

# 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

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

On line 5, we use the dedicated **beanoves** key to declare named slide ranges. On line 6, we declare a slide range named ‘A’, starting at slide 1 and with length 2. On line 13,

the extended *named overlay specification*  $\langle A.1 \rangle$  stands for 1, on line 16,  $\langle A.2 \rangle$  stands for 2 whereas on line 19,  $\langle A.3 \rangle$  stands for 3. On line 7, 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  $\langle B.last \rangle$  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 8.

## 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 `\BeanovesDefine` and `\Beanoves` commands below inside a **beamer** frame environment, or use the `beanoves` option of this environment. The value of the `beanoves` option is exactly the argument of the `\BeanovesDefine` and `\Beanoves` commands. When used, the `\Beanoves` command is executed for each frame, whereas the `\BeanovesDefine` is executed only once. The `beanoves` option is executed only once as well but is a bit more verbose. It takes precedence over the `\BeanovesDefine` command, but not on the `\Beanoves` command.

<hr/> <hr/>	<code>beanoves={</code>
	<code>  <math>\langle name_1 \rangle = \langle spec_1 \rangle,</math></code>
	<code>  <math>\langle name_2 \rangle = \langle spec_2 \rangle,</math></code>
	<code>  <math>\dots,</math></code>
	<code>  <math>\langle name_n \rangle = \langle spec_n \rangle,</math></code>
	<code>}</code>
<hr/> <hr/>	
<code>\BeanovesDefine</code>	<code>\BeanovesDefine {</code>
	<code>  <math>\langle name_1 \rangle = \langle spec_1 \rangle,</math></code>
	<code>  <math>\langle name_2 \rangle = \langle spec_2 \rangle,</math></code>
	<code>  <math>\dots,</math></code>
	<code>  <math>\langle name_n \rangle = \langle spec_n \rangle,</math></code>
	<code>}</code>
<hr/> <hr/>	
<code>\Beanoves</code>	<code>\Beanoves{</code>
	<code>  <math>\langle name_1 \rangle = \langle spec_1 \rangle,</math></code>
	<code>  <math>\langle name_2 \rangle = \langle spec_2 \rangle,</math></code>
	<code>  <math>\dots,</math></code>
	<code>  <math>\langle name_n \rangle = \langle spec_n \rangle,</math></code>
	<code>}</code>

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

$$\langle name \rangle.1 = \langle spec_1 \rangle,$$

$$\langle name \rangle.2 = \langle spec_2 \rangle,$$

$$\dots,$$

$$\langle name \rangle.n = \langle spec_n \rangle.$$

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

$$\langle name \rangle.1.1 = \langle spec_{1.1} \rangle,$$

$$\langle name \rangle.1.2 = \langle spec_{1.2} \rangle,$$

$$\langle name \rangle.2.1 = \langle spec_{2.1} \rangle,$$

$$\langle name \rangle.2.2 = \langle spec_{2.2} \rangle$$

and so on.

The `\BeanovesDefine` command is used once at the very beginning of the `frame` environment body and thus only apply to this frame. The `\Beanoves` command can be used there multiple times. The former command does not override what is set by the `beanoves` frame option contrary to the latter. This behaviour may be useful to input the very same source code into different frames and have different combinations of slides.

### 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 \begin{frame} [
2   beanoves = {
3     A = 3:6,
4   }
5 ] {Frame \insertframenumber} {Slide \insertslidenumber}
6 \ttfamily
7 \BeanovesEval(A.1) ==3,
8 \BeanovesEval(A.2) ==4,
9 \BeanovesEval(A.-1)==1,
10 \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 \begin{frame} [
2   beanoves = {
3     A = 3:6,
4   }
5 ] {Frame \insertframenumber} {Slide \insertslidenumber}
6 \ttfamily
7 \BeanovesEval(A.length) == 6,
8 \BeanovesEval(A.1)      == 3,
9 \BeanovesEval(A.2)      == 4,
10 \BeanovesEval(A.-1)     == 1,
11 \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 incremented<sup>1</sup>.

$\langle name \rangle .n$  : 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.

### 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

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

<sup>1</sup>This is actually an experimental feature.

## 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 ‘`?(<queries>)`’. Each one is then evaluated and replaced by its static counterpart. The overall result is finally forwarded to the `beamer` class.

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

query	static value	limitation
:	-	
::	-	
<code>&lt;first expr&gt;</code>	<code>&lt;first&gt;</code>	
<code>&lt;first expr&gt;:</code>	<code>&lt;first&gt; -</code>	no <code>&lt;name&gt;.range</code>
<code>&lt;first expr&gt;::</code>	<code>&lt;first&gt; -</code>	no <code>&lt;name&gt;.range</code>
<code>&lt;first expr&gt;:&lt;length expr&gt;</code>	<code>&lt;first&gt; - &lt;last&gt;</code>	no <code>&lt;name&gt;.range</code>
<code>&lt;first expr&gt;::&lt;end expr&gt;</code>	<code>&lt;first&gt; - &lt;last&gt;</code>	no <code>&lt;name&gt;.range</code>

Here `<first expr>`, `<length expr>` and `<end expr>` both denote algebraic expressions possibly involving named overlay specifications and counters. As integers, they respectively evaluate to `<first>`, `<length>` and `<last>`.

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

Notice that nesting `?(...)` expressions is not supported.

<sup>1</sup> `<*package>`

## 5 Implementation

Identify the internal prefix (L<sup>A</sup>T<sub>E</sub>X3 DocStrip convention).

<sup>2</sup> `<@@=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:n #1 {
10 % \msg_term:nnn { beanoves } { :n } { #1 }
11 }
12 \cs_generate_variant:Nn \__beanoves_DEBUG:n { x, V }
13 \int_zero_new:N \l__beanoves_group_int
14 \cs_set:Npn \__beanoves_group_begin: {
15   \group_begin:
16   \int_incr:N \l__beanoves_group_int
17 \__beanoves_DEBUG:x {GROUP~DOWN:~\int_use:N \l__beanoves_group_int}
18 }

```

```

19 \cs_set:Npn \__beanoves_group_end: {
20   \group_end:
21   \__beanoves_DEBUG:x {GROUP~UP:~\int_use:N \l__beanoves_group_int}
22 }

```

## 5.2 Local variables

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

```

23 \int_zero_new:N \l__beanoves_split_int
24 \int_zero_new:N \l__beanoves_depth_int
25 \int_zero_new:N \g__beanoves_append_int
26 \bool_new:N \l__beanoves_no_counter_bool
27 \bool_new:N \l__beanoves_no_range_bool
28 \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 ‘//’.

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

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

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

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

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

```

29 \prop_new:N \g__beanoves_prop

```

(End definition for  $\backslash\text{g\_beanoves\_prop}$ .)

---

```

\__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<sup>A</sup>T<sub>E</sub>X3 eponym functions, except `\__beanoves_gprovide:nn` which meaning is straightforward.

```

30 \cs_new:Npn \__beanoves_gput:nn {
31   \prop_gput:Nnn \g__beanoves_prop
32 }
33 \cs_new:Npn \__beanoves_gprovide:nn #1 #2 {
34   \prop_if_in:NnF \g__beanoves_prop { #1 } {
35     \prop_gput:Nnn \g__beanoves_prop { #1 } { #2 }
36   }
37 }
38 \cs_new:Npn \__beanoves_item:n {
39   \prop_item:Nn \g__beanoves_prop
40 }
41 \cs_new:Npn \__beanoves_get:nN {
42   \prop_get:NnN \g__beanoves_prop
43 }
44 \cs_new:Npn \__beanoves_gremove:n {
45   \prop_gremove:Nn \g__beanoves_prop
46 }
47 \cs_new:Npn \__beanoves_gclear:n #1 {
48   \clist_map_inline:nn { A, L, Z, C, CO, /A, /L, /Z, /N } {
49     \__beanoves_gremove:n { #1 / ##1 }
50   }
51 }
52 \cs_new:Npn \__beanoves_gclear_cache:n #1 {
53   \clist_map_inline:nn { /A, /L, /Z, /N } {
54     \__beanoves_gremove:n { #1 / ##1 }
55   }
56 }
57 \cs_new:Npn \__beanoves_gclear: {
58   \prop_gclear:N \g__beanoves_prop
59 }
60 \cs_generate_variant:Nn \__beanoves_gput:nn { nV }
61 \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 {&lt;key&gt;}</code> <code>\__beanoves_if_in:nTF {&lt;key&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code> Convenient shortcuts to test for the existence of some key, it makes the code more concise and readable.
--	---

---

```

62 \prg_new_conditional:Npnn \__beanoves_if_in:n #1 { p, T, F, TF } {
63   \prop_if_in:NnTF \g__beanoves_prop { #1 } {
64     \prg_return_true:
65   } {
66     \prg_return_false:
67   }
68 }
69 \prg_generate_conditional_variant:Nnn \__beanoves_if_in:n {V} { p, T, F, TF }

```

---

<code>\__beanoves_get:nNTF</code>	<code>\__beanoves_get:nNTF {&lt;key&gt;} {&lt;tl variable&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</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.

```

70 \prg_new_conditional:Npnn \__beanoves_get:nN #1 #2 { T, F, TF } {
71   \prop_get:NnNTF \g__beanoves_prop { #1 } #2 {
72     \prg_return_true:
73   } {
74     \prg_return_false:
75   }
76 }

```

Utility message.

```

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

```

### 5.3.2 Regular expressions

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

```

78 \regex_const:Nn \c__beanoves_name_regex {
79   [[[:alpha:]]_][[:alnum:]]_*
80 }

```

(End definition for `\c__beanoves_name_regex`.)

<code>\c__beanoves_path_regex</code>	A sequence of <i>&lt;positive integer&gt;</i> items representing a path.
--------------------------------------	--

```

81 \regex_const:Nn \c__beanoves_path_regex {
82   (? \. \d+ )*
83 }

```

(End definition for `\c__beanoves_path_regex`.)

<code>\c__beanoves_key_regex</code> <code>\c__beanoves_A_key_Z_regex</code>	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.
--	---

```

84 \regex_const:Nn \c__beanoves_key_regex {
85   \ur{c__beanoves_name_regex} \ur{c__beanoves_path_regex}
86 }

```



```

87 \regex_const:Nn \c__beanoves_A_key_Z_regex {
88   \A \ur{c__beanoves_key_regex} \Z
89 }

```

(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.

```

90 \regex_const:Nn \c__beanoves_dotted_regex {
91   \A \ur{c__beanoves_name_regex} (?: \. [\^.] + ) * \Z
92 }

```

(End definition for `\c__beanoves_dotted_regex`.)

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

```

93 \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.

```

94 \regex_const:Nn \c__beanoves_int_regex {
95   (?: [-+] ) ? \d +
96 }

```

(End definition for `\c__beanoves_int_regex`.)

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

```

97 \regex_const:Nn \c__beanoves_list_regex {
98   \A \[ \s *

```

Capture groups:

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

```

99   ( [\^] % [---
100   ] * ? )
101   \s * \] \Z
102 }

```

(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.

```

103 \regex_const:Nn \c__beanoves_split_regex {
104   \s * ( ? :

```

We start with ‘++’ instrussions <sup>2</sup>.

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

```

105   \+ \+ ( \ur{c__beanoves_name_regex} )

```

<sup>2</sup>At the same time an instruction and an expression... this is a synonym of expression

- 2: optionally followed by an integer path

106       ( \ur{c\_\_beanoves\_path\_regex} ) (?: \. n )?

We continue with other expressions

- 3:  $\langle name \rangle$  of a slide range

107       | ( \ur{c\_\_beanoves\_name\_regex} )

- 4: optionally followed by an integer path

108       ( \ur{c\_\_beanoves\_path\_regex} )

Next comes another branching

109       (?:

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

110           \.(e)ngth

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

111           | \.(a)st

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

112           | \.(n)ext

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

113           | \.(r)ange

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

114           | \.(n)

- 10: 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.

115           (?: \s\* \+= \s\*  
116           ( (?: \ur{c\_\_beanoves\_int\_regex} | \ur{c\_\_beanoves\_key\_regex} )  
117           (?: [+ \- \*/] (?: \d+ | \ur{c\_\_beanoves\_key\_regex} ) ) \*  
118           | \ ( .\*? \ ) (?: \Z | \s+ )  
119           )  
120           ) ?

121       ) ?

122     ) \s\*

123   }

(End definition for \c\_\_beanoves\_split\_regex.)

### 5.3.3 Defining named slide ranges

---

`\__beanoves_parse:Nnn`    `\__beanoves_parse:nn <command> {<key>} {<definition>}`

---

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:nnnn {<key>} {<first>} {<length>} {<last>}`  
`\__beanoves_range:nVVV`    `\__beanoves_range_alt:nnnn {<key>} {<first>} {<length>} {<last>}`  
`\__beanoves_range_alt:nnnn`    Auxiliary function called within a group. Setup the model to define a range. The alt  
`\__beanoves_range_alt:nVVV`    variant does not override an already existing value.

---

```

124 \cs_new:Npn \__beanoves_range:Nnnnn #1 #2 #3 #4 #5 {
125   \tl_if_empty:nTF { #3 } {
126     \tl_if_empty:nTF { #4 } {
127       \tl_if_empty:nTF { #5 } {
128         \msg_error:nnn { beanoves } { :n } { Not~a~range::~~#2 }
129       } {
130         #1 { #2/Z } { #5 }
131       }
132     } {
133       #1 { #2/L } { #4 }
134       \tl_if_empty:nF { #5 } {
135         #1 { #2/Z } { #5 }
136         #1 { #2/A } { #2.last - (#2.length) + 1 }
137       }
138     }
139   } {
140     #1 { #2/A } { #3 }
141     \tl_if_empty:nTF { #4 } {
142       \tl_if_empty:nF { #5 } {
143         #1 { #2/Z } { #5 }
144         #1 { #2/L } { #2.last - (#2.1) + 1 }
145       }
146     } {
147       #1 { #2/L } { #4 }
148       #1 { #2/Z } { #2.1 + #2.length - 1 }
149     }
150   }
151 }
152 \cs_new:Npn \__beanoves_range:nnnn #1 {
153   \__beanoves_gclear:n { #1 }
154   \__beanoves_range:Nnnnn \__beanoves_gput:nn { #1 }
155 }
156 \cs_generate_variant:Nn \__beanoves_range:nnnn { nVVV }
157 \cs_new:Npn \__beanoves_range_alt:nnnn #1 {
158   \__beanoves_gclear_cache:n { #1 }
159   \__beanoves_range:Nnnnn \__beanoves_gprovide:nn { #1 }
160 }
161 \cs_generate_variant:Nn \__beanoves_range_alt:nnnn { nVVV }

```

---

```

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

Shortcut to \__beanoves_parse:nn <command> {<key>} { 1 }

162 \cs_new:Npn \__beanoves_parse:Nn #1 #2 {
163   \__beanoves_parse:Nnn #1 { #2 } { 1 }
164 }

165 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
166 \cs_new:Npn \__beanoves_do_parse:Nnn #1 #2 #3 {

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

167   \tl_clear:N \l_a_tl
168   \tl_clear:N \l_b_tl
169   \tl_clear:N \l_c_tl
170   \regex_split:NnN \c__beanoves_colons_regex { #3 } \l_split_seq
171   \seq_pop_left:NNT \l_split_seq \l_a_tl {

\l_a_tl may contain the <start>.

172   \seq_pop_left:NNT \l_split_seq \l_b_tl {
173     \tl_if_empty:NTF \l_b_tl {

This is a one colon range.

174     \seq_pop_left:NN \l_split_seq \l_b_tl
\l_b_tl may contain the <length>.

175     \seq_pop_left:NNT \l_split_seq \l_c_tl {
176       \tl_if_empty:NTF \l_c_tl {

A :: was expected:

177 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(1):~#3 }
178   } {
179     \int_compare:nNnT { \tl_count:N \l_c_tl } > { 1 } {
180 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(2):~#3 }
181   }
182   \seq_pop_left:NN \l_split_seq \l_c_tl
\l_c_tl may contain the <end>.

183   \seq_if_empty:NF \l_split_seq {
184 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(3):~#3 }
185   }
186   }
187   }
188   } {

This is a two colon range.

189   \int_compare:nNnT { \tl_count:N \l_b_tl } > { 1 } {
190 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(4):~#3 }
191   }
192   \seq_pop_left:NN \l_split_seq \l_c_tl
\l_c_tl contains the <end>.

193   \seq_pop_left:NNTF \l_split_seq \l_b_tl {
194     \tl_if_empty:NTF \l_b_tl {
195       \seq_pop_left:NN \l_split_seq \l_b_tl

```

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

```

196         \seq_if_empty:NF \l_split_seq {
197 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(5):~#3 }
198     }
199     } {
200 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(6):~#3 }
201     }
202     } {
203         \tl_clear:N \l_b_tl
204     }
205 }
206 }
207 }

```

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.

```

208 \bool_if:nF {
209     \tl_if_empty_p:N \l_a_tl
210     || \tl_if_empty_p:N \l_b_tl
211     || \tl_if_empty_p:N \l_c_tl
212 } {
213 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(7):~#3 }
214 }
215 #1 { #2 } \l_a_tl \l_b_tl \l_c_tl
216 }

217 \cs_new:Npn \__beanoves_parse:Nnn #1 #2 #3 {
218     \__beanoves_group_begin:
219     \regex_match:NnTF \c__beanoves_A_key_Z_regex { #2 } {

```

We got a valid key.

```

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

```

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

```

221     \exp_args:NNx
222     \seq_set_from_clist:Nn \l_match_seq {
223         \seq_item:Nn \l_match_seq { 2 }
224     }
225     \seq_map_indexed_inline:Nn \l_match_seq {
226         \__beanoves_do_parse:Nnn #1 { #2.##1 } { ##2 }
227     }
228     } {
229         \__beanoves_do_parse:Nnn #1 { #2 } { #3 }
230     }
231     } {
232         \msg_error:nnn { beanoves } { :n } { Invalid~key:~#1 }
233     }
234     \__beanoves_group_end:
235 }

```

---

**\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`.

```
236 \NewDocumentCommand \BeanovesDefine { m } {
237   \Beanoves * { #1 }
238   \RenewDocumentCommand \BeanovesDefine { m } { }
239 }
240 \NewDocumentCommand \Beanoves { sm } {
241   \IfBooleanTF {#1} {
242     \keyval_parse:nnn {
243       \__beanoves_parse:Nn \__beanoves_range_alt:nVVV
244     } {
245       \__beanoves_parse:Nnn \__beanoves_range_alt:nVVV
246     }
247   } {
248     \keyval_parse:nnn {
249       \__beanoves_parse:Nn \__beanoves_range:nVVV
250     } {
251       \__beanoves_parse:Nnn \__beanoves_range:nVVV
252     }
253   }
254   { #2 }
255   \ignorespaces
256 }
```

If we use this command in the frame body, it will be executed for each different frame. If we use the frame option `beanoves` instead, the command is executed only once, at the cost of a more verbose code.

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

### 5.3.4 Scanning named overlay specifications

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

---

**\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 ??.)

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

```
258 \cs_set_eq:NN \__beanoves_beamer@masterdecode \beamer@masterdecode
259 \cs_set:Npn \beamer@masterdecode #1 {
260   \__beanoves_group_begin:
261   \tl_clear:N \l_ans_tl
262   \__beanoves_scan:nnn { #1 } \__beanoves_eval:nN \l_ans_tl
263   \exp_args:NNV
264   \__beanoves_group_end:
265   \__beanoves_beamer@masterdecode \l_ans_tl
266 }
```

<u>\_beanoves_scan:nNN</u>	<p>\_beanoves_scan:nNN {<math>\langle</math>named overlay expression<math>\rangle</math>} <math>\langle</math>eval<math>\rangle</math> <math>\langle</math>tl variable<math>\rangle</math></p> <p>Scan the <math>\langle</math>named overlay expression<math>\rangle</math> argument and feed the <math>\langle</math>tl variable<math>\rangle</math> replacing ?(...) instructions by their static counterpart with help from the <math>\langle</math>eval<math>\rangle</math> function, which is \_beanoves_eval:nN. A group is created to use local variables:</p> <p>\l_ans_tl: is the token list that will be appended to <math>\langle</math>tl variable<math>\rangle</math> on return.</p>
\l__beanoves_depth_int	<p>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.</p> <p>(End definition for \l__beanoves_depth_int.)</p>
g__beanoves_append_int	<p>Decrement each time \_beanoves_append:nN is called. To avoid catch circular definitions.</p> <p>(End definition for g__beanoves_append_int.)</p>
\l_query_tl	<p>Storage for the overlay query expression to be evaluated.</p> <p>(End definition for \l_query_tl. This variable is documented on page ??.)</p>
\l_token_seq	<p>The <math>\langle</math>overlay expression<math>\rangle</math> is split into the sequence of its tokens.</p> <p>(End definition for \l_token_seq. This variable is documented on page ??.)</p>
\l_ask_bool	<p>Whether a loop may continue. Controls the continuation of the main loop that scans the tokens of the <math>\langle</math>named overlay expression<math>\rangle</math> looking for a question mark.</p> <p>(End definition for \l_ask_bool. This variable is documented on page ??.)</p>
\l_query_bool	<p>Whether a loop may continue. Controls the continuation of the secondary loop that scans the tokens of the <math>\langle</math>named overlay expression<math>\rangle</math> looking for an opening parenthesis follow the question mark. It then controls the loop looking for the balanced closing parenthesis.</p> <p>(End definition for \l_query_bool. This variable is documented on page ??.)</p>
\l_token_tl	<p>Storage for just one token.</p> <p>(End definition for \l_token_tl. This variable is documented on page ??.)</p>
267 \cs_new:Npn \_beanoves_scan:nNN #1 #2 #3 {	
268   \_beanoves_group_begin:	
269   \tl_clear:N \l_ans_tl	
270   \int_zero:N \l__beanoves_depth_int	
271   \seq_clear:N \l_token_seq	
	Explode the $\langle$ named overlay expression $\rangle$ into a list of tokens:
272   \regex_split:nnN {} { #1 } \l_token_seq	
	Run the top level loop to scan for a '?':
273   \bool_set_true:N \l_ask_bool	
274   \bool_while_do:Nn \l_ask_bool {	
275     \seq_pop_left:NN \l_token_seq \l_token_tl	
276     \quark_if_no_value:NTF \l_token_tl {	
	We reached the end of the sequence (and the token list), we end the loop here.

```

277         \bool_set_false:N \l_ask_bool
278     } {
\l_token_tl contains a ‘normal’ token.
279     \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.
280         \bool_set_true:N \l_query_bool
281         \bool_while_do:Nn \l_query_bool {
Get next token.
282         \seq_pop_left:NN \l_token_seq \l_token_tl
283         \quark_if_no_value:Ntf \l_token_tl {
No opening parenthesis found, raise.
284         \msg_fatal:nxx { beanoves } { :n } {Missing-('%---)
285             ~after~a~?:~#1}
286     } {
287         \tl_if_eq:NnT \l_token_tl { ( %}
288     } {
We found the ‘(’ after the ‘?’. Increment the parenthesis depth to 1 (on first passage).
289         \int_incr:N \l__beanoves_depth_int
Record the forthcoming content in the \l_query_tl variable, up to the next balancing
‘)’.
290         \tl_clear:N \l_query_tl
291         \bool_while_do:Nn \l_query_bool {
Get next token.
292         \seq_pop_left:NN \l_token_seq \l_token_tl
293         \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.
294         \msg_error:nxx { beanoves } { :n } {Missing-%(---
295             `):~#1 }
296         \int_do_while:nNnn \l__beanoves_depth_int > 1 {
297             \int_decr:N \l__beanoves_depth_int
298             \tl_put_right:Nn \l_query_tl {%(---
299             )}
300         }
301         \int_zero:N \l__beanoves_depth_int
302         \bool_set_false:N \l_query_bool
303         \bool_set_false:N \l_ask_bool
304     } {
305         \tl_if_eq:NnTF \l_token_tl { ( %---)
306     } {
We found a ‘(’, increment the depth and append the token to \l_query_tl.
307         \int_incr:N \l__beanoves_depth_int
308         \tl_put_right:NV \l_query_tl \l_token_tl
309     } {

```



This is not a ‘(’.

```

310         \tl_if_eq:NnTF \l_token_tl { %(
311         )
312     } {

```

We found a ‘)’, decrement the depth.

```

313         \int_decr:N \l__beanoves_depth_int
314         \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.

```

315     \exp_args:NV #2 \l_query_tl \l_ans_tl
316     \bool_set_false:N \l_query_bool
317 } {

```

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

```

318         \tl_put_right:NV \l_query_tl \l_token_tl
319     }

```

Above ends the code for a positive depth.

```

320     } {

```

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

```

321         \tl_put_right:NV \l_query_tl \l_token_tl
322     }
323 }
324 }

```

Above ends the code for Not a ‘(’

```

325     }
326 }

```

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

```

327     }

```

Above ends the code for not a no value quark.

```

328 }

```

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.

```

329     \quark_if_no_value:NT \l_token_tl {
330         \bool_set_false:N \l_query_bool
331         \bool_set_false:N \l_ask_bool
332     }
333 } {

```

This is not a ‘?’, append the token to right of \l\_ans\_tl and continue.

```

334         \tl_put_right:NV \l_ans_tl \l_token_tl
335     }

```

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

```

336     }
337 }

```

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

```

338 \exp_args:NNNV
339 \__beanoves_group_end:
340 \tl_put_right:Nn #3 \l_ans_tl
341 }

```

Each new frame has its own set of slide ranges, we clear the property list on entering a new frame environment. Frame environments nested into other frame environments are not supported.

```

342 \AddToHook
343 { env/beamers@framepauses/before }
344 { \prop_gc_clear:N \g__beanoves_prop }

```

### 5.3.5 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.

```

345 \cs_new:Npn \__beanoves_fp_round:nN #1 #2 {
346   \__beanoves_DEBUG:x { ROUND:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2}
347   \tl_if_empty:NTF { #1 } {
348     \__beanoves_DEBUG:x { ROUND1:~EMPTY }
349   } {
350     \__beanoves_DEBUG:x { ROUND1:~\tl_to_str:n{#1} }
351     \tl_put_right:Nx #2 {
352       \fp_eval:n { round(#1) }
353     }
354   }
355 }
356 \cs_generate_variant:Nn \__beanoves_fp_round:nN { VN, xN }
357 \cs_new:Npn \__beanoves_fp_round:N #1 {
358   \__beanoves_DEBUG:x { ROUND:\string#1=\tl_to_str:V #1}
359   \tl_if_empty:VTF #1 {
360     \__beanoves_DEBUG:x { ROUND2:~EMPTY }
361   } {
362     \__beanoves_DEBUG:x { ROUND2:~\exp_args:Nx\tl_to_str:n{#1} }
363     \tl_set:Nx #1 {
364       \fp_eval:n { round(#1) }
365     }
366   }
367 }

```

---

$\backslash\_beanoves\_raw\_first:nNTF$	$\backslash\_beanoves\_raw\_first:nNTF \{ \langle name \rangle \} \langle tl\ variable \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
---	--

---

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.

```

368 \cs_set:Npn \__beanoves_return_true:nnN #1 #2 #3 {
369   \tl_if_empty:NTF \l_ans_tl {
370     \__beanoves_group_end:

```

```

371 \__beanoves_DEBUG:n { RETURN_FALSE/key=#1/type=#2/EMPTY }
372 \__beanoves_gremove:n { #1//#2 }
373 \prg_return_false:
374 } {
375 \__beanoves_fp_round:N \l_ans_tl
376 \__beanoves_gput:nV { #1//#2 } \l_ans_tl
377 \exp_args:NNNV
378 \__beanoves_group_end:
379 \tl_put_right:Nn #3 \l_ans_tl
380 \__beanoves_DEBUG:x { RETURN_TRUE/key=#1/type=#2/ans=\l_ans_tl/ }
381 \prg_return_true:
382 }
383 }
384 \cs_set:Npn \__beanoves_return_false:nn #1 #2 {
385 \__beanoves_DEBUG:n { RETURN_FALSE/key=#1/type=#2/ }
386 \__beanoves_group_end:
387 \__beanoves_gremove:n { #1//#2 }
388 \prg_return_false:
389 }
390 \prg_new_conditional:Npnn \__beanoves_raw_first:nN #1 #2 { T, F, TF } {
391 \__beanoves_DEBUG:x { RAW_FIRST/
392 key=\tl_to_str:n{#1}/\string #2=/\tl_to_str:V #2/}
393 \__beanoves_if_in:nTF { #1//A } {
394 \__beanoves_DEBUG:n { RAW_FIRST/#1/CACHED }
395 \tl_put_right:Nx #2 { \__beanoves_item:n { #1//A } }
396 \prg_return_true:
397 } {
398 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/NOT_CACHED }
399 \__beanoves_group_begin:
400 \tl_clear:N \l_ans_tl
401 \__beanoves_get:nNTF { #1/A } \l_a_tl {
402 \__beanoves_DEBUG:x { RAW_FIRST/key=#1/A=\l_a_tl }
403 \__beanoves_if_append:VNTF \l_a_tl \l_ans_tl {
404 \__beanoves_return_true:nnN { #1 } A #2
405 } {
406 \__beanoves_return_false:nn { #1 } A
407 }
408 } {
409 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/A/F }
410 \__beanoves_get:nNTF { #1/L } \l_a_tl {
411 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/L=\l_a_tl }
412 \__beanoves_get:nNTF { #1/Z } \l_b_tl {
413 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/Z=\l_b_tl }
414 \__beanoves_if_append:xNTF {
415 \l_b_tl - ( \l_a_tl ) + 1
416 } \l_ans_tl {
417 \__beanoves_return_true:nnN { #1 } A #2
418 } {
419 \__beanoves_return_false:nn { #1 } A
420 }
421 } {
422 \__beanoves_DEBUG:n { RAW_FIRST/key=#1/Z/F/ }
423 \__beanoves_return_false:nn { #1 } A
424 }

```

```

425     } {
426     \__beanoves_DEBUG:n { RAW_FIRST/key=#1/L/F/ }
427         \__beanoves_return_false:nn { #1 } A
428     }
429 }
430 }
431 }

```

---

\\_\_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.

```

432 \prg_new_conditional:Npnn \__beanoves_if_first:nN #1 #2 { T, F, TF } {
433 \__beanoves_DEBUG:x { IF_FIRST/\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2}
434 \__beanoves_raw_first:nNTF { #1 } #2 {
435     \prg_return_true:
436 } {
437     \__beanoves_get:nNTF { #1/C } \l_a_tl {
438 \__beanoves_DEBUG:n { IF_FIRST/#1/C/T/\l_a_tl }
439     \bool_set_true:N \l_no_counter_bool
440     \__beanoves_if_append:xNTF \l_a_tl \l_ans_tl {
441         \__beanoves_return_true:nnN { #1 } A #2
442     } {
443         \__beanoves_return_false:nn { #1 } A
444     }
445 } {
446     \regex_match:NnTF \c__beanoves_A_key_Z_regex { #1 } {
447         \__beanoves_gput:nn { #1/A } { 1 }
448         \tl_set:Nn #2 { 1 }
449 \__beanoves_DEBUG:x{IF_FIRST_MATCH:
450     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /}
451     \__beanoves_return_true:nnN { #1 } A #2
452 } {
453 \__beanoves_DEBUG:x{IF_FIRST_NO_MATCH:
454     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /}
455     \__beanoves_return_false:nn { #1 } A
456 }
457 }
458 }
459 }

```

---

\\_\_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.

---

```

460 \cs_new:Npn \__beanoves_first:nN #1 #2 {
461     \__beanoves_if_first:nNF { #1 } #2 {
462         \msg_error:nnn { beanoves } { :n } { Range-with-no~first:~#1 }
463     }
464 }
465 \cs_generate_variant:Nn \__beanoves_first:nN { VN }

```

---

```

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

```

---

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

```

466 \prg_new_conditional:Npnn \__beanoves_raw_length:nN #1 #2 { T, F, TF } {
467 \__beanoves_DEBUG:n { RAW_LENGTH/#1 }
468 \__beanoves_if_in:nTF { #1//L } {
469 \tl_put_right:Nx #2 { \__beanoves_item:n { #1//L } }
470 \__beanoves_DEBUG:x { RAW_LENGTH/CACHED/#1/\__beanoves_item:n { #1//L } }
471 \prg_return_true:
472 } {
473 \__beanoves_DEBUG:x { RAW_LENGTH/NOT_CACHED/key=#1/ }
474 \__beanoves_gput:nn { #1//L } { 0 }
475 \__beanoves_group_begin:
476 \tl_clear:N \l_ans_tl
477 \__beanoves_if_in:nTF { #1/L } {
478 \__beanoves_if_append:xNTF {
479 \__beanoves_item:n { #1/L }
480 } \l_ans_tl {
481 \__beanoves_return_true:nnN { #1 } L #2
482 } {
483 \__beanoves_return_false:nn { #1 } L
484 }
485 } {
486 \__beanoves_get:nNTF { #1/A } \l_a_tl {
487 \__beanoves_get:nNTF { #1/Z } \l_b_tl {
488 \__beanoves_if_append:xNTF {
489 \l_b_tl - (\l_a_tl) + 1
490 } \l_ans_tl {
491 \__beanoves_return_true:nnN { #1 } L #2
492 } {
493 \__beanoves_return_false:nn { #1 } L
494 }
495 } {
496 \__beanoves_return_false:nn { #1 } L
497 }
498 } {
499 \__beanoves_return_false:nn { #1 } L
500 }
501 }
502 }
503 }
504 \prg_generate_conditional_variant:Nnn
505 \__beanoves_raw_length:nN { VN } { T, F, TF }

```

---

```

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

```

---

Append the length of the <name> slide range to <tl variable>

```

506 \cs_new:Npn \__beanoves_length:nN #1 #2 {
507 \__beanoves_raw_length:nNF { #1 } #2 {
508 \msg_error:nnn { beanoves } { :n } { Range~with~no~length:~#1 }
509 }

```

```

510 }
511 \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.

```

512 \prg_new_conditional:Npnn \__beanoves_raw_last:nN #1 #2 { T, F, TF } {
513   \__beanoves_DEBUG:n { RAW_LAST/#1 }
514   \__beanoves_if_in:nTF { #1//Z } {
515     \tl_put_right:Nx #2 { \__beanoves_item:n { #1//Z } }
516     \prg_return_true:
517   } {
518     \__beanoves_gput:nn { #1//Z } { 0 }
519     \__beanoves_group_begin:
520     \tl_clear:N \l_ans_tl
521     \__beanoves_if_in:nTF { #1/Z } {
522       \__beanoves_DEBUG:x { NORMAL_RAW_LAST:~\__beanoves_item:n { #1/Z } }
523       \__beanoves_if_append:xNTF {
524         \__beanoves_item:n { #1/Z }
525       } \l_ans_tl {
526         \__beanoves_return_true:nnN { #1 } Z #2
527       } {
528         \__beanoves_return_false:nn { #1 } Z
529       }
530     } {
531       \__beanoves_get:nNTF { #1/A } \l_a_tl {
532         \__beanoves_get:nNTF { #1/L } \l_b_tl {
533           \__beanoves_if_append:xNTF {
534             \l_a_tl + (\l_b_tl) - 1
535           } \l_ans_tl {
536             \__beanoves_return_true:nnN { #1 } Z #2
537           } {
538             \__beanoves_return_false:nn { #1 } Z
539           }
540         } {
541           \__beanoves_return_false:nn { #1 } Z
542         }
543       } {
544         \__beanoves_return_false:nn { #1 } Z
545       }
546     }
547   }
548 }
549 \prg_generate_conditional_variant:Nnn
550   \__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 <name> slide range to <tl variable>

```

---

```

551 \cs_new:Npn \__beanoves_last:nN #1 #2 {
552   \__beanoves_raw_last:nNF { #1 } #2 {
553     \msg_error:nnn { beanoves } { :n } { Range~with~no~last:~#1 }

```

```

554   }
555 }
556 \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.

```

557 \prg_new_conditional:Npnn \__beanoves_if_next:nN #1 #2 { T, F, TF } {
558   \__beanoves_if_in:nTF { #1//N } {
559     \tl_put_right:Nx #2 { \__beanoves_item:n { #1//N } }
560     \prg_return_true:
561   } {
562     \__beanoves_group_begin:
563     \cs_set:Npn \__beanoves_return_true: {
564       \tl_if_empty:NTF \l_ans_tl {
565         \__beanoves_group_end:
566         \prg_return_false:
567       } {
568         \__beanoves_fp_round:N \l_ans_tl
569         \__beanoves_gput:nV { #1//N } \l_ans_tl
570         \exp_args:NNNV
571         \__beanoves_group_end:
572         \tl_put_right:Nn #2 \l_ans_tl
573         \prg_return_true:
574       }
575     }
576     \cs_set:Npn \__beanoves_return_false: {
577       \__beanoves_group_end:
578       \prg_return_false:
579     }
580     \tl_clear:N \l_a_tl
581     \__beanoves_raw_last:nNTF { #1 } \l_a_tl {
582       \__beanoves_if_append:xNTF {
583         \l_a_tl + 1
584       } \l_ans_tl {
585         \__beanoves_return_true:
586       } {
587         \__beanoves_return_false:
588       }
589     } {
590       \__beanoves_return_false:
591     }
592   }
593 }
594 \prg_generate_conditional_variant:Nnn
595   \__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>*.

```

596 \cs_new:Npn \__beanoves_next:nN #1 #2 {
597   \__beanoves_if_next:nNF { #1 } #2 {

```

```

598 \msg_error:nnn { beanoves } { :n } { Range-with-no-next:~#1 }
599 }
600 }
601 \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.

```

602 \prg_new_conditional:Npnn \__beanoves_if_free_counter:Nn #1 #2 { T, F, TF } {
603 \__beanoves_DEBUG:x { IF_FREE: \string #1/
604   key=\tl_to_str:n{#2}/value=\__beanoves_item:n {#2/C}/ }
605 \__beanoves_group_begin:
606 \tl_clear:N \l_ans_tl
607 \__beanoves_get:nNF { #2/C } \l_ans_tl {
608   \__beanoves_raw_first:nNF { #2 } \l_ans_tl {
609     \__beanoves_raw_last:nNF { #2 } \l_ans_tl { }
610   }
611 }
612 \__beanoves_DEBUG:x { IF_FREE_2:\string \l_ans_tl=\tl_to_str:V \l_ans_tl/}
613 \tl_if_empty:NTF \l_ans_tl {
614   \__beanoves_group_end:
615   \regex_match:NnTF \c__beanoves_A_key_Z_regex { #2 } {
616     \__beanoves_gput:nn { #2/C } { 1 }
617     \tl_set:Nn #1 { 1 }
618 \__beanoves_DEBUG:x { IF_FREE_MATCH_TRUE:\string #1=\tl_to_str:V #1 /
619   key=\tl_to_str:n{#2} }
620   \prg_return_true:
621   } {
622 \__beanoves_DEBUG:x { IF_FREE_NO_MATCH_FALSE: \string #1=\tl_to_str:V #1 /
623   key=\tl_to_str:n{#2} }
624   \prg_return_false:
625   }
626 } {
627   \__beanoves_gput:nV { #2/C } \l_ans_tl
628   \exp_args:NNNV
629   \__beanoves_group_end:
630   \tl_set:Nn #1 \l_ans_tl
631 \__beanoves_DEBUG:x { IF_FREE_TRUE(2): \string #1=\tl_to_str:V #1 /
632   key=\tl_to_str:n{#2} }
633   \prg_return_true:
634   }
635 }
636 \prg_generate_conditional_variant:Nnn
637 \__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.

```

638 \prg_new_conditional:Npnn \__beanoves_if_counter:nN #1 #2 { T, F, TF } {

```



```

639 \__beanoves_DEBUG:x { IF_COUNTER:key=
640   \tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 }
641   \__beanoves_group_begin:
642   \__beanoves_if_free_counter:NnTF \l_ans_tl { #1 } {

If there is a <first>, use it to bound the result from below.

643   \tl_clear:N \l_a_tl
644   \__beanoves_raw_first:nNT { #1 } \l_a_tl {
645     \fp_compare:nNnT { \l_ans_tl } < { \l_a_tl } {
646       \tl_set:NV \l_ans_tl \l_a_tl
647     }
648   }

If there is a <last>, use it to bound the result from above.

649   \tl_clear:N \l_a_tl
650   \__beanoves_raw_last:nNT { #1 } \l_a_tl {
651     \fp_compare:nNnT { \l_ans_tl } > { \l_a_tl } {
652       \tl_set:NV \l_ans_tl \l_a_tl
653     }
654   }
655   \exp_args:NNx
656   \__beanoves_group_end:
657   \__beanoves_fp_round:nN \l_ans_tl #2
658 \__beanoves_DEBUG:x {IF_COUNTER_TRUE:key=\tl_to_str:n{#1}/
659   \string #2=\tl_to_str:V #2 }
660   \prg_return_true:
661 } {
662 \__beanoves_DEBUG:x {IF_COUNTER_FALSE:key=\tl_to_str:n{#1}/
663   \string #2=\tl_to_str:V #2 }
664   \prg_return_false:
665 }
666 }
667 \prg_generate_conditional_variant:Nnn
668   \__beanoves_if_counter:nN { VN } { T, F, TF }

```

---

$\backslash\_beanoves\_if\_index:nN\textit{TF}$ $\backslash\_beanoves\_if\_index:VVN\textit{TF}$	$\backslash\_beanoves\_if\_index:nN\textit{TF}$ { <i>&lt;name&gt;</i> } { <i>&lt;integer path&gt;</i> } { <i>&lt;tl variable&gt;</i> } { <i>&lt;true code&gt;</i> } { <i>&lt;false code&gt;</i> }
---	--

---

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. If the computation is possible, *<true code>* is executed, otherwise *<false code>* is executed. The computation may fail when too many recursion calls are made.

```

669 \prg_new_conditional:Npnn \__beanoves_if_index:nN #1 #2 #3 { T, F, TF } {
670 \__beanoves_DEBUG:x { IF_INDEX:key=#1/index=#2/\string#3/ }
671   \__beanoves_group_begin:
672   \tl_set:Nn \l_name_tl { #1 }
673   \regex_split:nnNTF { \. } { #2 } \l_split_seq {
674     \seq_pop_left:NN \l_split_seq \l_a_tl
675     \seq_pop_right:NN \l_split_seq \l_a_tl
676     \seq_map_inline:Nn \l_split_seq {
677       \tl_set_eq:NN \l_b_tl \l_name_tl
678       \tl_put_right:Nn \l_b_tl { . ##1 }
679     }
680     \exp_args:Nx
681     \__beanoves_get:nN { \l_b_tl / A } \l_c_tl

```

```

681     \quark_if_no_value:N\TF \l_c_tl {
682       \tl_set_eq:NN \l_name_tl \l_b_tl
683     } {
684       \tl_set_eq:NN \l_name_tl \l_c_tl
685     }
686 \__beanoves_DEBUG:x { IF_INDEX_SPLIT:##1/
687   \string\l_name_tl=\tl_to_str:N \l_name_tl}
688 }
689 \tl_clear:N \l_b_tl
690 \exp_args:Nx
691 \__beanoves_raw_first:n\TF { \l_name_tl.\l_a_tl } \l_b_tl {
692   \tl_set_eq:NN \l_ans_tl \l_b_tl
693 } {
694   \tl_clear:N \l_b_tl
695   \exp_args:NV
696   \__beanoves_raw_first:n\TF \l_name_tl \l_b_tl {
697     \tl_set_eq:NN \l_ans_tl \l_b_tl
698   } {
699     \tl_set_eq:NN \l_ans_tl \l_name_tl
700   }
701   \tl_put_right:Nx \l_ans_tl { + (\l_a_tl) - 1}
702 }
703 \__beanoves_DEBUG:x { IF_INDEX_TRUE:key=#1/index=#2/
704   \string\l_ans_tl=\tl_to_str:N \l_ans_tl }
705   \exp_args:NNx
706   \__beanoves_group_end:
707   \__beanoves_fp_round:nN \l_ans_tl #3
708   \prg_return_true:
709 } {
710 \__beanoves_DEBUG:x { IF_INDEX_FALSE:key=#1/index=#2/ }
711   \prg_return_false:
712 }
713 }

```

---

$\backslash\_beanoves\_if\_incr:nnTF$ $\backslash\_beanoves\_if\_incr:nnTF$ $\backslash\_beanoves\_if\_incr:(VnN VVN)TF$	$\backslash\_beanoves\_if\_incr:nnTF \{ \langle name \rangle \} \{ \langle offset \rangle \} \{ \langle true code \rangle \} \{ \langle false code \rangle \}$ $\backslash\_beanoves\_if\_incr:nnTF \{ \langle name \rangle \} \{ \langle offset \rangle \} \{ \langle tl variable \rangle \} \{ \langle true code \rangle \} \{ \langle false code \rangle \}$
--	--

---

Increment the free counter position accordingly. When requested, put the result in the  $\langle tl variable \rangle$ . The result will lay within the declared range.

```

714 \prg_new_conditional:Npnn \__beanoves_if_incr:nn #1 #2 { T, F, TF } {
715   \__beanoves_DEBUG:x { IF_INCR:\tl_to_str:n{#1}/\tl_to_str:n{#2} }
716   \__beanoves_group_begin:
717   \tl_clear:N \l_a_tl
718   \__beanoves_if_free_counter:N\TF \l_a_tl { #1 } {
719     \tl_clear:N \l_b_tl
720     \__beanoves_if_append:x\TF { \l_a_tl + (#2) } \l_b_tl {
721       \__beanoves_fp_round:N \l_b_tl
722       \__beanoves_gput:nV { #1/C } \l_b_tl
723     }
724   \__beanoves_DEBUG:x { IF_INCR_TRUE:#1/#2 }
725   \prg_return_true:

```

```

726     } {
727         \__beanoves_group_end:
728 \__beanoves_DEBUG:x { IF_INCR_FALSE(1):#1/#2 }
729         \prg_return_false:
730     }
731 } {
732     \__beanoves_group_end:
733 \__beanoves_DEBUG:x { IF_INCR_FALSE(2):#1/#2 }
734     \prg_return_false:
735 }
736 }
737 \prg_new_conditional:Npnn \__beanoves_if_incr:nnN #1 #2 #3 { T, F, TF } {
738     \__beanoves_if_incr:nnTF { #1 } { #2 } {
739         \__beanoves_if_counter:nNTF { #1 } #3 {
740             \prg_return_true:
741         } {
742             \prg_return_false:
743         }
744     } {
745         \prg_return_false:
746     }
747 }
748 \prg_generate_conditional_variant:Nnn
749 \__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.

```

750 \prg_new_conditional:Npnn \__beanoves_if_range:nN #1 #2 { T, F, TF } {
751 \__beanoves_DEBUG:x{ RANGE:key=#1/\string#2/}
752 \bool_if:NTF \l__beanoves_no_range_bool {
753     \prg_return_false:
754 } {
755     \__beanoves_group_begin:
756     \tl_clear:N \l_a_tl
757     \tl_clear:N \l_b_tl
758     \tl_clear:N \l_ans_tl
759     \__beanoves_raw_first:nNTF { #1 } \l_a_tl {
760         \__beanoves_raw_last:nNTF { #1 } \l_b_tl {
761             \exp_args:NNNx
762             \__beanoves_group_end:
763             \tl_put_right:Nn #2 { \l_a_tl - \l_b_tl }
764 \__beanoves_DEBUG:x{ RANGE_TRUE_A_Z:key=#1/\string#2=#2/}
765             \prg_return_true:
766         } {
767             \exp_args:NNNx
768             \__beanoves_group_end:
769             \tl_put_right:Nn #2 { \l_a_tl - }
770 \__beanoves_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
771             \prg_return_true:
772         }
773     } {
774         \__beanoves_raw_last:nNTF { #1 } \l_b_tl {

```

```

775 \__beanoves_DEBUG:x{ RANGE_TRUE_Z:key=#1/\string#2=#2/}
776     \exp_args:NNNx
777     \__beanoves_group_end:
778     \tl_put_right:Nn #2 { - \l_b_tl }
779     \prg_return_true:
780 } {
781 \__beanoves_DEBUG:x{ RANGE_FALSE:key=#1/}
782     \__beanoves_group_end:
783     \prg_return_false:
784 }
785 }
786 }
787 }
788 \prg_generate_conditional_variant:Nnn
789 \__beanoves_if_range:nN { VN } { T, F, TF }

```

---

$\backslash\_beanoves\_range:nN$   
 $\backslash\_beanoves\_range:VN$

---

$\backslash\_beanoves\_range:nN \{ \langle name \rangle \} \langle tl \ variable \rangle$   
Append the range of the  $\langle name \rangle$  slide range to the  $\langle tl \ variable \rangle$ .

```

790 \cs_new:Npn \__beanoves_range:nN #1 #2 {
791     \__beanoves_if_range:nNF { #1 } #2 {
792         \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
793     }
794 }
795 \cs_generate_variant:Nn \__beanoves_range:nN { VN }

```

### 5.3.6 Evaluation

---

$\backslash\_beanoves\_resolve:nnN$   
 $\backslash\_beanoves\_resolve:VVN$   
 $\backslash\_beanoves\_resolve:nnNN$   
 $\backslash\_beanoves\_resolve:VVNN$

---

$\backslash\_beanoves\_resolve:nnN \{ \langle name \rangle \} \{ \langle path \rangle \} \langle tl \ variable \rangle$   
 $\backslash\_beanoves\_resolve:nnNN \{ \langle name \rangle \} \{ \langle path \rangle \} \langle tl \ name \ variable \rangle \langle tl \ last \ variable \rangle$   
Resolve the  $\langle name \rangle$  and  $\langle path \rangle$  into a key that is put into the  $\langle tl \ name \ variable \rangle$ .  
 $\langle name_0 \rangle . \langle i_1 \rangle . \langle i_2 \rangle \dots \langle i_n \rangle$  is turned into  $\langle name_1 \rangle . \langle i_2 \rangle \dots \langle i_n \rangle$  where  $\langle name_0 \rangle . \langle i_1 \rangle$  is  $\langle name_1 \rangle$ ,  
then  $\langle name_2 \rangle . \langle i_3 \rangle \dots \langle i_n \rangle$  where  $\langle name_1 \rangle . \langle i_2 \rangle$  is  $\langle name_2 \rangle$ ... In the second version, the last  
path component is first removed from  $\{ \langle path \rangle \}$  and stored in  $\langle tl \ last \ variable \rangle$ .

```

796 \cs_new:Npn \__beanoves_resolve:nnN #1 #2 #3 {
797     \__beanoves_group_begin:
798     \tl_set:Nn \l_a_tl { #1 }
799     \regex_split:nnNT { \. } { #2 } \l_split_seq {
800         \seq_pop_left:NN \l_split_seq \l_b_tl
801         \cs_set:Npn \:n ##1 {
802             \tl_set_eq:NN \l_b_tl \l_a_tl
803             \tl_put_right:Nn \l_b_tl { . ##1 }
804             \exp_args:Nx
805             \__beanoves_get:nNTF { \l_b_tl / A } \l_c_tl {
806                 \exp_args:NNx
807                 \regex_match:NnTF \c__beanoves_A_key_Z_regex \l_c_tl {
808                     \tl_set_eq:NN \l_a_tl \l_c_tl
809                 } {
810                     \cs_set:Npn \:n ####1 {
811                         \tl_set_eq:NN \l_b_tl \l_a_tl
812                         \tl_put_right:Nn \l_b_tl { . ####1 }

```

```

813         \tl_set_eq:NN \l_a_tl \l_b_tl
814     }
815 }
816 } {
817     \tl_set_eq:NN \l_a_tl \l_b_tl
818 }
819 }
820 \seq_map_function:NN \l_split_seq \:n
821 }
822 \exp_args:NNNV
823 \__beanoves_group_end:
824 \tl_set:Nn #3 \l_a_tl
825 }
826 \cs_generate_variant:Nn \__beanoves_resolve:nnN { VVN }
827 \cs_new:Npn \__beanoves_tl_put_right_braced:Nn #1 #2 {
828     \tl_put_right:Nn #1 { { #2 } }
829 }
830 \cs_generate_variant:Nn \__beanoves_tl_put_right_braced:Nn { NV }
831 \cs_new:Npn \__beanoves_resolve:nnNN #1 #2 #3 #4 {
832     \__beanoves_group_begin:
833     \regex_extract_once:nnNT { (\.\d+)*? (\.\d+) \Z} { #2 } \l_match_seq {
834         \exp_args:Nnx
835         \__beanoves_resolve:nnN { #1 } { \seq_item:Nn \l_match_seq 2 } \l_name_tl
836         \tl_set:Nn \l_a_tl {
837             \tl_set:Nn #3
838         }
839         \exp_args:NNV
840         \__beanoves_tl_put_right_braced:Nn \l_a_tl \l_name_tl
841         \tl_put_right:Nn \l_a_tl {
842             \tl_set:Nn #4
843         }
844         \exp_args:NNx
845         \__beanoves_tl_put_right_braced:Nn \l_a_tl {
846             \seq_item:Nn \l_match_seq 3
847         }
848     }
849     \exp_last_unbraced:NV
850     \__beanoves_group_end:
851     \l_a_tl
852 }
853 \cs_generate_variant:Nn \__beanoves_resolve:nnNN { VVNN }

```

---

<code>\__beanoves_if_append:nNTF</code>	<code>\__beanoves_if_append:nNTF {&lt;integer expression&gt;} &lt;tl variable&gt; {&lt;true</code>
<code>\__beanoves_if_append:(VN xN)TF</code>	<code>code)} {&lt;false code&gt;}</code>

---

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

`\l_ans_tl` To feed *<tl variable>* with.

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

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

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

`\l__beanoves_split_int` Is the index of the non queries, before all the caught groups.

(End definition for `\l__beanoves_split_int`.)

`\l_name_tl` Storage for `\l_split_seq` items that represent names.

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

`\l_path_tl` Storage for `\l_split_seq` items that represent integer paths.

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

Catch circular definitions.

```

854 \prg_new_conditional:Npnn \__beanoves_if_append:nN #1 #2 { T, F, TF } {
855   \__beanoves_DEBUG:x { IF_APPEND:tl_to_str:n { #1 } / \string #2}
856   \int_gdecr:N \g__beanoves_append_int
857   \int_compare:nNnTF \g__beanoves_append_int > 0 {
858     \__beanoves_DEBUG:x { IF_APPEND...}
859     \__beanoves_group_begin:

```

Local variables:

```

860   \int_zero:N \l__beanoves_split_int
861   \seq_clear:N \l_split_seq
862   \tl_clear:N \l_name_tl
863   \tl_clear:N \l_path_tl
864   \tl_clear:N \l_group_tl
865   \tl_clear:N \l_ans_tl
866   \tl_clear:N \l_a_tl

```

Implementation:

```

867   \regex_split:NnN \c__beanoves_split_regex { #1 } \l_split_seq
868   \__beanoves_DEBUG:x { SPLIT_SEQ: / \seq_use:Nn \l_split_seq / / }
869   \int_set:Nn \l__beanoves_split_int { 1 }
870   \tl_set:Nx \l_ans_tl {
871     \seq_item:Nn \l_split_seq { \l__beanoves_split_int }
872   }

```

---

<code>\switch:nTF</code>	<code>\switch:nTF {&lt;capture group number&gt;} {&lt;black code&gt;} {&lt;white code&gt;}</code>
--------------------------	---

---

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

```

873     \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
874       \tl_set:Nx ##2 {
875         \seq_item:Nn \l_split_seq { \l__beanoves_split_int + ##1 }
876       }
877     \__beanoves_DEBUG:x { IF_APPEND_SWITCH/##1/\string##2/\tl_to_str:N##2/}
878     \tl_if_empty:NTF ##2 { %SWITCH~APPEND~WHITE/##1/\
879       ##4 } { %SWITCH~APPEND~BLACK/##1/\
880       ##3
881     }
882   }

```

\prg\_return\_true: and \prg\_return\_false: are wrapped locally to close the group and return the proper value.

```

883     \cs_set:Npn \__beanoves_return_true: {
884       \__beanoves_fp_round:
885       \exp_args:NNNV
886       \__beanoves_group_end:
887       \tl_put_right:Nn #2 \l_ans_tl
888     \__beanoves_DEBUG:x { IF_APPEND_TRUE:\tl_to_str:n { #1 } /
889     \string #2=\tl_to_str:V #2 }
890     \prg_return_true:
891   }
892   \cs_set:Npn \__beanoves_fp_round: {
893     \__beanoves_fp_round:N \l_ans_tl
894   }
895   \cs_set:Npn \next: {
896     \__beanoves_return_true:
897   }
898   \cs_set:Npn \__beanoves_return_false: {
899     \__beanoves_group_end:
900     \__beanoves_DEBUG:x { IF_APPEND_FALSE:\tl_to_str:n { #1 } /
901     \string #2=\tl_to_str:V #2 }
902     \prg_return_false:
903   }
904   \cs_set:Npn \break: {
905     \bool_set_false:N \l__beanoves_continue_bool
906     \cs_set:Npn \next: {
907       \__beanoves_return_false:
908     }
909   }

```

Main loop.

```

910   \bool_set_true:N \l__beanoves_continue_bool
911   \bool_while_do:Nn \l__beanoves_continue_bool {
912     \int_compare:nNnTF {
913       \l__beanoves_split_int } < { \seq_count:N \l_split_seq
914     } {
915       \switch:nNTF 1 \l_name_tl {

```

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

```

916       \switch:nNTF 2 \l_path_tl {
917         \__beanoves_resolve:VVN \l_name_tl \l_path_tl \l_name_tl
918       } { }
919       \__beanoves_if_incr:VnNF \l_name_tl 1 \l_ans_tl {

```

```

920         \break:
921     }
922 } {
923     \switch:nNTF 3 \l_name_tl {

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

924     \tl_set:Nn \l_b_tl {
925         \switch:nNTF 4 \l_path_tl {
926             \__beanoves_resolve:VFN \l_name_tl \l_path_tl \l_name_tl
927         } { }
928     }
929     \switch:nNTF 5 \l_a_tl {

• Case ...length.

930     \l_b_tl
931     \__beanoves_raw_length:VNF \l_name_tl \l_ans_tl {
932         \break:
933     }
934 } {
935     \switch:nNTF 6 \l_a_tl {

• Case ...last.

936     \l_b_tl
937     \__beanoves_raw_last:VNF \l_name_tl \l_ans_tl {
938         \break:
939     }
940 } {
941     \switch:nNTF 7 \l_a_tl {

• Case ...next.

942     \l_b_tl
943     \__beanoves_if_next:VNF \l_name_tl \l_ans_tl {
944         \break:
945     }
946 } {
947     \switch:nNTF 8 \l_a_tl {

• Case ...range.

948     \l_b_tl
949     \__beanoves_if_range:VNTF \l_name_tl \l_ans_tl {
950         \cs_set_eq:NN \__beanoves_fp_round: \relax
951     } {
952         \break:
953     }

954 } {
955     \switch:nNTF 9 \l_a_tl {

• Case ...n.

956     \l_b_tl
957     \switch:nNTF { 10 } \l_a_tl {

```



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

```

958 \__beanoves_if_incr:VWNF \l_name_tl \l_a_tl \l_ans_tl {
959   \break:
960 }
961   } {
962 \__beanoves_DEBUG:x {+++++++~NAME=\l_name_tl}
963   \__beanoves_if_counter:VNF \l_name_tl \l_ans_tl {
964     \break:
965   }
966   }
967   } {

```

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

```

968   \switch:nNTF 4 \l_path_tl {
969 \exp_args:NVV
970 \__beanoves_if_index:nnNF \l_name_tl \l_path_tl \l_ans_tl {
971   \break:
972 }
973   } {
974 \exp_args:Nx
975 \__beanoves_if_counter:nNF { \l_name_tl } \l_ans_tl {
976   \break:
977 }
978   }
979   }
980   }
981   }
982   }
983   }
984   } {

```

No name.

```

985   }
986   }
987   \int_add:Nn \l__beanoves_split_int { 11 }
988   \tl_put_right:Nx \l_ans_tl {
989     \seq_item:Nn \l_split_seq { \l__beanoves_split_int }
990   }
991   } {
992     \bool_set_false:N \l__beanoves_continue_bool
993   }
994   }
995   \next:
996   } {
997     \msg_error:nxx
998     { beanoves } { :n } { Too-many-calls:~\tl_to_str:n { #1 } }
999     \__beanoves_return_false:
1000   }
1001   }
1002 \prg_generate_conditional_variant:Nnn
1003 \__beanoves_if_append:nN { VN, xN } { T, F, TF }

```

---

<code>\_beanoves_if_eval_query:nNTF</code>	<code>\_beanoves_if_eval_query:nNTF {&lt;overlay query&gt;} &lt;tl variable&gt; {&lt;true code&gt;} {&lt;false code&gt;}</code>
--	---

---

Evaluates the single *<overlay query>*, 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 *<seq variable>*. Ranges are supported with the colon syntax. This is executed within a local group. Below are local variables and constants.

`\l_a_tl` Storage for the first index of a range.

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

`\l_b_tl` Storage for the last index of a range, or its length.

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

`\c__beanoves_A_cln_Z_regex` Used to parse slide range overlay specifications. Next are the capture groups.

(End definition for `\c__beanoves_A_cln_Z_regex`.)

```

1004 \regex_const:Nn \c__beanoves_A_cln_Z_regex {
1005   \A \s* (?
    • 2: <first>
    ( [^:]* ) \s* :
    • 3: second optional colon
    (:)? \s*
    • 4: <length>
    ( [^:]* )
    • 5: standalone <first>
    | ( [^:]+ )
    ) \s* \Z
1011 }

1012 \prg_new_conditional:Npnn \_beanoves_if_eval_query:nN #1 #2 { T, F, TF } {
1013   \_beanoves_DEBUG:x { EVAL_QUERY:#1/
1014     \tl_to_str:n{#1}/\string#2=\tl_to_str:N #2}
1015   \int_gset:Nn \g__beanoves_append_int { 128 }
1016   \regex_extract_once:NnNTF \c__beanoves_A_cln_Z_regex {
1017     #1
1018   } \l_match_seq {
1019   \_beanoves_DEBUG:x { EVAL_QUERY:#1/
1020     \string\l_match_seq/\seq_use:Nn \l_match_seq //}
1021     \bool_set_false:N \l__beanoves_no_counter_bool
1022     \bool_set_false:N \l__beanoves_no_range_bool

```

---

<code>\switch:nNTF</code>	<code>\switch:nNTF {&lt;capture group number&gt;} &lt;tl variable&gt; {&lt;black code&gt;} {&lt;white code&gt;}</code>
---------------------------	--

---

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

```

1023     \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1024 \__beanoves_DEBUG:x { SWITCH:##1/ }
1025     \tl_set:Nx ##2 {
1026     \seq_item:Nn \l_match_seq { ##1 }
1027     }
1028 \__beanoves_DEBUG:x { \string ##2/ \tl_to_str:N ##2/}
1029     \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1030     }
1031     \switch:nNTF 5 \l_a_tl {

```

Single expression

```

1032     \bool_set_false:N \l__beanoves_no_range_bool
1033     \__beanoves_if_append:VNTF \l_a_tl #2 {
1034     \prg_return_true:
1035     } {
1036     \prg_return_false:
1037     }
1038     } {
1039     \switch:nNTF 2 \l_a_tl {
1040     \switch:nNTF 4 \l_b_tl {
1041     \switch:nNTF 3 \l_a_tl {

```

$\langle first \rangle :: \langle last \rangle$  range

```

1042     \__beanoves_if_append:VNTF \l_a_tl #2 {
1043     \tl_put_right:Nn #2 { - }
1044     \__beanoves_if_append:VNTF \l_b_tl #2 {
1045     \prg_return_true:
1046     } {
1047     \prg_return_false:
1048     }
1049     } {
1050     \prg_return_false:
1051     }
1052     } {

```

$\langle first \rangle : \langle length \rangle$  range

```

1053     \__beanoves_if_append:VNTF \l_a_tl #2 {
1054     \tl_put_right:Nx #2 { - }
1055     \tl_put_right:Nx \l_a_tl { - ( \l_b_tl ) + 1 }
1056     \__beanoves_if_append:VNTF \l_a_tl #2 {
1057     \prg_return_true:
1058     } {
1059     \prg_return_false:
1060     }
1061     } {
1062     \prg_return_false:
1063     }
1064     }
1065     } {

```

$\langle first \rangle$ : and  $\langle first \rangle ::$  range

```

1066     \__beanoves_if_append:VNTF \l_a_tl #2 {
1067     \tl_put_right:Nn #2 { - }
1068     \prg_return_true:
1069     } {

```

```

1070         \prg_return_false:
1071     }
1072 }
1073 } {
1074     \switch:nNTF 4 \l_b_tl {
1075         \switch:nNTF 3 \l_a_tl {
1076             ::<last> range
1077             \tl_put_right:Nn #2 { - }
1078             \__beanoves_if_append:VNTF \l_a_tl #2 {
1079                 \prg_return_true:
1080             } {
1081                 \prg_return_false:
1082             }
1083 } {
1084 \msg_error:nnx { beanoves } { :n } { Syntax~error(Missing~first):~#1 }
1085 } {
1086     : or :: range
1087     \seq_put_right:Nn #2 { - }
1088 }
1089 }
1090 } {
Error
1091     \msg_error:nnn { beanoves } { :n } { Syntax~error:~#1 }
1092 }
1093 }

```

---

\\_\_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 ??.)

\l\_ans\_seq Storage of the evaluated result.

(End definition for \l\_ans\_seq. This variable is documented on page ??.)

\c\_\_beanoves\_comma\_regex Used to parse slide range overlay specifications.

```

1094 \regex_const:Nn \c__beanoves_comma_regex { \s* , \s* }

```

(End definition for \c\_\_beanoves\_comma\_regex.)

No other variable is used.

```

1095 \cs_new:Npn \__beanoves_eval:nN #1 #2 {
1096     EVAL:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2\\
1097     \__beanoves_group_begin:

```

Local variables declaration

```
1098 \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  $\langle first \rangle :: \langle last \rangle$  range expressions as well. We first split the expression around commas, into  $\backslash l\_query\_seq$ .

```
1099 \regex_split:NnN \c__beanoves_comma_regex { #1 } \l_query_seq
```

Then each component is evaluated and the result is stored in  $\backslash l\_ans\_seq$  that we have clear before use.

```
1100 \seq_map_inline:Nn \l_query_seq {
1101   \tl_clear:N \l_ans_tl
1102   \__beanoves_if_eval_query:nNTF { ##1 } \l_ans_tl {
1103     \seq_put_right:NV \l_ans_seq \l_ans_tl
1104   } {
1105     \seq_map_break:n {
1106       \msg_fatal:nnn { beanoves } { :n } { Circular~dependency~in~#1}
1107     }
1108   }
1109 }
```

We have managed all the comma separated components, we collect them back and append them to  $\langle tl\ variable \rangle$ .

```
1110 \exp_args:NNNx
1111 \__beanoves_group_end:
1112 \tl_put_right:Nn #2 { \seq_use:Nn \l_ans_seq , }
1113 }
1114 \cs_generate_variant:Nn \__beanoves_eval:nN { VN, xN }
```

---

$\backslash$ BeanovesEval	$\backslash$ BeanovesEval [ $\langle tl\ variable \rangle$ ] [ $\langle overlay\ queries \rangle$ ]
---------------------------	---

---

$\langle overlay\ queries \rangle$  is the argument of  $?(\dots)$  instructions. This is a comma separated list of single  $\langle overlay\ query \rangle$ 's.

This function evaluates the  $\langle overlay\ queries \rangle$  and store the result in the  $\langle tl\ variable \rangle$  when provided or leave the result in the input stream. Forwards to  $\backslash \_beanoves\_eval:nN$  within a group.  $\backslash l\_ans\_tl$  is used locally to store the result.

```
1115 \NewExpandableDocumentCommand \BeanovesEval { s o m } {
1116   \__beanoves_group_begin:
1117   \tl_clear:N \l_ans_tl
1118   \IfBooleanTF { #1 } {
1119     \bool_set_true:N \l__beanoves_no_counter_bool
1120   } {
1121     \bool_set_false:N \l__beanoves_no_counter_bool
1122   }
1123   \__beanoves_eval:nN { #3 } \l_ans_tl
1124   \IfValueTF { #2 } {
1125     \exp_args:NNNV
1126     \__beanoves_group_end:
1127     \tl_set:Nn #2 \l_ans_tl
1128   } {
1129     \exp_args:NV
1130     \__beanoves_group_end: \l_ans_tl
1131   }
1132 }
```

### 5.3.7 Reseting slide ranges

---

```

\BeanovesReset \beanovesReset [⟨first value⟩] {⟨Slide list name⟩}

1133 \NewDocumentCommand \BeanovesReset { 0{1} m } {
1134   \__beanoves_reset:nn { #1 } { #2 }
1135   \ignorespaces
1136 }

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 (not usefull).

1137 \cs_new:Npn \__beanoves_reset:nn #1 #2 {
1138   \bool_if:nTF {
1139     \__beanoves_if_in_p:n { #2/A } || \__beanoves_if_in_p:n { #2/Z }
1140   } {
1141     \__beanoves_gremove:n { #2/C }
1142     \__beanoves_gremove:n { #2//A }
1143     \__beanoves_gremove:n { #2//L }
1144     \__beanoves_gremove:n { #2//Z }
1145     \__beanoves_gremove:n { #2//N }
1146     \__beanoves_gput:nn { #2/C0 } { #1 }
1147   } {
1148     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1149   }
1150 }

1151 \makeatother
1152 \ExplSyntaxOff
1153 ⟨/package⟩

```