tl } > { 1 } {
215 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(4):~#3 }
216 }
217 \seq_pop_left:NN \l_split_seq \l_c_tl

```

`\l_c_tl` contains the $\langle end \rangle$.

```

218 \seq_pop_left:NNTF \l_split_seq \l_b_tl {
219 \tl_if_empty:NTF \l_b_tl {
220 \seq_pop_left:NN \l_split_seq \l_b_tl

```

`\l_b_tl` may contain the $\langle length \rangle$.

```

221 \seq_if_empty:NF \l_split_seq {
222 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(5):~#3 }
223 }
224 } {
225 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(6):~#3 }
226 }
227 } {
228 \tl_clear:N \l_b_tl
229 }
230 }
231 }
232 }

```

Providing both the $\langle start \rangle$, $\langle length \rangle$ and $\langle end \rangle$ of a range is not allowed, even if they happen to be consistent.

```

233   \bool_if:nF {
234     \tl_if_empty_p:N \l_a_tl
235     || \tl_if_empty_p:N \l_b_tl
236     || \tl_if_empty_p:N \l_c_tl
237   } {
238     \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(7):~#3 }
239   }
240   #1 { #2 } \l_a_tl \l_b_tl \l_c_tl
241 }

242 \cs_new:Npn \__beanoves_parse:Nnn #1 #2 #3 {
243   \__beanoves_group_begin:
244   \regex_match:NnTF \c__beanoves_A_key_Z_regex { #2 } {

```

We got a valid key.

```

245   \regex_extract_once:NnNTF \c__beanoves_list_regex { #3 } \l_match_seq {

```

This is a comma separated list, extract each item and go recursive.

```

246     \exp_args:NNx
247     \seq_set_from_clist:Nn \l_match_seq {
248       \seq_item:Nn \l_match_seq { 2 }
249     }
250     \seq_map_indexed_inline:Nn \l_match_seq {
251       \__beanoves_do_parse:Nnn #1 { #2.##1 } { ##2 }
252     }
253   } {
254     \__beanoves_do_parse:Nnn #1 { #2 } { #3 }
255   }
256 } {
257   \msg_error:nnn { beanoves } { :n } { Invalid~key:~#1 }
258 }
259 \__beanoves_group_end:
260 }

```

\Beanoves

\Beanoves {*<key--value list>*}

The keys are the slide range specifiers. When no value is provided, it defaults to 1. On the contrary, *<key-value>* items are parsed by `__beanoves_parse:Nnn`.

```
261 \NewDocumentCommand \Beanoves { sm } {
262   \tl_if_eq:NnT \@currenvir { document } {
263     \__beanoves_gclear:
264   }
265   \IfBooleanTF {#1} {
266     \keyval_parse:nnn {
267       \__beanoves_parse:Nn \__beanoves_range_alt:nVVV
268     } {
269       \__beanoves_parse:Nnn \__beanoves_range_alt:nVVV
270     }
271   } {
272     \keyval_parse:nnn {
273       \__beanoves_parse:Nn \__beanoves_range:nVVV
274     } {
275       \__beanoves_parse:Nnn \__beanoves_range:nVVV
276     }
277   }
278   { #2 }
279   \ignorespaces
280 }
```

If we use the frame `beanoves` option, we can provide default values to the various name ranges.

```
281 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}
```

5.3.5 Scanning named overlay specifications

Patch some beamer commands to support `?(...)` instructions in overlay specifications.

\beamer@frame

\beamer@frame {*<overlay specification>*}

\beamer@masterdecode

\beamer@masterdecode {*<overlay specification>*}

Preprocess *<overlay specification>* before `beamer` uses it.

\l_ans_tl

Storage for the translated overlay specification, where `?(...)` instructions are replaced by their static counterparts.

(End definition for `\l_ans_tl`. This variable is documented on page 37.)

Save the original macro `\beamer@masterdecode` and then override it to properly preprocess the argument.

```
282 \cs_set_eq:NN \__beanoves_beamer@frame \beamer@frame
283 \cs_set:Npn \beamer@frame < #1 > {
284   \__beanoves_group_begin:
285   \tl_clear:N \l_ans_tl
286   \__beanoves_scan:nNN { #1 } \__beanoves_eval:nN \l_ans_tl
287   \exp_args:NNNV
288   \__beanoves_group_end:
289   \__beanoves_beamer@frame < \l_ans_tl >
290 }
291 \cs_set_eq:NN \__beanoves_beamer@masterdecode \beamer@masterdecode
```

```

292 \cs_set:Npn \beamer@masterdecode #1 {
293   \__beanoves_group_begin:
294   \tl_clear:N \l_ans_tl
295   \__beanoves_scan:nNN { #1 } \__beanoves_eval:nN \l_ans_tl
296   \exp_args:NNV
297   \__beanoves_group_end:
298   \__beanoves_beamer@masterdecode \l_ans_tl
299 }

```

__beanoves_scan:nNN __beanoves_scan:nNN {*<named overlay expression>*} *<eval>* *<tl variable>*

Scan the *<named overlay expression>* argument and feed the *<tl variable>* replacing ?(...) instructions by their static counterpart with help from the *<eval>* function, which is __beanoves_eval:nN. A group is created to use local variables:

\l_ans_tl: is the token list that will be appended to *<tl variable>* on return.

\l__beanoves_depth_int Store the depth level in parenthesis grouping used when finding the proper closing parenthesis balancing the opening parenthesis that follows immediately a question mark in a ?(...) instruction.

(End definition for \l__beanoves_depth_int.)

g__beanoves_append_int Decrement each time __beanoves_append:nN is called. To avoid catch circular definitions.

(End definition for g__beanoves_append_int.)

\l_query_tl Storage for the overlay query expression to be evaluated.

(End definition for \l_query_tl. This variable is documented on page ??.)

\l_token_seq The *<overlay expression>* is split into the sequence of its tokens.

(End definition for \l_token_seq. This variable is documented on page ??.)

\l_ask_bool Whether a loop may continue. Controls the continuation of the main loop that scans the tokens of the *<named overlay expression>* looking for a question mark.

(End definition for \l_ask_bool. This variable is documented on page ??.)

\l_query_bool Whether a loop may continue. Controls the continuation of the secondary loop that scans the tokens of the *<named overlay expression>* looking for an opening parenthesis follow the question mark. It then controls the loop looking for the balanced closing parenthesis.

(End definition for \l_query_bool. This variable is documented on page ??.)

\l_token_tl Storage for just one token.

(End definition for \l_token_tl. This variable is documented on page ??.)

```

300 \cs_new:Npn \__beanoves_scan:nNN #1 #2 #3 {
301   \__beanoves_group_begin:
302   \tl_clear:N \l_ans_tl
303   \int_zero:N \l__beanoves_depth_int
304   \seq_clear:N \l_token_seq

```


Explode the *<named overlay expression>* into a list of tokens:

```
305 \regex_split:nnN {} { #1 } \l_token_seq
```

Run the top level loop to scan for a ‘?’:

```
306 \bool_set_true:N \l_ask_bool
307 \bool_while_do:Nn \l_ask_bool {
308   \seq_pop_left:NN \l_token_seq \l_token_tl
309   \quark_if_no_value:NTF \l_token_tl {
```

We reached the end of the sequence (and the token list), we end the loop here.

```
310   \bool_set_false:N \l_ask_bool
311 } {
```

`\l_token_tl` contains a ‘normal’ token.

```
312   \tl_if_eq:NnTF \l_token_tl { ? } {
```

We found a ‘?’, we first gobble tokens until the next ‘(’, whatever they may be. In general, no tokens should be silently ignored.

```
313   \bool_set_true:N \l_query_bool
314   \bool_while_do:Nn \l_query_bool {
```

Get next token.

```
315   \seq_pop_left:NN \l_token_seq \l_token_tl
316   \quark_if_no_value:NTF \l_token_tl {
```

No opening parenthesis found, raise.

```
317   \msg_fatal:nmx { beanoves } { :n } {Missing~'('%---)
318   ~after~a~?:~#1}
319 } {
320   \tl_if_eq:NnT \l_token_tl { ( %}
321 } {
```

We found the ‘(’ after the ‘?’. Increment the parenthesis depth to 1 (on first passage).

```
322   \int_incr:N \l__beanoves_depth_int
```

Record the forthcoming content in the `\l_query_tl` variable, up to the next balancing ‘)’:

```
323   \tl_clear:N \l_query_tl
324   \bool_while_do:Nn \l_query_bool {
```

Get next token.

```
325   \seq_pop_left:NN \l_token_seq \l_token_tl
326   \quark_if_no_value:NTF \l_token_tl {
```

We reached the end of the sequence and the token list with no closing ‘)’. We raise and end both bool while loops. As recovery we feed `\l_query_tl` with the missing ‘)’. `\l__depth_int` is 0 whenever `\l_query_bool` is false.

```
327   \msg_error:nmx { beanoves } { :n } {Missing~%((---
328   ~)':~#1 }
329   \int_do_while:nNnn \l__beanoves_depth_int > 1 {
330     \int_decr:N \l__beanoves_depth_int
331     \tl_put_right:Nn \l_query_tl {%(---
332   )}
333 }
334 \int_zero:N \l__beanoves_depth_int
335 \bool_set_false:N \l_query_bool
336 \bool_set_false:N \l_ask_bool
```

```

337         } {
338         \tl_if_eq:NnTF \l_token_tl { ( %---)
339         } {

```

We found a ‘(’, increment the depth and append the token to \l_query_tl.

```

340         \int_incr:N \l__beanoves_depth_int
341         \tl_put_right:NV \l_query_tl \l_token_tl
342     } {

```

This is not a ‘(’.

```

343         \tl_if_eq:NnTF \l_token_tl { %(
344         )
345     } {

```

We found a ‘)’, decrement the depth.

```

346         \int_decr:N \l__beanoves_depth_int
347         \int_compare:nNnTF \l__beanoves_depth_int = 0 {

```

The depth level has reached 0: we found our balancing parenthesis of the ?(...) instruction. We can append the evaluated slide ranges token list to \l_ans_tl and stop the inner loop.

```

348     \exp_args:NV #2 \l_query_tl \l_ans_tl
349     \bool_set_false:N \l_query_bool
350 } {

```

The depth has not yet reached level 0. We append the ‘)’ to \l_query_tl because it is not the end of sequence marker.

```

351         \tl_put_right:NV \l_query_tl \l_token_tl
352     }

```

Above ends the code for a positive depth.

```

353     } {

```

The scanned token is not a ‘(’ nor a ‘)’, we append it as is to \l_query_tl.

```

354         \tl_put_right:NV \l_query_tl \l_token_tl
355     }
356 }
357 }

```

Above ends the code for Not a ‘(’

```

358     }
359 }

```

Above ends the code for: Found the ‘(’ after the ‘?’

```

360     }

```

Above ends the code for not a no value quark.

```

361 }

```

Above ends the code for the bool while loop to find the ‘(’ after the ‘?’.

If we reached the end of the token list, then end both the current loop and its containing loop.

```

362     \quark_if_no_value:NT \l_token_tl {
363         \bool_set_false:N \l_query_bool
364         \bool_set_false:N \l_ask_bool
365     }
366 } {

```

This is not a '?', append the token to right of \l_ans_tl and continue.

```
367     \tl_put_right:NV \l_ans_tl \l_token_tl
368   }
```

Above ends the code for the bool while loop to find a '(' after the '?'

```
369   }
370 }
```

Above ends the outer bool while loop to find '?' characters. We can append our result to $\langle tl\ variable \rangle$

```
371   \exp_args:NNNV
372   \__beanoves_group_end:
373   \tl_put_right:Nn #3 \l_ans_tl
374 }
```

I

5.3.6 Evaluation bricks

$\backslash_beanoves_fp_round:nN$ $\backslash_beanoves_fp_round:N$	$\backslash_beanoves_fp_round:nN \{\langle expression \rangle\} \langle tl\ variable \rangle$ $\backslash_beanoves_fp_round:N \langle tl\ variable \rangle$
---	--

Shortcut for $\backslash fp_eval:n\{round(\langle expression \rangle)\}$ appended to $\langle tl\ variable \rangle$. The second variant replaces the variable content with its rounded floating point evaluation.

```
375 \cs_new:Npn \__beanoves_fp_round:nN #1 #2 {
376   \__beanoves_DEBUG:x { ROUND:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2}
377   \tl_if_empty:nTF { #1 } {
378     \__beanoves_DEBUG:x { ROUND1:~EMPTY }
379   } {
380     \__beanoves_DEBUG:x { ROUND1:~\tl_to_str:n{#1} }
381     \tl_put_right:Nx #2 {
382       \fp_eval:n { round(#1) }
383     }
384   }
385 }
386 \cs_generate_variant:Nn \__beanoves_fp_round:nN { VN, xN }
387 \cs_new:Npn \__beanoves_fp_round:N #1 {
388   \__beanoves_DEBUG:x { ROUND:\string#1=\tl_to_str:V #1}
389   \tl_if_empty:VTF #1 {
390     \__beanoves_DEBUG:x { ROUND2:~EMPTY }
391   } {
392     \__beanoves_DEBUG:x { ROUND2:~\exp_args:Nx\tl_to_str:n{#1} }
393     \tl_set:Nx #1 {
394       \fp_eval:n { round(#1) }
395     }
396   }
397 }
```

$\backslash_beanoves_raw_first:nNTF$ $\backslash_beanoves_raw_first:(xN VN)TF$	$\backslash_beanoves_raw_first:nNTF \{\langle name \rangle\} \langle tl\ variable \rangle \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	--

Append the first index of the $\langle name \rangle$ slide range to the $\langle tl\ variable \rangle$. Cache the result. Execute $\langle true\ code \rangle$ when there is a $\langle first \rangle$, $\langle false\ code \rangle$ otherwise.

```

398 \cs_set:Npn \__beanoves_return_true:nnN #1 #2 #3 {
399   \tl_if_empty:NTF \l_ans_tl {
400     \__beanoves_group_end:
401   \__beanoves_DEBUG:n { RETURN_FALSE/key=#1/type=#2/EMPTY }
402   \__beanoves_gremove:n { #1//#2 }
403   \prg_return_false:
404 } {
405   \__beanoves_fp_round:N \l_ans_tl
406   \__beanoves_gput:nV { #1//#2 } \l_ans_tl
407   \exp_args:NNNV
408   \__beanoves_group_end:
409   \tl_put_right:Nn #3 \l_ans_tl
410 \__beanoves_DEBUG:x { RETURN_TRUE/key=#1/type=#2/ans=\l_ans_tl/ }
411 \prg_return_true:
412 }
413 }
414 \cs_set:Npn \__beanoves_return_false:nn #1 #2 {
415   \__beanoves_DEBUG:n { RETURN_FALSE/key=#1/type=#2/ }
416   \__beanoves_group_end:
417   \__beanoves_gremove:n { #1//#2 }
418   \prg_return_false:
419 }
420 \prg_new_conditional:Npnn \__beanoves_raw_first:nN #1 #2 { T, F, TF } {
421   \__beanoves_DEBUG:x { RAW_FIRST/
422     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2/}
423   \__beanoves_if_in:nTF { #1//A } {
424     \__beanoves_DEBUG:n { RAW_FIRST/#1/CACHED }
425     \tl_put_right:Nx #2 { \__beanoves_item:n { #1//A } }
426     \prg_return_true:
427   } {
428     \__beanoves_DEBUG:n { RAW_FIRST/key=#1/NOT_CACHED }
429     \__beanoves_group_begin:
430     \tl_clear:N \l_ans_tl
431     \__beanoves_get:nNTF { #1/A } \l_a_tl {
432       \__beanoves_DEBUG:x { RAW_FIRST/key=#1/A=\l_a_tl }
433       \__beanoves_if_append:VNTF \l_a_tl \l_ans_tl {
434         \__beanoves_return_true:nnN { #1 } A #2
435       } {
436         \__beanoves_return_false:nn { #1 } A
437       }
438     } {
439       \__beanoves_DEBUG:n { RAW_FIRST/key=#1/A/F }
440       \__beanoves_get:nNTF { #1/L } \l_a_tl {
441         \__beanoves_DEBUG:n { RAW_FIRST/key=#1/L=\l_a_tl }
442         \__beanoves_get:nNTF { #1/Z } \l_b_tl {
443           \__beanoves_DEBUG:n { RAW_FIRST/key=#1/Z=\l_b_tl }
444           \__beanoves_if_append:xNTF {
445             \l_b_tl - ( \l_a_tl ) + 1
446           } \l_ans_tl {
447             \__beanoves_return_true:nnN { #1 } A #2
448           } {
449             \__beanoves_return_false:nn { #1 } A
450           }
451         } {

```

```

452 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/Z/F/ }
453 \__beanoves_return_false:nn { #1 } A
454 }
455 } {
456 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/L/F/ }
457 \__beanoves_return_false:nn { #1 } A
458 }
459 }
460 }
461 }
462 \prg_generate_conditional_variant:Nnn
463 \__beanoves_raw_first:nN { VN, xN } { T, F, TF }

```

__beanoves_if_first:nNTF __beanoves_if_first:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Append the first index of the <name> slide range to the <tl variable>. If no first index was explicitly given, use the counter when available and 1 hen not. Cache the result. Execute <true code> when there is a <first>, <false code> otherwise.

```

464 \prg_new_conditional:Npnn \__beanoves_if_first:nN #1 #2 { T, F, TF } {
465 \__beanoves_DEBUG:x { IF_FIRST/\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2}
466 \__beanoves_raw_first:nNTF { #1 } #2 {
467 \prg_return_true:
468 } {
469 \__beanoves_get:nNTF { #1/C } \l_a_tl {
470 \__beanoves_DEBUG:n { IF_FIRST/#1/C/T/\l_a_tl }
471 \bool_set_true:N \l_no_counter_bool
472 \__beanoves_if_append:xNTF \l_a_tl \l_ans_tl {
473 \__beanoves_return_true:nnN { #1 } A #2
474 } {
475 \__beanoves_return_false:nn { #1 } A
476 }
477 } {
478 \regex_match:NnTF \c__beanoves_A_key_Z_regex { #1 } {
479 \__beanoves_gput:nn { #1/A } { 1 }
480 \tl_set:Nn #2 { 1 }
481 \__beanoves_DEBUG:x{IF_FIRST_MATCH:
482 key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /}
483 \__beanoves_return_true:nnN { #1 } A #2
484 } {
485 \__beanoves_DEBUG:x{IF_FIRST_NO_MATCH:
486 key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /}
487 \__beanoves_return_false:nn { #1 } A
488 }
489 }
490 }
491 }

```

__beanoves_first:nN __beanoves_first:nN {<name>} <tl variable>

__beanoves_first:VN

Append the start of the <name> slide range to the <tl variable>. Cache the result.

```

492 \cs_new:Npn \__beanoves_first:nN #1 #2 {
493 \__beanoves_if_first:nNF { #1 } #2 {
494 \msg_error:nnn { beanoves } { :n } { Range-with-no-first:~#1 }

```

```

495 }
496 }
497 \cs_generate_variant:Nn \__beanoves_first:nN { VN }

```

```

\__beanoves_raw_length:nTF \__beanoves_raw_length:nTF {<name>} <tl variable> {<true code>} {<false
code>}

```

Append the length of the $\langle name \rangle$ slide range to $\langle tl variable \rangle$ Execute $\langle true code \rangle$ when there is a $\langle length \rangle$, $\langle false code \rangle$ otherwise.

```

498 \prg_new_conditional:Npnn \__beanoves_raw_length:nN #1 #2 { T, F, TF } {
499 \__beanoves_DEBUG:n { RAW_LENGTH/#1 }
500 \__beanoves_if_in:nTF { #1//L } {
501 \tl_put_right:Nx #2 { \__beanoves_item:n { #1//L } }
502 \__beanoves_DEBUG:x { RAW_LENGTH/CACHED/#1/\__beanoves_item:n { #1//L } }
503 \prg_return_true:
504 } {
505 \__beanoves_DEBUG:x { RAW_LENGTH/NOT_CACHED/key=#1/ }
506 \__beanoves_gput:nn { #1//L } { 0 }
507 \__beanoves_group_begin:
508 \tl_clear:N \l_ans_tl
509 \__beanoves_if_in:nTF { #1/L } {
510 \__beanoves_if_append:xNTF {
511 \__beanoves_item:n { #1/L }
512 } \l_ans_tl {
513 \__beanoves_return_true:nnN { #1 } L #2
514 } {
515 \__beanoves_return_false:nn { #1 } L
516 }
517 } {
518 \__beanoves_get:nTF { #1/A } \l_a_tl {
519 \__beanoves_get:nTF { #1/Z } \l_b_tl {
520 \__beanoves_if_append:xNTF {
521 \l_b_tl - (\l_a_tl) + 1
522 } \l_ans_tl {
523 \__beanoves_return_true:nnN { #1 } L #2
524 } {
525 \__beanoves_return_false:nn { #1 } L
526 }
527 } {
528 \__beanoves_return_false:nn { #1 } L
529 }
530 } {
531 \__beanoves_return_false:nn { #1 } L
532 }
533 }
534 }
535 }
536 \prg_generate_conditional_variant:Nnn
537 \__beanoves_raw_length:nN { VN } { T, F, TF }

```

```

\__beanoves_length:nN
\__beanoves_length:VN

```

```

\__beanoves_length:nN {<name>} <tl variable>

```

Append the length of the $\langle name \rangle$ slide range to $\langle tl variable \rangle$

```

538 \cs_new:Npn \__beanoves_length:nN #1 #2 {
539   \__beanoves_raw_length:nNF { #1 } #2 {
540     \msg_error:nnn { beanoves } { :n } { Range~with~no~length:~#1 }
541   }
542 }
543 \cs_generate_variant:Nn \__beanoves_length:nN { VN }

```

```

\__beanoves_raw_last:nNTF \__beanoves_raw_last:nNTF {<name>} <tl variable> {<true code>} {<false code>}

```

Put the last index of the $\langle name \rangle$ range to the right of the $\langle tl variable \rangle$, when possible. Execute $\langle true code \rangle$ when a last index was given, $\langle false code \rangle$ otherwise.

```

544 \prg_new_conditional:Nppn \__beanoves_raw_last:nN #1 #2 { T, F, TF } {
545   \__beanoves_DEBUG:n { RAW_LAST/#1 }
546   \__beanoves_if_in:nTF { #1//Z } {
547     \tl_put_right:Nx #2 { \__beanoves_item:n { #1//Z } }
548     \prg_return_true:
549   } {
550     \__beanoves_gput:nn { #1//Z } { 0 }
551     \__beanoves_group_begin:
552     \tl_clear:N \l_ans_tl
553     \__beanoves_if_in:nTF { #1/Z } {
554       \__beanoves_DEBUG:x { NORMAL_RAW_LAST:~\__beanoves_item:n { #1/Z } }
555       \__beanoves_if_append:xNTF {
556         \__beanoves_item:n { #1/Z }
557       } \l_ans_tl {
558         \__beanoves_return_true:nnN { #1 } Z #2
559       } {
560         \__beanoves_return_false:nn { #1 } Z
561       }
562     } {
563       \__beanoves_get:nNTF { #1/A } \l_a_tl {
564         \__beanoves_get:nNTF { #1/L } \l_b_tl {
565           \__beanoves_if_append:xNTF {
566             \l_a_tl + (\l_b_tl) - 1
567           } \l_ans_tl {
568             \__beanoves_return_true:nnN { #1 } Z #2
569           } {
570             \__beanoves_return_false:nn { #1 } Z
571           }
572         } {
573           \__beanoves_return_false:nn { #1 } Z
574         }
575       } {
576         \__beanoves_return_false:nn { #1 } Z
577       }
578     }
579   }
580 }
581 \prg_generate_conditional_variant:Nnn
582   \__beanoves_raw_last:nN { VN } { T, F, TF }

```

```

\__beanoves_last:nN \__beanoves_last:nN {<name>} <tl variable>

```

```

\__beanoves_last:VN
Append the last index of the  $\langle name \rangle$  slide range to  $\langle tl variable \rangle$ 

```

```

583 \cs_new:Npn \__beanoves_last:nN #1 #2 {
584   \__beanoves_raw_last:nNF { #1 } #2 {
585     \msg_error:nnn { beanoves } { :n } { Range-with-no~last:~#1 }
586   }
587 }
588 \cs_generate_variant:Nn \__beanoves_last:nN { VN }

```

```

\__beanoves_if_next:nNTF \__beanoves_if_next:nNTF {<name>} <tl variable> {<true code>} {<false code>}

```

Append the index after the <name> slide range to the <tl variable>. Execute <true code> when there is a <next> index, <false code> otherwise.

```

589 \prg_new_conditional:Npnn \__beanoves_if_next:nN #1 #2 { T, F, TF } {
590   \__beanoves_if_in:nTF { #1//N } {
591     \tl_put_right:Nx #2 { \__beanoves_item:n { #1//N } }
592     \prg_return_true:
593   } {
594     \__beanoves_group_begin:
595     \cs_set:Npn \__beanoves_return_true: {
596       \tl_if_empty:NTF \l_ans_tl {
597         \__beanoves_group_end:
598         \prg_return_false:
599       } {
600         \__beanoves_fp_round:N \l_ans_tl
601         \__beanoves_gput:nV { #1//N } \l_ans_tl
602         \exp_args:NNNV
603         \__beanoves_group_end:
604         \tl_put_right:Nn #2 \l_ans_tl
605         \prg_return_true:
606       }
607     }
608     \cs_set:Npn \__beanoves_return_false: {
609       \__beanoves_group_end:
610       \prg_return_false:
611     }
612     \tl_clear:N \l_a_tl
613     \__beanoves_raw_last:nNTF { #1 } \l_a_tl {
614       \__beanoves_if_append:xNTF {
615         \l_a_tl + 1
616       } \l_ans_tl {
617         \__beanoves_return_true:
618       } {
619         \__beanoves_return_false:
620       }
621     } {
622       \__beanoves_return_false:
623     }
624   }
625 }
626 \prg_generate_conditional_variant:Nnn
627 \__beanoves_if_next:nN { VN } { T, F, TF }

```

```

\__beanoves_next:nN \__beanoves_next:nN {<name>} <tl variable>

```

```

\__beanoves_next:VN

```

Append the index after the <name> slide range to the <tl variable>.


```

628 \cs_new:Npn \__beanoves_next:nN #1 #2 {
629   \__beanoves_if_next:nNF { #1 } #2 {
630     \msg_error:nnn { beanoves } { :n } { Range-with-no-next:~#1 }
631   }
632 }
633 \cs_generate_variant:Nn \__beanoves_next:nN { VN }

```

```

\__beanoves_if_free_counter:nNTF \__beanoves_if_free_counter:nNTF <tl variable> {<name>} {<true code>}
\__beanoves_if_free_counter:nVTF {<false code>}

```

Set the *<tl variable>* to the value of the counter associated to the *{<name>}* slide range.

```

634 \prg_new_conditional:Nppn \__beanoves_if_free_counter:Nn #1 #2 { T, F, TF } {
635   \__beanoves_DEBUG:x { IF_FREE: \string #1/
636     key=\tl_to_str:n{#2}/value=\__beanoves_item:n {#2/C}/ }
637   \__beanoves_group_begin:
638   \tl_clear:N \l_ans_tl
639   \__beanoves_get:nNF { #2/C } \l_ans_tl {
640     \__beanoves_raw_first:nNF { #2 } \l_ans_tl {
641       \__beanoves_raw_last:nNF { #2 } \l_ans_tl { }
642     }
643   }
644   \__beanoves_DEBUG:x { IF_FREE_2:\string \l_ans_tl=\tl_to_str:V \l_ans_tl/}
645   \tl_if_empty:NTF \l_ans_tl {
646     \__beanoves_group_end:
647     \regex_match:NnTF \c__beanoves_A_key_Z_regex { #2 } {
648       \__beanoves_gput:nn { #2/C } { 1 }
649       \tl_set:Nn #1 { 1 }
650     }
651     \__beanoves_DEBUG:x { IF_FREE_MATCH_TRUE:\string #1=\tl_to_str:V #1 /
652       key=\tl_to_str:n{#2} }
653     \prg_return_true:
654   } {
655     \__beanoves_DEBUG:x { IF_FREE_NO_MATCH_FALSE: \string #1=\tl_to_str:V #1/
656       key=\tl_to_str:n{#2} }
657     \prg_return_false:
658   } {
659     \__beanoves_gput:nV { #2/C } \l_ans_tl
660     \exp_args:NNNV
661     \__beanoves_group_end:
662     \tl_set:Nn #1 \l_ans_tl
663   }
664   \__beanoves_DEBUG:x { IF_FREE_TRUE(2): \string #1=\tl_to_str:V #1 /
665     key=\tl_to_str:n{#2} }
666   \prg_return_true:
667 }
668 \prg_generate_conditional_variant:Nnn
669   \__beanoves_if_free_counter:Nn { NV } { T, F, TF }

```

```

\__beanoves_if_counter:nNTF \__beanoves_if_counter:nNTF {<name>} <tl variable> {<true code>} {<false
\__beanoves_if_counter:nVTF code>}

```

Append the value of the counter associated to the *{<name>}* slide range to the right of *<tl variable>*. The value always lays in between the range, whenever possible.

```

670 \prg_new_conditional:Npnn \__beanoves_if_counter:nN #1 #2 { T, F, TF } {
671 \__beanoves_DEBUG:x { IF_COUNTER:key=
672 \tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 }
673 \__beanoves_group_begin:
674 \__beanoves_if_free_counter:NnTF \l_ans_tl { #1 } {

```

If there is a $\langle first \rangle$, use it to bound the result from below.

```

675 \tl_clear:N \l_a_tl
676 \__beanoves_raw_first:nNT { #1 } \l_a_tl {
677 \fp_compare:nNnT { \l_ans_tl } < { \l_a_tl } {
678 \tl_set:NV \l_ans_tl \l_a_tl
679 }
680 }

```

If there is a $\langle last \rangle$, use it to bound the result from above.

```

681 \tl_clear:N \l_a_tl
682 \__beanoves_raw_last:nNT { #1 } \l_a_tl {
683 \fp_compare:nNnT { \l_ans_tl } > { \l_a_tl } {
684 \tl_set:NV \l_ans_tl \l_a_tl
685 }
686 }
687 \exp_args:NNx
688 \__beanoves_group_end:
689 \__beanoves_fp_round:nN \l_ans_tl #2
690 \__beanoves_DEBUG:x {IF_COUNTER_TRUE:key=\tl_to_str:n{#1}/
691 \string #2=\tl_to_str:V #2 }
692 \prg_return_true:
693 } {
694 \__beanoves_DEBUG:x {IF_COUNTER_FALSE:key=\tl_to_str:n{#1}/
695 \string #2=\tl_to_str:V #2 }
696 \prg_return_false:
697 }
698 }
699 \prg_generate_conditional_variant:Nnn
700 \__beanoves_if_counter:nN { VN } { T, F, TF }

```

<u>__beanoves_if_index:nnNTF</u> <u>__beanoves_if_index:VVNTF</u> <u>__beanoves_if_index:nnnNTF</u>	<u>__beanoves_if_index:nnNTF</u> $\{\langle name \rangle\}$ $\{\langle integer path \rangle\}$ $\langle tl variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ <u>__beanoves_if_index:nnnNTF</u> $\{\langle name \rangle\}$ $\{\langle integer path \rangle\}$ $\{\langle integer shift \rangle\}$ $\langle tl variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
--	---

Append the index associated to the $\{\langle name \rangle\}$ and $\{\langle integer path \rangle\}$ slide range to the right of $\langle tl variable \rangle$. When $\langle integer shift \rangle$ is 1, this is the first index, when $\langle integer shift \rangle$ is 2, this is the second index, and so on. When $\langle integer shift \rangle$ is 0, this is the index, before the first one, and so on. If the computation is possible, $\langle true code \rangle$ is executed, otherwise $\langle false code \rangle$ is executed. The computation may fail when too many recursion calls are made.

```

701 \prg_new_conditional:Npnn \__beanoves_if_index:nnN #1 #2 #3 { T, F, TF } {
702 \__beanoves_DEBUG:x { IF_INDEX:key=#1/index=#2/\string#3/ }
703 \__beanoves_group_begin:
704 \tl_set:Nn \l_name_tl { #1 }
705 \regex_split:nnNTF { \. } { #2 } \l_split_seq {
706 \seq_pop_left:NN \l_split_seq \l_a_tl
707 \seq_pop_right:NN \l_split_seq \l_a_tl

```

```

708 \seq_map_inline:Nn \l_split_seq {
709   \tl_set_eq:NN \l_b_tl \l_name_tl
710   \tl_put_right:Nn \l_b_tl { . ##1 }
711   \exp_args:Nx
712   \__beanoves_get:nN { \l_b_tl / A } \l_c_tl
713   \quark_if_no_value:NTF \l_c_tl {
714     \tl_set_eq:NN \l_name_tl \l_b_tl
715   } {
716     \tl_set_eq:NN \l_name_tl \l_c_tl
717   }
718 \__beanoves_DEBUG:x { IF_INDEX_SPLIT:##1/
719   \string\l_name_tl=\tl_to_str:N \l_name_tl}
720 }
721 \tl_clear:N \l_b_tl
722 \__beanoves_raw_first:xNTF { \l_name_tl.\l_a_tl } \l_b_tl {
723   \tl_set_eq:NN \l_ans_tl \l_b_tl
724 } {
725   \tl_clear:N \l_b_tl
726   \__beanoves_raw_first:VNTF \l_name_tl \l_b_tl {
727     \tl_set_eq:NN \l_ans_tl \l_b_tl
728   } {
729     \tl_set_eq:NN \l_ans_tl \l_name_tl
730   }
731   \tl_put_right:Nx \l_ans_tl { + (\l_a_tl) - 1}
732 }
733 \__beanoves_DEBUG:x { IF_INDEX_TRUE:key=#1/index=#2/
734   \string\l_ans_tl=\tl_to_str:N \l_ans_tl }
735   \exp_args:NNx
736   \__beanoves_group_end:
737   \__beanoves_fp_round:nN \l_ans_tl #3
738   \prg_return_true:
739 } {
740 \__beanoves_DEBUG:x { IF_INDEX_FALSE:key=#1/index=#2/ }
741   \prg_return_false:
742 }
743 }
744 \prg_generate_conditional_variant:Nnn
745 \__beanoves_if_index:nnN { VVN } { T, F, TF }

```

$\backslash_beanoves_if_index:nnnNTF$ $\backslash_beanoves_if_index:VVVNTF$	$\backslash_beanoves_if_index:nnnNTF \{ \langle name \rangle \} \{ \langle integer\ path \rangle \} \{ \langle integer\ index \rangle \} \langle tl$ $variable \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
--	--

Append the value of the counter associated to the $\{ \langle name \rangle \}$ slide range to the right of $\langle tl\ variable \rangle$. The value always lays in between the range, whenever possible. If the computation is possible, $\langle true\ code \rangle$ is executed, otherwise $\langle false\ code \rangle$ is executed. The computation may fail when too many recursion calls are made.

```

746 \prg_new_conditional:Npnn \__beanoves_if_index_original:nnnN #1 #2 #3 #4 { T, F, TF } {
747   \__beanoves_DEBUG:x { IF_INDEX:key=#1/path=#2/index=#3/\string#4/ }
748   \__beanoves_group_begin:
749   \tl_set:Nn \l_a_tl { #1 }
750   \regex_split:nnNTF { \. } { #2 } \l_split_seq {
751     \cs_set:Npn \:n ##1 {
752       \tl_set_eq:NN \l_b_tl \l_a_tl

```

```

753     \tl_put_right:Nn \l_b_tl { . ##1 }
754     \exp_args:Nx
755     \__beanoves_get:nNTF { \l_b_tl / A } \l_c_tl {
756         \exp_args:NNx
757         \regex_match:NnTF \c__beanoves_A_key_Z_regex \l_c_tl {
758             \tl_set_eq:NN \l_a_tl \l_c_tl
759         } {
760             \cs_set:Npn \:n ####1 {
761                 \tl_set_eq:NN \l_b_tl \l_a_tl
762                 \tl_put_right:Nn \l_b_tl { . ####1 }
763                 \tl_set_eq:NN \l_a_tl \l_b_tl
764             }
765             \string\l_a_tl=\tl_to_str:N \l_a_tl
766         }
767     } {
768         \tl_set_eq:NN \l_a_tl \l_b_tl
769     }
770     \__beanoves_DEBUG:x { IF_INDEX_SPLIT:##1/
771         \string\l_a_tl=\tl_to_str:N \l_a_tl
772     }
773     \seq_map_function:NN \l_split_seq \:n
774     \tl_clear:N \l_b_tl
775     \__beanoves_raw_first:VNTF \l_a_tl \l_b_tl {
776         \tl_set_eq:NN \l_ans_tl \l_b_tl
777     } {
778         \tl_set_eq:NN \l_ans_tl \l_a_tl
779     }
780     \tl_put_right:Nx \l_ans_tl { + (#3) - 1}
781     \__beanoves_DEBUG:x { IF_INDEX_TRUE:key=#1/path=#2/index=#3/
782         \string\l_ans_tl=\tl_to_str:N \l_ans_tl }
783     \exp_args:NNx
784     \__beanoves_group_end:
785     \__beanoves_fp_round:nN \l_ans_tl #4
786     \prg_return_true:
787 } {
788     \tl_if_empty:nTF { #2 } {
789         \tl_clear:N \l_b_tl
790         \__beanoves_raw_first:VNTF \l_a_tl \l_b_tl {
791             \tl_set_eq:NN \l_ans_tl \l_b_tl
792         } {
793             \tl_set_eq:NN \l_ans_tl \l_a_tl
794         }
795     }
796     \tl_put_right:Nx \l_ans_tl { + (#3) - 1}
797     \__beanoves_DEBUG:x { IF_INDEX_TRUE:key=#1/path=#2/index=#3/
798         \string\l_ans_tl=\tl_to_str:N \l_ans_tl }
799     \exp_args:NNx
800     \__beanoves_group_end:
801     \__beanoves_fp_round:nN \l_ans_tl #4
802     \prg_return_true:
803 } {
804     \__beanoves_DEBUG:x { IF_INDEX_FALSE:key=#1/path=#2/index=#3/ }
805     \prg_return_false:
806 }

```

```

807 }
808 }
809 \prg_new_conditional:Npnn \__beanoves_if_index:nnnN #1 #2 #3 #4 { T, F, TF } {
810 \__beanoves_DEBUG:x { IF_INDEX:key=#1/path=#2/index=#3/\string#4/ }
811 \__beanoves_group_begin:
812 \tl_set:Nn \l_a_tl { #1 }
813 \seq_set_split:Nnn \l_split_seq { . } { #2 }
814 \seq_remove_all:Nn \l_split_seq {}
815 \__beanoves_DEBUG:x {SPLIT_SEQ:/\seq_use:Nn \l_split_seq / /}
816 \cs_set:Npn \:n ##1 {
817 \tl_set_eq:NN \l_b_tl \l_a_tl
818 \tl_put_right:Nn \l_b_tl { . ##1 }
819 \exp_args:Nx
820 \__beanoves_get:nNTF { \l_b_tl / A } \l_c_tl {
821 \exp_args:NNx
822 \regex_match:NnTF \c__beanoves_A_key_Z_regex \l_c_tl {
823 \tl_set_eq:NN \l_a_tl \l_c_tl
824 } {
825 \tl_set_eq:NN \l_a_tl \l_b_tl
826 }
827 } {
828 \tl_set_eq:NN \l_a_tl \l_b_tl
829 }
830 \__beanoves_DEBUG:x { IF_INDEX_SPLIT:##1/
831 \string\l_a_tl=\tl_to_str:N \l_a_tl}
832 }
833 \seq_map_function:NN \l_split_seq \:n
834 \tl_clear:N \l_b_tl
835 \__beanoves_raw_first:VNTF \l_a_tl \l_b_tl {
836 \tl_set_eq:NN \l_ans_tl \l_b_tl
837 \tl_put_right:Nx \l_ans_tl { + (#3) - 1}
838 \__beanoves_DEBUG:x { IF_INDEX_TRUE:key=#1/path=#2/index=#3/
839 \string\l_ans_tl=\tl_to_str:N \l_ans_tl }
840 \exp_args:NNx
841 \__beanoves_group_end:
842 \__beanoves_fp_round:nN \l_ans_tl #4
843 \prg_return_true:
844 } {
845 \__beanoves_DEBUG:x { IF_INDEX_FALSE:key=#1/path=#2/index=#3/ }
846 \prg_return_false:
847 }
848 }
849 \prg_generate_conditional_variant:Nnn
850 \__beanoves_if_index:nnnN { VVVN } { T, F, TF }

```

<u>__beanoves_if_incr:nnTF</u>	__beanoves_if_incr:nnTF {<name>} {<offset>} {<true code>} {<false
<u>__beanoves_if_incr:nnTF</u>	code>}
<u>__beanoves_if_incr:(VnN VVN)TF</u>	__beanoves_if_incr:nnNTF {<name>} {<offset>} {<tl variable>} {<true code>}
	{<false code>}

Increment the free counter position accordingly. When requested, put the result in the *<tl variable>*. The result will lay within the declared range.

```

851 \prg_new_conditional:Npnn \__beanoves_if_incr:nn #1 #2 { T, F, TF } {

```

```

852 \__beanoves_DEBUG:x { IF_INCR:\tl_to_str:n{#1}/\tl_to_str:n{#2} }
853 \__beanoves_group_begin:
854 \tl_clear:N \l_a_tl
855 \__beanoves_if_free_counter:NnTF \l_a_tl { #1 } {
856   \tl_clear:N \l_b_tl
857   \__beanoves_if_append:xNTF { \l_a_tl + (#2) } \l_b_tl {
858     \__beanoves_fp_round:N \l_b_tl
859     \__beanoves_gput:nV { #1/C } \l_b_tl
860     \__beanoves_group_end:
861 \__beanoves_DEBUG:x { IF_INCR_TRUE:#1/#2 }
862   \prg_return_true:
863 } {
864   \__beanoves_group_end:
865 \__beanoves_DEBUG:x { IF_INCR_FALSE(1):#1/#2 }
866   \prg_return_false:
867 }
868 } {
869   \__beanoves_group_end:
870 \__beanoves_DEBUG:x { IF_INCR_FALSE(2):#1/#2 }
871   \prg_return_false:
872 }
873 }
874 \prg_new_conditional:Npnn \__beanoves_if_incr:nnN #1 #2 #3 { T, F, TF } {
875   \__beanoves_if_incr:nnTF { #1 } { #2 } {
876     \__beanoves_if_counter:nNTF { #1 } #3 {
877       \prg_return_true:
878     } {
879       \prg_return_false:
880     }
881   } {
882     \prg_return_false:
883   }
884 }
885 \prg_generate_conditional_variant:Nnn
886 \__beanoves_if_incr:nnN { VnN, VVN } { T, F, TF }

```

__beanoves_if_range:nNTF __beanoves_if_range:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Append the range of the <name> slide range to the <tl variable>. Execute <true code> when there is a <range>, <false code> otherwise.

```

887 \prg_new_conditional:Npnn \__beanoves_if_range:nN #1 #2 { T, F, TF } {
888   \__beanoves_DEBUG:x{ RANGE:key=#1/\string#2/}
889   \bool_if:NNTF \l__beanoves_no_range_bool {
890     \prg_return_false:
891   } {
892     \__beanoves_if_in:nTF { #1/ } {
893       \tl_put_right:Nn { 0-0 }
894     } {
895       \__beanoves_group_begin:
896       \tl_clear:N \l_a_tl
897       \tl_clear:N \l_b_tl
898       \tl_clear:N \l_ans_tl
899       \__beanoves_raw_first:nNTF { #1 } \l_a_tl {
900         \__beanoves_raw_last:nNTF { #1 } \l_b_tl {

```

```

901         \exp_args:NNNx
902         \__beanoves_group_end:
903         \tl_put_right:Nn #2 { \l_a_tl - \l_b_tl }
904     \__beanoves_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
905     \prg_return_true:
906 } {
907     \exp_args:NNNx
908     \__beanoves_group_end:
909     \tl_put_right:Nn #2 { \l_a_tl - }
910 \__beanoves_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
911     \prg_return_true:
912 }
913 } {
914     \__beanoves_raw_last:nNTF { #1 } \l_b_tl {
915 \__beanoves_DEBUG:x{ RANGE_TRUE_Z:key=#1/\string#2=#2/}
916         \exp_args:NNNx
917         \__beanoves_group_end:
918         \tl_put_right:Nn #2 { - \l_b_tl }
919         \prg_return_true:
920     } {
921 \__beanoves_DEBUG:x{ RANGE_FALSE:key=#1/}
922         \__beanoves_group_end:
923         \prg_return_false:
924     }
925 }
926 }
927 }
928 }
929 \prg_generate_conditional_variant:Nnn
930 \__beanoves_if_range:nN { VN } { T, F, TF }

```

```

\__beanoves_range:nN
\__beanoves_range:VN

```

`__beanoves_range:nN {<name>} <tl variable>`
Append the range of the <name> slide range to the <tl variable>.

```

931 \cs_new:Npn \__beanoves_range:nN #1 #2 {
932     \__beanoves_if_range:nNF { #1 } #2 {
933         \msg_error:nnn { beanoves } { :n } { No-range~available:~#1 }
934     }
935 }
936 \cs_generate_variant:Nn \__beanoves_range:nN { VN }

```

5.3.7 Evaluation

```

\__beanoves_resolve:nnN
\__beanoves_resolve:VVN
\__beanoves_resolve:nnNN
\__beanoves_resolve:VVNN

```

`__beanoves_resolve:nnN {<name>} {<path>} <tl variable>`
`__beanoves_resolve:nnNN {<name>} {<path>} <tl name variable> <tl last variable>`
Resolve the <name> and <path> into a key that is put into the <tl name variable>. <name_{012n}> is turned into <name_{12n}> where <name₀₁> is <name_{123n}> where <name₁₂> is <name₂

```

937 \cs_new:Npn \__beanoves_resolve:nnN #1 #2 #3 {
938     \__beanoves_group_begin:

```

```

939 \tl_set:Nn \l_a_tl { #1 }
940 \regex_split:nnNT { \. } { #2 } \l_split_seq {
941   \seq_pop_left:NN \l_split_seq \l_b_tl
942   \cs_set:Npn \:n ##1 {
943     \tl_set_eq:NN \l_b_tl \l_a_tl
944     \tl_put_right:Nn \l_b_tl { . ##1 }
945     \exp_args:Nx
946     \__beanoves_get:nNTF { \l_b_tl / A } \l_c_tl {
947       \exp_args:NNx
948       \regex_match:NnTF \c__beanoves_A_key_Z_regex \l_c_tl {
949         \tl_set_eq:NN \l_a_tl \l_c_tl
950       } {
951         \cs_set:Npn \:n ####1 {
952           \tl_set_eq:NN \l_b_tl \l_a_tl
953           \tl_put_right:Nn \l_b_tl { . ####1 }
954           \tl_set_eq:NN \l_a_tl \l_b_tl
955         }
956       }
957     } {
958       \tl_set_eq:NN \l_a_tl \l_b_tl
959     }
960   }
961   \seq_map_function:NN \l_split_seq \:n
962 }
963 \exp_args:NNNV
964 \__beanoves_group_end:
965 \tl_set:Nn #3 \l_a_tl
966 }
967 \cs_generate_variant:Nn \__beanoves_resolve:nnN { VVN }
968 \cs_new:Npn \__beanoves_tl_put_right_braced:Nn #1 #2 {
969   \tl_put_right:Nn #1 { { #2 } }
970 }
971 \cs_generate_variant:Nn \__beanoves_tl_put_right_braced:Nn { NV }
972 \cs_new:Npn \__beanoves_resolve:nnNN #1 #2 #3 #4 {
973   \__beanoves_group_begin:
974   \regex_extract_once:nnNT { (\.\d+)*? (\.\d+) \Z } { #2 } \l_match_seq {
975     \exp_args:Nnx
976     \__beanoves_resolve:nnN { #1 } { \seq_item:Nn \l_match_seq 2 } \l_name_tl
977     \tl_set:Nn \l_a_tl {
978       \tl_set:Nn #3
979     }
980     \exp_args:NNV
981     \__beanoves_tl_put_right_braced:Nn \l_a_tl \l_name_tl
982     \tl_put_right:Nn \l_a_tl {
983       \tl_set:Nn #4
984     }
985     \exp_args:NNx
986     \__beanoves_tl_put_right_braced:Nn \l_a_tl {
987       \seq_item:Nn \l_match_seq 3
988     }
989   }
990   \exp_last_unbraced:NV
991   \__beanoves_group_end:
992   \l_a_tl

```



```

993 }
994 \cs_generate_variant:Nn \__beanoves_resolve:nnNN { VVNN }

```

$\backslash_beanoves_if_append:nNTF$ $\backslash_beanoves_if_append:(VN xN)TF$	$\backslash_beanoves_if_append:nNTF \{ \langle integer\ expression \rangle \} \langle tl\ variable \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
---	---

Evaluates the $\langle integer\ expression \rangle$, replacing all the named specifications by their static counterpart then put the result to the right of the $\langle tl\ variable \rangle$. Executed within a group. Heavily used by $\backslash_beanoves_eval_query:nN$, where $\langle integer\ expression \rangle$ was initially enclosed in ‘ $?(...)$ ’. Local variables:

$\backslash l_ans_tl$ To feed $\langle tl\ variable \rangle$ with.

(End definition for $\backslash l_ans_tl$. This variable is documented on page 37.)

$\backslash l_split_seq$ The sequence of caught query groups and non queries.

(End definition for $\backslash l_split_seq$. This variable is documented on page 37.)

$\backslash l_beanoves_split_int$ Is the index of the non queries, before all the caught groups.

(End definition for $\backslash l_beanoves_split_int$.)

$\backslash l_name_tl$ Storage for $\backslash l_split_seq$ items that represent names.

(End definition for $\backslash l_name_tl$. This variable is documented on page 37.)

$\backslash l_path_tl$ Storage for $\backslash l_split_seq$ items that represent integer paths.

(End definition for $\backslash l_path_tl$. This variable is documented on page 37.)

Catch circular definitions.

```

995 \prg_new_conditional:Npnn \__beanoves_if_append:nN #1 #2 { T, F, TF } {
996   \__beanoves_DEBUG:x { IF_APPEND:\tl_to_str:n { #1 } / \string #2}
997   \int_gdecr:N \g__beanoves_append_int
998   \int_compare:nNnTF \g__beanoves_append_int > 0 {
999     \__beanoves_DEBUG:x { IF_APPEND...}
1000     \__beanoves_group_begin:

```

Local variables:

```

1001   \int_zero:N \l__beanoves_split_int
1002   \seq_clear:N \l_split_seq
1003   \tl_clear:N \l_name_tl
1004   \tl_clear:N \l_path_tl
1005   \tl_clear:N \l_group_tl
1006   \tl_clear:N \l_ans_tl
1007   \tl_clear:N \l_a_tl

```

Implementation:

```

1008   \regex_split:NnN \c__beanoves_split_regex { #1 } \l_split_seq
1009   \__beanoves_DEBUG:x { IF_APPEND_SPLIT_SEQ: / \seq_use:Nn \l_split_seq / / }
1010   \int_set:Nn \l__beanoves_split_int { 1 }
1011   \tl_set:Nx \l_ans_tl {
1012     \seq_item:Nn \l_split_seq { \l__beanoves_split_int }
1013   }

```

```
\switch:nTF {<capture group number>} {<black code>} {<white code>}
```

Helper function to locally set the `\l_group_tl` variable to the captured group *<capture group number>* and branch.

```

1014 \cs_set:Npn \switch:nTF ##1 ##2 ##3 ##4 {
1015   \tl_set:Nx ##2 {
1016     \seq_item:Nn \l_split_seq { \l__beanoves_split_int + ##1 }
1017   }
1018   \__beanoves_DEBUG:x { IF_APPEND_SWITCH/##1/\string##2/\tl_to_str:N##2/}
1019   \tl_if_empty:NTF ##2 { %SWITCH~APPEND~WHITE/##1/\
1020     ##4 } { %SWITCH~APPEND~BLACK/##1/\
1021     ##3
1022   }
1023 }
```

`\prg_return_true:` and `\prg_return_false:` are wrapped locally to close the group and return the proper value.

```

1024 \cs_set:Npn \__beanoves_return_true: {
1025   \__beanoves_fp_round:
1026   \exp_args:NNNV
1027   \__beanoves_group_end:
1028   \tl_put_right:Nn #2 \l_ans_tl
1029   \__beanoves_DEBUG:x { IF_APPEND_TRUE:\tl_to_str:n { #1 } /
1030   \string #2=\tl_to_str:V #2 }
1031   \prg_return_true:
1032 }
1033 \cs_set:Npn \__beanoves_fp_round: {
1034   \__beanoves_fp_round:N \l_ans_tl
1035 }
1036 \cs_set:Npn \next: {
1037   \__beanoves_return_true:
1038 }
1039 \cs_set:Npn \__beanoves_return_false: {
1040   \__beanoves_group_end:
1041   \__beanoves_DEBUG:x { IF_APPEND_FALSE:\tl_to_str:n { #1 } /
1042   \string #2=\tl_to_str:V #2 }
1043   \prg_return_false:
1044 }
1045 \cs_set:Npn \break: {
1046   \bool_set_false:N \l__beanoves_continue_bool
1047   \cs_set:Npn \next: {
1048     \__beanoves_return_false:
1049   }
1050 }
```

Main loop.

```

1051 \bool_set_true:N \l__beanoves_continue_bool
1052 \bool_while_do:Nn \l__beanoves_continue_bool {
1053   \int_compare:nNnTF {
1054     \l__beanoves_split_int } < { \seq_count:N \l_split_seq
1055   } {
1056     \switch:nTF 2 \l_name_tl {
```

- Case ++*<name>**<integer path>*.n.

```

1057         \switch:nNTF 3 \l_path_tl {
1058             \__beanoves_resolve:VWN \l_name_tl \l_path_tl \l_name_tl
1059         } { }
1060         \__beanoves_if_incr:VnNF \l_name_tl 1 \l_ans_tl {
1061             \break:
1062         }
1063     } {
1064         \switch:nNTF 5 \l_name_tl {

```

- Cases $\langle name \rangle \langle integer path \rangle \dots$

```

1065         \tl_set:Nn \l_b_tl {
1066             \switch:nNTF 6 \l_path_tl {
1067                 \__beanoves_resolve:VWN \l_name_tl \l_path_tl \l_name_tl
1068             } { }
1069         }
1070         \switch:nNTF 7 \l_a_tl {
1071             \l_b_tl
1072             \__beanoves_raw_length:VNF \l_name_tl \l_ans_tl {
1073                 \break:
1074             }

```

- Case ...length.

```

1075         } {
1076             \switch:nNTF 8 \l_a_tl {
1077                 \l_b_tl
1078                 \__beanoves_raw_last:VNF \l_name_tl \l_ans_tl {
1079                     \break:
1080                 }

```

- Case ...last.

```

1081         } {
1082             \switch:nNTF 9 \l_a_tl {
1083                 \l_b_tl
1084                 \__beanoves_if_next:VNF \l_name_tl \l_ans_tl {
1085                     \break:
1086                 }

```

- Case ...next.

```

1087         } {
1088             \switch:nNTF { 10 } \l_a_tl {
1089                 \l_b_tl
1090                 \__beanoves_if_range:VNTF \l_name_tl \l_ans_tl {
1091                     \cs_set_eq:NN \__beanoves_fp_round: \relax
1092                 } {
1093                     \break:
1094                 }

```

- Case ...range.

```

1095         } {
1096             \switch:nNTF { 11 } \l_a_tl {

```

- Case ...n.

```

1097             \l_b_tl
1098             \switch:nNTF { 12 } \l_a_tl {

```

- Case ...+=*<integer>*.

```

1099 \__beanoves_if_incr:VVNF \l_name_tl \l_a_tl \l_ans_tl {
1100   \break:
1101 }
1102   } {
1103 \__beanoves_DEBUG:x {+++++++~NAME=\l_name_tl}
1104   \__beanoves_if_counter:VNF \l_name_tl \l_ans_tl {
1105     \break:
1106   }
1107 }

1108   } {

```

- Case ...*<integer path>*.

```

1109             \switch:nNTF { 13 } \l_a_tl {
1110 \switch:nNTF 6 \l_path_tl { } {
1111   \tl_clear:N \l_path_tl
1112 }
1113 \__beanoves_if_index:VVNF \l_name_tl \l_path_tl \l_a_tl \l_ans_tl {
1114   \break:
1115 }
1116   } {
1117 \switch:nNTF 6 \l_path_tl {
1118   \__beanoves_if_index:VVNF \l_name_tl \l_path_tl \l_ans_tl {
1119     \break:
1120   }
1121 } {
1122   \exp_args:Nx
1123   \__beanoves_if_counter:nNF { \l_name_tl } \l_ans_tl {
1124     \break:
1125   }
1126 }
1127   }

1128   }
1129   }
1130   }
1131   }
1132   }
1133   } {

```

No name.

```

1134   }
1135 }
1136 \int_add:Nn \l__beanoves_split_int { 14 }
1137 \tl_put_right:Nx \l_ans_tl {
1138   \seq_item:Nn \l_split_seq { \l__beanoves_split_int }

```

```

1139     }
1140   } {
1141     \bool_set_false:N \l__beanoves_continue_bool
1142   }
1143 }
1144 \next:
1145 } {
1146   \msg_error:nnx
1147     { beanoves } { :n } { Too-many-calls:~\tl_to_str:n { #1 } }
1148   \__beanoves_return_false:
1149 }
1150 }
1151 \prg_generate_conditional_variant:Nnn
1152   \__beanoves_if_append:nN { VN, xN } { T, F, TF }

```

`__beanoves_if_eval_query:nNTF` `__beanoves_if_eval_query:nNTF {<overlay query> } <tl variable> {<true code>} {<false code>}`

Evaluates the single $\langle overlay\ query \rangle$, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of the $\langle seq\ variable \rangle$. Ranges are supported with the colon syntax. This is executed within a local group. Below are local variables and constants.

$\backslash l_a_{tl}$ Storage for the first index of a range.

(End definition for $\backslash l_a_{tl}$. This variable is documented on page 42.)

$\backslash l_b_{tl}$ Storage for the last index of a range, or its length.

(End definition for $\backslash l_b_{tl}$. This variable is documented on page 42.)

$\backslash c_{\text{beanoves_A_cln_Z_regex}}$ Used to parse slide range overlay specifications. Next are the capture groups.

(End definition for $\backslash c_{\text{beanoves_A_cln_Z_regex}}$.)

```

1153 \regex_const:Nn \c__beanoves_A_cln_Z_regex {
1154   \A \s* (?

```

- 2: $\langle first \rangle$

```

1155     ( [^:]* ) \s* :

```

- 3: second optional colon

```

1156     (:)? \s*

```

- 4: $\langle length \rangle$

```

1157     ( [^:]* )

```

- 5: standalone $\langle first \rangle$

```

1158     | ( [^:]+ )
1159   ) \s* \Z
1160 }

```

```

1161 \prg_new_conditional:Npnn \__beanoves_if_eval_query:nN #1 #2 { T, F, TF } {
1162 \__beanoves_DEBUG:x { EVAL_QUERY:#1/
1163   \tl_to_str:n{#1}/\string#2=\tl_to_str:N #2}
1164   \int_gset:Nn \g__beanoves_append_int { 128 }
1165   \regex_extract_once:NnNTF \c__beanoves_A_cln_Z_regex {
1166     #1
1167   } \l_match_seq {
1168 \__beanoves_DEBUG:x { EVAL_QUERY:#1/
1169   \string\l_match_seq/\seq_use:Nn \l_match_seq //}
1170   \bool_set_false:N \l__beanoves_no_counter_bool
1171   \bool_set_false:N \l__beanoves_no_range_bool

```

```

\switch:nNTF \switch:nNTF {<capture group number>} <tl variable> {<black code>} {<white code>}

```

Helper function to locally set the *<tl variable>* to the captured group *<capture group number>* and branch depending on the emptiness of this variable.

```

1172   \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1173 \__beanoves_DEBUG:x { EQ_SWITCH:##1/ }
1174   \tl_set:Nx ##2 {
1175     \seq_item:Nn \l_match_seq { ##1 }
1176   }
1177 \__beanoves_DEBUG:x { \string ##2/ \tl_to_str:N ##2/}
1178   \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1179 }
1180 \switch:nNTF 5 \l_a_tl {

```

Single expression

```

1181   \bool_set_false:N \l__beanoves_no_range_bool
1182   \__beanoves_if_append:VNTF \l_a_tl #2 {
1183     \prg_return_true:
1184   } {
1185     \prg_return_false:
1186   }
1187 } {
1188   \switch:nNTF 2 \l_a_tl {
1189     \switch:nNTF 4 \l_b_tl {
1190       \switch:nNTF 3 \l_c_tl {

```

<first>::<last> range

```

1191       \__beanoves_if_append:VNTF \l_a_tl #2 {
1192         \tl_put_right:Nn #2 { - }
1193       \__beanoves_if_append:VNTF \l_b_tl #2 {
1194         \prg_return_true:
1195       } {
1196         \prg_return_false:
1197       }
1198     } {
1199       \prg_return_false:
1200     }
1201   } {

```

<first>::<length> range

```

1202       \__beanoves_if_append:VNTF \l_a_tl #2 {
1203         \tl_put_right:Nx #2 { - }

```

```

1204         \tl_put_right:Nx \l_a_tl { + ( \l_b_tl ) - 1 }
1205         \__beanoves_if_append:VNTF \l_a_tl #2 {
1206             \prg_return_true:
1207         } {
1208             \prg_return_false:
1209         }
1210     } {
1211         \prg_return_false:
1212     }
1213 }
1214 } {
1215     <first>: and <first>:: range
1216     \__beanoves_if_append:VNTF \l_a_tl #2 {
1217         \tl_put_right:Nn #2 { - }
1218         \prg_return_true:
1219     } {
1220         \prg_return_false:
1221     }
1222 } {
1223     \switch:nNTF 4 \l_b_tl {
1224         \switch:nNTF 3 \l_c_tl {
1225             <last> range
1226             \tl_put_right:Nn #2 { - }
1227             \__beanoves_if_append:VNTF \l_a_tl #2 {
1228                 \prg_return_true:
1229             } {
1230                 \prg_return_false:
1231             }
1232         } {
1233             \msg_error:nxx { beanoves } { :n } { Syntax-error(Missing-first):~#1 }
1234         } {
1235             : or :: range
1236             \seq_put_right:Nn #2 { - }
1237         }
1238     }
1239 } {
Error
1240     \msg_error:nnn { beanoves } { :n } { Syntax-error:~#1 }
1241 }
1242 }

```

`_beanoves_eval:nN` `_beanoves_eval:nN {<overlay query list>} <tl variable>`

This is called by the *named overlay specifications* scanner. Evaluates the comma separated list of *<overlay query>*'s, replacing all the named overlay specifications and integer expressions by their static counterparts by calling `_beanoves_eval_query:nN`, then append the result to the right of the *<tl variable>*. This is executed within a local group. Below are local variables and constants used throughout the body of this function.

`\l_query_seq` Storage for a sequence of *<query>*'s obtained by splitting a comma separated list.

(End definition for `\l_query_seq`. This variable is documented on page 45.)

`\l_ans_seq` Storage of the evaluated result.

(End definition for `\l_ans_seq`. This variable is documented on page 45.)

`\c__beanoves_comma_regex` Used to parse slide range overlay specifications.

```

1243 \regex_const:Nn \c__beanoves_comma_regex { \s* , \s* }

```

(End definition for `\c__beanoves_comma_regex`.)

No other variable is used.

```

1244 \cs_new:Npn \_beanoves_eval:nN #1 #2 {
1245   \_beanoves_DEBUG:x {EVAL:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2}
1246   \_beanoves_group_begin:

```

Local variables declaration

```

1247   \seq_clear:N \l_ans_seq

```

In this main evaluation step, we evaluate the integer expression and put the result in a variable which content will be copied after the group is closed. We authorize comma separated expressions and *<first>::<last>* range expressions as well. We first split the expression around commas, into `\l_query_seq`.

```

1248   \regex_split:NnN \c__beanoves_comma_regex { #1 } \l_query_seq

```

Then each component is evaluated and the result is stored in `\l_ans_seq` that we have clear before use.

```

1249   \seq_map_inline:Nn \l_query_seq {
1250     \tl_clear:N \l_ans_tl
1251     \_beanoves_if_eval_query:nNTF { ##1 } \l_ans_tl {
1252       \seq_put_right:NV \l_ans_seq \l_ans_tl
1253     } {
1254       \seq_map_break:n {
1255         \msg_fatal:nnn { beanoves } { :n } { Circular~dependency~in~#1}
1256       }
1257     }
1258   }

```

We have managed all the comma separated components, we collect them back and append them to *<tl variable>*.

```

1259   \exp_args:NNNx
1260   \_beanoves_group_end:
1261   \tl_put_right:Nn #2 { \seq_use:Nn \l_ans_seq , }
1262 }
1263 \cs_generate_variant:Nn \_beanoves_eval:nN { VN, xN }

```

\BeanovesEval

\BeanovesEval [*<tl variable>*] {*<overlay queries>*}

<overlay queries> is the argument of ?(...) instructions. This is a comma separated list of single *<overlay query>*'s.

This function evaluates the *<overlay queries>* and store the result in the *<tl variable>* when provided or leave the result in the input stream. Forwards to `__beanoves_eval:nN` within a group. `\l_ans_tl` is used locally to store the result.

```
1264 \NewExpandableDocumentCommand \BeanovesEval { s o m } {  
1265   \__beanoves_group_begin:  
1266   \tl_clear:N \l_ans_tl  
1267   \IfBooleanTF { #1 } {  
1268     \bool_set_true:N \__beanoves_no_counter_bool  
1269   } {  
1270     \bool_set_false:N \__beanoves_no_counter_bool  
1271   }  
1272   \__beanoves_eval:nN { #3 } \l_ans_tl  
1273   \IfValueTF { #2 } {  
1274     \exp_args:NNNV  
1275     \__beanoves_group_end:  
1276     \tl_set:Nn #2 \l_ans_tl  
1277   } {  
1278     \exp_args:NV  
1279     \__beanoves_group_end: \l_ans_tl  
1280   }  
1281 }
```

5.3.8 Reseting slide ranges

\BeanovesReset

\beanovesReset [*<first value>*] {*<Slide list name>*}

```
1282 \NewDocumentCommand \BeanovesReset { O{1} m } {  
1283   \__beanoves_reset:nn { #1 } { #2 }  
1284   \ignorespaces  
1285 }
```

Forwards to `__beanoves_reset:nn`.

__beanoves_reset:nn

__beanoves_reset:nn {*<first value>*} {*<slide list name>*}

Reset the counter to the given *<first value>*. Clean the cached values also.

```
1286 \cs_new:Npn \__beanoves_reset:nn #1 #2 {  
1287   \bool_if:nTF {  
1288     \__beanoves_if_in_p:n { #2/A } || \__beanoves_if_in_p:n { #2/Z }  
1289   } {  
1290     \__beanoves_gremove:n { #2/C }  
1291     \__beanoves_gremove:n { #2//A }  
1292     \__beanoves_gremove:n { #2//L }  
1293     \__beanoves_gremove:n { #2//Z }  
1294     \__beanoves_gremove:n { #2//N }  
1295     \__beanoves_gput:nn { #2/C0 } { #1 }  
1296   } {  
1297     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }  
1298   }  
1299 }
```

```
1300 \makeatother
1301 \ExplSyntaxOff
1302 \end{package}
```