

beamer named overlay specification with beanoves

Jérôme Laurens

v1.0 2022/10/28

Abstract

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

Contents

1 Minimal example

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

```
1 \documentclass {beamer}
2 \RequirePackage {beanoves-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 \insertframenum}
11 {\Large Slide \insertslidenum}
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* $\langle A.1 \rangle$ stands for 1, on line 15, $\langle A.2 \rangle$ stands for 2 whereas on line 18, $\langle A.3 \rangle$ 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 $\langle B.last \rangle$ is replaced by 5. The next slide number after slide range ‘B’ is 6 which is also the start of the third slide range due to line 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 = {
  \name_1=\spec_1,
  \name_2=\spec_2,
  ...,
  \name_n=\spec_n,
}
```

```
\Beanoves{
  \name_1=\spec_1,
  \name_2=\spec_2,
  ...,
  \name_n=\spec_n,
}
```

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 incremented¹.

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

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

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

Notice that “.n” can generally be omitted.

3.2 Named slide lists

After the definition

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

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

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

4 ?(...) query expressions

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

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

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

¹This is actually an experimental feature.

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 $?(\mathbf{A.next})$, $?(\mathbf{A.last+1})$, $?(\mathbf{A.1+A.length})$ 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.

```
1 \*package
```

5 Implementation

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

```
2 @@=bnvs
```

5.1 Package declarations

```
3 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
4 \ProvidesExplPackage
5 \*!debug
6 {beanoves}
7 \!/debug
8 \*!gubed
9 {beanoves-debug}
10 \!/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 \*!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_a: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_a:nn
44 { \prg_replicate:nn {\l__bnvs_group_int + 1} {#1} }
45 }
46 \cs_generate_variant:Nn \__bnvs_DEBUG:nn { nx, nV }
47 </!gubed>
48 <!*!debug>
49 \cs_set_eq:NN \__bnvs_group_begin: \group_begin:
50 </!debug>

```

5.3 Local variables

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

```

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.

`\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 \g__bnvs_call_int.)
```

`__bnvs_call_reset:`

`__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` { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

Decrement the `\g__bnvs_call_int` counter globally and execute $\langle true\ code \rangle$ if we have not reached 0, $\langle false\ code \rangle$ 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` $\langle key \rangle$ – $\langle value \rangle$ property list to store the named slide lists. The basic keys are, assuming $\langle id \rangle!$ $\langle name \rangle$ is a fully qualified slide list name,

$\langle id \rangle!$ $\langle name \rangle$ /A for the first index

$\langle id \rangle!$ $\langle name \rangle$ /L for the length when provided

$\langle id \rangle!$ $\langle name \rangle$ /Z for the last index when provided

$\langle id \rangle!$ $\langle name \rangle$ /C for the counter value, when used

$\langle id \rangle!$ $\langle name \rangle$ /C0 for initial value of the counter (when reset)

Other keys are eventually used to cache results when some attributes are defined from other slide ranges. They are characterized by a ‘//’.

$\langle id \rangle!$ $\langle name \rangle$ //A for the cached static value of the first index

$\langle id \rangle!$ $\langle name \rangle$ //Z for the cached static value of the last index

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

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

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

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_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^AT_EX3 eponym functions, except `__bnvs_gprovide:nn` which meaning is straightforward.

```

92 \cs_new:Npn \__bnvs_gput:nn #1 #2 {
93   \*!gubed
94   \__bnvs_DEBUG:x {\string\__bnvs_gput:nn/key:#1/value:#2/}
95   \*!gubed
96   \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
97 }
98 \cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
99   \*!gubed
100   \__bnvs_DEBUG:x {\string\__bnvs_gprovide:nn/key:#1/value:#2/}
101   \*!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> <code>__bnvs_if_in_p:V *</code> <code>__bnvs_if_in:nTF *</code> <code>__bnvs_if_in:VTF *</code>	<code>__bnvs_if_in_p:n {<key>}</code> <code>__bnvs_if_in:nTF {<key>} {<true code>} {<false code>}</code> 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> <code>__bnvs_get:nnNTF</code>	<code>__bnvs_get:nNTF {<key>} <tl variable> {<true code>} {<false code>}</code> <code>__bnvs_get:nnNTF {<id>} {<key>} <tl variable> {<true code>} {<false code>}</code>
---	--

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

```

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

```

5.5.2 Regular expressions

`\c__bnvs_name_regex` 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`.)

`\c__bnvs_id_regex` 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 $\langle \textit{positive integer} \rangle$ 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 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 $\langle id \rangle$

3: question mark, when $\langle id \rangle$ 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 ‘++’ instrussions².

- 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`.)

²At 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   \*!gubed)
199   \__bnvs_DEBUG_on:
200   \__bnvs_DEBUG:x {THIS_IS_KEY}
201   \__bnvs_DEBUG_off:
202   \!gubed)
203 }
204 \AddToHook{env/beamer@frameslide/before}{
205   \bool_set_true:N \l__bnvs_in_frame_bool
206   \*!gubed)
207   \__bnvs_DEBUG_on:
208   \__bnvs_DEBUG:x {THIS_IS_BEFORE}
209   \__bnvs_DEBUG_off:
210   \!gubed)
211 }
212 \AddToHook{env/beamer@frameslide/after}{
213   \bool_set_false:N \l__bnvs_in_frame_bool
214   \*!gubed)
215   \__bnvs_DEBUG_on:
216   \__bnvs_DEBUG:x {THIS_IS_AFTER}
217   \__bnvs_DEBUG_off:
218   \!gubed)
219 }
220 \AddToHook{cmd/frame/before}{
221   \tl_set:Nn \l__bnvs_id_current_tl { ?! }
222   \*!gubed)
223   \__bnvs_DEBUG_on:
224   \__bnvs_DEBUG:x {THIS_IS_FRAME}
225   \__bnvs_DEBUG_off:
226   \!gubed)
227 }

```

5.5.4 Defining named slide ranges

<u>__bnvs_parse:Nnn</u>	__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 auxiliary 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   \*!gubed)
230   __bnvs_DEBUG:x {string__bnvs_range:Nnnnn/string#1/#2/#3/#4/#5/}
231   \!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 } \\l__bnvs_id_tl \\l__bnvs_name_tl {
272     \exp_args:Nx \\_bnvs_gput:nn { \\l__bnvs_name_tl/ } { }
273     \exp_args:NNNV
274     \\_bnvs_group_end:
275     \tl_set:Nn \\l__bnvs_id_current_tl \\l__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   <!*gubed>
284   \\_bnvs_DEBUG:x {\\string\\_bnvs_do_parse:Nnn/\\string#1/#2/#3}
285   <!/gubed>
```

This is not a list.

```

286   \tl_clear:N \\l__bnvs_a_tl
287   \tl_clear:N \\l__bnvs_b_tl
288   \tl_clear:N \\l__bnvs_c_tl
289   \regex_split:NnN \\c__bnvs_colons_regex { #3 } \\l__bnvs_split_seq
290   \seq_pop_left:NNT \\l__bnvs_split_seq \\l__bnvs_a_tl {
  \\l_a_tl may contain the <start>.
```

```

291     \seq_pop_left:NNT \\l__bnvs_split_seq \\l__bnvs_b_tl {
292       \tl_if_empty:NNTF \\l__bnvs_b_tl {
```

This is a one colon range.

```

293       \seq_pop_left:NN \\l__bnvs_split_seq \\l__bnvs_b_tl
  \\l_b_tl may contain the <length>.
```

```

294       \seq_pop_left:NNT \\l__bnvs_split_seq \\l__bnvs_c_tl {
295       \tl_if_empty:NNTF \\l__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 \\l__bnvs_c_tl } > { 1 } {
299 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(2):~#3 }
300   }
301     \seq_pop_left:NN \\l__bnvs_split_seq \\l__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:nNtT { \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:NTF \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 }

```

__bnvs_id_name_set:nNNTF __bnvs_id_name_set:nNNTF { $\langle key \rangle$ } $\langle id\ tl\ var \rangle$ $\langle full\ name\ tl\ var \rangle$ { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

If the $\langle key \rangle$ is a key, put the name it defines into the $\langle name\ tl\ var \rangle$ with the current frame id prefix \l__bnvs_id_tl if none was given, then execute $\langle true\ code \rangle$. Otherwise execute $\langle false\ code \rangle$.

```

337 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
338     \__bnvs_group_begin:
339     \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {

```



```

340     #1
341 } \l__bnvs_match_seq {
342     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
343     \tl_if_empty:NTF #2 {
344         \exp_args:NNNx
345         \__bnvs_group_end:
346         \tl_set:Nn #3 { \l__bnvs_id_current_tl #1 }
347         \tl_set_eq:NN #2 \l__bnvs_id_current_tl
348     } {
349         \cs_set:Npn \:n ##1 {
350             \__bnvs_group_end:
351             \tl_set:Nn #2 { ##1 }
352             \tl_set:Nn \l__bnvs_id_current_tl { ##1 }
353         }
354         \exp_args:NV
355         \:n #2
356         \tl_set:Nn #3 { #1 }
357     }
358     <!*gubed>
359     \__bnvs_DEBUG:x { \string\__bnvs_id_name_set:nNN\space TRUE/}
360     \__bnvs_DEBUG:x { #1/ \string#2:#2/\string#3:#3/ }
361     \__bnvs_DEBUG:x { \string\l__bnvs_id_current_tl:\l__bnvs_id_current_tl/ }
362     </!gubed>
363     \prg_return_true:
364 } {
365     \__bnvs_group_end:
366     <!*gubed>
367     \__bnvs_DEBUG:x { \string\__bnvs_id_name_set:nNN\space FALSE
368     /#1/\string#2/\string#3/
369 }
370 </!gubed>
371     \prg_return_false:
372 }
373 }

374 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
375     <!*gubed>
376     \__bnvs_DEBUG:x { \string\__bnvs_parse:Nnn/\string#1/#2/#3/}
377     </!gubed>
378     \__bnvs_group_begin:
379     \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
380     <!*gubed>
381     \__bnvs_DEBUG:x {key:#2/ID:\l__bnvs_id_tl/NAME:\l__bnvs_name_tl/}
382     </!gubed>
383     \regex_extract_once:NnNTF \c__bnvs_list_regex {
384         #3
385     } \l__bnvs_match_seq {

```

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

```

386     \exp_args:NNx
387     \seq_set_from_clist:Nn \l__bnvs_match_seq {
388         \seq_item:Nn \l__bnvs_match_seq { 2 }
389     }
390     \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
391         \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl.##1 } { ##2 }

```

```

392     }
393   } {
394     \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
395   }
396 } {
397   \msg_error:nnn { beanoves } { :n } { Invalid~key:~#2 }
398 }

```

We export \l__bnvs_id_tl:

```

399   \exp_args:NNNV
400   \__bnvs_group_end:
401   \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
402 }

```

\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.

```

403 \NewDocumentCommand \Beanoves { sm } {
404   \tl_if_eq:NnT \@currenvir { document } {
405     \__bnvs_gclear:
406   }
407   \IfBooleanTF {#1} {
408     \keyval_parse:nnn {
409       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
410     } {
411       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
412     }
413   } {
414     \keyval_parse:nnn {
415       \__bnvs_parse:Nn \__bnvs_range:nVVV
416     } {
417       \__bnvs_parse:Nnn \__bnvs_range:nVVV
418     }
419   }
420   { #2 }
421   \ignorespaces
422 }

```

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

```

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

```

5.5.5 Scanning named overlay specifications

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

<code>\beamer@frame</code> <code>\beamer@masterdecode</code>	<code>\beamer@frame {<overlay specification>}</code> <code>\beamer@masterdecode {<overlay specification>}</code>
---	---

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.

```

424 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
425 \cs_set:Npn \beamer@frame < #1 > {
426   \__bnvs_group_begin:
427   \tl_clear:N \l__bnvs_ans_tl
428   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
429   \exp_args:NNNV
430   \__bnvs_group_end:
431   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
432 }
433 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
434 \cs_set:Npn \beamer@masterdecode #1 {
435   \__bnvs_group_begin:
436   \tl_clear:N \l__bnvs_ans_tl
437   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
438   \exp_args:NNV
439   \__bnvs_group_end:
440   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
441 }

```

_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.)

```

442 \cs_new:Npn \_bnvs_scan:nNN #1 #2 #3 {
443   \_bnvs_group_begin:
444   \tl_clear:N \l__bnvs_ans_tl
445   \int_zero:N \l__bnvs_depth_int
446   \seq_clear:N \l__bnvs_token_seq

  Explode the <named overlay expression> into a list of tokens:
447   \regex_split:nnN {} { #1 } \l__bnvs_token_seq

  Run the top level loop to scan for a '?':
448   \bool_set_true:N \l__bnvs_ask_bool
449   \bool_while_do:Nn \l__bnvs_ask_bool {
450     \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
451     \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.
452     \bool_set_false:N \l__bnvs_ask_bool
453   } {

  \l_token_tl contains a 'normal' token.
454   \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.

```
455         \bool_set_true:N \l__bnvs_query_bool
456         \bool_while_do:Nn \l__bnvs_query_bool {
```

Get next token.

```
457         \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
458         \quark_if_no_value:NTF \l__bnvs_token_tl {
```

No opening parenthesis found, raise.

```
459         \msg_fatal:nxx { beanoves } { :n } {Missing~'('%---)
460         ~after~a~?:~#1}
461     } {
462         \tl_if_eq:NnT \l__bnvs_token_tl { ( %)
463     } {
```

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

```
464         \int_incr:N \l__bnvs_depth_int
```

Record the forthcoming content in the \l_query_tl variable, up to the next balancing ')':

```
465         \tl_clear:N \l__bnvs_query_tl
466         \bool_while_do:Nn \l__bnvs_query_bool {
```

Get next token.

```
467         \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
468         \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.

```
469         \msg_error:nxx { beanoves } { :n } {Missing~%((---
470         ~)':~#1 }
471         \int_do_while:nNnn \l__bnvs_depth_int > 1 {
472             \int_decr:N \l__bnvs_depth_int
473             \tl_put_right:Nn \l__bnvs_query_tl {%(---
474         )}
475     }
476     \int_zero:N \l__bnvs_depth_int
477     \bool_set_false:N \l__bnvs_query_bool
478     \bool_set_false:N \l__bnvs_ask_bool
479 } {
480     \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
481 } {
```

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

```
482         \int_incr:N \l__bnvs_depth_int
483         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
484     } {
```

This is not a '('.

```
485         \tl_if_eq:NnTF \l__bnvs_token_tl { %(
486         )
487     } {
```

We found a ')', decrement the depth.

```

488             \int_decr:N \l__bnvs_depth_int
489             \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.

```

490     \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
491     \bool_set_false:N \l__bnvs_query_bool
492     } {

```

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

```

493             \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
494             }

```

Above ends the code for a positive depth.

```

495     } {

```

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

```

496             \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
497             }
498         }
499     }

```

Above ends the code for Not a '('

```

500     }
501 }

```

Above ends the code for: Found the '(' after the '?'

```

502 }

```

Above ends the code for not a no value quark.

```

503 }

```

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.

```

504     \quark_if_no_value:NT \l__bnvs_token_tl {
505         \bool_set_false:N \l__bnvs_query_bool
506         \bool_set_false:N \l__bnvs_ask_bool
507     }
508 } {

```

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

```

509     \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
510 }

```

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

```

511 }
512 }

```

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

```

513 \exp_args:NNNV
514 \__bnvs_group_end:
515 \tl_put_right:Nn #3 \l__bnvs_ans_tl
516 }

```

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 \__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.

```

517 \exp_args_generate:n { VVx }
518 \prg_new_conditional:Npnn \__bnvs_extract_key:NNN
519   #1 #2 #3 { T, F, TF } {
520   <!*gubed>
521   \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN/
522     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn#3./
523   }
524   </!gubed>
525   \__bnvs_group_begin:
526   \exp_args:NNV
527   \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

```

528   \exp_args:Nx
529   \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
530     \tl_put_left:NV #2 { #1 }
531   <!*gubed>
532   \__bnvs_DEBUG:x { VERIF~\tl_to_str:V #2 }
533   </!gubed>
534   }
535   \exp_args:NNnx
536   \seq_set_split:Nnn \l__bnvs_split_seq . {
537     \seq_item:Nn \l__bnvs_match_seq 4
538   }
539   \seq_remove_all:Nn \l__bnvs_split_seq { }
540   \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
541   \seq_if_empty:NNTF \l__bnvs_split_seq {
```

No new integer path component is added.

```

542   \cs_set:Npn \:nn ##1 ##2 {
543     \__bnvs_group_end:
544     \tl_set:Nn #1 { ##1 }
545     \tl_set:Nn #2 { ##2 }
546   }
547   \exp_args:NVV \:nn #1 #2
548   <!*gubed>
549   \__bnvs_DEBUG:x { END/\string#1:#1/\string#2:#2/ }
550   </!gubed>
551   } {
```

Some new integer path components are added.

```

552   <!*gubed>
```

```

553 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN/\string#1:#1/
554 \string#2:#2/\string#3:\seq_use:Nn#3./
555 \string\l__bnvs_split_seq:\seq_use:Nn\l__bnvs_split_seq./
556 }
557 </!gubed>
558 \cs_set:Npn \:nnn ##1 ##2 ##3 {
559 \__bnvs_group_end:
560 \tl_set:Nn #1 { ##1 }
561 \tl_set:Nn #2 { ##2 }
562 \seq_set_split:Nnn #3 . { ##3 }
563 \seq_remove_all:Nn #3 { }
564 }
565 \exp_args:NVVx
566 \:nnn #1 #2 {
567 \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
568 }
569 <!*gubed>
570 \__bnvs_DEBUG:x { END/\string#1:#1/\string#2:#2/
571 \string#3:\seq_use:Nn #3 . /
572 \string\l__bnvs_split_seq:\seq_use:Nn \l__bnvs_split_seq . /
573 }
574 </!gubed>%</!gubed>
575 }
576 <!*gubed>
577 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN\space TRUE/
578 \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3 . /
579 }
580 </!gubed>
581 \prg_return_true:
582 } {
583 \__bnvs_group_end:
584 <!*gubed>
585 \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN\space FALSE/
586 \string#1/\string#2/\string#3/
587 }
588 </!gubed>
589 \prg_return_false:
590 }
591 }

```

```

\__bnvs_resolve:NNNTF \__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 $\backslash l_a_seq$, $\backslash l_b_seq$.

```

592 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
593   #1 #2 #3 { T, F, TF } {
594   <!*gubed>
595   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNN/
596   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
597   }
598   </!gubed>
599   \__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.

```

600 \seq_set_eq:NN \l__bnvs_a_seq #3
601 \seq_clear:N \l__bnvs_b_seq
602 \cs_set:Npn \loop: {
603   \__bnvs_call:TF {
604     \tl_set_eq:NN \l__bnvs_a_tl #2
605     \seq_if_empty:NNTF \l__bnvs_a_seq {
606       \exp_args:Nx
607       \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
608         \cs_set:Nn \loop: { \return_true: }
609       } {
610         \get_extract:F {

```

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

```

611     \cs_set:Nn \loop: { \return_true: }
612   }
613 } {
614   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
615   \get_extract:F {
616     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
617       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
618     }
619   }
620 }
621 }
622 \loop:
623 } {
624 <!*gubed>
625 \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNN\space~TOO~MANY~CALLS/
626   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
627 }
628 </!gubed>
629 \__bnvs_group_end:
630 \prg_return_false:
631 }
632 }
633 \cs_set:Npn \get_extract:F ##1 {
634   \exp_args:Nx
635   \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
636 <!*gubed>
637 \__bnvs_DEBUG:x { RESOLUTION:~\l__bnvs_a_tl / A=>\l__bnvs_b_tl}
638 </!gubed>
639   \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
640     \tl_set_eq:NN #2 \l__bnvs_b_tl
641     \seq_set_eq:NN #3 \l__bnvs_b_seq
642     \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
643     \seq_clear:N \l__bnvs_b_seq
644   } { ##1 }
645 } { ##1 }
646 }
647 \cs_set:Npn \return_true: {
648   \cs_set:Npn \:nnn #####1 #####2 #####3 {
649     \__bnvs_group_end:
650     \tl_set:Nn #1 { #####1 }
651     \tl_set:Nn #2 { #####2 }
652     \seq_set_split:Nnn #3 . { #####3 }
653     \seq_remove_all:Nn #3 { }
654   }
655   \exp_args:NVVx
656   \:nnn #1 #2 {
657     \seq_use:Nn #3 .
658   }
659 <!*gubed>
660 \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNN\space TRUE/
661   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
662 }
663 </!gubed>

```

```

664     \prg_return_true:
665   }
666   \loop:
667 }

```

```

__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

```

668 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
669   #1 #2 #3 { T, F, TF } {
670   <!*gubed>
671   \__bnvs_DEBUG:x { \string\__bnvs_resolve_n:NNN/
672   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
673   }
674   </!gubed>
675   \__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.

```

676   \seq_set_eq:NN \l__bnvs_a_seq #3
677   \seq_clear:N \l__bnvs_b_seq
678   \cs_set:Npn \loop: {
679     \__bnvs_call:TF {
680       \tl_set_eq:NN \l__bnvs_a_tl #2
681       \seq_if_empty:NNTF \l__bnvs_a_seq {
682         \exp_args:Nx
683         \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
684           \cs_set:Nn \loop: { \return_true: }
685         } {
686           \seq_if_empty:NNTF \l__bnvs_b_seq {
687             \cs_set:Nn \loop: { \return_true: }
688           } {
689             \get_extract:F {

```

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

```

690       \cs_set:Nn \loop: { \return_true: }
691     }
692   }
693 } {
694 } {
695   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
696   \get_extract:F {
697     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
698       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
699     }
700   }

```

```

701     }
702     \loop:
703     } {
704     (*!gubed)
705     \__bnvs_DEBUG:x { \string\__bnvs_resolve_n:NNN\space~TOO~MANY~CALLS/
706     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
707     }
708     <\/!gubed>
709     \__bnvs_group_end:
710     \prg_return_false:
711     }
712     }
713     \cs_set:Npn \get_extract:F ##1 {
714     \exp_args:Nx
715     \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
716     (*!gubed)
717     \__bnvs_DEBUG:x { RESOLUTION:~\l__bnvs_a_tl / A=>\l__bnvs_b_tl}
718     <\/!gubed>
719     \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
720     \tl_set_eq:NN #2 \l__bnvs_b_tl
721     \seq_set_eq:NN #3 \l__bnvs_b_seq
722     \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
723     \seq_clear:N \l__bnvs_b_seq
724     } { ##1 }
725     } { ##1 }
726     }
727     \cs_set:Npn \return_true: {
728     \cs_set:Npn \:nnn #####1 #####2 #####3 {
729     \__bnvs_group_end:
730     \tl_set:Nn #1 { #####1 }
731     \tl_set:Nn #2 { #####2 }
732     \seq_set_split:Nnn #3 . { #####3 }
733     \seq_remove_all:Nn #3 { }
734     }
735     \exp_args:NVVx
736     \:nnn #1 #2 {
737     \seq_use:Nn #3 .
738     }
739     (*!gubed)
740     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve_n:NNN\space TRUE/
741     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./
742     }
743     <\/!gubed>
744     \prg_return_true:
745     }
746     \loop:
747     }

```

```

__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, $\{ \langle false \text{ code} \rangle \}$ is executed directly. $\langle id \text{ tl var} \rangle$, $\langle name \text{ tl var} \rangle$ and $\langle path \text{ seq var} \rangle$ are meant to contain proper information. To resolve a path, $\langle name_0 \rangle . \langle i_1 \rangle . \langle i_2 \rangle \dots \langle i_n \rangle$ is turned into $\langle name_1 \rangle . \langle i_2 \rangle \dots \langle i_n \rangle$ where $\langle name_0 \rangle . \langle i_1 \rangle$ is $\langle name_1 \rangle$, then $\langle name_2 \rangle . \langle i_3 \rangle \dots \langle i_n \rangle$ where $\langle name_1 \rangle . \langle i_2 \rangle$ is $\langle name_2 \rangle \dots$. If the above rule does not apply, $\langle name_0 \rangle . \langle i_1 \rangle . \langle i_2 \rangle \dots \langle i_n \rangle$ may turn into $\langle name_2 \rangle . \langle i_3 \rangle \dots \langle i_n \rangle$ when $\langle name_0 \rangle . \langle i_1 \rangle . \langle i_2 \rangle$ is $\langle name_2 \rangle \dots$. 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 $\backslash_bnvs_if_append : \dots$ conditionals.

```

748 \prg_new_conditional:Npnn \__bnvs_resolve:NNNN
749   #1 #2 #3 #4 { T, F, TF } {
750   <!*gubed>
751   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN / }
752   \__bnvs_DEBUG:x { \string#1 / \string#2:#2/\string#3:#3 / }
753   \__bnvs_DEBUG:x { \string#4 : \seq_use:Nn #4 . / }
754   </!gubed>
755   #1 {
756     \__bnvs_group_begin:
757     \tl_set_eq:NN \l__bnvs_a_tl #3
758     \seq_set_eq:NN \l__bnvs_a_seq #4
759     \tl_clear:N \l__bnvs_b_tl
760     \seq_clear:N \l__bnvs_b_seq
761     \cs_set:Npn \return_true: {
762       \cs_set:Npn \:nnn #####1 #####2 #####3 {
763         \__bnvs_group_end:
764         \tl_set:Nn #2 { #####1 }
765         \tl_set:Nn #3 { #####2 }
766         \seq_set_split:Nnn #4 . { #####3 }
767         \seq_remove_all:Nn #4 { }
768       }
769       \exp_args:NVVx
770       \:nnn #2 #3 {
771         \seq_use:Nn #4 .
772       }
773     <!*gubed>
774     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNNN \space TRUE / }
775     \__bnvs_DEBUG:x { \string#1 / \string#2 : #2 / \string#3 : #3 / }
776     \__bnvs_DEBUG:x { \string#4 : \seq_use:Nn #4 . / }
777     </!gubed>
778     \prg_return_true:
779   }
780   \cs_set:Npn \branch:n ##1 {
781     \seq_pop_right:NTF \l__bnvs_a_seq \l__bnvs_b_tl {
782       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_b_tl
783     }
784     <!*gubed>
785     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN \space POP~TRUE~##1 }

```

```

785 \__bnvs_DEBUG:x {\string\l__bnvs_b_tl : \l__bnvs_b_tl }
786 \__bnvs_DEBUG:x {\string\l__bnvs_a_seq : \seq_count:N \l__bnvs_a_seq / }
787 \__bnvs_DEBUG:x {\seq_use:Nn \l__bnvs_a_seq ./ }
788 \__bnvs_DEBUG:x {\string\l__bnvs_b_seq : \seq_count:N \l__bnvs_b_seq / }
789 \__bnvs_DEBUG:x {\seq_use:Nn \l__bnvs_b_seq . / }
790 </!gubed>
791 \tl_set:Nn \l__bnvs_a_tl { #3 . }
792 \tl_put_right:Nx \l__bnvs_a_tl { \seq_use:Nn \l__bnvs_a_seq . }
793 } {
794 \cs_set_eq:NN \loop: \return_true:
795 }
796 }
797 \cs_set:Npn \branch:FF ##1 ##2 {
798 \exp_args:Nx
799 \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
800 \__bnvs_extract_key:NNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
801 \tl_set_eq:NN #3 \l__bnvs_b_tl
802 \seq_set_eq:NN #4 \l__bnvs_b_seq
803 \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
804 } { ##1 }
805 } { ##2 }
806 }
807 \cs_set:Npn \extract_key:F {
808 \__bnvs_extract_key:NNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
809 \tl_set_eq:NN #3 \l__bnvs_b_tl
810 \seq_set_eq:NN #4 \l__bnvs_b_seq
811 \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
812 }
813 }
814 \cs_set:Npn \loop: {
815 \__bnvs_call:TF {
816 \exp_args:Nx
817 \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {

```

If there is a length, no resolution occurs.

```

818 \branch:n { 1 }
819 } {
820 \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
821 \seq_clear:N \l__bnvs_b_seq
822 \tl_set:Nn \l__bnvs_a_tl { #3 . }
823 \tl_put_right:Nx \l__bnvs_a_tl {
824 \seq_use:Nn \l__bnvs_a_seq . .
825 }
826 \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
827 \branch:FF {

```

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

```

828 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
829 } {

```

Unknown key <\l_a_tl>/L.

```

830 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
831 }
832 } {
833 \branch:FF {

```

```

834         \cs_set_eq:NN \loop: \return_true:
835     } {
836         \cs_set:Npn \loop: {
837             \__bnvs_group_end:
838         } (*!gubed)
839         \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN \space FALSE / }
840         \__bnvs_DEBUG:x { \string#1/\string#2 : #2/\string#3 : #3 / }
841         \__bnvs_DEBUG:x { \string#4 : \seq_use:Nn #4 . / }
842         \__bnvs_DEBUG:x { \g__bnvs_call_int : \int_use:N\g__bnvs_call_int / }
843     } (*!gubed)
844     \prg_return_false:
845 }
846 }
847 }
848 }
849 } {
850     \cs_set:Npn \loop: {
851         \__bnvs_group_end:
852     } (*!gubed)
853     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space FALSE / }
854     \__bnvs_DEBUG:x { \string#1 / \string#2 : #2 / \string#3 : #3 / }
855     \__bnvs_DEBUG:x { \string#4 : \seq_use:Nn #4 . / }
856     \__bnvs_DEBUG:x { \g__bnvs_call_int : \int_use:N\g__bnvs_call_int / }
857 } (*!gubed)
858     \prg_return_false:
859 }
860 }
861 \loop:
862 }
863 \loop:
864 } {
865     \prg_return_true:
866 }
867 }
868 \prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
869     #1 #2 #3 #4 { T, F, TF } {
870     } (*!gubed)
871     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN/ }
872     \__bnvs_DEBUG:x { \string#1 / \string#2 : #2 / \string#3 : #3 / }
873     \__bnvs_DEBUG:x { \string#4 : \seq_use:Nn #4 . / }
874 } (*!gubed)
875     #1 {
876         \__bnvs_group_begin:

```

\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.

```

877     \tl_set_eq:NN \l__bnvs_a_tl #3
878     \seq_set_eq:NN \l__bnvs_a_seq #4
879     \cs_set:Npn \return_true: {
880         \cs_set:Npn \:nnn #####1 #####2 #####3 {
881             \__bnvs_group_end:
882             \tl_set:Nn #2 { #####1 }
883             \tl_set:Nn #3 { #####2 }
884             \seq_set_split:Nnn #4 . { #####3 }

```

```

885     \seq_remove_all:Nn #4 { }
886   }
887   \exp_args:NVVx
888   \:nnn #2 #3 {
889     \seq_use:Nn #4 .
890   }
891   <!*gubed>
892   \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNNN\space TRUE/ }
893   \__bnvs_DEBUG:x { \string#1 / \string#2 : #2 / \string#3 : #3/ }
894   \__bnvs_DEBUG:x { \string#4 : \seq_use:Nn #4 . / }
895   </!gubed>
896   \prg_return_true:
897   }
898   \cs_set:Npn \branch:n ##1 {
899     \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
900       <!*gubed>
901       \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space POP-TRUE-##1 / }
902       \__bnvs_DEBUG:x { \string\l__bnvs_b_tl : \l__bnvs_b_tl / }
903       \__bnvs_DEBUG:x { \string\l__bnvs_a_seq : \seq_count:N\l__bnvs_a_seq / }
904       \__bnvs_DEBUG:x { \seq_use:Nn \l__bnvs_a_seq . / }
905       </!gubed>
906       \tl_put_right:Nn \l__bnvs_a_tl { . }
907       \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
908     } {
909       \cs_set_eq:NN \loop: \return_true:
910     }
911   }
912   \cs_set:Npn \loop: {
913     \__bnvs_call:TF {
914       \exp_args:Nx
915       \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
916         \branch:n { 1 }
917       } {
918         \exp_args:Nx
919         \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
920           \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
921             \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
922             \tl_set_eq:NN #3 \l__bnvs_b_tl
923             \seq_set_eq:NN #4 \l__bnvs_a_seq
924           } {
925             \branch:n { 2 }
926           }
927         } {
928           \branch:n { 3 }
929         }
930       }
931     } {
932       \cs_set:Npn \loop: {
933         \__bnvs_group_end:
934         <!*gubed>
935         \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space FALSE / }
936         \__bnvs_DEBUG:x { \string#1 / \string#2 : #2 / \string#3 : #3 / }
937         \__bnvs_DEBUG:x { \string#4:\seq_use:Nn #4 . / }
938         \__bnvs_DEBUG:x { \string\g__bnvs_call_int : \int_use:N\g__bnvs_call_int / }

```



```

939 </!gubed>
940     \prg_return_false:
941     }
942 }
943 \loop:
944 }
945 \loop:
946 } {
947     \prg_return_true:
948 }
949 }

```

5.5.7 Evaluation bricks

<code>_bnvs_fp_round:nN</code> <code>_bnvs_fp_round:N</code>	<code>_bnvs_fp_round:nN {<expression>} <tl variable></code> <code>_bnvs_fp_round:N <tl variable></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.

```

950 \cs_new:Npn \_bnvs_fp_round:nN #1 #2 {
951 <!*gubed>
952 \_bnvs_DEBUG:x { ROUND:\tl_to_str:n{#1} / \string#2=\tl_to_str:V #2}
953 </!gubed>
954 \tl_if_empty:nTF { #1 } {
955 <!*gubed>
956 \_bnvs_DEBUG:x { ...ROUND:~EMPTY }
957 </!gubed>
958 } {
959     \tl_put_right:Nx #2 {
960         \fp_eval:n { round(#1) }
961     }
962 <!*gubed>
963 \_bnvs_DEBUG:x { ...ROUND:~\tl_to_str:V #2 => \string#2}
964 </!gubed>
965 }
966 }
967 \cs_generate_variant:Nn \_bnvs_fp_round:nN { VN, xN }
968 \cs_new:Npn \_bnvs_fp_round:N #1 {
969     \tl_if_empty:VTF #1 {
970 <!*gubed>
971 \_bnvs_DEBUG:x { ROUND:~EMPTY }
972 </!gubed>
973 } {
974 <!*gubed>
975 \_bnvs_DEBUG:x { ROUND-IN:~\tl_to_str:V #1 }
976 </!gubed>
977     \tl_set:Nx #1 {
978         \fp_eval:n { round(#1) }
979     }
980 <!*gubed>
981 \_bnvs_DEBUG:x { ROUND-OUT:~\tl_to_str:V #1 }
982 </!gubed>
983 }

```

984 }

_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.

```

985 \cs_set:Npn \_bnvs_return_true:nnN #1 #2 #3 {
986   \tl_if_empty:NTF \l__bnvs_ans_tl {
987     \_bnvs_group_end:
988     <!*gubed>
989     \_bnvs_DEBUG:n { RETURN_FALSE/key=#1/type=#2/EMPTY }
990     </!gubed>
991     \_bnvs_gremove:n { #1//#2 }
992     \prg_return_false:
993   } {
994     \_bnvs_fp_round:N \l__bnvs_ans_tl
995     \_bnvs_gput:nV { #1//#2 } \l__bnvs_ans_tl
996     \exp_args:NNNV
997     \_bnvs_group_end:
998     \tl_put_right:Nn #3 \l__bnvs_ans_tl
999     <!*gubed>
1000    \_bnvs_DEBUG:x { RETURN_TRUE/key=#1/type=#2/ans=\l__bnvs_ans_tl/ }
1001    </!gubed>
1002    \prg_return_true:
1003  }
1004 }
1005 \cs_set:Npn \_bnvs_return_false:nn #1 #2 {
1006   <!*gubed>
1007   \_bnvs_DEBUG:n { RETURN_FALSE/key=#1/type=#2/ }
1008   </!gubed>
1009   \_bnvs_group_end:
1010   \_bnvs_gremove:n { #1//#2 }
1011   \prg_return_false:
1012 }
1013 \prg_new_conditional:Npnn \_bnvs_raw_first:nN #1 #2 { T, F, TF } {
1014   <!*gubed>
1015   \_bnvs_DEBUG:x { RAW_FIRST/
1016     key=\tl_to_str:n{#1}/\string #2=/\tl_to_str:V #2/
1017   }
1018   </!gubed>
1019   \_bnvs_if_in:nTF { #1/A } {
1020     <!*gubed>
1021     \_bnvs_DEBUG:n { RAW_FIRST/#1/CACHED }
1022     </!gubed>
1023     \tl_put_right:Nx #2 { \_bnvs_item:n { #1/A } }
1024     \prg_return_true:
1025   } {
1026     <!*gubed>
1027     \_bnvs_DEBUG:n { RAW_FIRST/key=#1/NOT_CACHED }
1028     </!gubed>
1029     \_bnvs_group_begin:
1030     \tl_clear:N \l__bnvs_ans_tl
1031     \_bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {

```

```

1032 <!*gubed>
1033 \__bnvs_DEBUG:x { RAW_FIRST/key=#1/A=\l__bnvs_a_tl }
1034 </!gubed>
1035 \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1036 \__bnvs_return_true:nnN { #1 } A #2
1037 } {
1038 \__bnvs_return_false:nn { #1 } A
1039 }
1040 } {
1041 <!*gubed>
1042 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/A/F }
1043 </!gubed>
1044 \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
1045 <!*gubed>
1046 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/L=\l__bnvs_a_tl }
1047 </!gubed>
1048 \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
1049 <!*gubed>
1050 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/Z=\l__bnvs_b_tl }
1051 </!gubed>
1052 \__bnvs_if_append:xNTF {
1053 \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
1054 } \l__bnvs_ans_tl {
1055 \__bnvs_return_true:nnN { #1 } A #2
1056 } {
1057 \__bnvs_return_false:nn { #1 } A
1058 }
1059 } {
1060 <!*gubed>
1061 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/Z/F/ }
1062 </!gubed>
1063 \__bnvs_return_false:nn { #1 } A
1064 }
1065 } {
1066 <!*gubed>
1067 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/L/F/ }
1068 </!gubed>
1069 \__bnvs_return_false:nn { #1 } A
1070 }
1071 }
1072 }
1073 }
1074 \prg_generate_conditional_variant:Nnn
1075 \__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 when not. Cache the result. Execute <true code> when there is a <first>, <false code> otherwise.

```

1076 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
1077 <!*gubed>
1078 \__bnvs_DEBUG:x { IF_FIRST/\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2}

```

```

1079 </!gubed>
1080 \__bnvs_raw_first:nNTF { #1 } #2 {
1081   \prg_return_true:
1082   } {
1083     \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {
1084 <!*!gubed>
1085 \__bnvs_DEBUG:n { IF_FIRST/#1/C/T/\l__bnvs_a_tl }
1086 </!gubed>
1087   \bool_set_true:N \l_no_counter_bool
1088   \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1089     \__bnvs_return_true:nn { #1 } A #2
1090   } {
1091     \__bnvs_return_false:nn { #1 } A
1092   }
1093   } {
1094     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1095       \__bnvs_gput:nn { #1/A } { 1 }
1096       \tl_set:Nn #2 { 1 }
1097 <!*!gubed>
1098 \__bnvs_DEBUG:x{IF_FIRST_MATCH:
1099   key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /
1100 }
1101 </!gubed>
1102   \__bnvs_return_true:nn { #1 } A #2
1103   } {
1104 <!*!gubed>
1105 \__bnvs_DEBUG:x{IF_FIRST_NO_MATCH:
1106   key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2
1107 }
1108 </!gubed>
1109   \__bnvs_return_false:nn { #1 } A
1110   }
1111 }
1112 }
1113 }

```

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

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

```

1114 \cs_new:Npn \__bnvs_first:nN #1 #2 {
1115   \__bnvs_if_first:nNF { #1 } #2 {
1116     \msg_error:nnn { beanoves } { :n } { Range-with-no-first:~#1 }
1117   }
1118 }
1119 \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.

```

1120 \prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
1121 <!*!gubed>
1122 \__bnvs_DEBUG:x { \string\__bnvs_raw_length:nN/#1/\string#2/ }

```

```

1123 </!gubed>
1124 \__bnvs_if_in:nTF { #1//L } {
1125 \tl_put_right:Nx #2 { \__bnvs_item:n { #1//L } }
1126 <!*!gubed>
1127 \__bnvs_DEBUG:x { RAW_LENGTH/CACHED/key:#1/\__bnvs_item:n { #1//L } }
1128 </!gubed>
1129 \prg_return_true:
1130 } {
1131 <!*!gubed>
1132 \__bnvs_DEBUG:x { RAW_LENGTH/NOT_CACHED/key:#1/ }
1133 </!gubed>
1134 \__bnvs_gput:nn { #1//L } { 0 }
1135 \__bnvs_group_begin:
1136 \tl_clear:N \l__bnvs_ans_tl
1137 \__bnvs_if_in:nTF { #1/L } {
1138 \__bnvs_if_append:xNTF {
1139 \__bnvs_item:n { #1/L }
1140 } \l__bnvs_ans_tl {
1141 \__bnvs_return_true:nnN { #1 } L #2
1142 } {
1143 \__bnvs_return_false:nn { #1 } L
1144 }
1145 } {
1146 \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
1147 \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
1148 \__bnvs_if_append:xNTF {
1149 \l__bnvs_b_tl - (\l__bnvs_a_tl) + 1
1150 } \l__bnvs_ans_tl {
1151 \__bnvs_return_true:nnN { #1 } L #2
1152 } {
1153 \__bnvs_return_false:nn { #1 } L
1154 }
1155 } {
1156 \__bnvs_return_false:nn { #1 } L
1157 }
1158 } {
1159 \__bnvs_return_false:nn { #1 } L
1160 }
1161 }
1162 }
1163 }
1164 \prg_generate_conditional_variant:Nnn
1165 \__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.

```

1166 \prg_new_conditional:Npnn \__bnvs_raw_last:nN #1 #2 { T, F, TF } {
1167 <!*!gubed>
1168 \__bnvs_DEBUG:n { RAW_LAST/#1 }
1169 </!gubed>
1170 \__bnvs_if_in:nTF { #1//Z } {

```

```

1171     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//Z } }
1172     \prg_return_true:
1173   } {
1174     \__bnvs_gput:nn { #1//Z } { 0 }
1175     \__bnvs_group_begin:
1176     \tl_clear:N \l__bnvs_ans_tl
1177     \__bnvs_if_in:nTF { #1/Z } {
1178       <*\gubed>
1179       \__bnvs_DEBUG:x { NORMAL_RAW_LAST:~\__bnvs_item:n { #1/Z } }
1180       </!\gubed>
1181       \__bnvs_if_append:xNTF {
1182         \__bnvs_item:n { #1/Z }
1183       } \l__bnvs_ans_tl {
1184         \__bnvs_return_true:nnN { #1 } Z #2
1185       } {
1186         \__bnvs_return_false:nn { #1 } Z
1187       }
1188     } {
1189       \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
1190         \__bnvs_get:nNTF { #1/L } \l__bnvs_b_tl {
1191           \__bnvs_if_append:xNTF {
1192             \l__bnvs_a_tl + (\l__bnvs_b_tl) - 1
1193           } \l__bnvs_ans_tl {
1194             \__bnvs_return_true:nnN { #1 } Z #2
1195           } {
1196             \__bnvs_return_false:nn { #1 } Z
1197           }
1198         } {
1199           \__bnvs_return_false:nn { #1 } Z
1200         }
1201       } {
1202         \__bnvs_return_false:nn { #1 } Z
1203       }
1204     }
1205   }
1206 }
1207 \prg_generate_conditional_variant:Nnn
1208   \__bnvs_raw_last:nN { VN } { T, F, TF }

```

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

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

```

1209 \cs_new:Npn \__bnvs_last:nN #1 #2 {
1210   \__bnvs_raw_last:nNF { #1 } #2 {
1211     \msg_error:nnn { beanoves } { :n } { Range-with-no-last:~#1 }
1212   }
1213 }
1214 \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.

```

1215 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
1216   \__bnvs_if_in:nTF { #1//N } {
1217     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//N } }
1218     \prg_return_true:
1219   } {
1220     \__bnvs_group_begin:
1221     \cs_set:Npn \__bnvs_return_true: {
1222       \tl_if_empty:NTF \l__bnvs_ans_tl {
1223         \__bnvs_group_end:
1224         \prg_return_false:
1225       } {
1226         \__bnvs_fp_round:N \l__bnvs_ans_tl
1227         \__bnvs_gput:nV { #1//N } \l__bnvs_ans_tl
1228         \exp_args:NNNV
1229         \__bnvs_group_end:
1230         \tl_put_right:Nn #2 \l__bnvs_ans_tl
1231         \prg_return_true:
1232       }
1233     }
1234     \cs_set:Npn \return_false: {
1235       \__bnvs_group_end:
1236       \prg_return_false:
1237     }
1238     \tl_clear:N \l__bnvs_a_tl
1239     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_a_tl {
1240       \__bnvs_if_append:xNTF {
1241         \l__bnvs_a_tl + 1
1242       } \l__bnvs_ans_tl {
1243         \__bnvs_return_true:
1244       } {
1245         \return_false:
1246       }
1247     } {
1248       \return_false:
1249     }
1250   }
1251 }
1252 \prg_generate_conditional_variant:Nnn
1253   \__bnvs_if_next:nN { VN } { T, F, TF }

```

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

`__bnvs_next:VN` Append the index after the `<name>` slide range to the `<tl variable>`.

```

1254 \cs_new:Npn \__bnvs_next:nN #1 #2 {
1255   \__bnvs_if_next:nNF { #1 } #2 {
1256     \msg_error:nnn { beanoves } { :n } { Range~with~no~next:~#1 }
1257   }
1258 }
1259 \cs_generate_variant:Nn \__bnvs_next:nN { VN }

```

<code>_bnvs_if_index:nnNTF</code> <code>_bnvs_if_index:VVNTF</code> <code>_bnvs_if_index:nnnNTF</code>	<code>_bnvs_if_index:nnNTF {<name>} {<integer>} <tl variable> {<true code>} {<false code>}</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.

```

1260 \prg_new_conditional:Npnn \\_bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
1261   (*!gubed)
1262   \\_bnvs_DEBUG:x { IF_INDEX:key=#1/index=#2/\string#3/ }
1263   <!/gubed>
1264   \\_bnvs_group_begin:
1265   \tl_clear:N \\_bnvs_ans_tl
1266   \\_bnvs_raw_first:nNTF { #1 } \\_bnvs_ans_tl {
1267     \tl_put_right:Nn \\_bnvs_ans_tl { + (#2) - 1}
1268     \exp_args:NNV
1269     \\_bnvs_group_end:
1270     \\_bnvs_fp_round:nN \\_bnvs_ans_tl #3
1271   (*!gubed)
1272   \\_bnvs_DEBUG:x { IF_INDEX_TRUE:key=#1/index=#2/
1273     \string#3=\tl_to_str:N #3
1274   }
1275   <!/gubed>
1276   \prg_return_true:
1277   } {
1278   (*!gubed)
1279   \\_bnvs_DEBUG:x { IF_INDEX_FALSE:key=#1/index=#2/ }
1280   <!/gubed>
1281   \prg_return_false:
1282   }
1283 }
1284 \prg_generate_conditional_variant:Nnn
1285   \\_bnvs_if_index:nnN { VVN } { T, F, TF }

```

<code>_bnvs_if_range:nNTF</code>	<code>_bnvs_if_range:nNTF {<name>} <tl variable> {<true code>} {<false code>}</code>
------------------------------------	--

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

```

1286 \prg_new_conditional:Npnn \\_bnvs_if_range:nN #1 #2 { T, F, TF } {
1287   (*!gubed)
1288   \\_bnvs_DEBUG:x{ RANGE:key=#1/\string#2/}
1289   <!/gubed>
1290   \bool_if:NTF \\_bnvs_no_range_bool {
1291     \prg_return_false:
1292   } {
1293     \\_bnvs_if_in:nTF { #1/ } {
1294       \tl_put_right:Nn { 0-0 }
1295     } {
1296       \\_bnvs_group_begin:
1297       \tl_clear:N \\_bnvs_a_tl
1298       \tl_clear:N \\_bnvs_b_tl

```



```

1299     \tl_clear:N \l__bnvs_ans_tl
1300     \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
1301         \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
1302             \exp_args:NNNx
1303             \__bnvs_group_end:
1304             \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
1305         } {
1306             \__bnvs_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
1307         } {
1308             \prg_return_true:
1309         } {
1310             \exp_args:NNNx
1311             \__bnvs_group_end:
1312             \tl_put_right:Nn #2 { \l__bnvs_a_tl - }
1313         } {
1314             \__bnvs_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
1315         } {
1316             \prg_return_true:
1317         } {
1318             \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
1319                 \__bnvs_DEBUG:x{ RANGE_TRUE_Z:key=#1/\string#2=#2/}
1320             } {
1321                 \exp_args:NNNx
1322                 \__bnvs_group_end:
1323                 \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1324                 \prg_return_true:
1325             } {
1326                 \__bnvs_DEBUG:x{ RANGE_FALSE:key=#1/}
1327             } {
1328                 \__bnvs_group_end:
1329                 \prg_return_false:
1330             } {
1331                 \__bnvs_group_end:
1332                 \prg_return_false:
1333             } {
1334                 \__bnvs_group_end:
1335             } {
1336                 \__bnvs_group_end:
1337             } {
1338                 \prg_generate_conditional_variant:Nnn
1339                 \__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>`.

```

1340 \cs_new:Npn \__bnvs_range:nN #1 #2 {
1341     \__bnvs_if_range:nNF { #1 } #2 {
1342         \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1343     }
1344 }
1345 \cs_generate_variant:Nn \__bnvs_range:nN { VN }

```

```

\__bnvs_if_free_counter:nNTF \__bnvs_if_free_counter:nNTF {\name} \tl variable {\true code} {\false
\__bnvs_if_free_counter:VNTF code)}

```

Set the $\langle \text{tl variable} \rangle$ to the value of the counter associated to the $\{\langle \text{name} \rangle\}$ slide range.

```

1346 \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
1347   \*!gubed)
1348   \__bnvs_DEBUG:x { IF_FREE: key=\tl_to_str:n{#1}/
1349     value=\__bnvs_item:n {#1/C}/cs=\string #2/
1350 }
1351 \*!gubed)
1352   \__bnvs_group_begin:
1353   \tl_clear:N \l__bnvs_ans_tl
1354   \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
1355     \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
1356       \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
1357     }
1358   }
1359   \*!gubed)
1360   \__bnvs_DEBUG:x { IF_FREE_2: \string \l__bnvs_ans_tl= }
1361   \__bnvs_DEBUG:x { \tl_to_str:V \l__bnvs_ans_tl/ }
1362   \*!gubed)
1363   \tl_if_empty:NNTF \l__bnvs_ans_tl {
1364     \__bnvs_group_end:
1365     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1366       \__bnvs_gput:nn { #1/C } { 1 }
1367       \tl_set:Nn #2 { 1 }
1368     }
1369     \*!gubed)
1370     \__bnvs_DEBUG:x { IF_FREE_MATCH_TRUE:
1371       key=\tl_to_str:n{#1}\string #2=\tl_to_str:V #2 /
1372     }
1373     \*!gubed)
1374     \prg_return_true:
1375     } {
1376     \*!gubed)
1377     \__bnvs_DEBUG:x { IF_FREE_NO_MATCH_FALSE:
1378       key=\tl_to_str:n{#1}\string #2=\tl_to_str:V #2/
1379     }
1380     \*!gubed)
1381     \prg_return_false:
1382     }
1383     } {
1384     \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl
1385     \exp_args:NNNV
1386     \__bnvs_group_end:
1387     \tl_set:Nn #2 \l__bnvs_ans_tl
1388     \*!gubed)
1389     \__bnvs_DEBUG:x { IF_FREE_TRUE(2): /
1390     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2
1391   }
1392   \*!gubed)
1393   \prg_return_true:
1394   }
1395   }
1396   \prg_generate_conditional_variant:Nnn

```

```
1396 \__bnvs_if_free_counter:nN { VN } { T, F, TF }
```

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

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.

```
1397 \prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {
1398   <!*gubed>
1399   \__bnvs_DEBUG:x { IF_COUNTER:key=
1400     \tl_to_str:n{#1}/\string #2=\tl_to_str:V #2
1401   }
1402   </!gubed>
1403   \__bnvs_group_begin:
1404   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_ans_tl {
```

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

```
1405     \tl_clear:N \l__bnvs_a_tl
1406     \__bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
1407       \fp_compare:nNt { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
1408         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1409       }
1410     }
```

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

```
1411     \tl_clear:N \l__bnvs_a_tl
1412     \__bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
1413       \fp_compare:nNt { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
1414         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1415       }
1416     }
1417     \exp_args:NNV
1418     \__bnvs_group_end:
1419     \__bnvs_fp_round:nN \l__bnvs_ans_tl #2
1420   <!*gubed>
1421   \__bnvs_DEBUG:x {IF_COUNTER_TRUE:key=\tl_to_str:n{#1}/
1422     \string #2=\tl_to_str:V #2
1423   }
1424   </!gubed>
1425   \prg_return_true:
1426   } {
1427   <!*gubed>
1428   \__bnvs_DEBUG:x {IF_COUNTER_FALSE:key=\tl_to_str:n{#1}/
1429     \string #2=\tl_to_str:V #2
1430   }
1431   </!gubed>
1432   \prg_return_false:
1433   }
1434 }
1435 \prg_generate_conditional_variant:Nnn
1436 \__bnvs_if_counter:nN { VN } { T, F, TF }
```

```
\__bnvs_if_incr:nnTF \__bnvs_if_incr:nnTF {<name>} {<offset>} {<true code>} {<false code>}
\__bnvs_if_incr:nnNTF \__bnvs_if_incr:nnNTF {<name>} {<offset>} <tl variable> {<true code>} {<false
\__bnvs_if_incr:(VnN|VVN)TF 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.

```

1437 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {
1438   \*!gubed>
1439   \__bnvs_DEBUG:x { IF_INCR:\tl_to_str:n{#1}/\tl_to_str:n{#2} }
1440   \*!gubed>
1441   \__bnvs_group_begin:
1442   \tl_clear:N \l__bnvs_a_tl
1443   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_a_tl {
1444     \tl_clear:N \l__bnvs_b_tl
1445     \__bnvs_if_append:xNTF { \l__bnvs_a_tl + (#2) } \l__bnvs_b_tl {
1446       \__bnvs_fp_round:N \l__bnvs_b_tl
1447       \__bnvs_gput:nV { #1/C } \l__bnvs_b_tl
1448     }
1449   }
1450   \__bnvs_DEBUG:x { IF_INCR_TRUE:#1/#2 }
1451   \*!gubed>
1452   \prg_return_true:
1453   } {
1454     \__bnvs_group_end:
1455     \*!gubed>
1456     \__bnvs_DEBUG:x { IF_INCR_FALSE(1):#1/#2 }
1457     \*!gubed>
1458     \prg_return_false:
1459     }
1460     } {
1461       \__bnvs_group_end:
1462       \*!gubed>
1463       \__bnvs_DEBUG:x { IF_INCR_FALSE(2):#1/#2 }
1464       \*!gubed>
1465       \prg_return_false:
1466       }
1467     }
1468   \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1469     \__bnvs_if_incr:nnTF { #1 } { #2 } {
1470       \__bnvs_if_counter:nNTF { #1 } #3 {
1471         \prg_return_true:
1472         } {
1473           \prg_return_false:
1474           }
1475         } {
1476           \prg_return_false:
1477           }
1478       }
1479   \prg_generate_conditional_variant:Nnn
1480     \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }

```

5.5.8 Evaluation

<u>_bnvs_if_append:nNTF</u> <u>_bnvs_if_append:(VN xN)TF</u>	<p>_bnvs_if_append:nNTF {<integer expression>} <tl variable> {<true code>} {<false code>}</p> <p>Evaluates the <integer expression>, replacing all the named specifications by their static counterpart then put the result to the right of the <tl variable>. Executed within a group. Heavily used by _bnvs_eval_query:nN, where <integer expression> was initially enclosed in '?(...)'. Local variables:</p>
\l__bnvs_ans_tl	<p>To feed <tl variable> 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>
1481 \int_new:N \l__bnvs_split_int	
\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>
1482 \prg_new_conditional:Npnn _bnvs_if_append:nN #1 #2 { T, F, TF } {	
1483 <!*gubed>	
1484 _bnvs_DEBUG:x { \string_bnvs_if_append:nNTF/	
1485 \tl_to_str:n { #1 } / \string #2/	
1486 }	
1487 </!gubed>	
1488 _bnvs_call:TF {	
1489 <!*gubed>	
1490 _bnvs_DEBUG:x { IF_APPEND...}	
1491 </!gubed>	
1492 _bnvs_group_begin:	
Local variables:	
1493 \int_zero:N \l__bnvs_split_int	
1494 \seq_clear:N \l__bnvs_split_seq	
1495 \tl_clear:N \l__bnvs_id_tl	
1496 \tl_clear:N \l__bnvs_name_tl	
1497 \tl_clear:N \l__bnvs_path_tl	
1498 \tl_clear:N \l__bnvs_group_tl	
1499 \tl_clear:N \l__bnvs_ans_tl	
1500 \tl_clear:N \l__bnvs_a_tl	
Implementation:	
1501 \regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq	

```

1502 <!*gubed>
1503 \__bnvs_DEBUG:x { IF_APPEND_SPLIT_SEQ: /
1504 \#=\seq_count:N \l__bnvs_split_seq /
1505 \seq_use:Nn \l__bnvs_split_seq //
1506 }
1507 </!gubed>
1508 \int_set:Nn \l__bnvs_split_int { 1 }
1509 \tl_set:Nx \l__bnvs_ans_tl {
1510 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1511 }
1512 <!*gubed>
1513 \__bnvs_DEBUG:x { ANS: \l__bnvs_ans_tl }
1514 </!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.

```

1515 \cs_set:Npn \switch:nTF ##1 ##2 ##3 ##4 {
1516 \tl_set:Nx ##2 {
1517 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1518 }
1519 <!*gubed>
1520 \__bnvs_DEBUG:x { IF_APPEND_SWITCH/##1/
1521 \int_eval:n { \l__bnvs_split_int + ##1 } /
1522 \string##2=\tl_to_str:N##2/
1523 }
1524 </!gubed>
1525 \tl_if_empty:NTF ##2 {
1526 <!*gubed>
1527 \__bnvs_DEBUG:x { IF_APPEND_SWITCH_WHITE/##1/
1528 \int_eval:n { \l__bnvs_split_int + ##1 }
1529 }
1530 </!gubed>
1531 ##4 } {
1532 <!*gubed>
1533 \__bnvs_DEBUG:x { IF_APPEND_SWITCH_BLACK/##1/
1534 \int_eval:n { \l__bnvs_split_int + ##1 }
1535 }
1536 </!gubed>
1537 ##3
1538 }
1539 }

```

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

```

1540 \cs_set:Npn \return_true: {
1541 \fp_round:
1542 \exp_args:NNNV
1543 \__bnvs_group_end:
1544 \tl_put_right:Nn #2 \l__bnvs_ans_tl
1545 <!*gubed>
1546 \__bnvs_DEBUG:x { IF_APPEND_TRUE:\tl_to_str:n { #1 } /
1547 \string #2=\tl_to_str:V #2 /

```

```

1548 }
1549 \log_g_prop:
1550 </!gubed>
1551 \prg_return_true:
1552 }
1553 \cs_set:Npn \fp_round: {
1554   \__bnvs_fp_round:N \l__bnvs_ans_tl
1555 }
1556 \cs_set:Npn \return_false: {
1557   \__bnvs_group_end:
1558 <!*gubed>
1559 \__bnvs_DEBUG:x { IF_APPEND_FALSE:\tl_to_str:n { #1 } /
1560 \string #2=\tl_to_str:V #2 /
1561 }
1562 </!gubed>
1563 \prg_return_false:
1564 }
1565 \cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {
1566   \switch:nNTF { ##2 } \l__bnvs_id_tl { } {
1567     \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_current_tl
1568     \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1569   }
1570   \switch:nNTF { ##3 } \l__bnvs_path_tl {
1571     \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
1572     \seq_remove_all:Nn \l__bnvs_path_seq { }
1573 <!*gubed>
1574 \__bnvs_DEBUG:x { PATH_SEQ:\l__bnvs_path_tl==\seq_use:Nn\l__bnvs_path_seq .}
1575 </!gubed>
1576 } {
1577   \seq_clear:N \l__bnvs_path_seq
1578 }
1579 <!*gubed>
1580 \__bnvs_DEBUG:x { PATH_SEQ:\l__bnvs_path_tl==\seq_use:Nn\l__bnvs_path_seq .}
1581 \__bnvs_DEBUG:x { \string ##1 }
1582 </!gubed>
1583   ##1 \l__bnvs_id_tl \l__bnvs_name_tl \l__bnvs_path_seq {
1584     \cs_set:Npn \: {
1585       ##4
1586     }
1587   } {
1588     \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
1589   }
1590   \:
1591 }
1592 \cs_set:Npn \:T ##1 {
1593   \seq_if_empty:NTF \l__bnvs_path_seq { ##1 } {
1594     \cs_set_eq:NN \loop: \return_false:
1595   }
1596 }

```

Main loop.

```

1597   \cs_set:Npn \loop: {
1598 <!*gubed>
1599 \__bnvs_DEBUG:x { IF_APPEND_LOOP:\int_use:N\l__bnvs_split_int /
1600 \seq_count:N \l__bnvs_split_seq /

```

```

1601 }
1602 </!gubed>
1603 \int_compare:nNnTF {
1604   \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1605 } {
1606   \switch:nNTF 1 \l__bnvs_name_tl {

• Case ++<name><integer path>.n.

1607   \:NnnT \__bnvs_resolve_n:NNNTF 2 3 {
1608     \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1609       \cs_set_eq:NN \loop: \return_false:
1610     }
1611   }
1612 } {
1613   \switch:nNTF 4 \l__bnvs_name_tl {

• Cases <name><integer path>....

1614   \switch:nNTF 7 \l__bnvs_a_tl {
1615     \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1616       \:T {
1617         \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1618           \cs_set_eq:NN \loop: \return_false:
1619         }
1620       }
1621     }

• Case ...length.

1622   } {
1623     \switch:nNTF 8 \l__bnvs_a_tl {

• Case ...last.

1624     \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1625       \:T {
1626         \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1627           \cs_set_eq:NN \loop: \return_false:
1628         }
1629       }
1630     }

1631   } {
1632     \switch:nNTF 9 \l__bnvs_a_tl {

• Case ...next.

1633     \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1634       \:T {
1635         \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1636           \cs_set_eq:NN \loop: \return_false:
1637         }
1638       }
1639     }
1640   } {
1641     \switch:nNTF { 10 } \l__bnvs_a_tl {

```


- Case ...range.

```

1642 \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1643   \:T {
1644     \__bnvs_if_range:VNTF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1645       \cs_set_eq:NN \fp_round: \prg_do_nothing:
1646     } {
1647       \cs_set_eq:NN \loop: \return_false:
1648     }
1649   }
1650 }
1651           } {
1652           \switch:nNTF { 11 } \l__bnvs_a_t1 {

```

- Case ...n.

```

1653           \switch:nNTF { 12 } \l__bnvs_a_t1 {

```

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

```

1654 \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1655   \:T {
1656      $\langle *!gubed \rangle$ 
1657     \__bnvs_DEBUG:x {NAME=\l__bnvs_name_t1}
1658     \__bnvs_DEBUG:x {INCR=\l__bnvs_a_t1}
1659      $\langle /!gubed \rangle$ 
1660     \__bnvs_if_incr:VVNF \l__bnvs_name_t1 \l__bnvs_a_t1 \l__bnvs_ans_t1 {
1661       \cs_set_eq:NN \loop: \return_false:
1662     }
1663   }
1664 }
1665           } {
1666           \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1667             \seq_if_empty:NTF \l__bnvs_path_seq {
1668               \__bnvs_if_counter:VNF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1669                 \cs_set_eq:NN \loop: \return_false:
1670               }
1671             } {
1672               \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_t1
1673               \seq_if_empty:NTF \l__bnvs_path_seq {
1674                 \__bnvs_if_incr:VVNF \l__bnvs_name_t1 \l__bnvs_a_t1 \l__bnvs_ans_t1 {
1675                   \cs_set_eq:NN \loop: \return_false:
1676                 }
1677               } {
1678                 \msg_error:nxx { beanoves } { :n } { Too~many~. $\langle integer \rangle$ ~components:~#1 }
1679                 \cs_set_eq:NN \loop: \return_false:
1680               }
1681             }
1682           }
1683         }
1684       } {
1685         \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1686           \seq_if_empty:NTF \l__bnvs_path_seq {
1687             \__bnvs_if_counter:VNF \l__bnvs_name_t1 \l__bnvs_ans_t1 {

```

```

1688 \cs_set_eq:NN \loop: \return_false:
1689 }
1690
1691         } {
1692         \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1693         \seq_if_empty:NTF \l__bnvs_path_seq {
1694         \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1695         \cs_set_eq:NN \loop: \return_false:
1696         }
1697         } {
1698         \msg_error:nxx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1699         \cs_set_eq:NN \loop: \return_false:
1700         }
1701     }
1702 }
1703 }
1704 }
1705 }
1706 }
1707 } {

```

No name.

```

1708     }
1709 }
1710 <*>gubed>
1711 \__bnvs_DEBUG:x {ITERATE~ANS=\l__bnvs_ans_tl }
1712 </>gubed>
1713 \int_add:Nn \l__bnvs_split_int { 13 }
1714 \tl_put_right:Nx \l__bnvs_ans_tl {
1715 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1716 }
1717 <*>gubed>
1718 \__bnvs_DEBUG:x {ITERATE~ANS=\l__bnvs_ans_tl }
1719 </>gubed>
1720 \loop:
1721 } {
1722 <*>gubed>
1723 \__bnvs_DEBUG:x {END_OF_LOOP~ANS=\l__bnvs_ans_tl }
1724 </>gubed>
1725 \return_true:
1726 }
1727 }
1728 \loop:
1729 } {
1730 \msg_error:nxx { beanoves } { :n } { Too-many~calls:~ #1 }
1731 \prg_return_false:
1732 }
1733 }
1734 \prg_generate_conditional_variant:Nnn
1735 \__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 $\langle overlay\ query \rangle$, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of the $\langle seq\ variable \rangle$. Ranges are supported with the colon syntax. This is executed within a local group. Below are local variables and constants.

```

\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.)

```

```

1736 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1737   \A \s* (?

```

- 2: $\langle first \rangle$

```

1738     ( [^:]* ) \s* :

```

- 3: second optional colon

```

1739     (:)? \s*

```

- 4: $\langle length \rangle$

```

1740     ( [^:]* )

```

- 5: standalone $\langle first \rangle$

```

1741     | ( [^:]+ )
1742   ) \s* \Z
1743 }

```

```

1744 \prg_new_conditional:Npnn \__bnvs_if_eval_query:nN #1 #2 { T, F, TF } {
1745   \<!*gubed>
1746   \__bnvs_DEBUG:x { EVAL_QUERY:#1/
1747     \tl_to_str:n{#1}/\string#2=\tl_to_str:N #2
1748   }
1749   \</!gubed>
1750   \__bnvs_call_reset:
1751   \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
1752     #1
1753   } \l__bnvs_match_seq {
1754     \<!*gubed>
1755     \__bnvs_DEBUG:x { EVAL_QUERY:#1/
1756       \string\l__bnvs_match_seq/\seq_use:Nn \l__bnvs_match_seq //
1757     }
1758     \</!gubed>

```

```

1759     \bool_set_false:N \l__bnvs_no_counter_bool
1760     \bool_set_false:N \l__bnvs_no_range_bool

```

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

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

```

1761     \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1762     <!*gubed>
1763     \__bnvs_DEBUG:x { EQ_SWITCH:##1/ }
1764     </!gubed>
1765     \tl_set:Nx ##2 {
1766     \seq_item:Nn \l__bnvs_match_seq { ##1 }
1767     }
1768     <!*gubed>
1769     \__bnvs_DEBUG:x { \string ##2/ \tl_to_str:N ##2/}
1770     </!gubed>
1771     \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1772     }
1773     \switch:nNTF 5 \l__bnvs_a_tl {

```

● Single expression

```

1774     \bool_set_false:N \l__bnvs_no_range_bool
1775     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1776     \prg_return_true:
1777     } {
1778     \prg_return_false:
1779     }
1780     } {
1781     \switch:nNTF 2 \l__bnvs_a_tl {
1782     \switch:nNTF 4 \l__bnvs_b_tl {
1783     \switch:nNTF 3 \l__bnvs_c_tl {

```

● <first>::<last> range

```

1784     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1785     \tl_put_right:Nn #2 { - }
1786     \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1787     \prg_return_true:
1788     } {
1789     \prg_return_false:
1790     }
1791     } {
1792     \prg_return_false:
1793     }
1794     } {

```

● <first>:<length> range

```

1795     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1796     \tl_put_right:Nx #2 { - }
1797     \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1 }
1798     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1799     \prg_return_true:
1800     } {
1801     \prg_return_false:

```

```

1802         }
1803     } {
1804         \prg_return_false:
1805     }
1806 }
1807 } {
    ☛ <first>: and <first>:: range
1808     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1809         \tl_put_right:Nn #2 { - }
1810         \prg_return_true:
1811     } {
1812         \prg_return_false:
1813     }
1814 }
1815 } {
1816     \switch:nNTF 4 \l__bnvs_b_tl {
1817         \switch:nNTF 3 \l__bnvs_c_tl {
    ☛ ::<last> range
1818         \tl_put_right:Nn #2 { - }
1819         \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1820             \prg_return_true:
1821         } {
1822             \prg_return_false:
1823         }
1824     } {
1825 \msg_error:nxx { beanoves } { :n } { Syntax-error(Missing-first):~#1 }
1826     }
1827 } {
    ☛ : or :: range
1828     \seq_put_right:Nn #2 { - }
1829 }
1830 }
1831 }
1832 } {
    Error
1833     \msg_error:nnn { beanoves } { :n } { Syntax-error:~#1 }
1834 }
1835 }

```

`__bnvs_eval:nN` `__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.

1836 `\regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }`

(End definition for `\c__bnvs_comma_regex`.)

No other variable is used.

1837 `\cs_new:Npn __bnvs_eval:nN #1 #2 {`
1838 `<!*gubed>`
1839 `__bnvs_DEBUG:x {<string__bnvs_eval:nN:\tl_to_str:n{#1}/`
1840 `<string#2=\tl_to_str:V #2`
1841 `}`
1842 `</!gubed>`
1843 `__bnvs_group_begin:`

Local variables declaration

1844 `\seq_clear:N \l__bnvs_query_seq`
1845 `\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`.

1846 `\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.

1847 `\seq_map_inline:Nn \l__bnvs_query_seq {`
1848 `\tl_clear:N \l__bnvs_ans_tl`
1849 `__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {`
1850 `\seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl`
1851 `} {`
1852 `\seq_map_break:n {`
1853 `\msg_fatal:nnn { beanoves } { :n } { Circular~dependency~in~#1}`
1854 `}`
1855 `}`
1856 `}`

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

1857 `\exp_args:NNNx`
1858 `__bnvs_group_end:`

```

1859 \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
1860 }
1861 \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.

```

1862 \NewDocumentCommand \BeanovesEval { s o m } {
1863   \__bnvs_group_begin:
1864   \tl_clear:N \l__bnvs_ans_tl
1865   \IfBooleanTF { #1 } {
1866     \bool_set_true:N \l__bnvs_no_counter_bool
1867   } {
1868     \bool_set_false:N \l__bnvs_no_counter_bool
1869   }
1870   \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
1871   \IfValueTF { #2 } {
1872     \exp_args:NNNV
1873     \__bnvs_group_end:
1874     \tl_set:Nn #2 \l__bnvs_ans_tl
1875   } {
1876     \exp_args:NV
1877     \__bnvs_group_end: \l__bnvs_ans_tl
1878   }
1879 }

```

5.5.9 Reseting slide ranges

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

```

1880 \NewDocumentCommand \BeanovesReset { O{1} m } {
1881   \__bnvs_reset:nn { #1 } { #2 }
1882   \ignorespaces
1883 }

```

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.

```

1884 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1885   \bool_if:nTF {
1886     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1887   } {
1888     \__bnvs_gremove:n { #2/C }
1889     \__bnvs_gremove:n { #2//A }
1890     \__bnvs_gremove:n { #2//L }
1891     \__bnvs_gremove:n { #2//Z }

```

```

1892     \__bnvs_gremove:n { #2//N }
1893     \__bnvs_gput:nn { #2/C0 } { #1 }
1894   } {
1895     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1896   }
1897 }

1898 \makeatother
1899 \ExplSyntaxOff
1900 </package>

```