

# beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0      2023/01/07

## Abstract

This package allows the management of multiple named slide number sets in `beamer` documents. Named slide number sets are very handy both during edition and to manage complex and variable `beamer` overlay specifications. In particular, they allow to replace raw numbers in `beamer` `<...>` overlay specifications by logical identifiers. Demonstration files are [available for download](#) as part of the [development repository](#).

## Contents

<b>1</b>	<b>Minimal example</b>	<b>1</b>
<b>2</b>	<b>Named overlay sets</b>	<b>2</b>
2.1	Presentation . . . . .	2
2.2	Named overlay reference . . . . .	2
2.3	Defining named overlay sets . . . . .	3
<b>3</b>	<b>Named overlay resolution</b>	<b>3</b>
3.1	Simple definitions . . . . .	3
3.2	Counters . . . . .	5
3.3	Dotted paths . . . . .	6
3.4	Frame id . . . . .	6
<b>4</b>	<b>?(...) query expressions</b>	<b>7</b>
<b>5</b>	<b>Support</b>	<b>7</b>
<b>6</b>	<b>Implementation</b>	<b>7</b>
6.1	Package declarations . . . . .	8
6.2	logging . . . . .	8
6.3	Debugging and testing facilities . . . . .	8
6.4	Local variables . . . . .	8
6.5	Infinite loop management . . . . .	9
6.6	Overlay specification . . . . .	10
6.6.1	In slide range definitions . . . . .	10
6.7	Basic functions . . . . .	10
6.8	Functions with cache . . . . .	12
6.8.1	Implicit index counter . . . . .	13

6.8.2	Regular expressions	14
6.8.3	beamer.cls interface	16
6.8.4	Defining named slide ranges	16
6.8.5	Scanning named overlay specifications	24
6.8.6	Resolution	27
6.8.7	Evaluation bricks	35
6.8.8	Evaluation	47
6.8.9	Resetting counters	58

## 1 Minimal example

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

```

1 \documentclass {beamer}
2 \RequirePackage {beanoves}
3 \begin{document}
4 \Beanoves {
5     A = 1:2,
6     B = A.next:3,
7     C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \insertframenumber}
11 {\Large Slide \insertslidenumber}
12 \visible<?(A.1)> {Only on slide 1}\\
13 \visible<?(B.1)-?(B.last)> {Only on slide 3 to 5}\\
14 \visible<?(C.1)> {Only on slide 6}\\
15 \visible<?(A.2)> {Only on slide 2}\\
16 \visible<?(B.2::B.last)> {Only on slide 4 to 5}\\
17 \visible<?(C.2)> {Only on slide 7}\\
18 \visible<?(A.next)-> {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 overlay sets*. On line 5, we declare an overlay set named ‘A’, which is a range starting at slide 1 and with length 2. On line 12, the extended *named overlay specification* `?(A.1)` stands for 1 because 1 is the first index of the overlay set named A. On line 15, `?(A.2)` stands for 2 whereas on line 18, `?(A.next)` stands for 3. On line 6, we declare a second overlay set named ‘B’, starting after the 2 slides of ‘A’ namely 3. Its length is 3 meaning that its last slide number is 5, thus each `?(B.last)` is replaced by 5. The next slide number after slide range ‘B’ is 6 which is also the start of the third slide range due to line 7.

## 2 Named overlay sets

### 2.1 Presentation

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

### 2.2 Named overlay reference

`A.1`, `C.2` are *named overlay references*, as well as `A` and `Y!C.2`. More precisely, they are string identifiers, each one representing a well defined static integer to be used in `beamer` overlay specifications. They can take one of the next forms.

$\langle \text{short name} \rangle$  : like `A` and `C`,

$\langle \text{frame id} \rangle ! \langle \text{short name} \rangle$  : denoted by *qualified names*, like `X!A` and `Y!C`.

$\langle \text{short name} \rangle \langle \text{dotted path} \rangle$  : denoted by *full names* like `A.1` and `C.2`,

$\langle \text{frame id} \rangle ! \langle \text{short name} \rangle \langle \text{dotted path} \rangle$  : denoted by *qualified full names* like `X!A.1` and `Y!C.2`.

The *short names* and *frame ids* are alphanumerical case sensitive identifiers, with possible underscores but no space nor leading digit. Unicode symbols above `U+00A0` are allowed if the underlying `TeX` engine supports it. Identifiers consisting only of lowercase letters and underscores are reserved by the package.

The *dotted path* is a string  $\langle \text{component}_1 \rangle . \langle \text{component}_2 \rangle \dots \langle \text{component}_n \rangle$ , where each  $\langle \text{component}_i \rangle$  denotes either an integer, eventually signed, or a  $\langle \text{short name} \rangle$ . The *dotted path* can be empty for which `n` is 0.

The mapping from *named overlay references* to integers is defined at the global `TeX` level to allow its use in `\begin{frame}<...>` and to share the same overlay sets between different frames. Hence the *frame id* due to the need to possibly target a particular frame.

### 2.3 Defining named overlay sets

In order to define *named overlay setss*, we can either execute the next `\Beanoves` command 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.

---

<code>beanoves</code>	<code>beanoves = \{ \langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle, \langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle, \dots, \langle \text{ref}_n \rangle = \langle \text{spec}_n \rangle \}</code>
-----------------------	---

---

<code>\Beanoves</code>	<code>\Beanoves \{ \langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle, \langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle, \dots, \langle \text{ref}_n \rangle = \langle \text{spec}_n \rangle \}</code>
------------------------	--

---

Each  $\langle \text{ref} \rangle$  key is a *named overlay reference* whereas each  $\langle \text{spec} \rangle$  value is an *overlay set specifier*. When the same  $\langle \text{ref} \rangle$  key is used multiple times, only the last one is taken

into account. Possible  $\langle spec \rangle$  value are the *range specifiers*

$\langle first \rangle$ ,  $\langle first \rangle :$  and  $\langle first \rangle ::$ ,  $\langle first \rangle : \langle length \rangle$ ,  $\langle first \rangle :: \langle last \rangle$ ,  
 $: \langle length \rangle :: \langle last \rangle$  and  $:: \langle last \rangle : \langle length \rangle$ .

Here  $\langle first \rangle$ ,  $\langle length \rangle$  and  $\langle last \rangle$  are algebraic expression possibly involving any *named overlay reference* defined above. At least one of  $\langle first \rangle$  or  $\langle last \rangle$  must be provided.

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside  $\text{\LaTeX}$  environments, each call cumulates with the previous. Notice that the argument of this function can contain macros: they will be exhaustively expanded.

Also possible values are *list specifiers* which are comma separated lists of  $\langle ref \rangle = \langle spec \rangle$  definitions. The definition

$\langle key \rangle = [\langle ref_1 \rangle = \langle spec_1 \rangle, \langle ref_2 \rangle = \langle spec_2 \rangle, \dots, \langle ref_n \rangle = \langle spec_n \rangle]$

is a convenient shortcut for

$\langle key \rangle . \langle ref_1 \rangle = \langle value_1 \rangle$ ,

$\langle key \rangle . \langle ref_2 \rangle = \langle value_2 \rangle$ ,

$\dots$ ,

$\langle key \rangle . \langle ref_n \rangle = \langle value_n \rangle$ .

The rules above can apply individually to each line.

To support an array syntax, we can omit the  $\langle ref \rangle$  key. The first missing key is replaced by 1, the second by 2, and so on.

### 3 Named overlay resolution

Turning a *named overlay reference* into the static integer it represents, as in  $\langle ?(A.1) \rangle$  above is denoted *named overlay resolution* or simply *resolution*. This section is devoted to *resolution rules* depending on the definition of the named overlay set. Here  $\langle i \rangle$  denotes an integer whereas  $\langle first \rangle$ ,  $\langle length \rangle$  and  $\langle last \rangle$  stand for integers, or integer valued expressions.

#### 3.1 Simple definitions

$\langle name \rangle = \langle first \rangle$  For an unlimited range

reference	resolution
$\langle name \rangle . 1$	$\langle first \rangle$
$\langle name \rangle . 2$	$\langle first \rangle + 1$
$\langle name \rangle . \langle i \rangle$	$\langle first \rangle + \langle i \rangle - 1$

$\langle name \rangle = \langle first \rangle :$  as well as  $\langle first \rangle ::$ . For a range limited from below:

reference	resolution
$\langle name \rangle . 1$	$\langle first \rangle$
$\langle name \rangle . 2$	$\langle first \rangle + 1$
$\langle name \rangle . \langle i \rangle$	$\langle first \rangle + \langle i \rangle - 1$
$\langle name \rangle . previous$	$\langle first \rangle - 1$

$\langle name \rangle = :: \langle last \rangle$  For a range limited from above:

reference	resolution
$\langle name \rangle.1$	$\langle last \rangle$
$\langle name \rangle.0$	$\langle last \rangle - 1$
$\langle name \rangle.i$	$\langle last \rangle + \langle i \rangle - 1$
$\langle name \rangle.next$	$\langle last \rangle + 1$

$\langle name \rangle = \langle first \rangle : \langle length \rangle$  as well as variants  $\langle first \rangle :: \langle last \rangle$ ,  $:\langle length \rangle :: \langle last \rangle$  or  $:: \langle last \rangle : \langle length \rangle$ , which are equivalent provided  $\langle first \rangle + \langle length \rangle = \langle last \rangle + 1$ .

For a range limited from both above and below:

reference	resolution
$\langle name \rangle.1$	$\langle first \rangle$
$\langle name \rangle.2$	$\langle first \rangle + 1$
$\langle name \rangle.i$	$\langle first \rangle + \langle i \rangle - 1$
$\langle name \rangle.previous$	$\langle first \rangle - 1$
$\langle name \rangle.last$	$\langle last \rangle$
$\langle name \rangle.next$	$\langle last \rangle + 1$
$\langle name \rangle.length$	$\langle length \rangle$
$\langle name \rangle.range$	$\max(0, \langle first \rangle) \text{ '-' } \max(0, \langle last \rangle)$

Notice that the resolution of  $\langle name \rangle.range$  is not an algebraic difference, and negative integers do not make sense there while in `beamer` context.

For example

```

1 \Beanoves {
2   A = 3:6, % or equivalently A = 3::8, A = :6::8 and A = ::8:6
3 }
4 \begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
5 \ttfamily
6 \BeanovesEval(A.1)      == 3,
7 \BeanovesEval(A.-1)    == 1,
8 \BeanovesEval(A.previous) == 2,
9 \BeanovesEval(A.last)  == 8,
10 \BeanovesEval(A.next)  == 9,
11 \BeanovesEval(A.length) == 6,
12 \BeanovesEval(A.range) == 3-8,
13 \end{frame}

```

### 3.2 Counters

Each named overlay set defined has a dedicated value counter which is some kind of variable that can be used and incremented. A simple  $\langle name \rangle$  *named value reference* is resolved into the position of this value counter. For each frame, this variable is initialized to  $\langle name \rangle.first$  when finite or  $\langle name \rangle.last$  otherwise.

For each named overlay set defined, we also have an implicit index counter always starting at 1, its actual value is an integer denoted  $\langle n \rangle$ . The  $\langle name \rangle.n$  *named index reference* is resolved into  $\langle name \rangle.\langle n \rangle$ , which in turn is resolved according to the preceding rules.

Additionally, resolution rules are provided for the *named value references*:

$\langle \text{name} \rangle += \langle \text{integer expression} \rangle$  : resolve  $\langle \text{integer expression} \rangle$  into  $\langle \text{integer} \rangle$ , advance the value counter by  $\langle \text{integer} \rangle$  and use the new position. Here  $\langle \text{integer expression} \rangle$  is the longest character sequence with no space<sup>1</sup>.

$++\langle \text{name} \rangle$  : advance the value counter for  $\langle \text{name} \rangle$  by 1 and use the new position.

$\langle \text{name} \rangle ++$  : use the actual position and advance the value counter for  $\langle \text{name} \rangle$  by 1.

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

We have resolution rules as well for the *named index references*:

$\langle \text{name} \rangle . \text{n} += \langle \text{integer expression} \rangle$  : resolve  $\langle \text{integer expression} \rangle$  into  $\langle \text{integer} \rangle$ , advance the implicit index counter associate to  $\langle \text{name} \rangle$  by  $\langle \text{integer} \rangle$  and use the resolution of  $\langle \text{name} \rangle . \text{n}$ .

Here again,  $\langle \text{integer expression} \rangle$  denotes the longest character sequence with no space.

$\langle \text{name} \rangle . ++\text{n}$  as well as  $++\langle \text{name} \rangle . \text{n}$ : advance the implicit index counter associate to  $\langle \text{name} \rangle$  by 1 and use the resolution of  $\langle \text{name} \rangle . \text{n}$ ,

$\langle \text{name} \rangle . \text{n} ++$  : use the resolution of  $\langle \text{name} \rangle . \text{n}$  and increment the implicit index counter associate to  $\langle \text{name} \rangle$  by 1.

In order to decrement a counter, one can increment with a negative value, no dedicated syntax is provided yet.

These counters are reset to their default value for each new frame, which is 1 for the  $\langle \text{name} \rangle . \text{n}$  counter, and whichever  $\langle \text{name} \rangle . \text{first}$  or  $\langle \text{name} \rangle . \text{last}$  is defined for the  $\langle \text{name} \rangle$  counter.

### 3.3 Dotted paths

$\langle \text{name} \rangle . \langle i \rangle = \langle \text{range spec} \rangle$  All the preceding rules are overridden by this particular one and  $\langle \text{name} \rangle . \langle i \rangle$  resolves to the resolution of  $\langle \text{range spec} \rangle$ .

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,
3   A.3 = 0,
4 }
5 \begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
6 \ttfamily
7 \BeanovesEval(A.1) == 3,
8 \BeanovesEval(A.2) == 4,
9 \BeanovesEval(A.-1) == 1,
10 \BeanovesEval(A.3) == 0,
11 \end{frame}

```

<sup>1</sup>The parser for algebraic expression is very rudimentary.

$\langle \text{name} \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_k \rangle = \langle \text{range spec} \rangle$  When a dotted path has more than one component, a *named overlay reference* like A.1.2 needs some well defined resolution rule to avoid ambiguity. To resolve one level of such a reference  $\langle \text{name} \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_n \rangle$ , we replace the longest  $\langle \text{name} \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_k \rangle$  where  $0 \leq k \leq n$  by its definition  $\langle \text{name}' \rangle . \langle c'_1 \rangle \dots \langle c'_p \rangle$  if any (the path can be empty). **beanoves** uses this one level resolution as many times as possible, but no more than a predefined limit to catch circular reference that would lead to an infinite  $\text{\TeX}$  loop. One final resolution occurs with rules above if possible or an error is raised.

For a *named indexed reference* like  $\langle \text{name} \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_n \rangle . n$ , we must first resolve  $\langle \text{name} \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_n \rangle$  into  $\langle \text{name}' \rangle$  with an empty dotted path, then retrieve the value of  $\langle \text{name}' \rangle . n$  denoted as  $\langle n' \rangle$  and finally use the resolved  $\langle \text{name} \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_n \rangle . \langle n' \rangle$ .

### 3.4 Frame id

Except for very special situations, the *frame ids* can be left unspecified. When no *frame id* was explicitly provided, **beanoves** uses the *last frame id*. At the beginning of each frame, the *last frame id* is set to the *frame id* of the current frame, which is denoted *current frame id* and defaults to ?. Then it gets updated after each named reference resolution. For example, the first time A.1 reference is resolved within a given frame, it is first translated to  $\langle \text{current frame id} \rangle ! A.1$ , but when used just after Y!C.2, it becomes a shortcut to Y!A.1 because the *last frame id* was then Y.

In order to set the *frame id* of the current frame to  $\langle \text{frame id} \rangle$ , use the new **beanoves** *id* option of the **beamer** frame environment.

---

<b>beanoves id</b>	<b>beanoves id</b> = $\langle \text{frame id} \rangle$ ,
--------------------	--

---

We can use the same *frame id* for different frames to share named overlay sets.

## 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  $\langle ?(\langle \text{queries} \rangle) \rangle$ . Each one is then evaluated and replaced by its resolved static counterpart. The overall result is finally forwarded to the **beamer** class.

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

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

Here  $\langle first\ expr \rangle$ ,  $\langle length\ expr \rangle$  and  $\langle end\ expr \rangle$  both denote algebraic expressions possibly involving named slide references and counters. As integers, they are respectively resolved into  $\langle first \rangle$ ,  $\langle length \rangle$  and  $\langle last \rangle$ .

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

## 5 Support

See <https://github.com/jlaurens/beanoves>.

## 6 Implementation

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

```
1 <@@=bnvs>
```

Reserved namespace: identifiers containing the case insensitive string `beanoves` or the string `bnvs` delimited by two non characters. Not all the variables or functions names used by this package follow this convention, but in that case the global macro level is not polluted.

### 6.1 Package declarations

```
2 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
3 \ProvidesExplPackage
4   {beanoves}
5   {2023/01/07}
6   {1.0}
7   {Named overlay specifications for beamer}
```

### 6.2 logging

Utility message.

```
8 \msg_new:nnn { beanoves } { :n } { #1 }
9 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
10 \cs_new:Npn \__bnvs_warning:n {
11   \msg_warning:nnn { beanoves } { :n }
12 }
13 \cs_new:Npn \__bnvs_error:n {
14   \msg_error:nnn { beanoves } { :n }
```



```

15 }
16 \cs_new:Npn \__bnvs_error:x {
17   \msg_error:nnx { beanoves } { :n }
18 }
19 \cs_new:Npn \__bnvs_fatal:n {
20   \msg_fatal:nnn { beanoves } { :n }
21 }
22 \cs_new:Npn \__bnvs_fatal:x {
23   \msg_fatal:nnx { beanoves } { :n }
24 }

```

### 6.3 Debugging and testing facilities

Typesetting file `beanoves.dtx` creates both `beanoves` and `beanoves-debug` style files. The former is intended for everyday use whereas the latter contains supplemental debugging and testing facilities which are intentionally left undocumented.

### 6.4 Local variables

We make heavy use of local variables and function scopes. Many functions are executed within a  $\text{\TeX}$  group, which ensures no name collision with the caller stack. The number of variables used has not been optimized, nor the  $\text{\TeX}$  groups used. Optimization often goes against readability.

```

25 \tl_new:N \l__bnvs_id_last_tl
26 \tl_set:Nn \l__bnvs_id_last_tl { ?! }
27 \tl_new:N \l__bnvs_a_tl
28 \tl_new:N \l__bnvs_b_tl
29 \tl_new:N \l__bnvs_c_tl
30 \tl_new:N \l__bnvs_id_tl
31 \tl_new:N \l__bnvs_ans_tl
32 \tl_new:N \l__bnvs_name_tl
33 \tl_new:N \l__bnvs_path_tl
34 \tl_new:N \l__bnvs_group_tl
35 \tl_new:N \l__bnvs_query_tl
36 \tl_new:N \l__bnvs_token_tl
37 \tl_new:N \l__bnvs_root_tl
38 \int_new:N \g__bnvs_call_int
39 \int_new:N \l__bnvs_int
40 \seq_new:N \g__bnvs_def_seq
41 \seq_new:N \l__bnvs_a_seq
42 \seq_new:N \l__bnvs_b_seq
43 \seq_new:N \l__bnvs_ans_seq
44 \seq_new:N \l__bnvs_match_seq
45 \seq_new:N \l__bnvs_split_seq
46 \seq_new:N \l__bnvs_path_seq
47 \seq_new:N \l__bnvs_query_seq
48 \seq_new:N \l__bnvs_token_seq
49 \bool_new:N \l__bnvs_in_frame_bool
50 \bool_new:N \l__bnvs_parse_bool
51 \bool_set_false:N \l__bnvs_in_frame_bool

```

## 6.5 Infinite loop management

Unending recursivity is managed here.

`\g__bnvs_call_int` Some functions calls, as well as some loop bodies, decrement this counter. When this counter reaches 0, an error is raised or a computation is aborted.

(End definition for `\g__bnvs_call_int`.)

```
52 \int_const:Nn \c__bnvs_max_call_int { 2048 }
```

---

`\__bnvs_call_greset:` `\__bnvs_call_greset:`

---

Reset globally the call stack counter to its maximum value.

```
53 \cs_set:Npn \__bnvs_call_greset: {
54   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
55 }
```

---

`\__bnvs_call:TF` `\__bnvs_call_do:TF` `{\ true code }` `{\ false code }`

---

Decrement the `\g__bnvs_call_int` counter globally and execute `\ true code` if we have not reached 0, `\ false code` otherwise.

```
56 \prg_new_conditional:Npnn \__bnvs_call: { T, F, TF } {
57   \int_gdecr:N \g__bnvs_call_int
58   \int_compare:nNnTF \g__bnvs_call_int > 0 {
59     \prg_return_true:
60   } {
61     \prg_return_false:
62   }
63 }
```

## 6.6 Overlay specification

### 6.6.1 In slide range definitions

`\g__bnvs_prop` `\key-⟨value⟩` property list to store the named overlay sets. The basic keys are, assuming `\id!⟨name⟩` is a fully qualified overlay set name,

`\id!⟨name⟩/A` for the first index

`\id!⟨name⟩/L` for the length when provided

`\id!⟨name⟩/Z` for the last index when provided

`\id!⟨name⟩/V` for the counter value, when used

`\id!⟨name⟩/n` for the implicit index counter, when used.

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

`\id!⟨name⟩//A` for the cached static value of the first index

`\id!⟨name⟩//Z` for the cached static value of the last index

`\id!⟨name⟩//L` for the cached static value of the length

$\langle id \rangle! \langle name \rangle // P$  for the cached static value of the previous index

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

$\langle id \rangle! \langle name \rangle // V$  for the real counter value

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

<sup>64</sup> `\prop_new:N \g__bnvs_prop`

(End definition for `\g__bnvs_prop`.)

## 6.7 Basic functions

---

<code>\__bnvs_gput:nnn</code>	<code>\__bnvs_gput:nnn {&lt;subkey&gt;} {&lt;key&gt;} {&lt;value&gt;}</code>
<code>\__bnvs_gput:nnV</code>	<code>\__bnvs_gprovide:nnn {&lt;subkey&gt;} {&lt;key&gt;} {&lt;value&gt;}</code>
<code>\__bnvs_gprovide:nnn</code>	<code>\__bnvs_item:nn {&lt;subkey&gt;} {&lt;key&gt;}</code>
<code>\__bnvs_gprovide:nVn</code>	<code>\__bnvs_get:nnN {&lt;subkey&gt;} {&lt;key&gt;} &lt;tl variable&gt;</code>
<code>\__bnvs_item:nn</code>	<code>\__bnvs_gremove:nn {&lt;subkey&gt;} {&lt;key&gt;}</code>
<code>\__bnvs_gremove:nn</code>	<code>\__bnvs_clear:n {&lt;key&gt;}</code>
<code>\__bnvs_gremove:nV</code>	<code>\__bnvs_clear:</code>
<code>\__bnvs_gclear:n</code>	
<code>\__bnvs_gclear:</code>	

---

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L<sup>A</sup>T<sub>E</sub>X3 eponym functions, except `\__bnvs_gprovide:nn` which meaning is straightforward. The key argument is  $\langle key \rangle / \langle subkey \rangle$ .

```

65 \cs_new:Npn \__bnvs_gput:nnn #1 #2 {
66   \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
67 }
68 \cs_new:Npn \__bnvs_gprovide:nnn #1 #2 #3 {
69   \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
70     \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
71   }
72 }
73 \cs_new:Npn \__bnvs_item:nn #1 #2 {
74   \prop_item:Nn \g__bnvs_prop { #2 / #1 }
75 }
76 \cs_new:Npn \__bnvs_gremove:nn #1 #2 {
77   \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
78 }
79 \cs_new:Npn \__bnvs_gclear:n #1 {
80   \clist_map_inline:nn { A, L, Z, V, n } {
81     \__bnvs_gremove:nn { ##1 } { #1 }
82   }
83   \__bnvs_gclear_cache:n { #1 }
84 }
85 \cs_new:Npn \__bnvs_gclear: {
86   \prop_gclear:N \g__bnvs_prop
87 }
88 \cs_generate_variant:Nn \__bnvs_gput:nnn { nnV }
89 \cs_generate_variant:Nn \__bnvs_gremove:nn { nV }

```

---

<code>\__bnvs_if_in_p:nn *</code>	<code>\__bnvs_if_in_p:nn {&lt;subkey&gt;} {&lt;key&gt;}</code>
<code>\__bnvs_if_in_p:nV *</code>	<code>\__bnvs_if_in:nnTF {&lt;subkey&gt;} {&lt;key&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\__bnvs_if_in:nnTF *</code>	
<code>\__bnvs_if_in:nVTF *</code>	

---

Convenient shortcuts to test for the existence of  $\langle key \rangle / \langle subkey \rangle$ , it makes the code more concise and readable.

```

90 \prg_new_conditional:Npnn \__bnvs_if_in:nn #1 #2 { p, T, F, TF } {
91   \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
92     \prg_return_true:
93   } {
94     \prg_return_false:
95   }
96 }
97 \prg_generate_conditional_variant:Nnn
98   \__bnvs_if_in:nn {nV} { p, T, F, TF }

```

---

\\_\_bnvs\_get:nnNTF    \\_\_bnvs\_get:nnNTF {<subkey>} {<key>} <tl variable> {<true code>} {<false 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.

```

99 \prg_new_conditional:Npnn \__bnvs_get:nnN #1 #2 #3 { p, T, F, TF } {
100   \prop_get:NnNTF \g__bnvs_prop { #2 / #1 } #3 {
101     \prg_return_true:
102   } {
103     \prg_return_false:
104   }
105 }
106 \prg_generate_conditional_variant:Nnn
107   \__bnvs_get:nnN {nV} { p, T, F, TF }

```

## 6.8 Functions with cache

---

<code>\__bnvs_gput_cache:nnn</code>	<code>\__bnvs_gput_cache:nnn {&lt;subkey&gt;} {&lt;key&gt;} {&lt;value&gt;}</code>
<code>\__bnvs_gput_cache:(nnV nVn)</code>	<code>\__bnvs_item_cache:nn {&lt;subkey&gt;} {&lt;key&gt;}</code>
<code>\__bnvs_item_cache:nn</code>	<code>\__bnvs_gremove_cache:nn {&lt;subkey&gt;} {&lt;key&gt;}</code>
<code>\__bnvs_gremove_cache:nn</code>	<code>\__bnvs_clear_cache:n {&lt;key&gt;}</code>
<code>\__bnvs_gremove_cache:nV</code>	
<code>\__bnvs_gclear_cache:n</code>	
<code>\__bnvs_v_gclear:</code>	

---

Wrapper over the functions above for  $\langle key \rangle / \langle subkey \rangle$ .

```

108 \cs_new:Npn \__bnvs_gput_cache:nnn #1 {
109   \__bnvs_gput:nnn { / #1 }
110 }
111 \cs_new:Npn \__bnvs_item_cache:nn #1 #2 {
112   \prop_item:Nn \g__bnvs_prop { #2 / / #1 }
113 }
114 \cs_new:Npn \__bnvs_gremove_cache:nn #1 {
115   \__bnvs_gremove:nn { / #1 }
116 }
117 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
118   \clist_map_inline:nn { {}, A, L, Z, P, N, V } {
119     \__bnvs_gremove_cache:nn { ##1 } { #1 }
120   }
121 }
122 \cs_new:Npn \__bnvs_v_gclear: {
123   \__bnvs_group_begin:
124   \seq_clear:N \l__bnvs_a_seq
125   \prop_map_inline:Nn \g__bnvs_prop {
126     \regex_match:nnT { //V$ } { ##1 } {
127       \seq_put_right:Nn \l__bnvs_a_seq { ##1 }
128     }
129   }
130   \seq_map_inline:Nn \l__bnvs_a_seq {
131     \prop_gremove:Nn \g__bnvs_prop { ##1 }
132   }
133   \seq_clear:N \l__bnvs_a_seq
134   \prop_map_inline:Nn \g__bnvs_prop {
135     \regex_match:nnT { /V$ } { ##1 } {
136       \seq_put_right:Nn \l__bnvs_a_seq { ##1 }
137     }
138   }
139   \seq_map_inline:Nn \l__bnvs_a_seq {
140     \prop_get:NnNT \g__bnvs_prop { ##1 } \l__bnvs_a_tl {
141       \quark_if_no_value:NTF \l__bnvs_a_tl {
142         \prop_put:Nnn \g__bnvs_prop { ##1 } { \q_nil }
143       }
144     }
145   }
146   \__bnvs_group_end:
147 }
148 \cs_generate_variant:Nn \__bnvs_gremove_cache:nn { nV }
149 \cs_generate_variant:Nn \__bnvs_gput_cache:nnn { nVn, nnV }

```

---

<code>\_bnvs_if_in_cache_p:nn *</code> <code>\_bnvs_if_in_cache_p:nV *</code> <code>\_bnvs_if_in_cache:nnTF *</code> <code>\_bnvs_if_in_cache:nVTF *</code>	<code>\_bnvs_if_in_cache_p:n {⟨subkey⟩} {⟨key⟩}</code> <code>\_bnvs_if_in_cache:nTF {⟨subkey⟩} {⟨key⟩} {⟨true code⟩} {⟨false code⟩}</code> Convenient shortcuts to test for the existence of $\langle subkey \rangle / \langle key \rangle$ , it makes the code more concise and readable.
--	--

---

```

150 \prg_new_conditional:Npnn \_bnvs_if_in_cache:nn #1 #2 { p, T, F, TF } {
151   \_bnvs_if_in:nnTF { / #1 } { #2 } {
152     \prg_return_true:
153   } {
154     \prg_return_false:
155   }
156 }
157 \prg_generate_conditional_variant:Nnn
158   \_bnvs_if_in_cache:nn {nV} { p, T, F, TF }

```

---

<code>\_bnvs_get_cache:nnNTF</code>	<code>\_bnvs_get_cache:nnNTF {⟨subkey⟩} {⟨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  $\langle true code \rangle$  when the item is found,  $\langle false code \rangle$  otherwise. In the latter case, the content of the  $\langle tl variable \rangle$  is undefined. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```

159 \prg_new_conditional:Npnn \_bnvs_get_cache:nnN #1 #2 #3 { p, T, F, TF } {
160   \_bnvs_get:nnNTF { / #1 } { #2 } #3 {
161     \prg_return_true:
162   } {
163     \prg_return_false:
164   }
165 }
166 \prg_generate_conditional_variant:Nnn
167   \_bnvs_get_cache:nnN {nV} { p, T, F, TF }

```

### 6.8.1 Implicit index counter

The implicit index counter is local to the current frame. When used for the first time, it defaults to 1.

`\g__bnvs_n_prop`  $\langle key \rangle$ – $\langle value \rangle$  property list to store the named slide lists. The keys are  $\langle id \rangle!$  $\langle name \rangle$ .

```

168 \prop_new:N \g__bnvs_n_prop

```

(End definition for `\g__bnvs_n_prop`.)

---

<code>\_bnvs_n_gput:nn</code> <code>\_bnvs_n_gput:(nV Vn)</code> <code>\_bnvs_n_item:n</code> <code>\_bnvs_n_gremove:n</code> <code>\_bnvs_n_gclear:</code>	<code>\_bnvs_n_gput:nn {⟨key⟩} {⟨value⟩}</code> <code>\_bnvs_n_item:n {⟨key⟩}</code> <code>\_bnvs_n_gremove:n {⟨key⟩}</code> <code>\_bnvs_n_gclear:</code>
---	---

---

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L<sup>A</sup>T<sub>E</sub>X3 eponym functions.

```

169 \cs_new:Npn \_bnvs_n_gput:nn {
170   \prop_gput:Nnn \g__bnvs_n_prop
171 }

```

```

172 \cs_new:Npn \__bnvs_n_item:n #1 {
173   \prop_item:Nn \g__bnvs_n_prop { #1 }
174 }
175 \cs_new:Npn \__bnvs_n_gremove:n {
176   \prop_gremove:Nn \g__bnvs_n_prop
177 }
178 \cs_new:Npn \__bnvs_n_gclear: {
179   \prop_gclear:N \g__bnvs_n_prop
180 }

```

---

\\_\_bnvs\_n\_get:nNTF \\_\_bnvs\_n\_get:nNTF {<key>} <tl variable> {(true code)} {(false 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.

```

181 \prg_new_conditional:Npnn \__bnvs_n_get:nN #1 #2 { T, F, TF } {
182   \prop_get:NnNTF \g__bnvs_n_prop { #1 } #2 {
183     \prg_return_true:
184   } {
185     \prg_return_false:
186   }
187 }
188

```

## 6.8.2 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.

```

189 \regex_const:Nn \c__bnvs_name_regex {
190   [[:alpha:]]_ [[:alnum:]]_*
191 }

```

(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.

```

192 \regex_const:Nn \c__bnvs_id_regex {
193   (?: \ur{c__bnvs_name_regex} | [?] )? !
194 }

```

(End definition for \c\_\_bnvs\_id\_regex.)

\c\_\_bnvs\_path\_regex A sequence of .<positive integer> items representing a path.

```

195 \regex_const:Nn \c__bnvs_path_regex {
196   (?: \. \ur{c__bnvs_name_regex} | \. [-+]? \d+ )*
197 }

```

(End definition for \c\_\_bnvs\_path\_regex.)

\c\_\_bnvs\_A\_key\_Z\_regex A key is the name of an overlay set possibly followed by a dotted path. Matches the whole string.

(End definition for `\c__bnvs_A_key_Z_regex`.)

```
198 \regex_const:Nn \c__bnvs_A_key_Z_regex {
```

- 1: The range name including the slide  $\langle id \rangle$  and question mark if any
- 2: slide  $\langle id \rangle$  including the question mark

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

- 3: the path, if any.

```
200      ( \ur{c__bnvs_path_regex} ) \Z
201    }
```

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

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

(End definition for `\c__bnvs_colons_regex`.)

`\c__bnvs_split_regex` Used to parse slide list overlay specifications in queries. Next are the 7 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.

```
203 \regex_const:Nn \c__bnvs_split_regex {
204   \s* ( ? :
```

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

```
205   \+\+
```

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

```
206   ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_name_regex} )
```

- 3: optionally followed by a dotted path

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

- 4:  $\langle name \rangle$  of a slide range
- 5:  $\langle id \rangle$  of a slide range including the exclamation mark

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

- 6: optionally followed by a dotted path

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

We continue with other expressions

- 7: the  $\langle ++n \rangle$  attribute

```
210   (?: \.(\+)\+n
```

---

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



- 8: 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.

```
211 | \s* \+= \s* ( \S+ )
```

- 9: the post increment

```
212 | (\+)\+
```

```
213 )?
```

```
214 ) \s*
```

```
215 }
```

(End definition for `\c__bnvs_split_regex`.)

### 6.8.3 beamer.cls interface

Work in progress.

```
216 \RequirePackage{keyval}
217 \define@key{beamerframe}{beanoves-id}[]{}
218 \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }
219 }
220 \AddToHook{env/beamer@frameslide/before}{
221   \__bnvs_n_gclear:
222   \__bnvs_v_gclear:
223   \bool_set_true:N \l__bnvs_in_frame_bool
224 }
225 \AddToHook{env/beamer@frameslide/after}{
226   \bool_set_false:N \l__bnvs_in_frame_bool
227 }
```

### 6.8.4 Defining named slide ranges

---

```
\__bnvs_parse:nn \__bnvs_parse:nn {<key>} {<definition>}
```

---

Auxiliary function called within a group. `<key>` is the overlay reference key, including eventually a dotted path and a frame identifier, `<definition>` is the corresponding definition.

```
\l__bnvs_match_seq Local storage for the match result.
```

(End definition for `\l__bnvs_match_seq`.)

---

```
\__bnvs_range:nnnn \__bnvs_range:nnnn {<key>} {<first>} {<length>} {<last>}
```

---

Auxiliary function called within a group. Setup the model to define a range.

```
228 \cs_new:Npn \__bnvs_range:nnnn #1 {
```

```

229 \bool_if:NTF \l__bnvs_parse_bool {
230   \__bnvs_n_gremove:n { #1 }
231   \__bnvs_gclear:n { #1 }
232   \__bnvs_do_range:nnnn { #1 }
233 } {
234   \__bnvs_if_in:nnTF A { #1 } {
235     \use_none:nnn
236   } {
237     \__bnvs_if_in:nnTF L { #1 } {
238       \use_none:nnn
239     } {
240       \__bnvs_if_in:nnTF Z { #1 } {
241         \use_none:nnn
242       } {
243         \__bnvs_do_range:nnnn { #1 }
244       }
245     }
246   }
247 }
248 }
249 \cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
250 \cs_new:Npn \__bnvs_do_range:nnnn #1 #2 #3 #4 {
251   \tl_if_empty:nTF { #3 } {
252     \tl_if_empty:nTF { #2 } {
253       \tl_if_empty:nTF { #4 } {
254         \__bnvs_error:n { Not~a~range::~~#1 }
255       } {
256         \__bnvs_gput:nnn Z { #1 } { #4 }
257         \__bnvs_gput:nnn V { #1 } { \q_nil }
258       }
259     } {
260       \__bnvs_gput:nnn A { #1 } { #2 }
261       \__bnvs_gput:nnn V { #1 } { \q_nil }
262       \tl_if_empty:nF { #4 } {
263         \__bnvs_gput:nnn Z { #1 } { #4 }
264         \__bnvs_gput:nnn L { #1 } { \q_nil }
265       }
266     }
267   } {
268     \tl_if_empty:nTF { #2 } {
269       \__bnvs_gput:nnn L { #1 } { #3 }
270       \tl_if_empty:nF { #4 } {
271         \__bnvs_gput:nnn Z { #1 } { #4 }
272         \__bnvs_gput:nnn A { #1 } { \q_nil }
273         \__bnvs_gput:nnn V { #1 } { \q_nil }
274       }
275     } {
276       \__bnvs_gput:nnn A { #1 } { #2 }
277       \__bnvs_gput:nnn L { #1 } { #3 }
278       \__bnvs_gput:nnn Z { #1 } { \q_nil }
279       \__bnvs_gput:nnn V { #1 } { \q_nil }
280     }
281   }
282 }

```

---

---

`\__bnvs_parse:n`

`\__bnvs_parse:n {<key>}`

A key with no value has been parsed by `\keyval_parse`.

```
283 \cs_new:Npn \__bnvs_parse:n #1 {
284   \peek_catcode_ignore_spaces:NTF \c_group_begin_token {
285     \tl_if_empty:NTF \l__bnvs_root_tl {
286       \__bnvs_error:n { Unexpected~list~at~top~level. }
287     }
288     \__bnvs_group_begin:
289     \int_incr:N \l__bnvs_int
290     \tl_set:Nx \l__bnvs_root_tl { \int_use:N \l__bnvs_int . }
291     \cs_set:Npn \bnvs:nw ####1 ####2 \s_stop {
292       \regex_match:nnT { \S* } { ####2 } {
293         \__bnvs_error:n { Unexpected~####2 }
294       }
295       \keyval_parse:nnn {
296         \__bnvs_parse:n
297       } {
298         \__bnvs_parse:nn
299       } { ####1 }
300       \__bnvs_group_end:
301     }
302     \bnvs:nw
303   } {
304     \tl_if_empty:NTF \l__bnvs_root_tl {
305       \__bnvs_id_name_set:nNNTF { #1 } \l__bnvs_id_tl \l__bnvs_name_tl {
306         \__bnvs_parse_record:V \l__bnvs_name_tl
307       } {
308         \__bnvs_error:n { Unexpected~key:~#1 }
309       }
310     } {
311       \int_incr:N \l__bnvs_int
312       \__bnvs_parse_record:xn {
313         \l__bnvs_root_tl . \int_use:N \l__bnvs_int
314       } { #1 }
315     }
316     \use_none_delimit_by_s_stop:w
317   }
318   #1 \s_stop
319 }
```

---

---

`\__bnvs_parse_range:nNNNTF`

`\__bnvs_parse_range:nNNN {<input>} <first tl> <length tl> <last tl> {<true code>} {<false code>}`

Parse `<input>` as a range according to `\c__bnvs_colons_regex`.

```
320 \exp_args_generate:n { VVV }
321 \prg_new_conditional:Npnn \__bnvs_range_set:NNNn #1 #2 #3 #4 { T, F, TF } {
322   \__bnvs_group_begin:
```

This is not a list.

```
323   \tl_clear:N \l__bnvs_a_tl
324   \tl_clear:N \l__bnvs_b_tl
325   \tl_clear:N \l__bnvs_c_tl
```

```

326 \regex_split:NnNTF \c__bnvs_colons_regex { #4 } \l__bnvs_split_seq {
327 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_a_tl {

```

\l\_\_bnvs\_a\_tl may contain the *⟨start⟩*.

```

328 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_b_tl {
329 \tl_if_empty:NNTF \l__bnvs_b_tl {

```

This is a one colon range.

```

330 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl

```

\l\_\_bnvs\_b\_tl may contain the *⟨length⟩*.

```

331 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_c_tl {
332 \tl_if_empty:NNTF \l__bnvs_c_tl {

```

A :: was expected:

```

333 \__bnvs_error:n { Invalid~range~expression(1):~#4 }
334 } {
335 \int_compare:nNnT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
336 \__bnvs_error:n { Invalid~range~expression(2):~#4 }
337 }
338 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl

```

\l\_\_bnvs\_c\_tl may contain the *⟨end⟩*.

```

339 \seq_if_empty:NF \l__bnvs_split_seq {
340 \__bnvs_error:n { Invalid~range~expression(3):~#4 }
341 }
342 }
343 }
344 } {

```

This is a two colon range.

```

345 \int_compare:nNnT { \tl_count:N \l__bnvs_b_tl } > { 1 } {
346 \__bnvs_error:n { Invalid~range~expression(4):~#4 }
347 }
348 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl

```

\l\_\_bnvs\_c\_tl contains the *⟨end⟩*.

```

349 \seq_pop_left:NNTF \l__bnvs_split_seq \l__bnvs_b_tl {
350 \tl_if_empty:NNTF \l__bnvs_b_tl {
351 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl

```

\l\_b\_tl may contain the *⟨length⟩*.

```

352 \seq_if_empty:NF \l__bnvs_split_seq {
353 \__bnvs_error:n { Invalid~range~expression(5):~#4 }
354 }
355 } {
356 \__bnvs_error:n { Invalid~range~expression(6):~#4 }
357 }
358 } {
359 \tl_clear:N \l__bnvs_b_tl
360 }
361 }
362 }
363 }

```

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.

```

364     \bool_if:nF {
365         \tl_if_empty_p:N \l__bnvs_a_tl
366         || \tl_if_empty_p:N \l__bnvs_b_tl
367         || \tl_if_empty_p:N \l__bnvs_c_tl
368     } {
369         \__bnvs_error:n { Invalid~range-expression(7):~#3 }
370     }
371     \cs_set:Npn \:nnn ##1 ##2 ##3 {
372         \__bnvs_group_end:
373         \tl_set:Nn #1 { ##1 }
374         \tl_set:Nn #2 { ##2 }
375         \tl_set:Nn #3 { ##3 }
376     }
377     \exp_args:NVVV \:nnn \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
378     \prg_return_true:
379 } {
380     \__bnvs_group_end:
381     \prg_return_false:
382 }
383 }

```

---

```

\__bnvs_parse_record:n
\__bnvs_parse_record:nn

```

---

```

\__bnvs_parse_record:n {<full name>}
\__bnvs_parse_record:nn {<full name>} {<value>}

```

Auxiliary function for `\__bnvs_parse:n` and `\__bnvs_parse:nn`.

```

384 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
385 \cs_new:Npn \__bnvs_parse_record:n #1 {

```

This is not a list.

```

386     \bool_if:NTF \l__bnvs_parse_bool {
387         \__bnvs_gclear:n { #1 }
388         \__bnvs_gput:nnn V { #1 } { 1 }
389     } {
390         \__bnvs_gprovide:nnn V { #1 } { 1 }
391     }
392 }
393 \cs_generate_variant:Nn \__bnvs_parse_record:n { V }
394 \cs_new:Npn \__bnvs_parse_record:nn #1 #2 {

```

This is not a list.

```

395     \__bnvs_range_set:NNNnTF \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl { #2 } {
396         \__bnvs_range:nVVV { #1 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
397     } {
398         \bool_if:NTF \l__bnvs_parse_bool {
399             \__bnvs_gclear:n { #1 }
400             \__bnvs_gput:nnn V { #1 } { #2 }
401         } {
402             \__bnvs_gprovide:nnn V { #1 } { #2 }
403         }
404     }
405 }
406 \cs_generate_variant:Nn \__bnvs_parse_record:nn { xn, Vn }

```

---

```

__bnvs_id_name_set:nNNTF \__bnvs_id_name_set:nNNTF {<key>} <id tl var> <full name tl var> {< true code>} {<
false code>}

```

---

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

```

407 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
408   \__bnvs_group_begin:
409   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
410     #1
411   } \l__bnvs_match_seq {
412     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
413     \tl_if_empty:NTF #2 {
414       \exp_args:NNNx
415       \__bnvs_group_end:
416       \tl_set:Nn #3 { \l__bnvs_id_last_tl #1 }
417       \tl_set_eq:NN #2 \l__bnvs_id_last_tl
418     } {
419       \cs_set:Npn \:n ##1 {
420         \__bnvs_group_end:
421         \tl_set:Nn #2 { ##1 }
422         \tl_set:Nn \l__bnvs_id_last_tl { ##1 }
423       }
424       \exp_args:NV
425       \:n #2
426       \tl_set:Nn #3 { #1 }
427     }
428   \prg_return_true:
429 } {
430   \__bnvs_group_end:
431   \prg_return_false:
432 }
433 }

434 \cs_new:Npn \__bnvs_parse:nn #1 #2 {
435   \__bnvs_group_begin:
436   \tl_set:Nn \l__bnvs_a_tl { #1 }
437   \tl_put_left:NV \l__bnvs_a_tl \l__bnvs_root_tl
438   \exp_args:NV
439   \__bnvs_id_name_set:nNNTF \l__bnvs_a_tl \l__bnvs_id_tl \l__bnvs_name_tl {
440     \regex_match:nnTF { \S } { #2 } {
441       \peek_catcode_ignore_spaces:NTF \c_group_begin_token {

```

This is a comma separated list, go recursive.

```

442   \__bnvs_group_begin:
443   \tl_set:NV \l__bnvs_root_tl \l__bnvs_name_tl
444   \int_set:Nn \l__bnvs_int { 0 }
445   \cs_set:Npn \bnvs:nn ##1 ##2 \s_stop {
446     \regex_match:nnT { \S } { ##2 } {
447       \__bnvs_error:n { Unexpected~value~#2 }
448     }
449     \keyval_parse:nnn {
450       \__bnvs_parse:n
451     } {

```

```

452         \__bnvs_parse:nn
453     } { ##1 }
454     \__bnvs_group_end:
455 }
456 \bnvs:nn
457 } {
458     \__bnvs_parse_record:Vn \l__bnvs_name_tl { #2 }
459     \use_none_delimit_by_s_stop:w
460 } #2 \s_stop
461 } {

```

Empty value given: remove the reference.

```

462     \exp_args:NV
463     \__bnvs_gclear:n \l__bnvs_name_tl
464     \exp_args:NV
465     \__bnvs_n_gremove:n \l__bnvs_name_tl
466 }
467 } {
468     \__bnvs_error:n { Invalid~key:~#2 }
469 }

```

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

```

470     \exp_args:NNNV
471     \__bnvs_group_end:
472     \tl_set:Nn \l__bnvs_id_last_tl \l__bnvs_id_last_tl
473 }

474 \cs_new:Npn \__bnvs_parse_prepare:N #1 {
475     \tl_set:Nx #1 #1
476     \bool_set_false:N \l__bnvs_parse_bool
477     \bool_do_until:Nn \l__bnvs_parse_bool {
478         \tl_if_in:NnTF #1 {%---[
479     ]} {
480         \regex_replace_all:nnNF { \[ ([^\]]*) \] } { { { \1 } } } #1 {
481             \bool_set_true:N \l__bnvs_parse_bool
482         }
483     } {
484         \bool_set_true:N \l__bnvs_parse_bool
485     }
486 }
487 \tl_if_in:NnTF #1 {%---[
488 ]} {
489     \__bnvs_error:n { Unbalanced~%---[
490 ]}
491 } {
492     \tl_if_in:NnT #1 { [%---]
493     } {
494         \__bnvs_error:n { Unbalanced~[ %---]
495     }
496 }
497 }
498 }

```

---

`\Beanoves`

`\Beanoves {⟨key--value list⟩}`

The keys are the slide overlay references. When no value is provided, it defaults to 1. On the contrary, `⟨key-value⟩` items are parsed by `\__bnvs_parse:nn`.

```
499 \NewDocumentCommand \Beanoves { sm } {  
500   \tl_if_empty:NTF \@currenvir {
```

We are most certainly in the preamble, record the definitions globally for later use.

```
501     \seq_gput_right:Nn \g__bnvs_def_seq { #2 }  
502   } {  
503     \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, clear everything.

```
504     \__bnvs_gclear:  
505   }  
506   \__bnvs_group_begin:  
507   \tl_clear:N \l__bnvs_root_tl  
508   \int_zero:N \l__bnvs_int  
509   \tl_set:Nn \l__bnvs_a_tl { #2 }  
510   \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, use the global definitions.

```
511     \seq_if_empty:NF \g__bnvs_def_seq {  
512       \tl_put_left:Nx \l__bnvs_a_tl {  
513         \seq_use:Nn \g__bnvs_def_seq , ,  
514       }  
515     }  
516   }  
517   \__bnvs_parse_prepare:N \l__bnvs_a_tl  
518   \IfBooleanTF {#1} {  
519     \bool_set_false:N \l__bnvs_parse_bool  
520   } {  
521     \bool_set_true:N \l__bnvs_parse_bool  
522   }  
523   \exp_args:NnnV  
524   \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn } \l__bnvs_a_tl  
525   \exp_args:NNNV  
526   \__bnvs_group_end:  
527   \tl_set:Nn \l__bnvs_id_last_tl \l__bnvs_id_last_tl  
528   \ignorespaces  
529 }  
530 }
```

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

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



### 6.8.5 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` reads 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.

```

532 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
533 \cs_set:Npn \beamer@frame < #1 > {
534   \__bnvs_group_begin:
535   \tl_clear:N \l__bnvs_ans_tl
536   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
537   \exp_args:NNNV
538   \__bnvs_group_end:
539   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
540 }
541 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
542 \cs_set:Npn \beamer@masterdecode #1 {
543   \__bnvs_group_begin:
544   \tl_clear:N \l__bnvs_ans_tl
545   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
546   \exp_args:NNV
547   \__bnvs_group_end:
548   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
549 }
```

<u>\__bnvs_scan:nNN</u>	<p>\__bnvs_scan:nNN {<i>(named overlay expression)</i>} <i>&lt;eval&gt;</i> <i>&lt;tl variable&gt;</i></p> <p>Scan the <i>&lt;named overlay expression&gt;</i> argument and feed the <i>&lt;tl variable&gt;</i> replacing <i>?(...)</i> instructions by their static counterpart with help from the <i>&lt;eval&gt;</i> function, which is \__bnvs_eval:nN. A group is created to use local variables:</p>
\l__bnvs_ans_tl	<p>The token list that will be appended to <i>&lt;tl variable&gt;</i> on return.</p> <p>(End definition for \l__bnvs_ans_tl.)</p>
\l__bnvs_int	<p>Store the depth level in parenthesis grouping used when finding the proper closing parenthesis balancing the opening parenthesis that follows immediately a question mark in a <i>?(...)</i> instruction.</p> <p>(End definition for \l__bnvs_int.)</p>
\l__bnvs_query_tl	<p>Storage for the overlay query expression to be evaluated.</p> <p>(End definition for \l__bnvs_query_tl.)</p>
\l__bnvs_token_seq	<p>The <i>&lt;overlay expression&gt;</i> is split into the sequence of its tokens.</p> <p>(End definition for \l__bnvs_token_seq.)</p>
\l__bnvs_token_tl	<p>Storage for just one token.</p> <p>(End definition for \l__bnvs_token_tl.)</p>
	<pre> 550 \cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 { 551   \__bnvs_group_begin: 552   \tl_clear:N \l__bnvs_ans_tl 553   \seq_clear:N \l__bnvs_token_seq </pre> <p>Explode the <i>&lt;named overlay expression&gt;</i> into a list of tokens:</p> <pre> 554   \regex_split:nnN {} { #1 } \l__bnvs_token_seq </pre>
<u>\scan_question:</u>	<p>\scan_question:</p> <p>At top level state, scan the tokens of the <i>&lt;named overlay expression&gt;</i> looking for a ‘?’ character.</p> <pre> 555 \cs_set:Npn \scan_question: { 556   \seq_pop_left:NNT \l__bnvs_token_seq \l__bnvs_token_tl { 557     \tl_if_eq:NnTF \l__bnvs_token_tl { ? } { 558       \require_open: 559     } { 560       \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl 561       \scan_question: 562     } 563   } 564 } </pre>
<u>\require_open:</u>	<p>\require_open:</p> <p>We just found a ‘?’, we first gobble tokens until the next ‘(’, whatever they may be. In general, no tokens should be silently ignored.</p> <pre> 565 \cs_set:Npn \require_open: { </pre>

Get next token.

```
566     \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
567         \tl_if_eq:NnTF \l__bnvs_token_tl { ( %}
568     } {
```

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

```
569         \int_set:Nn \l__bnvs_int { 1 }
```

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

```
570         \tl_clear:N \l__bnvs_query_tl
571         \require_close:
572     } {
```

Ignore this token and loop.

```
573         \require_open:
574     }
575 } {
```

End reached but no opening parenthesis found, raise.

```
576     \__bnvs_fatal:x {Missing~'('%---)
577         ~after~a~?:~#1}
578 }
579 }
```

---

\require\_close:

---

\require\_close:

We found a ‘?’(‘, we record the forthcoming content in the \l\_\_bnvs\_query\_tl variable, up to the next balancing ‘)’.

```
580     \cs_set:Npn \require_close: {
```

Get next token.

```
581     \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
582         \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
583     } {
```

We found a ‘(‘, increment the depth and append the token to \l\_\_bnvs\_query\_tl, then scan again for a).

```
584         \int_incr:N \l__bnvs_int
585         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
586         \require_close:
587     } {
```

This is not a ‘(‘.

```
588         \tl_if_eq:NnTF \l__bnvs_token_tl { %(---
589         )
590     } {
```

We found a ‘)’’, we decrement and test the depth.

```
591         \int_decr:N \l__bnvs_int
592         \int_compare:nNnTF \l__bnvs_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 look for the next ?.

```

593         \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
594         \scan_question:
595     } {

```

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

```

596         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
597         \require_close:
598     }
599 } {

```

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

```

600         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
601         \require_close:
602     }
603 }
604 } {

```

Above ends the code for Not a ‘(’ We reached the end of the sequence and the token list with no closing ‘)’. We raise and terminate. As recovery we feed \l\_\_bnvs\_query\_tl with the missing ‘)’.

```

605     \__bnvs_error:x {Missing~%(---
606     `):~#1 }
607     \tl_put_right:Nx \l__bnvs_query_tl {
608         \prg_replicate:nn { \l__bnvs_int } {%(---
609         )}
610     }
611     \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
612 }
613 }

```

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

```

614     \scan_question:
615     \exp_args:NNNV
616     \__bnvs_group_end:
617     \tl_put_right:Nn #3 \l__bnvs_ans_tl
618 }

```

I

### 6.8.6 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_inp:NNNTF \__bnvs_inp: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 recorded key, on return, <name tl var> contains the resolved name, <path seq var> is prepended with new dotted path components, {\true code} is executed, otherwise {\false code} is executed.

```

619 \exp_args_generate:n { VVx }
620 \prg_new_conditional:Npnn \__bnvs_inp:NNN
621   #1 #2 #3 { T, F, TF } {
622   \__bnvs_group_begin:
623   \exp_args:NNV
624   \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

```

625   \exp_args:Nx
626   \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
627   \tl_put_left:NV #2 { #1 }
628   }
629   \exp_args:NNnx
630   \seq_set_split:Nnn \l__bnvs_split_seq . {
631   \seq_item:Nn \l__bnvs_match_seq 4
632   }
633   \seq_remove_all:Nn \l__bnvs_split_seq { }
634   \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
635   \seq_if_empty:NTF \l__bnvs_split_seq {

```

No new integer path component is added.

```

636   \cs_set:Npn \:nn ##1 ##2 {
637   \__bnvs_group_end:
638   \tl_set:Nn #1 { ##1 }
639   \tl_set:Nn #2 { ##2 }
640   }
641   \exp_args:NVV \:nn #1 #2
642   } {

```

Some new dotted path components are added.

```

643   \cs_set:Npn \:nnn ##1 ##2 ##3 {
644   \__bnvs_group_end:
645   \tl_set:Nn #1 { ##1 }
646   \tl_set:Nn #2 { ##2 }
647   \seq_set_split:Nnn #3 . { ##3 }
648   \seq_remove_all:Nn #3 { }
649   }
650   \exp_args:NVVx
651   \:nnn #1 #2 {
652   \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
653   }
654   }
655   \prg_return_true:
656 } {
657   \__bnvs_group_end:
658   \prg_return_false:
659 }
660 }

```

---

```

\__bnvs_resolve_n:NNN $\overline{TF}$  \__bnvs_resolve_x:TF {\true code} {\false code}
\__bnvs_resolve_n:TFF $\overline{TF}$  \__bnvs_resolve_x:NNNTF <id tl var> <name tl var> <path seq var> {\true code}
\__bnvs_resolve_x:NNN $\overline{TF}$  {\false code}
\__bnvs_resolve_x:TFF $\overline{TF}$  \__bnvs_resolve_n:TF {\true code} {\false code}
\__bnvs_resolve_n:NNNTF <id tl var> <name tl var> <path seq var> {\true code}
\__bnvs_resolve_x:NNNTF {\false code}

```

---

When too many nested calls occurred,  $\{\langle false\ code\rangle\}$  is executed directly otherwise  $\{\langle true\ code\rangle\}$  will be executed once resolution has occurred. The  $\langle id\ tl\ var\rangle$ ,  $\langle name\ tl\ var\rangle$  and  $\langle path\ seq\ var\rangle$  are meant to contain proper information on input and on output as well. On input,  $\{\langle id\ tl\ var\rangle\}$  contains a frame id,  $\{\langle name\ tl\ var\rangle\}$  contains a slide 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 level of a named one slide specification  $\langle qualified\ name\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ , we replace the shortest  $\langle qualified\ name\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_k\rangle$  where  $0\leq k\leq n$  by its definition  $\langle qualified\ name'\rangle.\langle j_1\rangle...\langle j_p\rangle$  if any. The  $\backslash\_bnvs\_resolve:NNNTF$  function uses this one level resolution as many times as possible, but no more than a predefined limit to catch circular reference that would lead to an infinite loop.

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  $\backslash\_bnvs\_id\_tl$  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.

In the  $\_n$  variant, the resolution is driven only when there is a non empty dotted path.

In the  $\_x$  variant, the resolution is driven one step further: if  $\langle path\ seq\ var\rangle$  is empty,  $\langle name\ tl\ var\rangle$  can contain anything.

```

661 \cs_new:Npn \__bnvs_resolve_x:TFF #1 #2 {
662   \__bnvs_resolve_x:NNNTF
663   \l__bnvs_id_tl
664   \l__bnvs_name_tl
665   \l__bnvs_path_seq {
666     \seq_if_empty:NTF \l__bnvs_path_seq { #1 } { #2 }
667   }
668 }
669 \prg_new_conditional:Npnn \__bnvs_resolve_x:NNN
670   #1 #2 #3 { T, F, TF } {
671   \__bnvs_group_begin:

```

Local variables:

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

- \l\_\_bnvs\_b\_seq contains the index path components to be resolved.

```

672 \seq_set_eq:NN \l__bnvs_a_seq #3
673 \seq_clear:N \l__bnvs_b_seq
674 \cs_set:Npn \loop: {
675   \__bnvs_call:TF {
676     \tl_set_eq:NN \l__bnvs_a_tl #2
677     \seq_if_empty:NTF \l__bnvs_a_seq {
678       \__bnvs_get:nVNTF L \l__bnvs_a_tl \l__bnvs_b_tl {
679         \cs_set:Nn \loop: { \return_true: }
680       } {
681         \resolve:F {

```

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

```

682       \cs_set:Nn \loop: { \return_true: }
683     }
684   }
685 } {
686   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
687   \resolve:F {
688     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
689       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
690     }
691   }
692 }
693 \loop:
694 } {
695   \__bnvs_group_end:
696   \prg_return_false:
697 }
698 }
699 \cs_set:Npn \resolve:F ##1 {
700   \__bnvs_get:nVNTF A \l__bnvs_a_tl \l__bnvs_b_tl {
701     \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
702       \tl_set_eq:NN #2 \l__bnvs_b_tl
703       \seq_set_eq:NN #3 \l__bnvs_b_seq
704       \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
705       \seq_clear:N \l__bnvs_b_seq
706     } {
707       \seq_if_empty:NTF \l__bnvs_b_seq {
708         \tl_set_eq:NN #2 \l__bnvs_b_tl
709         \seq_clear:N #3
710         \seq_clear:N \l__bnvs_a_seq
711       } {
712         ##1
713       }
714     }
715   } {
716     \__bnvs_get:nVNTF V \l__bnvs_a_tl \l__bnvs_b_tl {
717       \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
718         \tl_set_eq:NN #2 \l__bnvs_b_tl
719         \seq_set_eq:NN #3 \l__bnvs_b_seq
720         \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
721         \seq_clear:N \l__bnvs_b_seq

```

```

722     } {
723         \seq_if_empty:NTF \l__bnvs_b_seq {
724             \tl_set_eq:NN #2 \l__bnvs_b_tl
725             \seq_clear:N #3
726             \seq_clear:N \l__bnvs_a_seq
727         } {
728             ##1
729         }
730     }
731 } { ##1 }
732 }
733 }
734 \cs_set:Npn \return_true: {
735     \seq_pop_left:NNTF #3 \l__bnvs_a_tl {
736         \seq_if_empty:NTF #3 {
737             \tl_clear:N \l__bnvs_b_tl
738             \__bnvs_can_index:VTF #2 {
739                 \__bnvs_if_index:VVNTF #2 \l__bnvs_a_tl \l__bnvs_b_tl {
740                     \tl_set:NV #2 \l__bnvs_b_tl
741                 } {
742                     \tl_set:NV #2 \l__bnvs_a_tl
743                 }
744             } {
745                 \tl_set:NV #2 \l__bnvs_a_tl
746             }
747         } {
748             \__bnvs_error:x { Path~too~long:~#2.\l__bnvs_a_tl
749                 .\seq_use:Nn\l__bnvs_path_seq .}
750         }
751     } {
752         \tl_clear:N \l__bnvs_b_tl
753         \__bnvs_raw_value:VNT #2 \l__bnvs_b_tl {
754             \tl_set:NV #2 \l__bnvs_b_tl
755         }
756     }
757     \cs_set:Npn \:nnn ####1 ####2 ####3 {
758         \__bnvs_group_end:
759         \tl_set:Nn #1 { ####1 }
760         \tl_set:Nn #2 { ####2 }
761         \seq_set_split:Nnn #3 . { ####3 }
762         \seq_remove_all:Nn #3 { }
763     }
764     \exp_args:NVVx
765     \:nnn #1 #2 {
766         \seq_use:Nn #3 .
767     }
768     \prg_return_true:
769 }
770 \loop:
771 }
772 \cs_new:Npn \__bnvs_resolve_n:TFF #1 #2 {
773     \__bnvs_resolve_n:NNNTF
774     \l__bnvs_id_tl
775     \l__bnvs_name_tl

```



```

776 \l__bnvs_path_seq {
777   \seq_if_empty:NTF \l__bnvs_path_seq { #1 } { #2 }
778 }
779 }
780 \prg_new_conditional:Npnn \__bnvs_resolve_n: { T, F, TF } {
781   \__bnvs_resolve_n:NNNTF
782   \l__bnvs_name_tl
783   \l__bnvs_id_tl
784   \l__bnvs_path_seq {
785     \prg_return_true:
786   } {
787     \prg_return_false:
788   }
789 }
790 \prg_new_conditional:Npnn \__bnvs_resolve_n_old:NNN
791   #1 #2 #3 { T, F, TF } {
792   \__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.

```

793 \seq_set_eq:NN \l__bnvs_a_seq #3
794 \seq_clear:N \l__bnvs_b_seq
795 \cs_set:Npn \loop: {
796   \__bnvs_call:TF {
797     \tl_set_eq:NN \l__bnvs_a_tl #2
798     \seq_if_empty:NTF \l__bnvs_a_seq {
799       \__bnvs_get:nVNTF L \l__bnvs_a_tl \l__bnvs_b_tl {
800         \cs_set:Npn \loop: { \return_true: }
801       } {
802         \seq_if_empty:NTF \l__bnvs_b_seq {
803           \cs_set:Npn \loop: { \return_true: }
804         } {
805           \:F {

```

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

```

806       \cs_set:Npn \loop: { \return_true: }
807     }
808   }
809 }
810 } {
811   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
812   \:F {
813     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
814       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
815     }
816   }
817 }
818 \loop:
819 } {

```

```

820     \__bnvs_group_end:
821     \prg_return_false:
822 }
823 }
824 \cs_set:Npn \:F ##1 {
825     \__bnvs_get:nVNTF A \l__bnvs_a_tl \l__bnvs_b_tl {
826         \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
827             \tl_set_eq:NN #2 \l__bnvs_b_tl
828             \seq_set_eq:NN #3 \l__bnvs_b_seq
829             \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
830             \seq_clear:N \l__bnvs_b_seq
831         } { ##1 }
832     } { ##1 }
833 }
834 \cs_set:Npn \return_true: {
835     \cs_set:Npn \:nnn #####1 #####2 #####3 {
836         \__bnvs_group_end:
837         \tl_set:Nn #1 { #####1 }
838         \tl_set:Nn #2 { #####2 }
839         \seq_set_split:Nnn #3 . { #####3 }
840         \seq_remove_all:Nn #3 { }
841     }
842     \exp_args:NVVx
843     \:nnn #1 #2 { \seq_use:Nn #3 . }
844     \prg_return_true:
845 }
846 \loop:
847 }
848 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
849     #1 #2 #3 { T, F, TF } {
850     \__bnvs_group_begin:

```

Local variables:

- \l\_\_bnvs\_a\_tl contains the name with a partial index path currently resolved.
- \l\_\_bnvs\_id\_tl, \l\_\_bnvs\_name\_tl, \l\_\_bnvs\_path\_seq contains the resolution.
- \l\_\_bnvs\_a\_seq contains the dotted path components to be resolved. Initially empty.

```

851 \tl_set_eq:NN \l__bnvs_id_tl #1
852 \tl_set_eq:NN \l__bnvs_name_tl #2
853 \seq_set_eq:NN \l__bnvs_path_seq #3
854 \seq_set_eq:NN \l__bnvs_a_seq #3
855 \seq_clear:N \l__bnvs_b_seq
856 \cs_set:Npn \loop: {
857     \__bnvs_call:TF {
858         \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_name_tl
859         \seq_if_empty:NTF \l__bnvs_a_seq {
860             \seq_if_empty:NTF \l__bnvs_b_seq {
861                 \group_end_return_true:
862             } {
863                 \resolve:nF A {
864                     \resolve:nF V {
865                         \may_loop:

```

```

866         }
867     }
868 }
869 } {
870     \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
871     \resolve:nF A {
872         \resolve:nF V {
873             \may_loop:
874         }
875     }
876 }
877 } {
878     \__bnvs_group_end:
879     \prg_return_false:
880 }
881 }
882 \cs_set:Npn \may_loop: {
883     \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
884         \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
885         \loop:
886     } {
887         \group_end_return_true:
888     }
889 }
890 \cs_set:Npn \resolve:nF ##1 ##2 {
891     \__bnvs_get:nVNTF ##1 \l__bnvs_a_tl \l__bnvs_b_tl {
892         \__bnvs_inp:NNNTF \l__bnvs_id_tl \l__bnvs_b_tl \l__bnvs_b_seq {
893             \tl_set_eq:NN \l__bnvs_name_tl \l__bnvs_b_tl
894             \seq_set_eq:NN \l__bnvs_path_seq \l__bnvs_b_seq
895             \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
896             \seq_clear:N \l__bnvs_b_seq
897             \loop:
898         } {
899             \may_loop:
900         }
901     } {
902         ##2
903     }
904 }
905 \cs_set:Npn \group_end_return_true: {
906     \cs_set:Npn \:nnn #####1 #####2 #####3 {
907         \__bnvs_group_end:
908         \tl_set:Nn #1 { #####1 }
909         \tl_set:Nn #2 { #####2 }
910         \seq_set_split:Nnn #3 . { #####3 }
911         \seq_remove_all:Nn #3 { }
912     }
913     \exp_args:NVVx
914     \:nnn \l__bnvs_id_tl \l__bnvs_name_tl { \seq_use:Nn \l__bnvs_path_seq . }
915     \prg_return_true:
916 }
917 \loop:
918 }

```

---

```

\__bnvs_resolve_n:NNN $\overline{TF}$  \__bnvs_resolve_n:NNNTF  $\langle id\ tl\ var \rangle$   $\langle name\ tl\ var \rangle$   $\langle path\ seq\ var \rangle$   $\{\langle true\ code \rangle\}$ 
\__bnvs_resolve_x:NNN $\overline{TF}$   $\{\langle false\ code \rangle\}$ 
\__bnvs_resolve_x:NNNTF  $\langle id\ tl\ var \rangle$   $\langle name\ tl\ var \rangle$   $\langle path\ seq\ var \rangle$   $\{\langle true\ code \rangle\}$ 
 $\{\langle false\ code \rangle\}$ 

```

---

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 slide 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 level of a named one slide specification  $\langle qualified\ name \rangle.\langle c_1 \rangle.\langle c_2 \rangle...\langle c_n \rangle$ , we replace the shortest  $\langle qualified\ name \rangle.\langle c_1 \rangle.\langle c_2 \rangle...\langle c_k \rangle$  where  $0 \leq k \leq n$  by its definition  $\langle qualified\ name' \rangle.\langle c'_1 \rangle...\langle c'_p \rangle$  if any. The `\__bnvs_resolve:NNNTF` function uses this one level resolution as many times as possible, but no more than a predefined limit to catch circular reference that would lead to an infinite loop.

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 `\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.

NOTA BENE: Implementation details. None of the *tl* variables must be one of `\l__bnvs_a_tl`, `\l__bnvs_b_tl` or `\l__bnvs_c_tl`. None of the *seq* variables must be one of `\l__bnvs_a_seq`, `\l__bnvs_b_seq`.

In the `_x` variant, the resolution is driven one step further: if  $\langle path\ seq\ var \rangle$  is empty,  $\langle name\ tl\ var \rangle$  can contain anything, including an integer for example.

### 6.8.7 Evaluation bricks

We start by helpers.

---

```

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

```

---

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

```

919 \cs_new:Npn \__bnvs_round:nN #1 #2 {
920   \tl_if_empty:nTF { #1 } {
921     \tl_put_right:Nn #2 { 0 }
922   } {
923     \tl_put_right:Nx #2 { \fp_eval:n { round(#1) } }
924   }
925 }
926 \cs_new:Npn \__bnvs_round:N #1 {
927   \tl_if_empty:VTF #1 {
928     \tl_set:Nn #1 { 0 }

```

```

929 } {
930   \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
931 }
932 }

```

---

<code>\__bnvs_group_end_return_true:nnN</code> <code>\__bnvs_group_end_return_false:nn</code>	<code>\__bnvs_group_end_return_true:nnN {&lt;subkey&gt;} {&lt;key&gt;} &lt;tl variable&gt;</code> <code>\__bnvs_group_end_return_false:nn {&lt;subkey&gt;} {&lt;key&gt;}</code>
--	--

---

End a group and calls `\prg_return_true:` or `\prg_return_false:`. Before returning, the first one appends the content of `\l__bnvs_ans_tl` to the `<tl variable>` and cache this content under `<subkey>` whereas the second one cleans the canche for that `<subkey>`.

```

933 \cs_set:Npn \__bnvs_group_end_return_true:nnN #1 #2 #3 {
934   \tl_if_empty:NTF \l__bnvs_ans_tl {
935     \__bnvs_group_end:
936     \__bnvs_gremove_cache:nn { #1 } { #2 }
937     \prg_return_false:
938   } {
939     \__bnvs_round:N \l__bnvs_ans_tl
940     \__bnvs_gput_cache:nnV { #1 } { #2 } \l__bnvs_ans_tl
941     \exp_args:NNNV
942     \__bnvs_group_end:
943     \tl_put_right:Nn #3 \l__bnvs_ans_tl
944     \prg_return_true:
945   }
946 }
947 \cs_set:Npn \__bnvs_group_end_return_false:nn #1 #2 {
948   \__bnvs_group_end:
949   \__bnvs_gremove_cache:nn { #1 } { #2 }
950   \prg_return_false:
951 }

```

---

<code>\__bnvs_raw_first:nNTF</code> <code>\__bnvs_raw_first:(xN VN)TF</code>	<code>\__bnvs_raw_first:nNTF {&lt;name&gt;} &lt;tl variable&gt; {&lt;true code&gt;} {&lt;false code&gt;}</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.

```

952 \prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
953   \__bnvs_group_begin:
954   \__bnvs_get_cache:nnNTF A { #1 } #2 {
955     \exp_args:NNNV
956     \__bnvs_group_end:
957     \tl_put_right:Nn #2 #2
958     \prg_return_true:
959   } {
960     \__bnvs_get:nnNTF A { #1 } \l__bnvs_a_tl {
961       \tl_clear:N \l__bnvs_ans_tl
962       \quark_if_nil:NNTF \l__bnvs_a_tl {
963         \__bnvs_gput:nnn A { #1 } { \q_no_value }

```

The first index must be computed separately from the length and the last index.

```

964 \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
965 \tl_put_right:Nn \l__bnvs_ans_tl { - }
966 \tl_clear:N \l__bnvs_a_tl
967 \__bnvs_raw_length:nNTF { #1 } \l__bnvs_a_tl {
968 \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
969 \tl_put_right:Nn \l__bnvs_ans_tl { + 1 }
970 \__bnvs_group_end_return_true:nnN A { #1 } #2
971 } {
972 \__bnvs_error:n { Unavailable~length~for~#1~(\__bnvs_raw_first:nNTF/2) }
973 \__bnvs_group_end_return_false:nn A { #1 }
974 }
975 } {
976 \__bnvs_error:n { Unavailable~last~for~#1~(\__bnvs_raw_first:nNTF/1) }
977 \__bnvs_group_end_return_false:nn A { #1 }
978 }
979 } {
980 \quark_if_no_value:NTF \l__bnvs_a_tl {
981 \__bnvs_fatal:n {Circular~definition:~#1}
982 } {
983 \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
984 \__bnvs_group_end_return_true:nnN A { #1 } #2
985 } {
986 \__bnvs_group_end_return_false:nn A { #1 }
987 }
988 }
989 }
990 } {
991 \__bnvs_group_end_return_false:nn A { #1 }
992 }
993 }
994 }
995 \prg_generate_conditional_variant:Nnn
996 \__bnvs_raw_first:nN { VN, xN } { T, F, TF }

```

---

\\_\_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.

```

997 \prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
998 \__bnvs_group_begin:
999 \__bnvs_get_cache:nnNTF L { #1 } #2 {
1000 \exp_args:NNNV
1001 \__bnvs_group_end:
1002 \tl_put_right:Nn #2 #2
1003 \prg_return_true:
1004 } {
1005 \__bnvs_get:nnNTF L { #1 } \l__bnvs_a_tl {
1006 \tl_clear:N \l__bnvs_ans_tl
1007 \quark_if_nil:NTF \l__bnvs_a_tl {
1008 \__bnvs_gput:nnn L { #1 } { \q_no_value }

```

The length must be computed separately from the start and the last index.

```

1009     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
1010         \tl_put_right:Nn \l__bnvs_ans_tl { - }
1011         \tl_clear:N \l__bnvs_a_tl
1012         \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
1013             \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1014             \tl_put_right:Nn \l__bnvs_ans_tl { + 1 }
1015             \__bnvs_group_end_return_true:nnN L { #1 } #2
1016         } {
1017             \__bnvs_error:n { Unavailable~first~for~#1~(\__bnvs_raw_length:nNTF/2) }
1018             \__bnvs_group_end_return_false:nn L { #1 }
1019         }
1020     } {
1021         \__bnvs_error:n { Unavailable~last~for~#1~(\__bnvs_raw_length:nNTF/1) }
1022         \__bnvs_group_end_return_false:nn L { #1 }
1023     }
1024 } {
1025     \quark_if_no_value:NTF \l__bnvs_a_tl {
1026         \__bnvs_fatal:n {Circular~definition:~#1}
1027     } {
1028         \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1029             \__bnvs_group_end_return_true:nnN L { #1 } #2
1030         } {
1031             \__bnvs_group_end_return_false:nn L { #1 }
1032         }
1033     }
1034 }
1035 } {
1036     \__bnvs_group_end_return_false:nn L { #1 }
1037 }
1038 }
1039 }
1040 \prg_generate_conditional_variant:Nnn
1041 \__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.

```

1042 \prg_new_conditional:Npnn \__bnvs_raw_last:nN #1 #2 { T, F, TF } {
1043     \__bnvs_group_begin:
1044     \__bnvs_get_cache:nnNTF Z { #1 } #2 {
1045         \exp_args:NNNV
1046         \__bnvs_group_end:
1047         \tl_put_right:Nn #2 #2
1048         \prg_return_true:
1049     } {
1050         \__bnvs_get:nnNTF Z { #1 } \l__bnvs_a_tl {
1051             \tl_clear:N \l__bnvs_ans_tl
1052             \quark_if_nil:NTF \l__bnvs_a_tl {
1053                 \__bnvs_gput:nnn Z { #1 } { \q_no_value }

```

The last index must be computed separately from the start and the length.

```

1054 \tl_clear:N \l__bnvs_a_tl
1055 \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1056 \tl_put_right:Nn \l__bnvs_ans_tl { + }
1057 \tl_clear:N \l__bnvs_b_tl
1058 \__bnvs_raw_length:nNTF { #1 } \l__bnvs_b_tl {
1059 \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_b_tl
1060 \tl_put_right:Nn \l__bnvs_ans_tl { - 1 }
1061 \__bnvs_group_end_return_true:nnN Z { #1 } #2
1062 } {
1063 \__bnvs_error:n { Unavailable~length~for~#1~(\__bnvs_raw_last:nNTF/1) }
1064 \__bnvs_group_end_return_false:nn Z { #1 }
1065 }
1066 } {
1067 \__bnvs_error:n { Unavailable~start~for~#1~(\__bnvs_raw_last:nNTF/1) }
1068 \__bnvs_group_end_return_false:nn Z { #1 }
1069 }
1070 } {
1071 \quark_if_no_value:NTF \l__bnvs_a_tl {
1072 \__bnvs_fatal:n {Circular~definition:~#1}
1073 } {
1074 \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1075 \__bnvs_group_end_return_true:nnN Z { #1 } #2
1076 } {
1077 \__bnvs_group_end_return_false:nn Z { #1 }
1078 }
1079 }
1080 }
1081 } {
1082 \__bnvs_group_end_return_false:nn Z { #1 }
1083 }
1084 }
1085 }
1086 \prg_generate_conditional_variant:Nnn
1087 \__bnvs_raw_last:nN { VN } { T, F, TF }

```

---

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

---

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

```

1088 \prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {
1089 \__bnvs_group_begin:
1090 \tl_clear:N \l__bnvs_a_tl
1091 \tl_clear:N \l__bnvs_b_tl
1092 \tl_clear:N \l__bnvs_ans_tl
1093 \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
1094 \int_compare:nNnT { \l__bnvs_a_tl } < 0 {
1095 \tl_set:Nn \l__bnvs_a_tl { 0 }
1096 }
1097 \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {

```

Limited from above and below.



```

1098     \int_compare:nNtT { \l__bnvs_b_tl } < 0 {
1099         \tl_set:Nn \l__bnvs_b_tl { 0 }
1100     }
1101     \exp_args:NNNx
1102     \__bnvs_group_end:
1103     \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
1104     \prg_return_true:
1105 } {

```

Limited from below.

```

1106     \exp_args:NNNV
1107     \__bnvs_group_end:
1108     \tl_put_right:Nn #2 \l__bnvs_a_tl
1109     \tl_put_right:Nn #2 { - }
1110     \prg_return_true:
1111 }
1112 } {
1113     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {

```

Limited from above.

```

1114     \int_compare:nNtT { \l__bnvs_b_tl } < 0 {
1115         \tl_set:Nn \l__bnvs_b_tl { 0 }
1116     }
1117     \exp_args:NNNx
1118     \__bnvs_group_end:
1119     \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1120     \prg_return_true:
1121 } {
1122     \__bnvs_raw_value:nNTF { #1 } \l__bnvs_b_tl {

```

Unlimited range.

```

1123     \exp_args:NNNx
1124     \__bnvs_group_end:
1125     \tl_put_right:Nn #2 { - }
1126     \prg_return_true:
1127 } {
1128     \__bnvs_group_end:
1129     \prg_return_false:
1130 }
1131 }
1132 }
1133 }
1134 \prg_generate_conditional_variant:Nnn
1135 \__bnvs_if_range:nN { VN } { T, F, TF }

```

---

`\__bnvs_if_previous:nNTF`    `\__bnvs_if_previous: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.

```

1136 \prg_new_conditional:Npnn \__bnvs_if_previous:nN #1 #2 { T, F, TF } {

```

```

1137 \__bnvs_group_begin:
1138 \__bnvs_get_cache:nnNTF P { #1 } #2 {
1139   \exp_args:NNNV
1140   \__bnvs_group_end:
1141   \tl_put_right:Nn #2 #2
1142   \prg_return_true:
1143 } {
1144   \tl_clear:N \l__bnvs_ans_tl
1145   \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1146     \tl_put_right:Nn \l__bnvs_ans_tl { -1 }
1147     \__bnvs_group_end_return_true:nnN P { #1 } #2
1148   } {
1149     \__bnvs_group_end_return_false:nn P { #1 }
1150   }
1151 }
1152 }
1153 \prg_generate_conditional_variant:Nnn
1154 \__bnvs_if_previous:nN { VN } { T, F, TF }

```

---

\\_\_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.

```

1155 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
1156   \__bnvs_group_begin:
1157   \__bnvs_get_cache:nnNTF N { #1 } #2 {
1158     \exp_args:NNNV
1159     \__bnvs_group_end:
1160     \tl_put_right:Nn #2 #2
1161     \prg_return_true:
1162   } {
1163     \tl_clear:N \l__bnvs_ans_tl
1164     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
1165       \tl_put_right:Nn \l__bnvs_ans_tl { +1 }
1166       \__bnvs_group_end_return_true:nnN P { #1 } #2
1167     } {
1168       \__bnvs_group_end_return_false:nn P { #1 }
1169     }
1170   }
1171 }
1172 \prg_generate_conditional_variant:Nnn
1173 \__bnvs_if_next:nN { VN } { T, F, TF }

```

---

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

Append the value of the <name> overlay set to the <tl variable>. Cache the result under subkey V. Execute <true code> when there is a <value>, <false code> otherwise.

```

1174 \prg_new_conditional:Npnn \__bnvs_raw_value:nN #1 #2 { T, F, TF } {
1175   \__bnvs_group_begin:
1176   \__bnvs_get_cache:nnNTF V { #1 } #2 {
1177     \exp_args:NNNV
1178     \__bnvs_group_end:
1179     \tl_put_right:Nn #2 #2

```

```

1180 \prg_return_true:
1181 } {
1182 \__bnvs_get:nnNTF V { #1 } \l__bnvs_a_tl {
1183 \tl_clear:N \l__bnvs_ans_tl
1184 \quark_if_nil:NTF \l__bnvs_a_tl {
1185 \__bnvs_gput:nnn V { #1 } { \q_no_value }
1186 \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1187 \__bnvs_group_end_return_true:nnN V { #1 } #2
1188 } {
1189 \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
1190 \__bnvs_group_end_return_true:nnN V { #1 } #2
1191 } {
1192 \__bnvs_group_end_return_false:nn V { #1 }
1193 }
1194 }
1195 } {
1196 \quark_if_no_value:NTF \l__bnvs_a_tl {
1197 \__bnvs_fatal:n {Circular~definition:~#1}
1198 } {
1199 \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1200 \__bnvs_group_end_return_true:nnN V { #1 } #2
1201 } {
1202 \__bnvs_group_end_return_false:nn V { #1 }
1203 }
1204 }
1205 }
1206 } {
1207 \__bnvs_group_end_return_false:nn V { #1 }
1208 }
1209 }
1210 }
1211 \prg_generate_conditional_variant:Nnn
1212 \__bnvs_raw_value:nN{ V } { T, F, TF }

```

---

$\backslash\_bnvs\_can\_index:n\overline{TF}$ $\backslash\_bnvs\_can\_index:V\overline{TF}$ $\backslash\_bnvs\_if\_index:nn\overline{NTF}$ $\backslash\_bnvs\_if\_index:VVN\overline{TF}$	$\backslash\_bnvs\_can\_index:nTF \{ \langle name \rangle \} \{ \langle true code \rangle \} \{ \langle false code \rangle \}$ $\backslash\_bnvs\_if\_index:nnNTF \{ \langle name \rangle \} \{ \langle integer \rangle \} \langle tl variable \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}$ $\backslash\_bnvs\_can\_index:nTF \{ \langle name \rangle \} \{ \langle integer \rangle \} \langle tl variable \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}$
--	---

---

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

```

1213 \prg_new_conditional:Npnn \__bnvs_can_index:n #1 { p, T, F, TF } {
1214 \bool_if:nTF {
1215 \__bnvs_if_in_p:nn A { #1 }
1216 || \__bnvs_if_in_p:nn Z { #1 }
1217 || \__bnvs_if_in_p:nn V { #1 }
1218 } {

```

```

1219     \prg_return_true:
1220   } {
1221     \prg_return_false:
1222   }
1223 }
1224 \prg_generate_conditional_variant:Nnn
1225   \__bnvs_can_index:n { V } { p, T, F, TF }
1226 \prg_new_conditional:Npnn \__bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
1227   \__bnvs_group_begin:
1228   \cs_set:Npn \group_end_return_true:n ##1 {
1229     \tl_put_right:Nn \l__bnvs_ans_tl { + #2 - 1 }
1230     \exp_args:NNV
1231     \__bnvs_group_end:
1232     \__bnvs_round:nN \l__bnvs_ans_tl #3
1233     \prg_return_true:
1234   }
1235   \tl_clear:N \l__bnvs_ans_tl
1236   \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {

```

Limited overlay set.

```

1237     \group_end_return_true:n { A }
1238   } {
1239     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {

```

Right limited overlay set.

```

1240     \group_end_return_true:n { Z }
1241   } {
1242     \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {

```

Unlimited overlay set.

```

1243     \group_end_return_true:n { V }
1244   } {
1245     \__bnvs_group_end:
1246     \prg_return_false:
1247   }
1248 }
1249 }
1250 }
1251 \prg_generate_conditional_variant:Nnn
1252   \__bnvs_if_index:nnN { VVN } { T, F, TF }

```

---

$\backslash\_bnvs\_if\_n\_value:nNTF$ $\backslash\_bnvs\_if\_n\_value:VNTF$	$\backslash\_bnvs\_if\_n\_value:nNTF \{ \langle name \rangle \} \langle tl \ variable \rangle \{ \langle true \ code \rangle \} \{ \langle false \ code \rangle \}$
--	---

---

Append the value of the n counter associated to the  $\{ \langle name \rangle \}$  overlay set to the right of  $\langle tl \ variable \rangle$ . Initialize this counter to 1 on the first use.  $\langle false \ code \rangle$  is never executed.

```

1253 \prg_new_conditional:Npnn \__bnvs_if_n_value:nN #1 #2 { T, F, TF } {
1254   \__bnvs_n_get:nNF { #1 } #2 {
1255     \tl_set:Nn #2 { 1 }
1256     \__bnvs_n_gput:nn { #1 } { 1 }
1257   }
1258   \prg_return_true:
1259 }
1260 \prg_generate_conditional_variant:Nnn
1261   \__bnvs_if_n_value:nN { VN } { T, F, TF }

```

---

`\_bnvs_if_n_index:nNTF`  
`\_bnvs_if_n_index:VNTF`

---

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

Append the value of the n counter associated to the {<name>} overlay set to the right of <tl variable>. Initialize this counter to 1 on the first use.

```

1262 \prg_new_conditional:Npnn \_bnvs_if_n_index:nN #1 #2 { T, F, TF } {
1263   \_bnvs_group_begin:
1264   \_bnvs_if_n_value:nNF { #1 } \l__bnvs_a_tl { }
1265   \exp_args:NNnV
1266   \_bnvs_group_end:
1267   \_bnvs_if_index:nnNTF { #1 } \l__bnvs_a_tl #2 {
1268     \prg_return_true:
1269   } {
1270     \_bnvs_group_begin:
1271     \_bnvs_raw_value:nNTF {#1} \l__bnvs_ans_tl {
1272       \tl_put_right:Nn \l__bnvs_ans_tl { + #2 - 1 }
1273       \exp_args:NNV
1274       \_bnvs_group_end:
1275       \_bnvs_round:Nn \l__bnvs_ans_tl
1276       \prg_return_true:
1277     } {
1278       \_bnvs_group_end:
1279       \prg_return_false:
1280     }
1281   }
1282 }
1283 \prg_generate_conditional_variant:Nnn
1284   \_bnvs_if_n_index:nN { VN } { T, F, TF }

```

---

`\_bnvs_if_incr:nnTF`  
`\_bnvs_if_incr:nnNTF`  
`\_bnvs_if_incr:(VnN|VVN)TF`

---

`\_bnvs_if_incr:nnTF {<name>} {<offset>} {<true code>} {<false code>}`  
`\_bnvs_if_incr:nnNTF {<name>} {<offset>} <tl variable> {<true code>} {<false code>}`

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

```

1285 \prg_new_conditional:Npnn \_bnvs_if_incr:nn #1 #2 { T, F, TF } {
1286   \_bnvs_group_begin:
1287   \tl_clear:N \l__bnvs_ans_tl
1288   \_bnvs_if_append:nNTF { #2 } \l__bnvs_ans_tl {
1289     \int_compare:nNtF \l__bnvs_ans_tl = 0 {
1290       \tl_clear:N \l__bnvs_ans_tl
1291       \_bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
1292         \_bnvs_group_end:
1293         \prg_return_true:
1294       } {
1295         \_bnvs_group_end:
1296         \prg_return_false:
1297       }
1298     } {
1299       \tl_put_right:Nn \l__bnvs_ans_tl { + }
1300       \_bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
1301         \_bnvs_round:N \l__bnvs_ans_tl
1302         \_bnvs_gput_cache:nnV V { #1 } \l__bnvs_ans_tl
1303         \_bnvs_group_end:

```

```

1304         \prg_return_true:
1305     } {
1306         \__bnvs_group_end:
1307         \prg_return_false:
1308     }
1309 }
1310 } {
1311     \__bnvs_group_end:
1312     \prg_return_false:
1313 }
1314 }
1315 \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1316     \__bnvs_if_incr:nnTF { #1 } { #2 } {
1317         \__bnvs_raw_value:nNTF { #1 } #3 {
1318             \prg_return_true:
1319         } {
1320             \prg_return_false:
1321         }
1322     } {
1323         \prg_return_false:
1324     }
1325 }
1326 \prg_generate_conditional_variant:Nnn
1327     \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }

```

---

<code>\__bnvs_if_n_incr:nnTF</code>	<code>\__bnvs_if_n_incr:nnTF {&lt;name&gt;} {&lt;offset&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\__bnvs_if_n_incr:nnNTF</code>	<code>\__bnvs_if_n_incr:nnNTF {&lt;name&gt;} {&lt;offset&gt;} &lt;tl variable&gt; {&lt;true code&gt;}</code>
<code>\__bnvs_if_n_incr:(VnN VVN)TF</code>	<code>{&lt;false code&gt;}</code>

---

Increment the implicit index counter accordingly. When requested, put the result in the *<tl variable>*.

```

1328 \prg_new_conditional:Npnn \__bnvs_if_n_incr:nn #1 #2 { T, F, TF } {
1329     \__bnvs_group_begin:
1330     \tl_clear:N \l__bnvs_ans_tl
1331     \__bnvs_n_get:nNF { #1 } \l__bnvs_ans_tl {
1332         \tl_set:Nn \l__bnvs_ans_tl { 1 }
1333     }
1334     \tl_clear:N \l__bnvs_a_tl
1335     \__bnvs_if_append:nNTF { #2 } \l__bnvs_a_tl {
1336         \tl_put_right:Nn \l__bnvs_ans_tl { + }
1337         \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1338         \__bnvs_round:N \l__bnvs_ans_tl
1339         \__bnvs_n_gput:nV { #1 } \l__bnvs_ans_tl
1340     }
1341     \__bnvs_group_end:
1342     \prg_return_true:
1343 } {
1344     \__bnvs_group_end:
1345     \prg_return_false:
1346 }
1347 \prg_new_conditional:Npnn \__bnvs_if_n_incr:nnN #1 #2 #3 { T, F, TF } {
1348     \__bnvs_if_n_incr:nnTF { #1 } { #2 } {
1349         \__bnvs_n_get:nNTF { #1 } #3 {

```

```

1350     \prg_return_true:
1351   } {
1352     \prg_return_false:
1353   }
1354 } {
1355   \prg_return_false:
1356 }
1357 }
1358 \prg_generate_conditional_variant:Nnn
1359   \__bnvs_if_n_incr:nnN { VnN, VVN } { T, F, TF }

```

---

\\_\_bnvs\_if\_post:nnNTF

\\_\_bnvs\_if\_post:(VnN|VVN)TF

---

\\_\_bnvs\_if\_post:nnNTF {<name>} {<offset>} <tl variable> {<true code>} {<false code>}

Put the value of the free counter for the given <name> in the <tl variable> then increment this free counter position accordingly.

```

1360 \prg_new_conditional:Npnn \__bnvs_if_post:nnN #1 #2 #3 { T, F, TF } {
1361   \__bnvs_group_begin:
1362   \tl_clear:N \l__bnvs_ans_tl
1363   \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
1364     \__bnvs_if_incr:nnTF { #1 } { #2 } {
1365       \exp_args:NNNV
1366       \__bnvs_group_end:
1367       \tl_put_right:Nn #3 \l__bnvs_ans_tl
1368       \prg_return_true:
1369     } {
1370       \__bnvs_group_end:
1371       \prg_return_false:
1372     }
1373   } {
1374     \__bnvs_group_end:
1375     \prg_return_false:
1376   }
1377 }
1378 \prg_generate_conditional_variant:Nnn
1379   \__bnvs_if_post:nnN { VnN, VVN } { T, F, TF }

```

## 6.8.8 Evaluation

---

```

\__bnvs_if_append:nNTF
\__bnvs_if_append:(VN|xN)TF

```

---

```

\__bnvs_if_append:nNTF {<integer expression>} <tl variable> {<true code>} {<false
code>}}

```

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

`\l__bnvs_ans_tl` To feed  $\langle tl\ variable \rangle$  with.

(End definition for `\l__bnvs_ans_tl`.)

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

(End definition for `\l__bnvs_split_seq`.)

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

(End definition for `\l__bnvs_split_int`.)

```

1380 \int_new:N \l__bnvs_split_int

```

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

(End definition for `\l__bnvs_name_tl`.)

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

(End definition for `\l__bnvs_path_tl`.)

Catch circular definitions. Open a main T<sub>E</sub>X group to define local functions and variables, sometimes another grouping level is used. The main T<sub>E</sub>X group is closed in the `\return_...` functions.

```

1381 \prg_new_conditional:Nnn \__bnvs_if_append:nN { T, F, TF } {
1382   \__bnvs_call:TF {
1383     \__bnvs_group_begin:

```

This T<sub>E</sub>X group is closed just before returning. Local variables:

```

1384   \int_zero:N \l__bnvs_split_int
1385   \seq_clear:N \l__bnvs_split_seq
1386   \tl_clear:N \l__bnvs_id_tl
1387   \tl_clear:N \l__bnvs_name_tl
1388   \tl_clear:N \l__bnvs_path_tl
1389   \tl_clear:N \l__bnvs_group_tl
1390   \tl_clear:N \l__bnvs_ans_tl
1391   \tl_clear:N \l__bnvs_a_tl

```

Implementation:

```

1392   \regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq
1393   \int_set:Nn \l__bnvs_split_int { 1 }
1394   \tl_set:Nx \l__bnvs_ans_tl {
1395     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1396   }

```



---

`\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.

```

1397 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1398   \tl_set:Nx ##2 {
1399     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1400   }
1401   \tl_if_empty:NTF ##2 {
1402     ##4 } {
1403     ##3
1404   }
1405 }
1406 \cs_set:Npn \fp_round: {
1407   \__bnvs_round:N \l__bnvs_ans_tl
1408 }

```

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

```

1409 \cs_set:Npn \group_end_return_false: {
1410   \cs_set:Npn \loop: {
1411     \__bnvs_group_end:
1412     \prg_return_false:
1413   }
1414 }
1415 \cs_set:Npn \group_end_return_false:x ##1 {
1416   \__bnvs_error:x { ##1 }
1417   \group_end_return_false:
1418 }
1419 \cs_set:Npn \resolve_n:T ##1 {
1420   \__bnvs_resolve_n:TFF {
1421     ##1
1422   } {
1423     \group_end_return_false:x { Too-many~dotted-components:~#1 }
1424   } {
1425     \group_end_return_false:x { Unknown~dotted-path:~#1 }
1426   }
1427 }
1428 \cs_set:Npn \resolve_x:T ##1 {
1429   \__bnvs_resolve_x:TFF {
1430     ##1
1431   } {
1432     \group_end_return_false:x { Too-many~dotted-components:~#1 }
1433   } {
1434     \group_end_return_false:x { Unknown~dotted-path:~#1 }
1435   }
1436 }
1437 \cs_set:Npn \:nn ##1 ##2 {
1438   \switch:nNTF { ##1 } \l__bnvs_id_tl { } {
1439     \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_last_tl
1440     \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1441   }
1442   \switch:nNTF { ##2 } \l__bnvs_path_tl {
1443     \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl

```

```

1444     \seq_remove_all:Nn \l__bnvs_path_seq { }
1445   } {
1446     \seq_clear:N \l__bnvs_path_seq
1447   }
1448 }
1449 \cs_set:cpn {.n?:TF} ##1 ##2 {
1450   \seq_get_right:NNTF \l__bnvs_path_seq \l__bnvs_b_tl {
1451     \exp_args:NV
1452     \str_if_eq:nnTF \l__bnvs_b_tl { n } {
1453       \seq_pop_right:NN \l__bnvs_path_seq \l__bnvs_b_tl
1454       ##1
1455     } { ##2 }
1456   } { ##2 }
1457 }
1458 \cs_set:cpn {...++n:} {
1459   \__bnvs_group_begin:
1460   \__bnvs_resolve_n:TFF {
1461     \tl_clear:N \l__bnvs_b_tl
1462     \__bnvs_if_n_incr:VnNTF \l__bnvs_name_tl { 1 } \l__bnvs_b_tl {
1463       \exp_args:NNNV
1464       \__bnvs_group_end:
1465       \tl_set:Nn \l__bnvs_b_tl \l__bnvs_b_tl
1466       \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
1467       \resolve_x:T {
1468         \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
1469       }
1470     } {
1471       \__bnvs_group_end:
1472     }
1473   } {
1474     \__bnvs_group_end:
1475     \group_end_return_false:x { Too~many~dotted~components::~#1 }
1476   } {
1477     \__bnvs_group_end:
1478     \group_end_return_false:
1479   }
1480 }

```

Main loop. The explanations given here apply to quite every case. We start by recovering the frame id and the dotted path. Then we resolve the slide range name and path to the last possible name and a void integer path. We raise if we cannot obtain a void integer path.

```

1481   \cs_set:Npn \loop: {
1482     \int_compare:nNnTF {
1483       \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1484     } {
1485       \switch:nNTF { 1 } \l__bnvs_name_tl {
1486         \:nn { 2 } { 3 }
1487         \use:c {.n?:TF} {

```

- Case ++...n.

```

1488       \use:c { ...++n: }
1489     } {

```

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

```

1490         \resolve_n:T {
1491             \tl_clear:N \l__bnvs_ans_tl
1492             \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1493                 \group_end_return_false:
1494             }
1495         }
1496     }
1497 } {
1498     \switch:nNTF 4 \l__bnvs_name_tl {
1499         \:nn { 5 } { 6 }
1500         \switch:nNTF 7 \l__bnvs_a_tl {

```

- Case ...++n.

```

1501         \use:c { ...++n: }
1502     } {
1503         \switch:nNTF 8 \l__bnvs_a_tl {
1504             \use:c { .n?:TF } {

```

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

```

1505 \__bnvs_group_begin:
1506 \__bnvs_resolve_n:TFF {
1507     \tl_clear:N \l__bnvs_b_tl
1508     \__bnvs_if_n_incr:VVNTF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_b_tl {
1509         \exp_args:NNNV
1510         \__bnvs_group_end:
1511         \tl_set:Nn \l__bnvs_b_tl \l__bnvs_b_tl
1512         \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
1513         \resolve_x:T {
1514             \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
1515         }
1516     } {
1517         \__bnvs_group_end:
1518     }
1519 } {
1520     \__bnvs_group_end:
1521     \group_end_return_false:x { Too~many~dotted~components:~#1 }
1522 } {
1523     \__bnvs_group_end:
1524     \group_end_return_false:x { Unknown~dotted~path:~#1 }
1525 }
1526     } {

```

- Case A+=*<integer>*.

```

1527 \resolve_n:T {
1528     \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1529         \group_end_return_false:
1530     }
1531 }
1532     }
1533 } {
1534     \switch:nNTF 9 \l__bnvs_a_tl {

```

- Case ...++.

```

1535 \resolve_n:T {
1536   \__bnvs_if_post:VnNF \l__bnvs_name_tl { 1 } \l__bnvs_ans_tl {
1537     \return_false:
1538   }
1539 }
1540           } {

```

Only the path, branch according to the last component.

```

1541 \seq_pop_right:NNTF \l__bnvs_path_seq \l__bnvs_b_tl {
1542   \exp_args:NV
1543   \str_case:nnF \l__bnvs_b_tl {
1544     { n } {

```

- Case ...n.

```

1545   \__bnvs_group_begin:
1546   \resolve_n:T {
1547     \exp_args:NNV
1548     \__bnvs_group_end:
1549     \__bnvs_if_n_value:nNTF \l__bnvs_name_tl \l__bnvs_b_tl {
1550       \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
1551       \resolve_x:T {
1552         \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
1553       }
1554     } {
1555   \group_end_return_false:x { Undefined~dotted~path:~#1 }
1556   }
1557   }
1558   }
1559   { length } {

```

- Case ...length.

```

1560   \resolve_n:T {
1561     \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1562       \group_end_return_false:
1563     }
1564   }
1565   }
1566   { last } {

```

- Case ...last.

```

1567   \resolve_n:T {
1568     \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1569       \group_end_return_false:
1570     }
1571   }
1572   }
1573   { range } {

```

- Case ...range.

```

1574     \resolve_n:T {
1575         \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
1576             \cs_set_eq:NN \fp_round: \prg_do_nothing:
1577         } {
1578             \group_end_return_false:
1579         }
1580     }
1581 }
1582 { previous } {

```

- Case ...previous.

```

1583     \resolve_n:T {
1584         \__bnvs_if_previous:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1585             \group_end_return_false:
1586         }
1587     }
1588 }
1589 { next } {

```

- Case ...next.

```

1590     \resolve_n:T {
1591         \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1592             \group_end_return_false:
1593         }
1594     }
1595 }
1596 } {

```

- Case ...*<integer>*.

```

1597     \resolve_n:T {
1598         \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_b_tl \l__bnvs_ans_tl {
1599             \group_end_return_false:
1600         }
1601     }
1602 }
1603 } {

```

- Case ....

```

1604     \resolve_n:T {
1605         \__bnvs_raw_value:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1606             \group_end_return_false:
1607         }
1608     }
1609 }
1610     }
1611     }
1612     }
1613     } {

```

No name. Unreachable code.

```

1614     }
1615   }
1616   \int_add:Nn \l__bnvs_split_int { 10 }
1617   \tl_put_right:Nx \l__bnvs_ans_tl {
1618     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1619   }
1620   \loop:
1621 } {
1622   \fp_round:
1623   \exp_args:NNNV
1624   \__bnvs_group_end:
1625   \tl_put_right:Nn #2 \l__bnvs_ans_tl
1626   \prg_return_true:
1627 }
1628 }
1629 \loop:
1630 } {
1631   \__bnvs_error:x { Too~many~calls:~ #1 }
1632   \prg_return_false:
1633 }
1634 }
1635 \prg_generate_conditional_variant:Nnn
1636   \__bnvs_if_append:nN { VN } { T, F, TF }

```

---

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

Evaluates the single `<overlay query>`, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of the `<seq variable>`. Ranges are supported with the colon syntax. This is executed within a local `TEX` 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`.)

```

1637 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1638   \A \s* (?
    • 2: <first>
1639     ( [^:]* ) \s* :
    • 3: second optional colon
1640     (:)? \s*
    • 4: <length>
1641     ( [^:]* )
    • 5: standalone <first>
1642     | ( [^:]+ )
1643   ) \s* \Z
1644 }

1645 \prg_new_conditional:Npnn \__bnvs_if_eval_query:nN #1 #2 { T, F, TF } {
1646   \__bnvs_call_greset:
1647   \cs_set:Npn \return_true: {
1648     \prg_return_true:
1649   }
1650   \cs_set:Npn \return_false: {
1651     \prg_return_false:
1652   }
1653   \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
1654     #1
1655   } \l__bnvs_match_seq {

```

---

```
\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.

```
1656 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1657   \tl_set:Nx ##2 {
1658     \seq_item:Nn \l__bnvs_match_seq { ##1 }
1659   }
1660   \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1661 }
1662 \switch:nNTF 5 \l__bnvs_a_tl {
```

☛ Single expression

```
1663   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1664     \return_true:
1665   } {
1666     \return_false:
1667   }
1668 } {
1669   \switch:nNTF 2 \l__bnvs_a_tl {
1670     \switch:nNTF 4 \l__bnvs_b_tl {
1671       \switch:nNTF 3 \l__bnvs_c_tl {
```

☛ *<first>::<last>* range

```
1672   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1673     \tl_put_right:Nn #2 { - }
1674     \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1675       \return_true:
1676     } {
1677       \return_false:
1678     }
1679   } {
1680     \return_false:
1681   }
1682 } {
```

☛ *<first>:<length>* range

```
1683   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1684     \tl_put_right:Nx #2 { - }
1685     \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1 }
1686     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1687       \return_true:
1688     } {
1689       \return_false:
1690     }
1691   } {
1692     \return_false:
1693   }
1694 }
1695 }
```

☛ *<first>:* and *<first>::* range



```

1696         \_bnvs_if_append:VNTF \l\_bnvs_a_tl #2 {
1697             \tl_put_right:Nn #2 { - }
1698             \return_true:
1699         } {
1700             \return_false:
1701         }
1702     }
1703 } {
1704     \switch:nNTF 4 \l\_bnvs_b_tl {
1705         \switch:nNTF 3 \l\_bnvs_c_tl {
1706             \tl_put_right:Nn #2 { - }
1707             \_bnvs_if_append:VNTF \l\_bnvs_a_tl #2 {
1708                 \return_true:
1709             } {
1710                 \return_false:
1711             }
1712         } {
1713             \_bnvs_error:x { Syntax~error(Missing~first):~#1 }
1714         }
1715     } {
1716         \seq_put_right:Nn #2 { - }
1717     }
1718 }
1719 }
1720 } {

```

::*⟨last⟩* range

```

1706         \tl_put_right:Nn #2 { - }
1707         \_bnvs_if_append:VNTF \l\_bnvs_a_tl #2 {
1708             \return_true:
1709         } {
1710             \return_false:
1711         }
1712     } {
1713         \_bnvs_error:x { Syntax~error(Missing~first):~#1 }
1714     }
1715 } {

```

: or :: range

```

1716         \seq_put_right:Nn #2 { - }
1717     }
1718 }
1719 }
1720 } {

```

Error

```

1721     \_bnvs_error:n { Syntax~error:~#1 }
1722     \return_false:
1723 }
1724 }

```

---

`\__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.

1725 `\regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }`

(End definition for `\c__bnvs_comma_regex`.)

No other variable is used.

1726 `\cs_new:Npn \__bnvs_eval:nN #1 #2 {`

1727 `\__bnvs_group_begin:`

Local variables declaration

1728 `\seq_clear:N \l__bnvs_query_seq`

1729 `\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`.

1730 `\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.

1731 `\seq_map_inline:Nn \l__bnvs_query_seq {`  
 1732 `\tl_clear:N \l__bnvs_ans_tl`  
 1733 `\__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {`  
 1734 `\seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl`  
 1735 `} {`  
 1736 `\seq_map_break:n {`  
 1737 `\__bnvs_fatal:n { Circular/Undefined~dependency~in~#1 }`  
 1738 `}`  
 1739 `}`  
 1740 `}`

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

1741 `\exp_args:NNNx`  
 1742 `\__bnvs_group_end:`  
 1743 `\tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }`  
 1744 `}`  
 1745 `\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.

```
1746 \NewDocumentCommand \BeanovesEval { o m } {
1747   \__bnvs_group_begin:
1748   \tl_clear:N \l__bnvs_ans_tl
1749   \__bnvs_eval:nN { #2 } \l__bnvs_ans_tl
1750   \IfValueTF { #1 } {
1751     \exp_args:NNNV
1752     \__bnvs_group_end:
1753     \tl_set:Nn #1 \l__bnvs_ans_tl
1754   } {
1755     \exp_args:NV
1756     \__bnvs_group_end: \l__bnvs_ans_tl
1757   }
1758 }
```

### 6.8.9 Reseting counters

---

**\BeanovesReset**

---

**\beanovesReset** [*<first value>*] {*<key>*}

---

**\BeanovesReset\***

---

**\beanovesReset\*** [*<first value>*] {*<key>*}

Forwards to `\__bnvs_reset:nn` or `\__bnvs_reset_all:nn` when starred.

```
1759 \NewDocumentCommand \BeanovesReset { s O{} m } {
1760   \__bnvs_id_name_set:nNTF { #3 } \l__bnvs_id_tl \l__bnvs_name_tl {
1761     \IfBooleanTF { #1 } {
1762       \exp_args:NV \__bnvs_reset_all:nn
1763     } {
1764       \exp_args:NV \__bnvs_reset:nn
1765     }
1766     \l__bnvs_name_tl { #2 }
1767   } {
1768     \__bnvs_warning:n { Unknown~name:~#1 }
1769   }
1770   \ignorespaces
1771 }
```

---

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

---

**\\_\_bnvs\_reset:nn** {*<key>*} {*<first value>*}

---

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

---

The key must include the frame id. Reset the value counter to the given *<first value>*. The `_all` version also cleans the cached values.

```
1772 \cs_new:Npn \__bnvs_reset_all:nn #1 #2 {
1773   \bool_if:nTF {
1774     \__bnvs_if_in_p:nn A { #1 }
1775     || \__bnvs_if_in_p:nn Z { #1 }
1776     || \__bnvs_if_in_p:nn V { #1 }
1777   } {
```

```

1778     \__bnvs_gremove_cache:nn A { #1 }
1779     \__bnvs_gremove_cache:nn L { #1 }
1780     \__bnvs_gremove_cache:nn Z { #1 }
1781     \__bnvs_gremove_cache:nn P { #1 }
1782     \__bnvs_gremove_cache:nn N { #1 }
1783     \__bnvs_gremove_cache:nn V { #1 }
1784     \tl_if_empty:nF { #2 } {
1785         \__bnvs_gput_cache:nnn V { #1 } { #2 }
1786     }
1787 } {
1788     \__bnvs_warning:n { Unknown~name:~#1 }
1789 }
1790 }
1791 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1792     \__bnvs_if_in:nnTF V { #1 } {
1793         \__bnvs_gremove_cache:nn V { #1 }
1794         \tl_if_empty:nF { #2 } {
1795             \__bnvs_gput_cache:nnn V { #1 } { #2 }
1796         }
1797     } {
1798         \__bnvs_warning:n { Unknown~name:~#1 }
1799     }
1800 }
1801 \makeatother
1802 \ExplSyntaxOff

```