

# 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

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Minimal example</b>                          | <b>2</b> |
| <b>2</b> | <b>Named slide lists</b>                        | <b>2</b> |
| 2.1      | Presentation . . . . .                          | 2        |
| 2.2      | Defining named slide lists . . . . .            | 2        |
| <b>3</b> | <b>Named overlay specifications</b>             | <b>3</b> |
| 3.1      | Named slide ranges . . . . .                    | 3        |
| 3.2      | Named slide lists . . . . .                     | 5        |
| <b>4</b> | <b>?(...) query expressions</b>                 | <b>5</b> |
| <b>5</b> | <b>Implementation</b>                           | <b>5</b> |
| 5.1      | Package declarations . . . . .                  | 6        |
| 5.2      | logging and debugging facilities . . . . .      | 6        |
| 5.3      | Local variables . . . . .                       | 7        |
| 5.4      | Infinite loop management . . . . .              | 7        |
| 5.5      | Overlay specification . . . . .                 | 8        |
| 5.5.1    | In slide range definitions . . . . .            | 8        |
| 5.5.2    | Regular expressions . . . . .                   | 10       |
| 5.5.3    | beamer.cls interface . . . . .                  | 13       |
| 5.5.4    | Defining named slide ranges . . . . .           | 13       |
| 5.5.5    | Scanning named overlay specifications . . . . . | 19       |
| 5.5.6    | Resolution . . . . .                            | 23       |
| 5.5.7    | Evaluation bricks . . . . .                     | 33       |
| 5.5.8    | Evaluation . . . . .                            | 45       |
| 5.5.9    | Reseting slide ranges . . . . .                 | 55       |

# 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-debug}
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 of 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 at least 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 possibly 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

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

happens to be 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, % or equivalently A = 3::8 or A = :6::8,
3 }
4 }
5 \begin{frame} {Frame \insertframenum} {Slide \insertslidenumber}
6 \ttfamily
7 \BeanovesEval(A.1)      == 3,
8 \BeanovesEval(A.length) == 6,
9 \BeanovesEval(A.last)   == 8,
10 \BeanovesEval(A.next)   == 9,
11 \BeanovesEval(A.range)  == 3-8,
12 \end{frame}

```

Using these specifications 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

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

$\langle name \rangle.n+=\langle integer \rangle$  : 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

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

## 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 ‘? $\langle queries \rangle$ ’. 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  $?(\mathbf{A}.next)$ ,  $?(\mathbf{A}.last+1)$ ,  $?(\mathbf{A}.1+\mathbf{A}.length)$  give the same result as soon as the slide range named ‘A’ has been properly defined with a starting value and a length.

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

<sup>1</sup>  $\langle *package \rangle$

## 5 Implementation

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

<sup>2</sup>  $\langle @@=bnvs \rangle$

## 5.1 Package declarations

```

3 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
4 \ProvidesExplPackage
5 <*gubed&:!debug>
6   {beanoves}
7 </gubed&:!debug>
8 <*debug&:!gubed>
9   {beanoves-debug}
10 </debug&:!gubed>
11   {2022/10/28}
12   {1.0}
13   {Named overlay specifications for beamer}

```

## 5.2 logging and debugging facilities

Utility message.

```

14 \msg_new:nnn { beanoves } { :n } { #1 }
15 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
16 <*debug&:!gubed>
17 \cs_set:Npn \__bnvs_DEBUG_:nn #1 #2 {
18   \msg_term:nnn { beanoves } { :n } { #1~#2 }
19 }
20 \cs_new:Npn \__bnvs_DEBUG_on: {
21   \cs_set:Npn \__bnvs_DEBUG:n {
22     \exp_args:Nx
23     \__bnvs_DEBUG_:nn
24     { \prg_replicate:nn {\l__bnvs_group_int} { } \space }
25   }
26 }
27 \cs_new:Npn \__bnvs_DEBUG_off: {
28   \cs_set_eq:NN \__bnvs_DEBUG:n \use_none:n
29 }
30 \__bnvs_DEBUG_off:
31 \cs_generate_variant:Nn \__bnvs_DEBUG:n { x, V }
32 \int_zero_new:N \l__bnvs_group_int
33 \cs_set:Npn \__bnvs_group_begin: {
34   \group_begin:
35   \int_incr:N \l__bnvs_group_int
36 }
37 \cs_set_eq:NN \__bnvs_group_end: \group_end:
38 \cs_new:Npn \__bnvs_DEBUG__:nn #1 #2 {
39   \__bnvs_DEBUG:x { #1~#2 }
40 }
41 \cs_new:Npn \__bnvs_DEBUG:nn #1 {
42   \exp_args:Nx
43   \__bnvs_DEBUG__:nn
44   { \prg_replicate:nn {\l__bnvs_group_int + 1} {#1} }
45 }
46 \cs_generate_variant:Nn \__bnvs_DEBUG:nn { nx, nV }
47 </debug&:!gubed>
48 <*gubed&:!debug>
49 \cs_set_eq:NN \__bnvs_group_begin: \group_begin:
50 </gubed&:!debug>

```

### 5.3 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.

```

51 \int_new:N \l__bnvs_depth_int
52 \bool_new:N \l__bnvs_ask_bool
53 \bool_new:N \l__bnvs_query_bool
54 \bool_new:N \l__bnvs_no_counter_bool
55 \bool_new:N \l__bnvs_no_range_bool
56 \bool_new:N \l__bnvs_continue_bool
57 \bool_new:N \l__bnvs_in_frame_bool
58 \bool_set_false:N \l__bnvs_in_frame_bool
59 \tl_new:N \l__bnvs_id_current_tl
60 \tl_new:N \l__bnvs_a_tl
61 \tl_new:N \l__bnvs_b_tl
62 \tl_new:N \l__bnvs_c_tl
63 \tl_new:N \l__bnvs_id_tl
64 \tl_new:N \l__bnvs_ans_tl
65 \tl_new:N \l__bnvs_name_tl
66 \tl_new:N \l__bnvs_path_tl
67 \tl_new:N \l__bnvs_group_tl
68 \tl_new:N \l__bnvs_query_tl
69 \tl_new:N \l__bnvs_token_tl
70 \seq_new:N \l__bnvs_a_seq
71 \seq_new:N \l__bnvs_b_seq
72 \seq_new:N \l__bnvs_ans_seq
73 \seq_new:N \l__bnvs_match_seq
74 \seq_new:N \l__bnvs_split_seq
75 \seq_new:N \l__bnvs_path_seq
76 \seq_new:N \l__bnvs_query_seq
77 \seq_new:N \l__bnvs_token_seq

```

### 5.4 Infinite loop management

Unending recursivity is managed here.

$\backslash\text{g\_bnvs\_call\_int}$

```

78 \int_zero_new:N \g__bnvs_call_int
79 \int_const:Nn \c__bnvs_max_call_int { 2048 }

```

*(End definition for  $\backslash\text{g\_bnvs\_call\_int}$ .)*

---

$\backslash\text{__bnvs\_call\_reset}$ :  $\backslash\text{__bnvs\_call\_reset}$ :

---

Reset the call stack counter.

```

80 \cs_set:Npn \__bnvs_call_reset: {
81   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
82 }

```

---

\\_bnvs\\_call:TF    \\_bnvs\\_call\\_do:TF {< *true code* >} {< *false code* >}

---

Decrement the \\_g\\_bnvs\\_call\\_int counter globally and execute < *true code* > if we have not reached 0, < *false code* > otherwise.

```

83 \prg_new_conditional:Npnn \_bnvs\_call: { T, F, TF } {
84   \int_gdecr:N \_g\_bnvs\_call\_int
85   \int_compare:nNnTF \_g\_bnvs\_call\_int > 0 {
86     \prg_return_true:
87   } {
88     \prg_return_false:
89   }
90 }
```

## 5.5 Overlay specification

### 5.5.1 In slide range definitions

\\_g\\_bnvs\\_prop    <key>-<value> property list to store the named slide lists. The basic keys are, assuming <id>!<name> is a fully qualified slide list name,

<id>!<name>/A for the first index

<id>!<name>/L for the length when provided

<id>!<name>/Z for the last index when provided

<id>!<name>/C for the counter value, when used

<id>!<name>/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 '//'.

<id>!<name>//A for the cached static value of the first index

<id>!<name>//Z for the cached static value of the last index

<id>!<name>//L for the cached static value of the length

<id>!<name>//N for the cached static value of the next index

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

```

91 \prop_new:N \_g\_bnvs\_prop
```

(End definition for \\_g\\_bnvs\\_prop.)



---

```

\__bnvs_gput:nn
\__bnvs_gput:nV
\__bnvs_gprovide:nn
\__bnvs_gprovide:nV
\__bnvs_item:n
\__bnvs_get:nN
\__bnvs_gremove:n
\__bnvs_gclear:n
\__bnvs_gclear:
\__bnvs_gclear_cache:n
\__bnvs_gclear:

```

---

```

\__bnvs_gput:nn {<key>} {<value>}
\__bnvs_gprovide:nn {<key>} {<value>}
\__bnvs_item:n {<key>}
\__bnvs_get:n {<key>} {<tl variable>}
\__bnvs_gremove:n {<key>}
\__bnvs_gclear:n {<key>}
\__bnvs_gclear_cache:n {<key>}
\__bnvs_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 `\__bnvs_gprovide:nn` which meaning is straightforward.

```

92 \cs_new:Npn \__bnvs_gput:nn #1 #2 {
93   <*debug&!gubed>
94   \__bnvs_DEBUG:x {\string\__bnvs_gput:nn/key:#1/value:#2/}
95   </debug&!gubed>
96   \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
97 }
98 \cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
99   <*debug&!gubed>
100   \__bnvs_DEBUG:x {\string\__bnvs_gprovide:nn/key:#1/value:#2/}
101   </debug&!gubed>
102   \prop_if_in:NnF \g__bnvs_prop { #1 } {
103     \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
104   }
105 }
106 \cs_new:Npn \__bnvs_item:n {
107   \prop_item:Nn \g__bnvs_prop
108 }
109 \cs_new:Npn \__bnvs_get:nN {
110   \prop_get:NnN \g__bnvs_prop
111 }
112 \cs_new:Npn \__bnvs_gremove:n {
113   \prop_gremove:Nn \g__bnvs_prop
114 }
115 \cs_new:Npn \__bnvs_gclear:n #1 {
116   \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
117     \__bnvs_gremove:n { #1 / ##1 }
118   }
119 }
120 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
121   \clist_map_inline:nn { /A, /L, /Z, /N } {
122     \__bnvs_gremove:n { #1 / ##1 }
123   }
124 }
125 \cs_new:Npn \__bnvs_gclear: {
126   \prop_gclear:N \g__bnvs_prop
127 }
128 \cs_generate_variant:Nn \__bnvs_gput:nn { nV }
129 \cs_generate_variant:Nn \__bnvs_gprovide:nn { nV }

```

---

|  |   |
|--|---|
| <code>\__bnvs_if_in_p:n *</code><br><code>\__bnvs_if_in_p:V *</code><br><code>\__bnvs_if_in:nTF *</code><br><code>\__bnvs_if_in:VTF *</code> | <code>\__bnvs_if_in_p:n {&lt;key&gt;}</code><br><code>\__bnvs_if_in:nTF {&lt;key&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code><br>Convenient shortcuts to test for the existence of some key, it makes the code more concise and readable. |
|--|---|

---

```

130 \prg_new_conditional:Npnn \__bnvs_if_in:n #1 { p, T, F, TF } {
131   \prop_if_in:NnTF \g__bnvs_prop { #1 } {
132     \prg_return_true:
133   } {
134     \prg_return_false:
135   }
136 }
137 \prg_generate_conditional_variant:Nnn \__bnvs_if_in:n {V} { p, T, F, TF }

```

---

|   |  |
|---|--|
| <code>\__bnvs_get:nNTF</code><br><code>\__bnvs_get:nnNTF</code> | <code>\__bnvs_get:nNTF {&lt;key&gt;} {&lt;tl variable&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code><br><code>\__bnvs_get:nnNTF {&lt;id&gt;} {&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.

```

138 \prg_new_conditional:Npnn \__bnvs_get:nN #1 #2 { T, F, TF } {
139   \prop_get:NnNTF \g__bnvs_prop { #1 } #2 {
140     <*debug&!gubed>
141     \__bnvs_DEBUG:x { \string\__bnvs_get:nN\space TRUE/
142       #1/\string#2:#2/
143     }
144     </debug&!gubed>
145     \prg_return_true:
146   } {
147     <*debug&!gubed>
148     \__bnvs_DEBUG:x { \string\__bnvs_get:nN\space FALSE/#1/\string#2/ }
149     </debug&!gubed>
150     \prg_return_false:
151   }
152 }

```

### 5.5.2 Regular expressions

|                                  |   |
|----------------------------------|---|
| <code>\c__bnvs_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. |
|----------------------------------|---|

```

153 \regex_const:Nn \c__bnvs_name_regex {
154   [[:alpha:]]_+ [[:alnum:]]_*
155 }

```

(End definition for `\c__bnvs_name_regex`.)

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

```

156 \regex_const:Nn \c__bnvs_id_regex {
157   (?: \ur{c__bnvs_name_regex} | [?]* ) ? !
158 }

```

(End definition for `\c__bnvs_id_regex`.)

`\c__bnvs_path_regex` A sequence of `.⟨positive integer⟩` items representing a path.

```
159 \regex_const:Nn \c__bnvs_path_regex {
160   (? : \. [+-]? \d+ ) *
161 }
```

(End definition for `\c__bnvs_path_regex`.)

`\c__bnvs_key_regex` A key is the name of a slide range possibly followed by positive integer attributes using  
`\c__bnvs_A_key_Z_regex` a dot syntax. The ‘A\_key\_Z’ variant matches the whole string.

```
162 \regex_const:Nn \c__bnvs_key_regex {
163   \ur{c__bnvs_id_regex} ?
164   \ur{c__bnvs_name_regex}
165   \ur{c__bnvs_path_regex}
166 }
167 \regex_const:Nn \c__bnvs_A_key_Z_regex {
```

2: slide `⟨id⟩`

3: question mark, when `⟨id⟩` is empty

4: The range name

```
168   \A ( ( \ur{c__bnvs_id_regex} ? ) \ur{c__bnvs_name_regex} )
```

5: the path, if any.

```
169   ( \ur{c__bnvs_path_regex} ) \Z
170 }
171
```

(End definition for `\c__bnvs_key_regex` and `\c__bnvs_A_key_Z_regex`.)

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

```
172 \regex_const:Nn \c__bnvs_colons_regex { :(:+)? }
```

(End definition for `\c__bnvs_colons_regex`.)

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

```
173 \regex_const:Nn \c__bnvs_list_regex {
174   \A \[ \s*
```

Capture groups:

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

```
175   ( [^\] %[---
176   ]*? )
177   \s* \] \Z
178 }
```

(End definition for `\c__bnvs_list_regex`.)

`\c__bnvs_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.

```
179 \regex_const:Nn \c__bnvs_split_regex {
180   \s* ( ? :
```

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

- 1:  $\langle name \rangle$  of a slide range
- 2:  $\langle id \rangle$  of a slide range plus the exclamation mark

```
181   \+\+ ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_name_regex} )
```

- 3: optionally followed by an integer path

```
182   ( \ur{c__bnvs_path_regex} ) (?: \. n )?
```

We continue with other expressions

- 4: fully qualified  $\langle name \rangle$  of a slide range,
- 5:  $\langle id \rangle$  of a slide range plus the exclamation mark (to manage void  $\langle id \rangle$ )

```
183   | ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_name_regex} )
```

- 6: optionally followed by an integer path

```
184   ( \ur{c__bnvs_path_regex} )
```

Next comes another branching

```
185   (?:
```

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

```
186   \. l(e)ngth
```

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

```
187   | \. l(a)st
```

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

```
188   | \. ne(x)t
```

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

```
189   | \. (r)ange
```

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

```
190   | \. (n)
```

• 12: the poor man integer expression after ‘+=’, which is the longest sequence of black characters, which ends just before a space or at the very last character. This tricky definition allows quite any algebraic expression, even those involving parenthesis.

```
191   (?: \s* \+= \s* ( \S+ ) )?
```

```
192   )?
```

```
193   ) \s*
```

```
194   }
```

(End definition for `\c__bnvs_split_regex`.)

---

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

### 5.5.3 beamer.cls interface

Work in progress.

```

195 \RequirePackage{keyval}
196 \define@key{beamerframe}{beanoves~id}[] {
197   \tl_set:Nx \l__bnvs_id_current_tl { #1 ! }
198   <*debug&!gubed>
199   \__bnvs_DEBUG_on:
200   \__bnvs_DEBUG:x {THIS_IS_KEY}
201   \__bnvs_DEBUG_off:
202 </debug&!gubed>
203 }
204 \AddToHook{env/beamer@frameslide/before}{
205   \bool_set_true:N \l__bnvs_in_frame_bool
206   <*debug&!gubed>
207   \__bnvs_DEBUG_on:
208   \__bnvs_DEBUG:x {THIS_IS_BEFORE}
209   \__bnvs_DEBUG_off:
210 </debug&!gubed>
211 }
212 \AddToHook{env/beamer@frameslide/after}{
213   \bool_set_false:N \l__bnvs_in_frame_bool
214   <*debug&!gubed>
215   \__bnvs_DEBUG_on:
216   \__bnvs_DEBUG:x {THIS_IS_BEFORE}
217   \__bnvs_DEBUG_off:
218 </debug&!gubed>
219 }
220 \AddToHook{cmd/frame/before}{
221   \tl_set:Nn \l__bnvs_id_current_tl { ?! }
222   <*debug&!gubed>
223   \__bnvs_DEBUG_on:
224   \__bnvs_DEBUG:x {THIS_IS_FRAME}
225   \__bnvs_DEBUG_off:
226 </debug&!gubed>
227 }

```

### 5.5.4 Defining named slide ranges

---

\\_\_bnvs\_parse:Nnn    \\_\_bnvs\_parse:Nnn <command> {<key>} {<definition>}

---

Auxiliary function called within a group. <key> is the slide range key, including eventually a dotted integer path and a slide identifier, <definition> is the corresponding definition. <command> is \\_\_bnvs\_range:nVVV at runtime.

\l\_\_bnvs\_match\_seq    Local storage for the match result.  
 (End definition for \l\_\_bnvs\_match\_seq.)

---

```

__bnvs_range:nnnn
__bnvs_range:nVVV
__bnvs_range_alt:nnnn
__bnvs_range_alt:nVVV
__bnvs_range:Nnnnn

```

---

```

__bnvs_range:nnnn {<key>} {<first>} {<length>} {<last>}
__bnvs_range_alt:nnnn {<key>} {<first>} {<length>} {<last>}
__bnvs_range:Nnnnn <cmd> {<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.

Implementation detail: the core functionality is implemented in the function `__bnvs_range:Nnnnn` which first argument is `__bnvs_gput:nn` for `__bnvs_range:nnnn` and `__bnvs_gprovide:nn` for `__bnvs_range_alt:nnnn`.

```

228 \cs_new:Npn __bnvs_range:Nnnnn #1 #2 #3 #4 #5 {
229   \*debug&!gubed
230   __bnvs_DEBUG:x {\string__bnvs_range:Nnnnn/\string#1/#2/#3/#4/#5/}
231   \</debug&!gubed
232   \tl_if_empty:nTF { #3 } {
233     \tl_if_empty:nTF { #4 } {
234       \tl_if_empty:nTF { #5 } {
235         \msg_error:nnn { beanoves } { :n } { Not~a~range::~~#2 }
236       } {
237         #1 { #2/Z } { #5 }
238       }
239     } {
240       #1 { #2/L } { #4 }
241       \tl_if_empty:nF { #5 } {
242         #1 { #2/Z } { #5 }
243         #1 { #2/A } { #2.last - (#2.length) + 1 }
244       }
245     }
246   } {
247     #1 { #2/A } { #3 }
248     \tl_if_empty:nTF { #4 } {
249       \tl_if_empty:nF { #5 } {
250         #1 { #2/Z } { #5 }
251         #1 { #2/L } { #2.last - (#2.1) + 1 }
252       }
253     } {
254       #1 { #2/L } { #4 }
255       #1 { #2/Z } { #2.1 + #2.length - 1 }
256     }
257   }
258 }
259 \cs_new:Npn __bnvs_range:nnnn #1 {
260   __bnvs_gclear:n { #1 }
261   __bnvs_range:Nnnnn __bnvs_gput:nn { #1 }
262 }
263 \cs_generate_variant:Nn __bnvs_range:nnnn { nVVV }
264 \cs_new:Npn __bnvs_range_alt:nnnn #1 {
265   __bnvs_gclear_cache:n { #1 }
266   __bnvs_range:Nnnnn __bnvs_gprovide:nn { #1 }
267 }
268 \cs_generate_variant:Nn __bnvs_range_alt:nnnn { nVVV }

```

---

\\\_bnvs\_parse:Nn    \\\_bnvs\_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.

```

269 \cs_new:Npn \\_bnvs_parse:Nn #1 #2 {
270   \\_bnvs_group_begin:
271   \\_bnvs_id_name_set:nNNTF { #2 } \\_bnvs_id_tl \\_bnvs_name_tl {
272     \exp_args:Nx \\_bnvs_gput:nn { \\_bnvs_name_tl/ } { }
273     \exp_args:NNNV
274     \\_bnvs_group_end:
275     \tl_set:Nn \\_bnvs_id_current_tl \\_bnvs_id_current_tl
276   } {
277     \msg_error:nnn { beanoves } { :n } { Unexpected-key:~#2 }
278     \\_bnvs_group_end:
279   }
280 }
```

---

\\\_bnvs\_do\_parse:Nnn    \\\_bnvs\_do\_parse:Nnn <command> {<full name>}

---

Auxiliary function for \\\_bnvs\_parse:Nn. <command> is \\\_bnvs\_range:nVVV at run-time and must have signature nVVV.

```

281 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
282 \cs_new:Npn \\_bnvs_do_parse:Nnn #1 #2 #3 {
283   <*debug&!gubed>
284   \\_bnvs_DEBUG:x {\\string\\_bnvs_do_parse:Nnn/\\string#1/#2/#3}
285   </debug&!gubed>
```

This is not a list.

```

286   \tl_clear:N \\_bnvs_a_tl
287   \tl_clear:N \\_bnvs_b_tl
288   \tl_clear:N \\_bnvs_c_tl
289   \regex_split:NnN \\_bnvs_colons_regex { #3 } \\_bnvs_split_seq
290   \seq_pop_left:NNT \\_bnvs_split_seq \\_bnvs_a_tl {
```

\\\_bnvs\_a\_tl may contain the <start>.

```

291     \seq_pop_left:NNT \\_bnvs_split_seq \\_bnvs_b_tl {
292       \tl_if_empty:NNTF \\_bnvs_b_tl {
```

This is a one colon range.

```

293         \seq_pop_left:NN \\_bnvs_split_seq \\_bnvs_b_tl
```

\\\_bnvs\_b\_tl may contain the <length>.

```

294         \seq_pop_left:NNT \\_bnvs_split_seq \\_bnvs_c_tl {
295         \tl_if_empty:NNTF \\_bnvs_c_tl {
```

A :: was expected:

```

296   \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(1):~#3 }
297   } {
298     \int_compare:nNnT { \tl_count:N \\_bnvs_c_tl } > { 1 } {
299   \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(2):~#3 }
300   }
301   \seq_pop_left:NN \\_bnvs_split_seq \\_bnvs_c_tl
```

\l\_c\_tl may contain the  $\langle end \rangle$ .

```

302         \seq_if_empty:NF \l__bnvs_split_seq {
303   \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(3):~#3 }
304         }
305       }
306     }
307   } {

```

This is a two colon range.

```

308         \int_compare:nNt { \tl_count:N \l__bnvs_b_tl } > { 1 } {
309   \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(4):~#3 }
310         }
311       \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl

```

\l\_c\_tl contains the  $\langle end \rangle$ .

```

312       \seq_pop_left:NNTF \l__bnvs_split_seq \l__bnvs_b_tl {
313         \tl_if_empty:NNTF \l__bnvs_b_tl {
314           \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl

```

\l\_b\_tl may contain the  $\langle length \rangle$ .

```

315         \seq_if_empty:NF \l__bnvs_split_seq {
316   \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(5):~#3 }
317         }
318       } {
319   \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(6):~#3 }
320       }
321     } {
322       \tl_clear:N \l__bnvs_b_tl
323     }
324   }
325 }
326 }

```

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.

```

327   \bool_if:nF {
328     \tl_if_empty_p:N \l__bnvs_a_tl
329     || \tl_if_empty_p:N \l__bnvs_b_tl
330     || \tl_if_empty_p:N \l__bnvs_c_tl
331   } {
332   \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(7):~#3 }
333   }
334   #1 { #2 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
335 }
336 \cs_generate_variant:Nn \__bnvs_do_parse:Nnn { Nxn, Non }
337 \cs_new:Npn \__bnvs_parse_old:Nnn #1 #2 #3 {
338   \__bnvs_group_begin:
339   \regex_match:NnTF \c__bnvs_A_key_Z_regex { #2 } {

```

We got a valid key.

```

340   \regex_extract_once:NnNTF \c__bnvs_list_regex { #3 } \l__bnvs_match_seq {

```

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

```

341     \exp_args:NNx
342     \seq_set_from_clist:Nn \l__bnvs_match_seq {

```



```

343     \seq_item:Nn \l__bnvs_match_seq { 2 }
344   }
345   \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
346     \__bnvs_do_parse:Nnn #1 { #2.##1 } { ##2 }
347   }
348 } {
349   \__bnvs_do_parse:Nnn #1 { #2 } { #3 }
350 }
351 } {
352   \msg_error:nnn { beanoves } { :n } { Invalid~key:~#1 }
353 }
354 \__bnvs_group_end:
355 }

```

---

\\_\_bnvs\_id\_name\_set:nNNTF \\_\_bnvs\_id\_name\_set:nNNTF {<key>} <id tl var> <full name tl var> {< true code>} {< false code>}

If the <key> is a key, put the name it defines into the <name tl var> with the current frame id prefix \l\_\_bnvs\_id\_tl if none was given, then execute <true code>. Otherwise execute <false code>.

```

356 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
357   \__bnvs_group_begin:
358   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex { #1 } \l__bnvs_match_seq {
359     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
360     \tl_if_empty:NNTF #2 {
361       \exp_args:NNNx
362       \__bnvs_group_end:
363       \tl_set:Nn #3 { \l__bnvs_id_current_tl #1 }
364       \tl_set_eq:NN #2 \l__bnvs_id_current_tl
365     } {
366       \cs_set:Npn \:n ##1 {
367         \__bnvs_group_end:
368         \tl_set:Nn #2 { ##1 }
369         \tl_set:Nn \l__bnvs_id_current_tl { ##1 }
370       }
371       \exp_args:NV
372       \:n #2
373       \tl_set:Nn #3 { #1 }
374     }
375   } <*debug&!gubed>
376   \__bnvs_DEBUG:x { \string\__bnvs_id_name_set:nNN\space TRUE/#1/
377     \string#2:#2/\string#3:#3/\string\l__bnvs_id_current_tl:\l__bnvs_id_current_tl/
378   }
379   </debug&!gubed>
380   \prg_return_true:
381 } {
382   \__bnvs_group_end:
383   <*debug&!gubed>
384   \__bnvs_DEBUG:x { \string\__bnvs_id_name_set:nNN\space FALSE/#1/
385     \string#2/#2/\string#3/#3/
386   }
387   </debug&!gubed>
388   \prg_return_false:
389 }

```

```

390 }

391 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
392   <*debug&!gubed>
393   \__bnvs_DEBUG:x {\string\__bnvs_parse:Nnn/\string#1/#2/#3/}
394   </debug&!gubed>
395   \__bnvs_group_begin:
396   \__bnvs_id_name_set:nNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
397     <*debug&!gubed>
398     \__bnvs_DEBUG:x {key:#2/ID:\l__bnvs_id_tl/NAME:\l__bnvs_name_tl/}
399     </debug&!gubed>%</debug&!gubed>
400     \regex_extract_once:NnTF \c__bnvs_list_regex { #3 } \l__bnvs_match_seq {

```

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

```

401     \exp_args:NNx
402     \seq_set_from_clist:Nn \l__bnvs_match_seq {
403       \seq_item:Nn \l__bnvs_match_seq { 2 }
404     }
405     \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
406       \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl.##1 } { ##2 }
407     }
408   } {
409     \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
410   }
411 } {
412   \msg_error:nnn { beanoves } { :n } { Invalid-key:~#2 }
413 }

```

We export \l\_\_bnvs\_id\_tl:

```

414   \exp_args:NNNV
415   \__bnvs_group_end:
416   \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
417 }

```

---

**\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 `\__bnvs_parse:Nnn`.

```
418 \NewDocumentCommand \Beanoves { sm } {
419   \tl_if_eq:NnT \@currenvir { document } {
420     \__bnvs_gclear:
421   }
422   \IfBooleanTF {#1} {
423     \keyval_parse:nnn {
424       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
425     } {
426       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
427     }
428   } {
429     \keyval_parse:nnn {
430       \__bnvs_parse:Nn \__bnvs_range:nVVV
431     } {
432       \__bnvs_parse:Nnn \__bnvs_range:nVVV
433     }
434   }
435   { #2 }
436   \ignorespaces
437 }
```

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

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

### 5.5.5 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\_\_bnvs\_ans\_tl**

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

(End definition for `\l__bnvs_ans_tl`.)

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

```
439 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
440 \cs_set:Npn \beamer@frame < #1 > {
441   \__bnvs_group_begin:
442   \tl_clear:N \l__bnvs_ans_tl
443   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
444   \exp_args:NNNV
445   \__bnvs_group_end:
446   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
447 }
448 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
```

```

449 \cs_set:Npn \beamer@masterdecode #1 {
450   \__bnvs_group_begin:
451   \tl_clear:N \l__bnvs_ans_tl
452   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
453   \exp_args:NNV
454   \__bnvs_group_end:
455   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
456 }

```

---

\\_\_bnvs\_scan:nNN \\_\_bnvs\_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 \\_\_bnvs\_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\_\_bnvs\_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\_\_bnvs\_depth\_int.)

\l\_\_bnvs\_query\_tl Storage for the overlay query expression to be evaluated.

(End definition for \l\_\_bnvs\_query\_tl.)

\l\_\_bnvs\_token\_seq The *<overlay expression>* is split into the sequence of its tokens.

(End definition for \l\_\_bnvs\_token\_seq.)

\l\_\_bnvs\_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\_\_bnvs\_ask\_bool.)

\l\_\_bnvs\_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\_\_bnvs\_query\_bool.)

\l\_\_bnvs\_token\_tl Storage for just one token.

(End definition for \l\_\_bnvs\_token\_tl.)

```

457 \cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 {
458   \__bnvs_group_begin:
459   \tl_clear:N \l__bnvs_ans_tl
460   \int_zero:N \l__bnvs_depth_int
461   \seq_clear:N \l__bnvs_token_seq

```

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

```

462   \regex_split:nnN {} { #1 } \l__bnvs_token_seq

```

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

```

463   \bool_set_true:N \l__bnvs_ask_bool

```

```

464 \bool_while_do:Nn \l__bnvs_ask_bool {
465   \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
466   \quark_if_no_value:NTF \l__bnvs_token_tl {

```

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

```

467   \bool_set_false:N \l__bnvs_ask_bool
468 } {

```

\l\_token\_tl contains a ‘normal’ token.

```

469   \tl_if_eq:NnTF \l__bnvs_token_tl { ? } {

```

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

```

470   \bool_set_true:N \l__bnvs_query_bool
471   \bool_while_do:Nn \l__bnvs_query_bool {

```

Get next token.

```

472   \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
473   \quark_if_no_value:NTF \l__bnvs_token_tl {

```

No opening parenthesis found, raise.

```

474   \msg_fatal:nmx { beanoves } { :n } {Missing~(' %---)
475   ~after~a~?:~#1}
476 } {
477   \tl_if_eq:NnT \l__bnvs_token_tl { ( % )
478 } {

```

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

```

479   \int_incr:N \l__bnvs_depth_int

```

Record the forthcoming content in the \l\_query\_tl variable, up to the next balancing ‘)’.

```

480   \tl_clear:N \l__bnvs_query_tl
481   \bool_while_do:Nn \l__bnvs_query_bool {

```

Get next token.

```

482   \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
483   \quark_if_no_value:NTF \l__bnvs_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\_\_bnvs\_depth\_int is 0 whenever \l\_\_bnvs\_query\_bool is false.

```

484   \msg_error:nmx { beanoves } { :n } {Missing~%(---
485   ~):~#1 }
486   \int_do_while:nNnn \l__bnvs_depth_int > 1 {
487     \int_decr:N \l__bnvs_depth_int
488     \tl_put_right:Nn \l__bnvs_query_tl {%(---
489     )}
490   }
491   \int_zero:N \l__bnvs_depth_int
492   \bool_set_false:N \l__bnvs_query_bool
493   \bool_set_false:N \l__bnvs_ask_bool
494 } {
495   \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
496 } {

```

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

```

497         \int_incr:N \l__bnvs_depth_int
498         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
499     } {

```

This is not a ‘(’.

```

500         \tl_if_eq:NnTF \l__bnvs_token_tl { %(
501         )
502     } {

```

We found a ‘)’, decrement the depth.

```

503         \int_decr:N \l__bnvs_depth_int
504         \int_compare:nNnTF \l__bnvs_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.

```

505     \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
506     \bool_set_false:N \l__bnvs_query_bool
507 } {

```

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

```

508         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
509     }

```

Above ends the code for a positive depth.

```

510 } {

```

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

```

511         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
512     }
513 }
514 }

```

Above ends the code for Not a ‘(’

```

515     }
516 }

```

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

```

517 }

```

Above ends the code for not a no value quark.

```

518 }

```

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.

```

519     \quark_if_no_value:NT \l__bnvs_token_tl {
520         \bool_set_false:N \l__bnvs_query_bool
521         \bool_set_false:N \l__bnvs_ask_bool
522     }
523 } {

```

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

```

524     \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
525 }

```

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

```
526     }
527 }
```

Above ends the outer bool while loop to find ‘?’ characters. We can append our result to *<tl variable>*

```
528 \exp_args:NNNV
529 \__bnvs_group_end:
530 \tl_put_right:Nn #3 \l__bnvs_ans_tl
531 }
```

I

### 5.5.6 Resolution

Given a frame id, a name and an integer path, we resolve any intermediate standalone reference. For example, with A=B and B=C, A is resolved in C. But with A=B+1 and B=C, A is not resolved in C+1. With A=B:D and B=C, A is not resolved in C:D as well.

---

```
\__bnvs_extract_key:NNNTF <id tl var> <name tl var> <path seq var> {<true code>}
{<false code>}
```

---

Auxiliary function. *<id tl var>* contains a frame id whereas *<name tl var>* contains a range name. If we recognize a key, on return, *<name tl var>* contains the resolved name, *<path seq var>* is prepended with new integer path components, *{<true code>}* is executed, otherwise *{<false code>}* is executed.

```
532 \exp_args_generate:n { VVx }
533 \prg_new_conditional:Npnn \__bnvs_extract_key:NNN
534     #1 #2 #3 { T, F, TF } {
535 <*debug&!gubed>
536 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN/
537     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn#3./
538 }
539 </debug&!gubed>
540 \__bnvs_group_begin:
541 \exp_args:NNV
542 \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex #2 \l__bnvs_match_seq {
```

This is a correct key, update the path sequence accordingly

```
543 \exp_args:Nx
544 \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
545     \tl_put_left:NV #2 { #1 }
546 <*debug&!gubed>
547 \__bnvs_DEBUG:x { VERIF~\tl_to_str:V #2 }
548 </debug&!gubed>
549 }
550 \exp_args:NNnx
551 \seq_set_split:Nnn \l__bnvs_split_seq . { \seq_item:Nn \l__bnvs_match_seq 4 }
552 \seq_remove_all:Nn \l__bnvs_split_seq { }
553 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
554 \seq_if_empty:NTF \l__bnvs_split_seq {
```

No new integer path component is added.

```
555     \cs_set:Npn \:nn ##1 ##2 {
556       \__bnvs_group_end:
557       \tl_set:Nn #1 { ##1 }
558       \tl_set:Nn #2 { ##2 }
559     }
560     \exp_args:NVV \:nn #1 #2
561 <*debug&!gubed>
562 \__bnvs_DEBUG:x { END/\string#1:#1/\string#2:#2/ }
563 </debug&!gubed>
564 } {
```

Some new integer path components are added.

```
565 <*debug&!gubed>
566 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN/\string#1:#1/
567   \string#2:#2/\string#3:\seq_use:Nn#3./
568   \string\l__bnvs_split_seq:\seq_use:Nn\l__bnvs_split_seq./
569 }
570 </debug&!gubed>
571 \cs_set:Npn \:nnn ##1 ##2 ##3 {
572   \__bnvs_group_end:
573   \tl_set:Nn #1 { ##1 }
574   \tl_set:Nn #2 { ##2 }
575   \seq_set_split:Nnn #3 . { ##3 }
576   \seq_remove_all:Nn #3 { }
577 }
578 \exp_args:NVVx
579 \:nnn #1 #2 {
580   \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
581 }
582 <*debug&!gubed>
583 \__bnvs_DEBUG:x { END/\string#1:#1/\string#2:#2/
584   \string#3:\seq_use:Nn #3 . /
585   \string\l__bnvs_split_seq:\seq_use:Nn \l__bnvs_split_seq . /
586 }
587 </debug&!gubed>%</debug&!gubed>
588 }
589 <*debug&!gubed>
590 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN\space TRUE/
591   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3 . /
592 }
593 </debug&!gubed>
594 \prg_return_true:
595 } {
596   \__bnvs_group_end:
597 <*debug&!gubed>
598 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN\space FALSE/
599   \string#1/\string#2/\string#3/
600 }
601 </debug&!gubed>
602 \prg_return_false:
603 }
604 }
```



---

```

\__bnvs_resolve:NNNTF <id tl var> <name tl var> <path seq var> {<true code>}
{<false code>}

```

---

When too many nested calls occurred,  $\{\langle false\ code\rangle\}$  is executed directly.  $\langle id\ tl\ var\rangle$ ,  $\langle name\ tl\ var\rangle$  and  $\langle path\ seq\ var\rangle$  are meant to contain proper information. On input,  $\{\langle id\ tl\ var\rangle\}$  contains a frame id,  $\{\langle name\ tl\ var\rangle\}$  contains a range name and  $\{\langle path\ seq\ var\rangle\}$  contains the components of an integer path, possibly empty. On return,  $\langle id\ tl\ var\rangle$  contains the frame id used,  $\langle name\ tl\ var\rangle$  contains the resolved range name and  $\langle path\ seq\ var\rangle$  contains the sequence of integer path components that could not be resolved. To resolve a path,  $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$  is turned into  $\langle name_1\rangle.\langle i_2\rangle...\langle i_n\rangle$  where  $\langle name_0\rangle.\langle i_1\rangle$  is  $\langle name_1\rangle$ , then  $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$  where  $\langle name_1\rangle.\langle i_2\rangle$  is  $\langle name_2\rangle...$  If the above rule does not apply,  $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$  may turn into  $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$  when  $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle$  is  $\langle name_2\rangle...$  The algorithm is not yet more clever. The resolution algorithm is quite straightforward:

1. If  $\langle name\ tl\ var\rangle$  content is the name of an unlimited range, and the first item of this range is exactly another name range with eventually a heading frame identifier or a trailing integer path, then  $\langle name\ tl\ var\rangle$  is replaced by this name, the  $\langle id\ tl\ var\rangle$  and  $\backslash l\_bnvs\_id\_tl$  are updates accordingly and the  $\langle path\ seq\ var\rangle$  is prepended with the integer path.
2. If  $\langle path\ seq\ var\rangle$  is not empty, append to the right of  $\langle name\ tl\ var\rangle$  after a separating dot, all its left elements but the last one and loop. Otherwise return. None of the tl variables must be one of  $\backslash l\_a\_tl$ ,  $\backslash l\_b\_tl$  or  $\backslash l\_c\_tl$ . None of the seq variables must be one of  $\backslash l\_a\_seq$ ,  $\backslash l\_b\_seq$ .

```

605 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
606     #1 #2 #3 { T, F, TF } {
607   <*debug&!gubed>
608   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNN/
609     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
610   }
611   </debug&!gubed>
612   \__bnvs_group_begin:

```

Local variables:

- $\backslash l\_a\_tl$  contains the name with a partial index path currently resolved.
- $\backslash l\_a\_seq$  contains the index path components currently resolved.
- $\backslash l\_b\_tl$  contains the resolution.
- $\backslash l\_b\_seq$  contains the index path components to be resolved.

```

613   \seq_set_eq:NN \l__bnvs_a_seq #3
614   \seq_clear:N \l__bnvs_b_seq
615   \cs_set:Npn \loop: {
616     \__bnvs_call:TF {
617       \tl_set_eq:NN \l__bnvs_a_tl #2
618       \seq_if_empty:NNTF \l__bnvs_a_seq {
619         \exp_args:Nx
620         \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
621           \cs_set:Nn \loop: { \return_true: }
622         } {
623           \get_extract:F {

```

Unknown key <\l\_a\_tl>/A or the value for key <\l\_a\_tl>/A does not fit.

```

624         \cs_set:Nn \loop: { \return_true: }
625     }
626 } {
627 } {
628     \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
629     \get_extract:F {
630         \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
631             \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
632         }
633     }
634 }
635 \loop:
636 } {
637 <*debug&!gubed>
638 \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNN\space~TOO~MANY~CALLS/
639     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
640 }
641 </debug&!gubed>
642     \__bnvs_group_end:
643     \prg_return_false:
644 }
645 }
646 \cs_set:Npn \get_extract:F ##1 {
647     \exp_args:Nx
648     \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
649 <*debug&!gubed>
650 \__bnvs_DEBUG:x { RESOLUTION:~\l__bnvs_a_tl / A=>\l__bnvs_b_tl}
651 </debug&!gubed>
652     \__bnvs_extract_key:NNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
653         \tl_set_eq:NN #2 \l__bnvs_b_tl
654         \seq_set_eq:NN #3 \l__bnvs_b_seq
655         \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
656         \seq_clear:N \l__bnvs_b_seq
657     } { ##1 }
658 } { ##1 }
659 }
660 \cs_set:Npn \return_true: {
661     \cs_set:Npn \:nnn #####1 #####2 #####3 {
662         \__bnvs_group_end:
663         \tl_set:Nn #1 { #####1 }
664         \tl_set:Nn #2 { #####2 }
665         \seq_set_split:Nnn #3 . { #####3 }
666         \seq_remove_all:Nn #3 { }
667     }
668     \exp_args:NVVx
669     \:nnn #1 #2 {
670         \seq_use:Nn #3 .
671     }
672 <*debug&!gubed>
673 \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNN\space TRUE/
674     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
675 }
676 </debug&!gubed>

```

```

677     \prg_return_true:
678 }
679 \loop:
680 }

```

---

```

\__bnvs_resolve_n:NNNTF TF
\__bnvs_resolve_n:NNNTF <id tl var> <name tl var> <path seq var> {( true code)} {(
)} false code

```

---

The difference with the function above without `_n` is that resolution is performed only when there is an integer path afterwards

```

681 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
682   #1 #2 #3 { T, F, TF } {
683   <*debug&!gubed>
684   \__bnvs_DEBUG:x { \string\__bnvs_resolve_n:NNN/
685     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
686   }
687   </debug&!gubed>
688   \__bnvs_group_begin:

```

Local variables:

- `\l_a_tl` contains the name with a partial index path currently resolved.
- `\l_a_seq` contains the index path components currently resolved.
- `\l_b_tl` contains the resolution.
- `\l_b_seq` contains the index path components to be resolved.

```

689 \seq_set_eq:NN \l__bnvs_a_seq #3
690 \seq_clear:N \l__bnvs_b_seq
691 \cs_set:Npn \loop: {
692   \__bnvs_call:TF {
693     \tl_set_eq:NN \l__bnvs_a_tl #2
694     \seq_if_empty:NTF \l__bnvs_a_seq {
695       \exp_args:Nx
696       \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
697         \cs_set:Nn \loop: { \return_true: }
698       } {
699         \seq_if_empty:NTF \l__bnvs_b_seq {
700           \cs_set:Nn \loop: { \return_true: }
701         } {
702           \get_extract:F {

```

Unknown key `<\l_a_tl>/A` or the value for key `<\l_a_tl>/A` does not fit.

```

703       \cs_set:Nn \loop: { \return_true: }
704     }
705   }
706 } {
707   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
708   \get_extract:F {
709     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
710       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
711     }
712   }
713 }

```

```

714     }
715     \loop:
716     } {
717     <*debug&!gubed>
718     \__bnvs_DEBUG:x { \string\__bnvs_resolve_n:NNN\space~TOO~MANY~CALLS/
719     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
720     }
721     </debug&!gubed>
722     \__bnvs_group_end:
723     \prg_return_false:
724     }
725     }
726     \cs_set:Npn \get_extract:F ##1 {
727     \exp_args:Nx
728     \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
729     <*debug&!gubed>
730     \__bnvs_DEBUG:x { RESOLUTION:~\l__bnvs_a_tl / A=>\l__bnvs_b_tl}
731     </debug&!gubed>
732     \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
733     \tl_set_eq:NN #2 \l__bnvs_b_tl
734     \seq_set_eq:NN #3 \l__bnvs_b_seq
735     \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
736     \seq_clear:N \l__bnvs_b_seq
737     } { ##1 }
738     } { ##1 }
739     }
740     \cs_set:Npn \return_true: {
741     \cs_set:Npn \:nnn #####1 #####2 #####3 {
742     \__bnvs_group_end:
743     \tl_set:Nn #1 { #####1 }
744     \tl_set:Nn #2 { #####2 }
745     \seq_set_split:Nnn #3 . { #####3 }
746     \seq_remove_all:Nn #3 { }
747     }
748     \exp_args:NVVx
749     \:nnn #1 #2 {
750     \seq_use:Nn #3 .
751     }
752     <*debug&!gubed>
753     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve_n:NNN\space TRUE/
754     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
755     }
756     </debug&!gubed>
757     \prg_return_true:
758     }
759     \loop:
760     }

```

---

```

\__bnvs_resolve:NNNTF TF
\__bnvs_resolve:NNNTF <cs:nn> <id tl var> <name tl var> <path seq var> {< true
code>} {< >} false code

```

---

When too many nested calls occurred,  $\{false\}$  is executed directly.  $\langle id\ tl\ var \rangle$ ,  $\langle name\ tl\ var \rangle$  and  $\langle path\ seq\ var \rangle$  are meant to contain proper information. To resolve a path,  $\langle name_0 \rangle.\langle i_1 \rangle.\langle i_2 \rangle...\langle i_n \rangle$  is turned into  $\langle name_1 \rangle.\langle i_2 \rangle...\langle i_n \rangle$  where  $\langle name_0 \rangle.\langle i_1 \rangle$  is  $\langle name_1 \rangle$ , then  $\langle name_2 \rangle.\langle i_3 \rangle...\langle i_n \rangle$  where  $\langle name_1 \rangle.\langle i_2 \rangle$  is  $\langle name_2 \rangle...$ . If the above rule does not apply,  $\langle name_0 \rangle.\langle i_1 \rangle.\langle i_2 \rangle...\langle i_n \rangle$  may turn into  $\langle name_2 \rangle.\langle i_3 \rangle...\langle i_n \rangle$  when  $\langle name_0 \rangle.\langle i_1 \rangle.\langle i_2 \rangle$  is  $\langle name_2 \rangle...$ . We try to match the longest sequence of components first. The algorithm is not yet more clever. In general,  $\langle cs:nn \rangle$  is just  $\backslash use\_i:nn$  but for in place incrementation, we must resolve only when there is an integer path. See the implementation of the  $\backslash\_bnvs\_if\_append:...$  conditionals.

```

761 \prg_new_conditional:Npnn \__bnvs_resolve:NNNN
762   #1 #2 #3 #4 { T, F, TF } {
763   <*debug&!gubed>
764   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN/
765     \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
766   }
767   </debug&!gubed>
768   #1 {
769     \__bnvs_group_begin:

```

$\backslash l\_a\_tl$  contains the name with a partial index path currently resolved.  $\backslash l\_a\_seq$  contains the remaining index path components to be resolved.  $\backslash l\_b\_seq$  contains the current index path components to be resolved.

```

770     \tl_set_eq:NN \l__bnvs_a_tl #3
771     \seq_set_eq:NN \l__bnvs_a_seq #4
772     \tl_clear:N \l__bnvs_b_tl
773     \seq_clear:N \l__bnvs_b_seq
774     \cs_set:Npn \return_true: {
775       \cs_set:Npn \:nnn #####1 #####2 #####3 {
776         \__bnvs_group_end:
777         \tl_set:Nn #2 { #####1 }
778         \tl_set:Nn #3 { #####2 }
779         \seq_set_split:Nnn #4 . { #####3 }
780         \seq_remove_all:Nn #4 { }
781       }
782       \exp_args:NVVx
783       \:nnn #2 #3 {
784         \seq_use:Nn #4 .
785       }
786     <*debug&!gubed>
787     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNNN \space TRUE/
788       \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
789     }
790     </debug&!gubed>
791     \prg_return_true:
792   }
793   \cs_set:Npn \branch:n ##1 {
794     \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
795       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_b_tl
796     <*debug&!gubed>
797     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN \space POP~TRUE~##1 }

```

```

798 \__bnvs_DEBUG:x {\string\l__bnvs_b_tl:\l__bnvs_b_tl }
799 \__bnvs_DEBUG:x {\string\l__bnvs_a_seq:\seq_count:N\l__bnvs_a_seq/\seq_use:Nn \l__bnvs_a_seq
800 \__bnvs_DEBUG:x {\string\l__bnvs_b_seq:\seq_count:N\l__bnvs_b_seq/\seq_use:Nn \l__bnvs_b_seq
801 </debug&!\gubed>
      \tl_set:Nn \l__bnvs_a_tl { #3 . }
802 \tl_put_right:Nx \l__bnvs_a_tl { \seq_use:Nn \l__bnvs_a_seq . }
803 } {
804 \cs_set_eq:NN \loop: \return_true:
805 }
806 }
807 }
808 \cs_set:Npn \branch:FF ##1 ##2 {
809 \exp_args:Nx
810 \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
811 \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
812 \tl_set_eq:NN #3 \l__bnvs_b_tl
813 \seq_set_eq:NN #4 \l__bnvs_b_seq
814 \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
815 } { ##1 }
816 } { ##2 }
817 }
818 \cs_set:Npn \extract_key:F {
819 \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
820 \tl_set_eq:NN #3 \l__bnvs_b_tl
821 \seq_set_eq:NN #4 \l__bnvs_b_seq
822 \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
823 }
824 }
825 \cs_set:Npn \loop: {
826 \__bnvs_call:TF {
827 \exp_args:Nx
828 \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {

```

If there is a length, no resolution occurs.

```

829 \branch:n { 1 }
830 } {
831 \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
832 \seq_clear:N \l__bnvs_b_seq
833 \tl_set:Nn \l__bnvs_a_tl { #3 . }
834 \tl_put_right:Nx \l__bnvs_a_tl { \seq_use:Nn \l__bnvs_a_seq . . }
835 \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
836 \branch:FF {

```

The value for key <\l\_a\_tl>/L is not just a (qualified) name.

```

837 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
838 } {

```

Unknown key <\l\_a\_tl>/L.

```

839 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
840 }
841 } {
842 \branch:FF {
843 \cs_set_eq:NN \loop: \return_true:
844 } {
845 \cs_set:Npn \loop: {
846 \__bnvs_group_end:

```

```

847 <*debug&!gubed>
848 \_bnvs_DEBUG:x { \string\_bnvs_resolve:NNNN\space FALSE/
849 \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
850 \g__bnvs_call_int : \int_use:N\g__bnvs_call_int/
851 }
852 </debug&!gubed>
853 \prg_return_false:
854 }
855 }
856 }
857 }
858 } {
859 \cs_set:Npn \loop: {
860 \_bnvs_group_end:
861 <*debug&!gubed>
862 \_bnvs_DEBUG:x { \string\_bnvs_resolve:NNNN\space FALSE/
863 \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
864 \g__bnvs_call_int : \int_use:N\g__bnvs_call_int/
865 }
866 </debug&!gubed>
867 \prg_return_false:
868 }
869 }
870 \loop:
871 }
872 \loop:
873 } {
874 \prg_return_true:
875 }
876 }
877 \prg_new_conditional:Npnn \_bnvs_resolve_OLD:NNNN
878 #1 #2 #3 #4 { T, F, TF } {
879 <*debug&!gubed>
880 \_bnvs_DEBUG:x { \string\_bnvs_resolve:NNNN/
881 \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
882 }
883 </debug&!gubed>
884 #1 {
885 \_bnvs_group_begin:
886 \tl_set_eq:NN \l__bnvs_a_tl #3
887 \seq_set_eq:NN \l__bnvs_a_seq #4
888 \cs_set:Npn \return_true: {
889 \cs_set:Npn \:nnn ####1 ####2 ####3 {
890 \_bnvs_group_end:
891 \tl_set:Nn #2 { ####1 }
892 \tl_set:Nn #3 { ####2 }
893 \seq_set_split:Nnn #4 . { ####3 }
894 \seq_remove_all:Nn #4 { }
895 }
896 \exp_args:NVVx
897 \:nnn #2 #3 {

```

\l\_a\_tl contains the name with a partial index path to be resolved. \l\_a\_seq contains the remaining index path components to be resolved.

```

898         \seq_use:Nn #4 .
899     }
900     <*debug&!gubed>
901     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNNN\space TRUE/
902         \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
903     }
904     </debug&!gubed>
905     \prg_return_true:
906     }
907     \cs_set:Npn \branch:n ##1 {
908         \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
909     <*debug&!gubed>
910     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space POP~TRUE~##1/
911         \string\l__bnvs_b_tl:\l__bnvs_b_tl/\string\l__bnvs_a_seq:\seq_count:N\l__bnvs_a_seq/
912         \seq_use:Nn \l__bnvs_a_seq ./
913     }
914     </debug&!gubed>
915         \tl_put_right:Nn \l__bnvs_a_tl { . }
916         \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
917     } {
918         \cs_set_eq:NN \loop: \return_true:
919     }
920     }
921     \cs_set:Npn \loop: {
922         \__bnvs_call:TF {
923             \exp_args:Nx
924             \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
925                 \branch:n { 1 }
926             } {
927                 \exp_args:Nx
928                 \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
929                     \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
930                         \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
931                         \tl_set_eq:NN #3 \l__bnvs_b_tl
932                         \seq_set_eq:NN #4 \l__bnvs_a_seq
933                     } {
934                         \branch:n { 2 }
935                     }
936                 } {
937                     \branch:n { 3 }
938                 }
939             }
940         } {
941             \cs_set:Npn \loop: {
942                 \__bnvs_group_end:
943     <*debug&!gubed>
944     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space FALSE/
945         \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
946         \string\g__bnvs_call_int : \int_use:N\g__bnvs_call_int/
947     }
948     </debug&!gubed>
949         \prg_return_false:
950     }
951     }

```



```

952     \loop:
953   }
954   \loop:
955 } {
956   \prg_return_true:
957 }
958 }

```

### 5.5.7 Evaluation bricks

---

|   |  |
|---|--|
| <code>\_bnvs_fp_round:nN</code><br><code>\_bnvs_fp_round:N</code> | <code>\_bnvs_fp_round:nN {&lt;expression&gt;} &lt;tl variable&gt;</code><br><code>\_bnvs_fp_round:N &lt;tl variable&gt;</code> |
|---|--|

---

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

```

959 \cs_new:Npn \_bnvs_fp_round:nN #1 #2 {
960   <*debug&!gubed>
961   \_bnvs_DEBUG:x { ROUND:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2}
962   </debug&!gubed>
963   \tl_if_empty:nTF { #1 } {
964     <*debug&!gubed>
965     \_bnvs_DEBUG:x { ...ROUND:~EMPTY }
966     </debug&!gubed>
967   } {
968     \tl_put_right:Nx #2 {
969       \fp_eval:n { round(#1) }
970     }
971     <*debug&!gubed>
972     \_bnvs_DEBUG:x { ...ROUND:~\tl_to_str:V #2 => \string#2}
973     </debug&!gubed>
974   }
975 }
976 \cs_generate_variant:Nn \_bnvs_fp_round:nN { VN, xN }
977 \cs_new:Npn \_bnvs_fp_round:N #1 {
978   \tl_if_empty:VTF #1 {
979     <*debug&!gubed>
980     \_bnvs_DEBUG:x { ROUND:~EMPTY }
981     </debug&!gubed>
982   } {
983     <*debug&!gubed>
984     \_bnvs_DEBUG:x { ROUND~IN:~\tl_to_str:V #1 }
985     </debug&!gubed>
986     \tl_set:Nx #1 {
987       \fp_eval:n { round(#1) }
988     }
989     <*debug&!gubed>
990     \_bnvs_DEBUG:x { ROUND~OUT:~\tl_to_str:V #1 }
991     </debug&!gubed>
992   }
993 }

```

---

`\\_bnvs_raw_first:nNTF`  
`\\_bnvs_raw_first:(xN|VN)TF`

---

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

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

```

994 \cs_set:Npn \\_bnvs_return_true:nnN #1 #2 #3 {
995   \tl_if_empty:NTF \l__bnvs_ans_tl {
996     \\_bnvs_group_end:
997     <*debug&!gubed>
998     \\_bnvs_DEBUG:n { RETURN_FALSE/key=#1/type=#2/EMPTY }
999     </debug&!gubed>
1000     \\_bnvs_gremove:n { #1/#2 }
1001     \prg_return_false:
1002   } {
1003     \\_bnvs_fp_round:N \l__bnvs_ans_tl
1004     \\_bnvs_gput:nV { #1/#2 } \l__bnvs_ans_tl
1005     \exp_args:NNNV
1006     \\_bnvs_group_end:
1007     \tl_put_right:Nn #3 \l__bnvs_ans_tl
1008     <*debug&!gubed>
1009     \\_bnvs_DEBUG:x { RETURN_TRUE/key=#1/type=#2/ans=\l__bnvs_ans_tl/ }
1010     </debug&!gubed>
1011     \prg_return_true:
1012   }
1013 }
1014 \cs_set:Npn \\_bnvs_return_false:nn #1 #2 {
1015   <*debug&!gubed>
1016   \\_bnvs_DEBUG:n { RETURN_FALSE/key=#1/type=#2/ }
1017   </debug&!gubed>
1018   \\_bnvs_group_end:
1019   \\_bnvs_gremove:n { #1/#2 }
1020   \prg_return_false:
1021 }
1022 \prg_new_conditional:Npnn \\_bnvs_raw_first:nN #1 #2 { T, F, TF } {
1023   <*debug&!gubed>
1024   \\_bnvs_DEBUG:x { RAW_FIRST/
1025     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2/
1026   }
1027   </debug&!gubed>
1028   \\_bnvs_if_in:nTF { #1/A } {
1029     <*debug&!gubed>
1030     \\_bnvs_DEBUG:n { RAW_FIRST/#1/CACHED }
1031     </debug&!gubed>
1032     \tl_put_right:Nx #2 { \\_bnvs_item:n { #1/A } }
1033     \prg_return_true:
1034   } {
1035     <*debug&!gubed>
1036     \\_bnvs_DEBUG:n { RAW_FIRST/key=#1/NOT_CACHED }
1037     </debug&!gubed>
1038     \\_bnvs_group_begin:
1039     \tl_clear:N \l__bnvs_ans_tl
1040     \\_bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
1041     <*debug&!gubed>
1042     \\_bnvs_DEBUG:x { RAW_FIRST/key=#1/A=\l__bnvs_a_tl }
1043     </debug&!gubed>

```

```

1044     \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1045         \__bnvs_return_true:nnN { #1 } A #2
1046     } {
1047         \__bnvs_return_false:nn { #1 } A
1048     }
1049 } {
1050 <*debug&!gubed>
1051 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/A/F }
1052 </debug&!gubed>
1053     \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
1054 <*debug&!gubed>
1055 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/L=\l__bnvs_a_tl }
1056 </debug&!gubed>
1057     \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
1058 <*debug&!gubed>
1059 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/Z=\l__bnvs_b_tl }
1060 </debug&!gubed>
1061     \__bnvs_if_append:xNTF {
1062         \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
1063     } \l__bnvs_ans_tl {
1064         \__bnvs_return_true:nnN { #1 } A #2
1065     } {
1066         \__bnvs_return_false:nn { #1 } A
1067     }
1068 } {
1069 <*debug&!gubed>
1070 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/Z/F/ }
1071 </debug&!gubed>
1072     \__bnvs_return_false:nn { #1 } A
1073 }
1074 } {
1075 <*debug&!gubed>
1076 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/L/F/ }
1077 </debug&!gubed>
1078     \__bnvs_return_false:nn { #1 } A
1079 }
1080 }
1081 }
1082 }
1083 \prg_generate_conditional_variant:Nnn
1084     \__bnvs_raw_first:nN { VN, xN } { T, F, TF }

```

---

\\_\_bnvs\_if\_first:nNTF    \\_\_bnvs\_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.

```

1085 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
1086 <*debug&!gubed>
1087 \__bnvs_DEBUG:x { IF_FIRST/\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2}
1088 </debug&!gubed>
1089 \__bnvs_raw_first:nNTF { #1 } #2 {
1090     \prg_return_true:

```

```

1091 } {
1092   \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {
1093     <*debug&!gubed>
1094     \__bnvs_DEBUG:n { IF_FIRST/#1/C/T/\l__bnvs_a_tl }
1095     </debug&!gubed>
1096     \bool_set_true:N \l_no_counter_bool
1097     \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1098       \__bnvs_return_true:nnN { #1 } A #2
1099     } {
1100       \__bnvs_return_false:nn { #1 } A
1101     }
1102   } {
1103     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1104       \__bnvs_gput:nn { #1/A } { 1 }
1105       \tl_set:Nn #2 { 1 }
1106     <*debug&!gubed>
1107     \__bnvs_DEBUG:x{IF_FIRST_MATCH:
1108       key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /
1109     }
1110     </debug&!gubed>
1111     \__bnvs_return_true:nnN { #1 } A #2
1112   } {
1113     <*debug&!gubed>
1114     \__bnvs_DEBUG:x{IF_FIRST_NO_MATCH:
1115       key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2
1116     }
1117     </debug&!gubed>
1118     \__bnvs_return_false:nn { #1 } A
1119   }
1120 }
1121 }
1122 }

```

---

```

\__bnvs_first:nN
\__bnvs_first:VN

```

---

\\_\_bnvs\_first:nN {<name>} <tl variable>

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

```

1123 \cs_new:Npn \__bnvs_first:nN #1 #2 {
1124   \__bnvs_if_first:nNF { #1 } #2 {
1125     \msg_error:nnn { beanoves } { :n } { Range-with-no~first:~#1 }
1126   }
1127 }
1128 \cs_generate_variant:Nn \__bnvs_first:nN { VN }

```

---

```

\__bnvs_raw_length:nNTF

```

---

\\_\_bnvs\_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.

```

1129 \prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
1130   <*debug&!gubed>
1131   \__bnvs_DEBUG:x { \string\__bnvs_raw_length:nN/#1/\string#2/ }
1132   </debug&!gubed>
1133   \__bnvs_if_in:nTF { #1//L } {
1134     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//L } }

```

```

1135 <*debug&!gubed>
1136 \_bnvs_DEBUG:x { RAW_LENGTH/CACHED/key:#1/\_bnvs_item:n { #1//L } }
1137 </debug&!gubed>
1138 \prg_return_true:
1139 } {
1140 <*debug&!gubed>
1141 \_bnvs_DEBUG:x { RAW_LENGTH/NOT_CACHED/key:#1/ }
1142 </debug&!gubed>
1143 \_bnvs_gput:nn { #1//L } { 0 }
1144 \_bnvs_group_begin:
1145 \tl_clear:N \l__bnvs_ans_tl
1146 \_bnvs_if_in:nTF { #1/L } {
1147 \_bnvs_if_append:xNTF {
1148 \_bnvs_item:n { #1/L }
1149 } \l__bnvs_ans_tl {
1150 \_bnvs_return_true:nnN { #1 } L #2
1151 } {
1152 \_bnvs_return_false:nn { #1 } L
1153 }
1154 } {
1155 \_bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
1156 \_bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
1157 \_bnvs_if_append:xNTF {
1158 \l__bnvs_b_tl - (\l__bnvs_a_tl) + 1
1159 } \l__bnvs_ans_tl {
1160 \_bnvs_return_true:nnN { #1 } L #2
1161 } {
1162 \_bnvs_return_false:nn { #1 } L
1163 }
1164 } {
1165 \_bnvs_return_false:nn { #1 } L
1166 }
1167 } {
1168 \_bnvs_return_false:nn { #1 } L
1169 }
1170 }
1171 }
1172 }
1173 \prg_generate_conditional_variant:Nnn
1174 \_bnvs_raw_length:nN { VN } { T, F, TF }

```

---

\\_bnvs\_raw\_last:nNTF \\_bnvs\_raw\_last:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Put the last index of the fully qualified <name> range to the right of the <tl variable>, when possible. Execute <true code> when a last index was given, <false code> otherwise.

```

1175 \prg_new_conditional:Npnn \_bnvs_raw_last:nN #1 #2 { T, F, TF } {
1176 <*debug&!gubed>
1177 \_bnvs_DEBUG:n { RAW_LAST/#1 }
1178 </debug&!gubed>
1179 \_bnvs_if_in:nTF { #1//Z } {
1180 \tl_put_right:Nx #2 { \_bnvs_item:n { #1//Z } }
1181 \prg_return_true:
1182 } {

```

```

1183     \__bnvs_gput:nn { #1//Z } { 0 }
1184     \__bnvs_group_begin:
1185     \tl_clear:N \l__bnvs_ans_tl
1186     \__bnvs_if_in:nTF { #1/Z } {
1187     <*debug&!gubed>
1188     \__bnvs_DEBUG:x { NORMAL_RAW_LAST:~\__bnvs_item:n { #1/Z } }
1189     </debug&!gubed>
1190     \__bnvs_if_append:xNTF {
1191     \__bnvs_item:n { #1/Z }
1192     } \l__bnvs_ans_tl {
1193     \__bnvs_return_true:nnN { #1 } Z #2
1194     } {
1195     \__bnvs_return_false:nn { #1 } Z
1196     }
1197     } {
1198     \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
1199     \__bnvs_get:nNTF { #1/L } \l__bnvs_b_tl {
1200     \__bnvs_if_append:xNTF {
1201     \l__bnvs_a_tl + (\l__bnvs_b_tl) - 1
1202     } \l__bnvs_ans_tl {
1203     \__bnvs_return_true:nnN { #1 } Z #2
1204     } {
1205     \__bnvs_return_false:nn { #1 } Z
1206     }
1207     } {
1208     \__bnvs_return_false:nn { #1 } Z
1209     }
1210     } {
1211     \__bnvs_return_false:nn { #1 } Z
1212     }
1213     }
1214     }
1215     }
1216     \prg_generate_conditional_variant:Nnn
1217     \__bnvs_raw_last:nN { VN } { T, F, TF }

```

---

```

\__bnvs_last:nN
\__bnvs_last:VN

```

---

`\__bnvs_last:nN {<name>} <tl variable>`

Append the last index of the fully qualified <name> slide range to <tl variable>

```

1218 \cs_new:Npn \__bnvs_last:nN #1 #2 {
1219   \__bnvs_raw_last:nNF { #1 } #2 {
1220     \msg_error:nnn { beanoes } { :n } { Range-with-no-last:~#1 }
1221   }
1222 }
1223 \cs_generate_variant:Nn \__bnvs_last:nN { VN }

```

---

```

\__bnvs_if_next:nNTF

```

---

`\__bnvs_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.

```

1224 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
1225   \__bnvs_if_in:nTF { #1//N } {
1226     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//N } }

```

```

1227 \prg_return_true:
1228 } {
1229   \__bnvs_group_begin:
1230   \cs_set:Npn \__bnvs_return_true: {
1231     \tl_if_empty:NTF \l__bnvs_ans_tl {
1232       \__bnvs_group_end:
1233       \prg_return_false:
1234     } {
1235       \__bnvs_fp_round:N \l__bnvs_ans_tl
1236       \__bnvs_gput:nV { #1//N } \l__bnvs_ans_tl
1237       \exp_args:NNNV
1238       \__bnvs_group_end:
1239       \tl_put_right:Nn #2 \l__bnvs_ans_tl
1240       \prg_return_true:
1241     }
1242   }
1243   \cs_set:Npn \return_false: {
1244     \__bnvs_group_end:
1245     \prg_return_false:
1246   }
1247   \tl_clear:N \l__bnvs_a_tl
1248   \__bnvs_raw_last:nNTF { #1 } \l__bnvs_a_tl {
1249     \__bnvs_if_append:xNTF {
1250       \l__bnvs_a_tl + 1
1251     } \l__bnvs_ans_tl {
1252       \__bnvs_return_true:
1253     } {
1254       \return_false:
1255     }
1256   } {
1257     \return_false:
1258   }
1259 }
1260 }
1261 \prg_generate_conditional_variant:Nnn
1262 \__bnvs_if_next:nN { VN } { T, F, TF }

```

---

```

\__bnvs_next:nN
\__bnvs_next:VN

```

---

`\__bnvs_next:nN {<name>} <tl variable>`

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

```

1263 \cs_new:Npn \__bnvs_next:nN #1 #2 {
1264   \__bnvs_if_next:nNF { #1 } #2 {
1265     \msg_error:nnn { beanoves } { :n } { Range-with-no~next:~#1 }
1266   }
1267 }
1268 \cs_generate_variant:Nn \__bnvs_next:nN { VN }

```

---

```

__bnvs_if_index:nnNTF
__bnvs_if_index:VVNTF
__bnvs_if_index:nnnNTF

```

---

```

__bnvs_if_index:nnNTF {<name>} {<integer>} <tl variable> {<true code>} {<false
code>}

```

Append the index associated to the {<name>} and {<integer>} slide range to the right of <tl variable>. When <integer shift> is 1, this is the first index, when <integer shift> is 2, this is the second index, and so on. When <integer shift> is 0, this is the index, before the first one, and so on. 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.

```

1269 \prg_new_conditional:Npnn __bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
1270   <*debug&!gubed>
1271   __bnvs_DEBUG:x { IF_INDEX:key=#1/index=#2/\string#3/ }
1272   </debug&!gubed>
1273   __bnvs_group_begin:
1274   \tl_clear:N \l__bnvs_ans_tl
1275   __bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1276     \tl_put_right:Nn \l__bnvs_ans_tl { + (#2) - 1 }
1277     \exp_args:NNV
1278     __bnvs_group_end:
1279     __bnvs_fp_round:nN \l__bnvs_ans_tl #3
1280     <*debug&!gubed>
1281     __bnvs_DEBUG:x { IF_INDEX_TRUE:key=#1/index=#2/
1282       \string#3=\tl_to_str:N #3
1283     }
1284     </debug&!gubed>
1285     \prg_return_true:
1286   } {
1287     <*debug&!gubed>
1288     __bnvs_DEBUG:x { IF_INDEX_FALSE:key=#1/index=#2/ }
1289     </debug&!gubed>
1290     \prg_return_false:
1291   }
1292 }
1293 \prg_generate_conditional_variant:Nnn
1294   __bnvs_if_index:nnN { VVN } { T, F, TF }

```

---

```

__bnvs_if_range:nNTF

```

---

```

__bnvs_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.

```

1295 \prg_new_conditional:Npnn __bnvs_if_range:nN #1 #2 { T, F, TF } {
1296   <*debug&!gubed>
1297   __bnvs_DEBUG:x{ RANGE:key=#1/\string#2/}
1298   </debug&!gubed>
1299   \bool_if:NNTF \l__bnvs_no_range_bool {
1300     \prg_return_false:
1301   } {
1302     __bnvs_if_in:nTF { #1/ } {
1303       \tl_put_right:Nn { 0-0 }
1304     } {
1305       __bnvs_group_begin:
1306       \tl_clear:N \l__bnvs_a_tl
1307       \tl_clear:N \l__bnvs_b_tl

```



```

1308     \tl_clear:N \l__bnvs_ans_tl
1309     \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
1310         \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
1311             \exp_args:NNNx
1312             \__bnvs_group_end:
1313             \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
1314         } {
1315             \__bnvs_DEBUG:x{ RANGE_TRUE_A_Z:key=#1/\string#2=#2/}
1316         } {
1317             \prg_return_true:
1318             } {
1319                 \exp_args:NNNx
1320                 \__bnvs_group_end:
1321                 \tl_put_right:Nn #2 { \l__bnvs_a_tl - }
1322             } {
1323                 \__bnvs_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
1324             } {
1325                 \prg_return_true:
1326                 }
1327             } {
1328                 \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
1329                     \__bnvs_DEBUG:x{ RANGE_TRUE_Z:key=#1/\string#2=#2/}
1330                 } {
1331                     \exp_args:NNNx
1332                     \__bnvs_group_end:
1333                     \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1334                 } {
1335                     \prg_return_true:
1336                     } {
1337                         \__bnvs_DEBUG:x{ RANGE_FALSE:key=#1/}
1338                     } {
1339                         \__bnvs_group_end:
1340                         \prg_return_false:
1341                     }
1342                 }
1343             }
1344         }
1345     }
1346 }
1347 \prg_generate_conditional_variant:Nnn
1348   \__bnvs_if_range:nN { VN } { T, F, TF }

```

---

`\__bnvs_range:nN`  
`\__bnvs_range:VN`

---

`\__bnvs_range:nN {<name>} <tl variable>`

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

```

1349 \cs_new:Npn \__bnvs_range:nN #1 #2 {
1350   \__bnvs_if_range:nNF { #1 } #2 {
1351     \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1352   }
1353 }
1354 \cs_generate_variant:Nn \__bnvs_range:nN { VN }

```

---

```

1355 \__bnvs_if_free_counter:nNTF \__bnvs_if_free_counter:nNTF {<name>} <tl variable> {<true code>} {<false
1356 \__bnvs_if_free_counter:VNTF code>}

```

---

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

```

1355 \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
1356   <*debug&!gubed>
1357   \__bnvs_DEBUG:x { IF_FREE: key=\tl_to_str:n{#1}/
1358     value=\__bnvs_item:n {#1/C}/cs=\string #2/
1359 }
1360 </debug&!gubed>
1361   \__bnvs_group_begin:
1362   \tl_clear:N \l__bnvs_ans_tl
1363   \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
1364     \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
1365       \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
1366     }
1367   }
1368   <*debug&!gubed>
1369   \__bnvs_DEBUG:x { IF_FREE_2:\string \l__bnvs_ans_tl=\tl_to_str:V \l__bnvs_ans_tl/}
1370 </debug&!gubed>
1371   \tl_if_empty:NTF \l__bnvs_ans_tl {
1372     \__bnvs_group_end:
1373     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1374       \__bnvs_gput:nn { #1/C } { 1 }
1375       \tl_set:Nn #2 { 1 }
1376     <*debug&!gubed>
1377     \__bnvs_DEBUG:x { IF_FREE_MATCH_TRUE:
1378       key=\tl_to_str:n{#1}\string #2=\tl_to_str:V #2 /
1379     }
1380 </debug&!gubed>
1381     \prg_return_true:
1382     } {
1383   <*debug&!gubed>
1384   \__bnvs_DEBUG:x { IF_FREE_NO_MATCH_FALSE:
1385     key=\tl_to_str:n{#1}\string #2=\tl_to_str:V #2/
1386   }
1387 </debug&!gubed>
1388   \prg_return_false:
1389   }
1390 } {
1391   \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl
1392   \exp_args:NNNV
1393   \__bnvs_group_end:
1394   \tl_set:Nn #2 \l__bnvs_ans_tl
1395 <*debug&!gubed>
1396   \__bnvs_DEBUG:x { IF_FREE_TRUE(2): /
1397     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2
1398   }
1399 </debug&!gubed>
1400   \prg_return_true:
1401   }
1402 }
1403 \prg_generate_conditional_variant:Nnn
1404   \__bnvs_if_free_counter:nN { VN } { T, F, TF }

```

---

```

__bnvs_if_counter:nNTF
__bnvs_if_counter:VNTF

```

---

```

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

```

1405 \prg_new_conditional:Npnn __bnvs_if_counter:nN #1 #2 { T, F, TF } {
1406   <*debug&!gubed>
1407   __bnvs_DEBUG:x { IF_COUNTER:key=
1408     \tl_to_str:n{#1}/\string #2=\tl_to_str:V #2
1409   }
1410   </debug&!gubed>
1411   __bnvs_group_begin:
1412   __bnvs_if_free_counter:nNTF { #1 } \l__bnvs_ans_tl {

```

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

```

1413     \tl_clear:N \l__bnvs_a_tl
1414     __bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
1415       \fp_compare:nNnT { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
1416         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1417       }
1418     }

```

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

```

1419     \tl_clear:N \l__bnvs_a_tl
1420     __bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
1421       \fp_compare:nNnT { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
1422         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1423       }
1424     }
1425     \exp_args:NNV
1426     __bnvs_group_end:
1427     __bnvs_fp_round:nN \l__bnvs_ans_tl #2
1428   <*debug&!gubed>
1429   __bnvs_DEBUG:x {IF_COUNTER_TRUE:key=\tl_to_str:n{#1}/
1430     \string #2=\tl_to_str:V #2
1431   }
1432   </debug&!gubed>
1433   \prg_return_true:
1434   } {
1435   <*debug&!gubed>
1436   __bnvs_DEBUG:x {IF_COUNTER_FALSE:key=\tl_to_str:n{#1}/
1437     \string #2=\tl_to_str:V #2
1438   }
1439   </debug&!gubed>
1440   \prg_return_false:
1441   }
1442 }
1443 \prg_generate_conditional_variant:Nnn
1444   __bnvs_if_counter:nN { VN } { T, F, TF }

```

---

```

__bnvs_if_incr:nnTF
__bnvs_if_incr:nnNTF
__bnvs_if_incr:(VnN|VVN)TF

```

---

```

__bnvs_if_incr:nnTF {<name>} {<offset>} {<true code>} {<false code>}
__bnvs_if_incr:nnNTF {<name>} {<offset>} <tl variable> {<true code>} {<false
code>}

```

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

```

1445 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {
1446   \*debug&!gubed
1447   \__bnvs_DEBUG:x { IF_INCR:\tl_to_str:n{#1}/\tl_to_str:n{#2} }
1448   \*debug&!gubed
1449   \__bnvs_group_begin:
1450   \tl_clear:N \l__bnvs_a_tl
1451   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_a_tl {
1452     \tl_clear:N \l__bnvs_b_tl
1453     \__bnvs_if_append:xNTF { \l__bnvs_a_tl + (#2) } \l__bnvs_b_tl {
1454       \__bnvs_fp_round:N \l__bnvs_b_tl
1455       \__bnvs_gput:nV { #1/C } \l__bnvs_b_tl
1456       \__bnvs_group_end:
1457       \*debug&!gubed
1458       \__bnvs_DEBUG:x { IF_INCR_TRUE:#1/#2 }
1459       \*debug&!gubed
1460       \prg_return_true:
1461       } {
1462         \__bnvs_group_end:
1463         \*debug&!gubed
1464         \__bnvs_DEBUG:x { IF_INCR_FALSE(1):#1/#2 }
1465         \*debug&!gubed
1466         \prg_return_false:
1467         }
1468       } {
1469         \__bnvs_group_end:
1470         \*debug&!gubed
1471         \__bnvs_DEBUG:x { IF_INCR_FALSE(2):#1/#2 }
1472         \*debug&!gubed
1473         \prg_return_false:
1474         }
1475       }
1476   \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1477     \__bnvs_if_incr:nnTF { #1 } { #2 } {
1478       \__bnvs_if_counter:nNTF { #1 } #3 {
1479         \prg_return_true:
1480         } {
1481           \prg_return_false:
1482           }
1483       } {
1484         \prg_return_false:
1485         }
1486     }
1487   \prg_generate_conditional_variant:Nnn
1488   \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }

```

### 5.5.8 Evaluation

|   |   |
|---|---|
| <u>\__bnvs_if_append:nNTF</u><br><u>\__bnvs_if_append:(VN xN)TF</u> | <p>\__bnvs_if_append:nNTF {&lt;integer expression&gt;} &lt;tl variable&gt; {&lt;true code&gt;} {&lt;false code&gt;}</p> <p>Evaluates the &lt;integer expression&gt;, replacing all the named specifications by their static counterpart then put the result to the right of the &lt;tl variable&gt;. Executed within a group. Heavily used by \__bnvs_eval_query:nN, where &lt;integer expression&gt; was initially enclosed in ‘?(...)’. Local variables:</p>  |
| \l__bnvs_ans_tl   | <p>To feed &lt;tl variable&gt; with.</p> <p>(End definition for \l__bnvs_ans_tl.)</p>   |
| \l__bnvs_split_seq  | <p>The sequence of caught query groups and non queries.</p> <p>(End definition for \l__bnvs_split_seq.)</p>   |
| \l__bnvs_split_int  | <p>Is the index of the non queries, before all the caught groups.</p> <p>(End definition for \l__bnvs_split_int.)</p> <p>1489 \int_new:N \l__bnvs_split_int</p>   |
| \l__bnvs_name_tl  | <p>Storage for \l_split_seq items that represent names.</p> <p>(End definition for \l__bnvs_name_tl.)</p>   |
| \l__bnvs_path_tl  | <p>Storage for \l_split_seq items that represent integer paths.</p> <p>(End definition for \l__bnvs_path_tl.)</p> <p>Catch circular definitions.</p> <p>1490 \prg_new_conditional:Npnn \__bnvs_if_append:nN #1 #2 { T, F, TF } {</p> <p>1491 &lt;*debug&amp;!gubed&gt;</p> <p>1492 \__bnvs_DEBUG:x { \string\__bnvs_if_append:nNTF/</p> <p>1493 \tl_to_str:n { #1 } / \string #2/</p> <p>1494 }</p> <p>1495 &lt;/debug&amp;!gubed&gt;</p> <p>1496 \__bnvs_call:TF {</p> <p>1497 &lt;*debug&amp;!gubed&gt;</p> <p>1498 \__bnvs_DEBUG:x { IF_APPEND...}</p> <p>1499 &lt;/debug&amp;!gubed&gt;</p> <p>1500 \__bnvs_group_begin:</p> <p>Local variables:</p> <p>1501 \int_zero:N \l__bnvs_split_int</p> <p>1502 \seq_clear:N \l__bnvs_split_seq</p> <p>1503 \tl_clear:N \l__bnvs_id_tl</p> <p>1504 \tl_clear:N \l__bnvs_name_tl</p> <p>1505 \tl_clear:N \l__bnvs_path_tl</p> <p>1506 \tl_clear:N \l__bnvs_group_tl</p> <p>1507 \tl_clear:N \l__bnvs_ans_tl</p> <p>1508 \tl_clear:N \l__bnvs_a_tl</p> <p>Implementation:</p> <p>1509 \regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq</p> |

```

1510 <*debug&!gubed>
1511 \_bnvs_DEBUG:x { IF_APPEND_SPLIT_SEQ: /
1512 \#=\seq_count:N \l__bnvs_split_seq /
1513 \seq_use:Nn \l__bnvs_split_seq / /
1514 }
1515 </debug&!gubed>
1516 \int_set:Nn \l__bnvs_split_int { 1 }
1517 \tl_set:Nx \l__bnvs_ans_tl {
1518 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1519 }
1520 <*debug&!gubed>
1521 \_bnvs_DEBUG:x { ANS: \l__bnvs_ans_tl }
1522 </debug&!gubed>

```

---

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

---

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

```

1523 \cs_set:Npn \switch:nTF ##1 ##2 ##3 ##4 {
1524 \tl_set:Nx ##2 {
1525 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1526 }
1527 <*debug&!gubed>
1528 \_bnvs_DEBUG:x { IF_APPEND_SWITCH/##1/
1529 \int_eval:n { \l__bnvs_split_int + ##1 } /
1530 \string##2=\tl_to_str:N##2/
1531 }
1532 </debug&!gubed>
1533 \tl_if_empty:NTF ##2 {
1534 <*debug&!gubed>
1535 \_bnvs_DEBUG:x { IF_APPEND_SWITCH_WHITE/##1/
1536 \int_eval:n { \l__bnvs_split_int + ##1 }
1537 }
1538 </debug&!gubed>
1539 ##4 } {
1540 <*debug&!gubed>
1541 \_bnvs_DEBUG:x { IF_APPEND_SWITCH_BLACK/##1/
1542 \int_eval:n { \l__bnvs_split_int + ##1 }
1543 }
1544 </debug&!gubed>
1545 ##3
1546 }
1547 }

```

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

```

1548 \cs_set:Npn \return_true: {
1549 \fp_round:
1550 \exp_args:NNNV
1551 \_bnvs_group_end:
1552 \tl_put_right:Nn #2 \l__bnvs_ans_tl
1553 <*debug&!gubed>
1554 \_bnvs_DEBUG:x { IF_APPEND_TRUE:\tl_to_str:n { #1 } /
1555 \string #2=\tl_to_str:V #2 /

```

```

1556 }
1557 \log_g_prop:
1558 </debug&!gubed>
1559     \prg_return_true:
1560     }
1561     \cs_set:Npn \fp_round: {
1562         \__bnvs_fp_round:N \l__bnvs_ans_tl
1563     }
1564     \cs_set:Npn \return_false: {
1565         \__bnvs_group_end:
1566         <*debug&!gubed>
1567         \__bnvs_DEBUG:x { IF_APPEND_FALSE:\tl_to_str:n { #1 } /
1568             \string #2=\tl_to_str:V #2 /
1569         }
1570         </debug&!gubed>
1571         \prg_return_false:
1572     }
1573     \cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {
1574         \switch:nNTF { ##2 } \l__bnvs_id_tl { } {
1575             \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_current_tl
1576             \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1577         }
1578         \switch:nNTF { ##3 } \l__bnvs_path_tl {
1579             \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
1580             \seq_remove_all:Nn \l__bnvs_path_seq { }
1581         <*debug&!gubed>
1582         \__bnvs_DEBUG:x { PATH_SEQ:\l__bnvs_path_tl==\seq_use:Nn\l__bnvs_path_seq .}
1583         </debug&!gubed>
1584         } {
1585             \seq_clear:N \l__bnvs_path_seq
1586         }
1587         <*debug&!gubed>
1588         \__bnvs_DEBUG:x { PATH_SEQ:\l__bnvs_path_tl==\seq_use:Nn\l__bnvs_path_seq .}
1589         \__bnvs_DEBUG:x { \string ##1 }
1590         </debug&!gubed>
1591         ##1 \l__bnvs_id_tl \l__bnvs_name_tl \l__bnvs_path_seq {
1592             \cs_set:Npn \: {
1593                 ##4
1594             }
1595         } {
1596             \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
1597         }
1598         \:
1599     }
1600     \cs_set:Npn \:T ##1 {
1601         \seq_if_empty:NTF \l__bnvs_path_seq { ##1 } {
1602             \cs_set_eq:NN \loop: \return_false:
1603         }
1604     }

```

Main loop.

```

1605     \cs_set:Npn \loop: {
1606         <*debug&!gubed>
1607         \__bnvs_DEBUG:x { IF_APPEND_LOOP:\int_use:N\l__bnvs_split_int /
1608             \seq_count:N \l__bnvs_split_seq /

```

```

1609 }
1610 </debug&!gubed>
1611 \int_compare:nNnTF {
1612     \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1613 } {
1614     \switch:nNTF 1 \l__bnvs_name_tl {

• Case ++<name><integer path>.n.

1615     \:NnnT \__bnvs_resolve_n:NNNTF 2 3 {
1616         \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1617             \cs_set_eq:NN \loop: \return_false:
1618         }
1619     }
1620 } {
1621     \switch:nNTF 4 \l__bnvs_name_tl {

• Cases <name><integer path>....

1622     \switch:nNTF 7 \l__bnvs_a_tl {
1623         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1624             \:T {
1625                 \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1626                     \cs_set_eq:NN \loop: \return_false:
1627                 }
1628             }
1629         }

• Case ...length.

1630     } {
1631         \switch:nNTF 8 \l__bnvs_a_tl {

• Case ...last.

1632         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1633             \:T {
1634                 \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1635                     \cs_set_eq:NN \loop: \return_false:
1636                 }
1637             }
1638         }

1639     } {
1640         \switch:nNTF 9 \l__bnvs_a_tl {

• Case ...next.

1641         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1642             \:T {
1643                 \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1644                     \cs_set_eq:NN \loop: \return_false:
1645                 }
1646             }
1647         }
1648     } {
1649         \switch:nNTF { 10 } \l__bnvs_a_tl {

```



- Case ...range.

```

1650         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1651         \:T {
1652             \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
1653                 \cs_set_eq:NN \fp_round: \prg_do_nothing:
1654             } {
1655                 \cs_set_eq:NN \loop: \return_false:
1656             }
1657         }
1658     }
1659 } {
1660     \switch:nNTF { 11 } \l__bnvs_a_tl {

```

- Case ...n.

```

1661         \switch:nNTF { 12 } \l__bnvs_a_tl {

```

- Case ...+= $\langle integer \rangle$ .

```

1662         \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1663         \:T {
1664             <*debug&!gubed>
1665             \__bnvs_DEBUG:x {NAME=\l__bnvs_name_tl}
1666             \__bnvs_DEBUG:x {INCR=\l__bnvs_a_tl}
1667             </debug&!gubed>
1668             \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1669                 \cs_set_eq:NN \loop: \return_false:
1670             }
1671         }
1672     }
1673 } {
1674     \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1675         \seq_if_empty:NTF \l__bnvs_path_seq {
1676             \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1677                 \cs_set_eq:NN \loop: \return_false:
1678             }
1679         } {
1680             \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1681             \seq_if_empty:NTF \l__bnvs_path_seq {
1682                 \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1683                     \cs_set_eq:NN \loop: \return_false:
1684                 }
1685             } {
1686                 \msg_error:nnx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
1687                 \cs_set_eq:NN \loop: \return_false:
1688             }
1689         }
1690     }
1691 }

1692 } {
1693     \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1694         \seq_if_empty:NTF \l__bnvs_path_seq {
1695             \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {

```

```

1696 \cs_set_eq:NN \loop: \return_false:
1697 }
1698
1699         } {
1700         \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1701         \seq_if_empty:NTF \l__bnvs_path_seq {
1702         \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1703         \cs_set_eq:NN \loop: \return_false:
1704         }
1705         } {
1706         \msg_error:nxx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
1707         \cs_set_eq:NN \loop: \return_false:
1708         }
1709     }
1710 }
1711 }
1712 }
1713 }
1714 }
1715 } {

```

No name.

```

1716     }
1717 }
1718 <*debug&!gubed>
1719 \__bnvs_DEBUG:x {ITERATE~ANS=\l__bnvs_ans_tl }
1720 </debug&!gubed>
1721 \int_add:Nn \l__bnvs_split_int { 13 }
1722 \tl_put_right:Nx \l__bnvs_ans_tl {
1723 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1724 }
1725 <*debug&!gubed>
1726 \__bnvs_DEBUG:x {ITERATE~ANS=\l__bnvs_ans_tl }
1727 </debug&!gubed>
1728 \loop:
1729 } {
1730 <*debug&!gubed>
1731 \__bnvs_DEBUG:x {END_OF_LOOP~ANS=\l__bnvs_ans_tl }
1732 </debug&!gubed>
1733 \return_true:
1734 }
1735 }
1736 \loop:
1737 } {
1738 \msg_error:nxx { beanoves } { :n } { Too~many~calls:~ #1 }
1739 \prg_return_false:
1740 }
1741 }
1742 \prg_generate_conditional_variant:Nnn
1743 \__bnvs_if_append:nN { VN, xN } { T, F, TF }

```

---

`\__bnvs_if_eval_query:nNTF`

`\__bnvs_if_eval_query:nNTF {<overlay query>} <tl variable> {<true code>} {<false 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__bnvs_a_tl` Storage for the first index of a range.

(End definition for `\l__bnvs_a_tl`.)

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

(End definition for `\l__bnvs_b_tl`.)

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

(End definition for `\c__bnvs_A_cln_Z_regex`.)

```
1744 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1745   \A \s* (?:
```

- 2: *<first>*

```
1746     ( [^:]* ) \s* :
```

- 3: second optional colon

```
1747     ( : )? \s*
```

- 4: *<length>*

```
1748     ( [^:]* )
```

- 5: standalone *<first>*

```
1749     | ( [^:]+ )
```

```
1750   ) \s* \Z
```

```
1751 }
```

```
1752 \prg_new_conditional:Npnn \__bnvs_if_eval_query:nN #1 #2 { T, F, TF } {
```

```
1753   <*debug&!gubed>
```

```
1754   \__bnvs_DEBUG:x { EVAL_QUERY:#1/
```

```
1755     \tl_to_str:n{#1}/\string#2=\tl_to_str:N #2
```

```
1756   }
```

```
1757   </debug&!gubed>
```

```
1758   \__bnvs_call_reset:
```

```
1759   \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
```

```
1760     #1
```

```
1761   } \l__bnvs_match_seq {
```

```
1762   <*debug&!gubed>
```

```
1763   \__bnvs_DEBUG:x { EVAL_QUERY:#1/
```

```
1764     \string\l__bnvs_match_seq/\seq_use:Nn \l__bnvs_match_seq //
```

```
1765   }
```

```
1766   </debug&!gubed>
```

```

1767 \bool_set_false:N \l__bnvs_no_counter_bool
1768 \bool_set_false:N \l__bnvs_no_range_bool

```

---

**\switch:nNTF**  $\langle capture\ group\ number \rangle$   $\langle tl\ variable \rangle$   $\{ \langle black\ code \rangle \}$   $\{ \langle white\ code \rangle \}$

---

Helper function to locally set the  $\langle tl\ variable \rangle$  to the captured group  $\langle capture\ group\ number \rangle$  and branch depending on the emptiness of this variable.

```

1769 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1770 <*debug&!gubed>
1771 \__bnvs_DEBUG:x { EQ_SWITCH:##1/ }
1772 </debug&!gubed>
1773 \tl_set:Nx ##2 {
1774 \seq_item:Nn \l__bnvs_match_seq { ##1 }
1775 }
1776 <*debug&!gubed>
1777 \__bnvs_DEBUG:x { \string ##2/ \tl_to_str:N ##2/}
1778 </debug&!gubed>
1779 \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1780 }
1781 \switch:nNTF 5 \l__bnvs_a_tl {

```

☛ Single expression

```

1782 \bool_set_false:N \l__bnvs_no_range_bool
1783 \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1784 \prg_return_true:
1785 } {
1786 \prg_return_false:
1787 }
1788 } {
1789 \switch:nNTF 2 \l__bnvs_a_tl {
1790 \switch:nNTF 4 \l__bnvs_b_tl {
1791 \switch:nNTF 3 \l__bnvs_c_tl {

```

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

```

1792 \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1793 \tl_put_right:Nn #2 { - }
1794 \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1795 \prg_return_true:
1796 } {
1797 \prg_return_false:
1798 }
1799 } {
1800 \prg_return_false:
1801 }
1802 } {

```

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

```

1803 \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1804 \tl_put_right:Nx #2 { - }
1805 \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1}
1806 \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1807 \prg_return_true:
1808 } {
1809 \prg_return_false:

```

```

1810         }
1811     } {
1812         \prg_return_false:
1813     }
1814 }
1815 } {
1816     <first>: and <first>:: range
1817     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1818         \tl_put_right:Nn #2 { - }
1819         \prg_return_true:
1820     } {
1821         \prg_return_false:
1822     }
1823 } {
1824     \switch:nNTF 4 \l__bnvs_b_tl {
1825         \switch:nNTF 3 \l__bnvs_c_tl {
1826             \tl_put_right:Nn #2 { - }
1827             \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1828                 \prg_return_true:
1829             } {
1830                 \prg_return_false:
1831             }
1832         } {
1833             \msg_error:nmx { beanoves } { :n } { Syntax-error(Missing-first):~#1 }
1834         }
1835     } {
1836         : or :: range
1837         \seq_put_right:Nn #2 { - }
1838     }
1839 }
1840 } {
Error
1841     \msg_error:nnn { beanoves } { :n } { Syntax-error:~#1 }
1842 }
1843 }

```

---

```

__bnvs_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 `__bnvs_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__bnvs_query_seq  Storage for a sequence of <query>'s obtained by splitting a comma separated list.

```

(End definition for `\l__bnvs_query_seq`.)

```

\l__bnvs_ans_seq  Storage of the evaluated result.

```

(End definition for `\l__bnvs_ans_seq`.)

```

\c__bnvs_comma_regex  Used to parse slide range overlay specifications.

```

```

1844 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

1845 \cs_new:Npn \__bnvs_eval:nN #1 #2 {
1846   <*debug&!gubed>
1847   \__bnvs_DEBUG:x {<string>\__bnvs_eval:nN:<tl_to_str:n{#1}/
1848     <string#2=<tl_to_str:V #2
1849   }
1850   </debug&!gubed>
1851   \__bnvs_group_begin:

```

Local variables declaration

```

1852   \seq_clear:N \l__bnvs_query_seq
1853   \seq_clear:N \l__bnvs_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`.

```

1854   \regex_split:NnN \c__bnvs_comma_regex { #1 } \l__bnvs_query_seq

```

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

```

1855   \seq_map_inline:Nn \l__bnvs_query_seq {
1856     \tl_clear:N \l__bnvs_ans_tl
1857     \__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {
1858       \seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl
1859     } {
1860       \seq_map_break:n {
1861         \msg_fatal:nnn { beanoves } { :n } { Circular~dependency~in~#1}
1862       }
1863     }
1864   }

```

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

```

1865   \exp_args:NNNx
1866   \__bnvs_group_end:

```

```

1867 \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
1868 }
1869 \cs_generate_variant:Nn \__bnvs_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 `\__bnvs_eval:nN` within a group. `\l_ans_tl` is used locally to store the result.

```

1870 \NewDocumentCommand \BeanovesEval { s o m } {
1871   \__bnvs_group_begin:
1872   \tl_clear:N \l__bnvs_ans_tl
1873   \IfBooleanTF { #1 } {
1874     \bool_set_true:N \l__bnvs_no_counter_bool
1875   } {
1876     \bool_set_false:N \l__bnvs_no_counter_bool
1877   }
1878   \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
1879   \IfValueTF { #2 } {
1880     \exp_args:NNNV
1881     \__bnvs_group_end:
1882     \tl_set:Nn #2 \l__bnvs_ans_tl
1883   } {
1884     \exp_args:NV
1885     \__bnvs_group_end: \l__bnvs_ans_tl
1886   }
1887 }

```

### 5.5.9 Reseting slide ranges

---

**\BeanovesReset**

---

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

```

1888 \NewDocumentCommand \BeanovesReset { O{1} m } {
1889   \__bnvs_reset:nn { #1 } { #2 }
1890   \ignorespaces
1891 }

```

Forwards to `\__bnvs_reset:nn`.

---

**\\_\_bnvs\_reset:nn**

---

**\\_\_bnvs\_reset:nn** {*<first value>*} {*<slide range name>*}

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

```

1892 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1893   \bool_if:nTF {
1894     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1895   } {
1896     \__bnvs_gremove:n { #2/C }
1897     \__bnvs_gremove:n { #2//A }
1898     \__bnvs_gremove:n { #2//L }
1899     \__bnvs_gremove:n { #2//Z }

```

```

1900     \__bnvs_gremove:n { #2//N }
1901     \__bnvs_gput:nn { #2/C0 } { #1 }
1902   } {
1903     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1904   }
1905 }

1906 \makeatother
1907 \ExplSyntaxOff
1908 </package>

```