beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2022/10/28

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.

Contents

1 Minimal example

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

```
 1 \documentclass {beamer}
 2 \RequirePackage {beanoves}
 3 \begin{document}
 4 \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.

⟨***short name***⟩ : like `A` and `C`,

⟨***frame id***⟩`!`⟨***short name***⟩ : denoted by *qualified names*, like `X!A` and `Y!C`.

⟨***short name***⟩⟨***dotted path***⟩ : denoted by *full names* like `A.1` and `C.2`,

⟨***frame id***⟩`!`⟨***short name***⟩⟨***dotted path***⟩ : 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 `.`⟨*component*$_1$⟩`.`⟨*component*$_2$⟩`. . . .`⟨*component*$_n$⟩, where each ⟨*component*$_i$⟩ denotes either an integer, eventually signed, or a ⟨*short name*⟩. 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.

`beanoves` `beanoves = {`⟨*ref*$_1$⟩`=`⟨*spec*$_1$⟩`,` ⟨*ref*$_2$⟩`=`⟨*spec*$_2$⟩`,...,` ⟨*ref*$_n$⟩`=`⟨*spec*$_n$⟩`}`

`\Beanoves` `\Beanoves{`⟨*ref*$_1$⟩`=`⟨*spec*$_1$⟩`,` ⟨*ref*$_2$⟩`=`⟨*spec*$_2$⟩`,...,` ⟨*ref*$_n$⟩`=`⟨*spec*$_n$⟩`}`

Each ⟨*ref*⟩ key is a *named overlay reference* whereas each ⟨*spec*⟩ value is an *overlay set specifier*. When the same ⟨*ref*⟩ key is used multiple times, only the last one is taken

into account. Possible ⟨*spec*⟩ value are the *range specifiers*
⟨*first*⟩, ⟨*first*⟩`:` and ⟨*first*⟩`::`, ⟨*first*⟩`:`⟨*length*⟩, ⟨*first*⟩`::`⟨*last*⟩,
`:`⟨*length*⟩`::`⟨*last*⟩ and `::`⟨*last*⟩`:`⟨*length*⟩.
Here ⟨*first*⟩, ⟨*length*⟩ and ⟨*last*⟩ are algebraic expression possibly involving any *named overlay reference* defined above. At least one of ⟨*first*⟩ or ⟨*last*⟩ must be provided.

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside 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 ⟨*ref*⟩=⟨*spec*⟩ definitions. The definition
$\langle \mathit{key}\rangle$`=[`$\langle \mathit{ref}_1\rangle$`=`$\langle \mathit{spec}_1\rangle$`, `$\langle \mathit{ref}_2\rangle$`=`$\langle \mathit{spec}_2\rangle$`,..., `$\langle \mathit{ref}_n\rangle$`=`$\langle \mathit{spec}_n\rangle$`]`
is a convenient shortcut for
$\langle \mathit{key}\rangle$`.`$\langle \mathit{ref}_1\rangle$`=`$\langle \mathit{value}_1\rangle$`,`
$\langle \mathit{key}\rangle$`.`$\langle \mathit{ref}_2\rangle$`=`$\langle \mathit{value}_2\rangle$`,`
`...,`
$\langle \mathit{key}\rangle$`.`$\langle \mathit{ref}_n\rangle$`=`$\langle \mathit{value}_n\rangle$`.`
The rules above can apply individually to each line.

To support an array syntax, we can omit the ⟨*ref*⟩ 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 `<?(A.1)>` 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 ⟨*i*⟩ denotes an integer whereas ⟨*first*⟩, ⟨*length*⟩ and ⟨*last*⟩ stand for integers, or integer valued expressions.

3.1 Simple definitions

⟨name⟩ = ⟨*first*⟩ For an unlimited range

reference	resolution
⟨**name**⟩`.1`	⟨*first*⟩
⟨**name**⟩`.2`	⟨*first*⟩ + 1
⟨**name**⟩`.`⟨*i*⟩	⟨*first*⟩ + ⟨*i*⟩ − 1

⟨name⟩ = ⟨*first*⟩: as well as ⟨*first*⟩`::`. For a range limited from below:

reference	resolution
⟨**name**⟩`.1`	⟨*first*⟩
⟨**name**⟩`.2`	⟨*first*⟩ + 1
⟨**name**⟩`.`⟨*i*⟩	⟨*first*⟩ + ⟨*i*⟩ − 1
⟨**name**⟩`.previous`	⟨*first*⟩ − 1

⟨name⟩ = ::⟨*last*⟩ For a range limited from above:

reference	resolution
⟨*name*⟩`.1`	⟨*last*⟩
⟨*name*⟩`.0`	⟨*last*⟩ − 1
⟨*name*⟩`.`⟨*i*⟩	⟨*last*⟩ + ⟨*i*⟩ − 1
⟨*name*⟩`.next`	⟨*last*⟩ + 1

⟨*name*⟩ `=` ⟨*first*⟩`:`⟨*length*⟩ as well as variants ⟨*first*⟩`::`⟨*last*⟩, `:`⟨*length*⟩`::`⟨*last*⟩ or `::`⟨*last*⟩`:`⟨*length*⟩, which are equivalent provided ⟨*first*⟩ + ⟨*length*⟩ = ⟨*last*⟩ + 1.

For a range limited from both above and below:

reference	resolution
⟨*name*⟩`.1`	⟨*first*⟩
⟨*name*⟩`.2`	⟨*first*⟩ + 1
⟨*name*⟩`.`⟨*i*⟩	⟨*first*⟩ + ⟨*i*⟩ − 1)
⟨*name*⟩`.previous`	⟨*first*⟩ − 1
⟨*name*⟩`.last`	⟨*last*⟩
⟨*name*⟩`.next`	⟨*last*⟩ + 1
⟨*name*⟩`.length`	⟨*length*⟩
⟨*name*⟩`.range`	max(0, ⟨*first*⟩) `''-''` max(0, ⟨*last*⟩)

Notice that the resolution of ⟨*name*⟩`.range` is not an algebraic difference, and negative integers do not make sense there while in beamer context.

For example

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

3.2 Counters

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

Each named overlay set defined also has a dedicated value counter which is some kind of variable that can be used and incremented. A simple ⟨*name*⟩ reference is resolved into the position of this counter, which is the value of the counter bounded from below by the lowest value of the range and from above by the largest value of the range. For unbounded ranges, these values may be infinite.

Additionnaly, resolution rules are provided for the *named index references*:

⟨***name***⟩`.n+=`⟨***integer expression***⟩ : resolve ⟨*integer expression*⟩ into ⟨*integer*⟩, advance the implicit index counter associate to ⟨*name*⟩ by ⟨*integer*⟩ and use the resolution of ⟨*name*⟩`.n`.

Here ⟨*integer expression*⟩ denotes the longest character sequence with no space[1].

⟨***name***⟩**`.++n`** as well as `++`⟨*name*⟩`.n`: advance the implicit index counter associate to ⟨*name*⟩ by 1 and use the resolution of ⟨*name*⟩`.n`,

⟨***name***⟩**`.n++`** : use the resolution of ⟨*name*⟩`.n` and increment the implicit index counter associate to ⟨*name*⟩ by 1.

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

⟨***name***⟩**`+=`**⟨***integer expression***⟩ : resolve ⟨*integer expression*⟩ into ⟨*integer*⟩, advance the value counter by ⟨*integer*⟩ and use the new position. Here again, ⟨*integer expression*⟩ is the longest character sequence with no space.

`++`⟨***name***⟩ : advance the value counter for ⟨*name*⟩ by 1 and use the new position.

⟨***name***⟩**`++`** : use the actual position and advance the value counter for ⟨*name*⟩ by 1.

For example both `?(A.next)`, `?(A.last+1)`, `?(A.1+A.length)` give the same result as soon as the slide range named '`A`' has been properly defined with a starting value and a length.

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

3.3 Dotted paths

⟨***name***⟩**`.`**⟨***i***⟩ **`=`** ⟨***range spec***⟩ All the preceding rules are overriden by this particular one and ⟨*name*⟩`.`⟨*i*⟩ resolves to the resolution of ⟨*range spec*⟩.

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

```
\Beanoves {
  A = 3,
  A.3 = 0,
}
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\ttfamily
\BeanovesEval(A.1) == 3,
\BeanovesEval(A.2) == 4,
\BeanovesEval(A.-1)== 1,
\BeanovesEval(A.3) == 0,
\end{frame}
```

⟨***name***⟩**`.`**$\langle c_1\rangle$**`.`**$\langle c_2\rangle$**`...`**$\langle c_k\rangle$ **`=`** ⟨***range spec***⟩ 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 ⟨*name*⟩`.`$\langle c_1\rangle$`.`$\langle c_2\rangle$`...`$\langle c_n\rangle$, we replace the longest ⟨*name*⟩`.`$\langle c_1\rangle$`.`$\langle c_2\rangle$`...`$\langle c_k\rangle$ where

[1] The parser for algebraic expression is very rudimentary.

0≤k≤n by its definition ⟨*name'*⟩.⟨c'_1⟩...⟨c'_p⟩ 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 TEX loop. One final resolution occurs with rules above if possible or an error is raised.

For a *named indexed reference* like ⟨*name*⟩.⟨c_1⟩.⟨c_2⟩...⟨c_n⟩.n, we must first resolve ⟨*name*⟩.⟨c_1⟩.⟨c_2⟩...⟨c_n⟩ into ⟨*name'*⟩ with an empty dotted path, then retrieve the value of ⟨*name'*⟩.n denoted as ⟨*n'*⟩ and finally use the resolved ⟨*name*⟩.⟨c_1⟩.⟨c_2⟩...⟨c_n⟩.⟨*n'*⟩.

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 ⟨*current frame id*⟩`!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 ⟨*frame id*⟩, use the new `beanoves id` option of the beamer frame environment.

`beanoves id` `beanoves id=`⟨*frame id*⟩`,`

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 '`?(`⟨*queries*⟩`)`'. Each one is then evaluated and replaced by its resolved static counterpart. The overall result is finally forwarded to the beamer class.

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

query	resolution	limitation
⟨**first expr**⟩	⟨*first*⟩	
⟨**first expr**⟩:	⟨*first*⟩ -	no ⟨*name*⟩.range
⟨**first expr**⟩:⟨**length expr**⟩	⟨*first*⟩ - ⟨*last*⟩	no ⟨*name*⟩.range
::⟨**end expr**⟩:⟨**length expr**⟩	⟨*first*⟩ - ⟨*last*⟩	no ⟨*name*⟩.range
:	-	
⟨**first expr**⟩::	⟨*first*⟩ -	no ⟨*name*⟩.range
::⟨**end expr**⟩	- ⟨*last*⟩	no ⟨*name*⟩.range
⟨**first expr**⟩::⟨**end expr**⟩	⟨*first*⟩ - ⟨*last*⟩	no ⟨*name*⟩.range
:⟨**length expr**⟩::⟨**end expr**⟩	⟨*first*⟩ - ⟨*last*⟩	no ⟨*name*⟩.range
::	-	

Here ⟨*first expr*⟩, ⟨*length expr*⟩ and ⟨*end expr*⟩ both denote algebraic expressions possibly involving named slide references and counters. As integers, they are respectively resolved into ⟨*first*⟩, ⟨*length*⟩ and ⟨*last*⟩.

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

5 Support

See `https://github.com/jlaurens/beanoves`.

6 Implementation

Identify the internal prefix (LaTeX3 DocStrip convention).

```
⟨@@=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

```
\NeedsTeXFormat{LaTeX2e}[2020/01/01]
\ProvidesExplPackage
  {beanoves}
  {2022/10/28}
  {1.0}
  {Named overlay specifications for beamer}
```

6.2 logging

Utility message.

```
\msg_new:nnn { beanoves } { :n } { #1 }
\msg_new:nnn { beanoves } { :nn } { #1~(#2) }
\cs_new:Npn \__bnvs_warning:n {
  \msg_warning:nnn { beanoves } { :n }
}
\cs_new:Npn \__bnvs_error:n {
  \msg_error:nnn { beanoves } { :n }
}
\cs_new:Npn \__bnvs_error:x {
  \msg_error:nnx { beanoves } { :n }
}
\cs_new:Npn \__bnvs_fatal:n {
  \msg_fatal:nnn { beanoves } { :n }
}
\cs_new:Npn \__bnvs_fatal:x {
  \msg_fatal:nnx { beanoves } { :n }
}
```

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 TeX group, which ensures no name collision with the caller stack. In that case, variables need not follow exactly the LaTeX3 naming convention: we do not specialize with the module name. On execution, next initialization instructions declare the variables as side effect.

```
\tl_new:N \l__bnvs_id_last_tl
\tl_set:Nn \l__bnvs_id_last_tl { ?! }
\tl_new:N \l__bnvs_a_tl
\tl_new:N \l__bnvs_b_tl
\tl_new:N \l__bnvs_c_tl
\tl_new:N \l__bnvs_id_tl
\tl_new:N \l__bnvs_ans_tl
\tl_new:N \l__bnvs_name_tl
\tl_new:N \l__bnvs_path_tl
\tl_new:N \l__bnvs_group_tl
\tl_new:N \l__bnvs_query_tl
\tl_new:N \l__bnvs_token_tl
\tl_new:N \l__bnvs_root_tl
\int_new:N \g__bnvs_call_int
\int_new:N \l__bnvs_int
\seq_new:N \g__bnvs_def_seq
\seq_new:N \l__bnvs_a_seq
\seq_new:N \l__bnvs_b_seq
\seq_new:N \l__bnvs_ans_seq
\seq_new:N \l__bnvs_match_seq
\seq_new:N \l__bnvs_split_seq
\seq_new:N \l__bnvs_path_seq
\seq_new:N \l__bnvs_query_seq
\seq_new:N \l__bnvs_token_seq
\bool_new:N \l__bnvs_in_frame_bool
\bool_new:N \l__bnvs_parse_bool
\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`.*)

```
\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.

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

`\__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.

```
\prg_new_conditional:Npnn  \__bnvs_call: { T, F, TF } {
  \int_gdecr:N \g__bnvs_call_int
  \int_compare:nNnTF \g__bnvs_call_int > 0 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

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

⟨*id*⟩!⟨*name*⟩//P for the cached static value of the previous index

⟨*id*⟩!⟨*name*⟩//N for the cached static value of the next index

⟨*id*⟩!⟨*name*⟩//V for the real counter value

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

```
\prop_new:N \g__bnvs_prop
```

(*End definition for* `\g__bnvs_prop`.)

6.7 Basic functions

```
\__bnvs_gput:nnn
\__bnvs_gput:nnV
\__bnvs_gprovide:nnn
\__bnvs_gprovide:nVn
\__bnvs_item:nn
\__bnvs_gremove:nn
\__bnvs_gremove:nV
\__bnvs_gclear:n
\__bnvs_gclear:
```

```
\__bnvs_gput:nnn {⟨subkey⟩} {⟨key⟩} {⟨value⟩}
\__bnvs_gprovide:nnn {⟨subkey⟩} {⟨key⟩} {⟨value⟩}
\__bnvs_item:nn {⟨subkey⟩} {⟨key⟩}
\__bnvs_get:nnN {⟨subkey⟩} {⟨key⟩} ⟨tl variable⟩
\__bnvs_gremove:nn {⟨subkey⟩} {⟨key⟩}
\__bnvs_clear:n {⟨key⟩}
\__bnvs_clear:
```

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over LaTeX3 eponym functions, except `\__bnvs_gprovide:nn` which meaning is straightforward. The key argument is ⟨*key*⟩/⟨*subkey*⟩.

```
\cs_new:Npn \__bnvs_gput:nnn #1 #2 {
  \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
}
\cs_new:Npn \__bnvs_gprovide:nnn #1 #2 #3 {
  \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
    \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
  }
}
\cs_new:Npn \__bnvs_item:nn #1 #2 {
  \prop_item:Nn \g__bnvs_prop { #2 / #1 }
}
\cs_new:Npn \__bnvs_gremove:nn  #1 #2 {
  \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
}
\cs_new:Npn \__bnvs_gclear:n #1 {
  \__bnvs_gremove:nn {} { #1 }
  \clist_map_inline:nn { A, L, Z, V, n } {
    \__bnvs_gremove:nn { ##1 } { #1 }
  }
  \__bnvs_gclear_cache:n { #1 }
}
\cs_new:Npn \__bnvs_gclear: {
  \prop_gclear:N \g__bnvs_prop
}
\cs_generate_variant:Nn \__bnvs_gput:nnn { nnV }
\cs_generate_variant:Nn \__bnvs_gremove:nn { nV }
```

```
\__bnvs_if_in_p:nn ⋆
\__bnvs_if_in_p:nV ⋆
\__bnvs_if_in:nnTF ⋆
\__bnvs_if_in:nVTF ⋆
```

```
\__bnvs_if_in_p:nn {⟨subkey⟩} {⟨key⟩}
\__bnvs_if_in:nnTF {⟨subkey⟩} {⟨key⟩} {⟨true code⟩} {⟨false code⟩}
```

Convenient shortcuts to test for the existence of ⟨*subkey*⟩/⟨*key*⟩, it makes the code more concise and readable.

```
\prg_new_conditional:Npnn \__bnvs_if_in:nn #1 #2 { p, T, F, TF } {
  \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

```
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_in:nn {nV} { p, T, F, TF }
```

__bnvs_get:nnN*TF*

__bnvs_get:nnNTF {⟨*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.

```
\prg_new_conditional:Npnn \__bnvs_get:nnN #1 #2 #3 { p, T, F, TF } {
  \prop_get:NnNTF \g__bnvs_prop { #2 / #1 } #3 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_get:nnN {nV} { p, T, F, TF }
```

6.8 Functions with cache

```
\__bnvs_gput_cache:nnn
\__bnvs_gput_cache:(nnV|nVn)
\__bnvs_item_cache:nn
\__bnvs_gremove_cache:nn
\__bnvs_gremove_cache:nV
\__bnvs_gclear_cache:n
\__bnvs_v_gclear:
```

```
\__bnvs_gput_cache:nnn {⟨subkey⟩} {⟨key⟩} {⟨value⟩}
\__bnvs_item_cache:nn {⟨subkey⟩} {⟨key⟩}
\__bnvs_gremove_cache:nn {⟨subkey⟩} {⟨key⟩}
\__bnvs_clear_cache:n {⟨key⟩}
```

Wrapper over the functions above for ⟨*key*⟩/⟨*subkey*⟩.

```
\cs_new:Npn \__bnvs_gput_cache:nnn #1 {
  \__bnvs_gput:nnn { / #1 }
}
\cs_new:Npn \__bnvs_item_cache:nn #1 #2 {
  \prop_item:Nn \g__bnvs_prop { #2 / / #1 }
}
\cs_new:Npn \__bnvs_gremove_cache:nn  #1 {
  \__bnvs_gremove:nn { / #1 }
}
\cs_new:Npn \__bnvs_gclear_cache:n #1 {
  \clist_map_inline:nn { {}, A, L, Z, P, N, V } {
    \__bnvs_gremove_cache:nn { ##1 } { #1 }
  }
}
\cs_new:Npn \__bnvs_v_gclear: {
  \__bnvs_group_begin:
  \seq_clear:N \l__bnvs_a_seq
  \prop_map_inline:Nn \g__bnvs_prop {
    \regex_match:nnT { //V$ } { ##1 } {
      \seq_put_right:Nn \l__bnvs_a_seq { ##1 }
    }
  }
  \seq_map_inline:Nn \l__bnvs_a_seq {
    \prop_gremove:Nn \g__bnvs_prop { ##1 }
  }
  \__bnvs_group_end:
}
\cs_generate_variant:Nn \__bnvs_gremove_cache:nn { nV }
\cs_generate_variant:Nn \__bnvs_gput_cache:nnn { nVn, nnV }
```

```
\__bnvs_if_in_cache_p:nn ★
\__bnvs_if_in_cache_p:nV ★
\__bnvs_if_in_cache:nnTF ★
\__bnvs_if_in_cache:nVTF ★
```

```
\__bnvs_if_in_cache_p:n {⟨subkey⟩} {⟨key⟩}
\__bnvs_if_in_cache:nTF {⟨subkey⟩} {⟨key⟩} {⟨true code⟩} {⟨false code⟩}
```

Convenient shortcuts to test for the existence of ⟨*subkey*⟩/⟨*key*⟩, it makes the code more concise and readable.

```
\prg_new_conditional:Npnn \__bnvs_if_in_cache:nn #1 #2 { p, T, F, TF } {
  \__bnvs_if_in:nnTF { / #1 } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_in_cache:nn {nV} { p, T, F, TF }
```

__bnvs_get_cache:nnN*TF*

__bnvs_get_cache: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.

```
\prg_new_conditional:Npnn \__bnvs_get_cache:nnN #1 #2 #3 { p, T, F, TF } {
  \__bnvs_get:nnNTF { / #1 } { #2 } #3 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__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 ⟨*key*⟩–⟨*value*⟩ property list to store the named slide lists. The keys are ⟨*id*⟩!⟨*name*⟩.

```
\prop_new:N \g__bnvs_n_prop
```

(*End definition for \g__bnvs_n_prop.*)

__bnvs_n_gput:nn
__bnvs_n_gput:(nV|Vn)
__bnvs_n_item:n
__bnvs_n_gremove:n
__bnvs_n_gclear:

__bnvs_n_gput:nn {⟨*key*⟩} {⟨*value*⟩}
__bnvs_n_item:n {⟨*key*⟩}
__bnvs_n_gremove:n {⟨*key*⟩}
__bnvs_n_gclear:

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over LaTeX3 eponym functions.

```
\cs_new:Npn \__bnvs_n_gput:nn {
  \prop_gput:Nnn \g__bnvs_n_prop
}
\cs_new:Npn \__bnvs_n_item:n #1 {
  \prop_item:Nn \g__bnvs_n_prop { #1 }
}
\cs_new:Npn \__bnvs_n_gremove:n {
  \prop_gremove:Nn \g__bnvs_n_prop
}
\cs_new:Npn \__bnvs_n_gclear: {
  \prop_gclear:N \g__bnvs_n_prop
}
```

__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.

```
\prg_new_conditional:Npnn \__bnvs_n_get:nN #1 #2 { T, F, TF } {
  \prop_get:NnNTF \g__bnvs_n_prop { #1 } #2 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

```

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.

```
\regex_const:Nn \c__bnvs_name_regex {
  [[:alpha:]_][[:alnum:]_]*
}
```

(*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.

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

(*End definition for* `\c__bnvs_id_regex`.)

\c__bnvs_path_regex

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

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

(*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`.)

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

1: The range name including the slide ⟨*id*⟩ and question mark if any

2: slide ⟨*id*⟩ including the question mark

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

3: the path, if any.

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

\c__bnvs_colons_regex

For ranges defined by a colon syntax.

```
\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.

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

We start with '++' instrussions[2].

```
      \+\+
```

- 1: ⟨*name*⟩ of a slide range
- 2: ⟨*id*⟩ of a slide range including the exclamation mark

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

- 3: optionally followed by a dotted path

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

- 4: ⟨*name*⟩ of a slide range
- 5: ⟨*id*⟩ of a slide range including the exclamation mark

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

- 6: optionally followed by a dotted path

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

We continue with other expressions

- 7: the ⟨*++n*⟩ attribute

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

- 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.

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

- 9: the post increment

```
      | (\+)\+

    )?
  ) \s*
}
```

(*End definition for* `\c__bnvs_split_regex`.)

[2] At the same time an instruction and an expression... this is a synonym of exprection

6.8.3 beamer.cls interface

Work in progress.

```
\RequirePackage{keyval}
\define@key{beamerframe}{beanoves~id}[]{
  \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }
}
\AddToHook{env/beamer@frameslide/before}{
  \__bnvs_n_gclear:
  \__bnvs_v_gclear:
  \bool_set_true:N \l__bnvs_in_frame_bool
}
\AddToHook{env/beamer@frameslide/after}{
  \bool_set_false:N \l__bnvs_in_frame_bool
}
```

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.

```
\cs_new:Npn \__bnvs_range:nnnn #1 {
  \bool_if:NTF \l__bnvs_parse_bool {
    \__bnvs_n_gremove:n { #1 }
    \__bnvs_gclear:n { #1 }
    \__bnvs_do_range:nnnn { #1 }
  } {
    \__bnvs_if_in:nnTF A { #1 } {
      \use_none:nnn
    } {
      \__bnvs_if_in:nnTF L { #1 } {
        \use_none:nnn
      } {
        \__bnvs_if_in:nnTF Z { #1 } {
          \use_none:nnn
        } {
          \__bnvs_do_range:nnnn { #1 }
        }
      }
    }
  }
}
\cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
\cs_new:Npn \__bnvs_do_range:nnnn #1 #2 #3 #4 {
  \tl_if_empty:nTF { #3 } {
    \tl_if_empty:nTF { #2 } {
      \tl_if_empty:nTF { #4 } {
        \__bnvs_error:n { Not~a~range:~:~#1 }
      } {
        \__bnvs_gput:nnn Z { #1 } { #4 }
        \__bnvs_gput:nnn V { #1 } { \q_nil }
      }
    } {
      \__bnvs_gput:nnn A { #1 } { #2 }
      \__bnvs_gput:nnn V { #1 } { \q_nil }
      \tl_if_empty:nF { #4 } {
        \__bnvs_gput:nnn Z { #1 } { #4 }
        \__bnvs_gput:nnn L { #1 } { \q_nil }
      }
    }
  } {
    \tl_if_empty:nTF { #2 } {
      \__bnvs_gput:nnn L { #1 } { #3 }
      \tl_if_empty:nF { #4 } {
        \__bnvs_gput:nnn Z { #1 } { #4 }
        \__bnvs_gput:nnn A { #1 } { \q_nil }
        \__bnvs_gput:nnn V { #1 } { \q_nil }
      }
    } {
      \__bnvs_gput:nnn A { #1 } { #2 }
      \__bnvs_gput:nnn L { #1 } { #3 }
      \__bnvs_gput:nnn Z { #1 } { \q_nil }
      \__bnvs_gput:nnn V { #1 } { \q_nil }
    }
  }
}
```

__bnvs_parse:n

__bnvs_parse:n {⟨*key*⟩}

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

```
\cs_new:Npn \__bnvs_parse:n #1 {
  \peek_catcode_ignore_spaces:NTF \c_group_begin_token {
    \tl_if_empty:NTF \l__bnvs_root_tl {
      \__bnvs_error:n { Unexpected~list~at~top~level. }
    }
    \__bnvs_group_begin:
    \int_incr:N \l__bnvs_int
    \tl_set:Nx \l__bnvs_root_tl { \int_use:N \l__bnvs_int . }
    \cs_set:Npn \bnvs:nw ####1 ####2 \s_stop  {
      \regex_match:nnT { \S* } { ####2 } {
        \__bnvs_error:n { Unexpected~####2 }
      }
      \keyval_parse:nnn {
        \__bnvs_parse:n
      } {
        \__bnvs_parse:nn
      } { ####1 }
      \__bnvs_group_end:
    }
    \bnvs:nw
  } {
    \tl_if_empty:NTF \l__bnvs_root_tl {
      \__bnvs_id_name_set:nNNTF { #1 } \l__bnvs_id_tl \l__bnvs_name_tl {
        \__bnvs_parse_record:V \l__bnvs_name_tl
      } {
        \__bnvs_error:n { Unexpected~key:~#1 }
      }
    } {
      \int_incr:N \l__bnvs_int
      \__bnvs_parse_record:xn {
        \l__bnvs_root_tl . \int_use:N \l__bnvs_int
      } { #1 }
    }
    \use_none_delimit_by_s_stop:w
  }
  #1 \s_stop
}
```

__bnvs_parse_range:nNNN*TF*

__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.

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

This is not a list.

```
  \tl_clear:N \l__bnvs_a_tl
  \tl_clear:N \l__bnvs_b_tl
  \tl_clear:N \l__bnvs_c_tl
  \regex_split:NnNTF \c__bnvs_colons_regex { #4 } \l__bnvs_split_seq {
    \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_a_tl {
```

\l__bnvs_a_tl may contain the ⟨*start*⟩.

```
      \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_b_tl {
        \tl_if_empty:NTF \l__bnvs_b_tl {
```

This is a one colon range.

```
          \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
```

\l__bnvs_b_tl may contain the ⟨*length*⟩.

```
          \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_c_tl {
            \tl_if_empty:NTF \l__bnvs_c_tl {
```

A :: was expected:

```
              \__bnvs_error:n { Invalid~range~expression(1):~#4 }
            } {
              \int_compare:nNnT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
                \__bnvs_error:n { Invalid~range~expression(2):~#4 }
              }
              \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
```

\l__bnvs_c_tl may contain the ⟨*end*⟩.

```
              \seq_if_empty:NF \l__bnvs_split_seq {
                \__bnvs_error:n { Invalid~range~expression(3):~#4 }
              }
            }
          }
        } {
```

This is a two colon range.

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

\l__bnvs_c_tl contains the ⟨*end*⟩.

```
          \seq_pop_left:NNTF \l__bnvs_split_seq \l__bnvs_b_tl {
            \tl_if_empty:NTF \l__bnvs_b_tl {
              \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
```

\l_b_tl may contain the ⟨*length*⟩.

```
              \seq_if_empty:NF \l__bnvs_split_seq {
                \__bnvs_error:n { Invalid~range~expression(5):~#4 }
              }
            } {
              \__bnvs_error:n { Invalid~range~expression(6):~#4 }
            }
          } {
            \tl_clear:N \l__bnvs_b_tl
          }
        }
      }
    }
```

Providing both the ⟨*start*⟩, ⟨*length*⟩ and ⟨*end*⟩ of a range is not allowed, even if they happen to be consistent.

```
    \bool_if:nF {
      \tl_if_empty_p:N \l__bnvs_a_tl
      || \tl_if_empty_p:N \l__bnvs_b_tl
      || \tl_if_empty_p:N \l__bnvs_c_tl
    } {
      \__bnvs_error:n { Invalid~range~expression(7):~#3 }
    }
    \cs_set:Npn \:nnn ##1 ##2 ##3 {
      \__bnvs_group_end:
      \tl_set:Nn #1 { ##1 }
      \tl_set:Nn #2 { ##2 }
      \tl_set:Nn #3 { ##3 }
    }
    \exp_args:NVVV \:nnn \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
    \prg_return_true:
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
```

__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.

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

This is not a list.

```
  \bool_if:NTF \l__bnvs_parse_bool {
    \__bnvs_gclear:n { #1 }
    \__bnvs_gput:nnn V  { #1 } { 1 }
    \__bnvs_gput:nnn {} { #1 } { 1 }
  } {
    \__bnvs_gprovide:nnn V  { #1 } { 1 }
    \__bnvs_gprovide:nnn {} { #1 } { 1 }
  }
}
\cs_generate_variant:Nn \__bnvs_parse_record:n { V }
\cs_new:Npn \__bnvs_parse_record:nn #1 #2 {
```

This is not a list.

```
  \__bnvs_range_set:NNNnTF \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl { #2 } {
    \__bnvs_range:nVVV { #1 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
  } {
    \bool_if:NTF \l__bnvs_parse_bool {
      \__bnvs_gclear:n { #1 }
      \__bnvs_gput:nnn V  { #1 } { #2 }
      \__bnvs_gput:nnn {} { #1 } { #2 }
    } {
      \__bnvs_gprovide:nnn V  { #1 } { #2 }
      \__bnvs_gprovide:nnn {} { #1 } { #2 }
    }
  }
}
\cs_generate_variant:Nn \__bnvs_parse_record:nn { xn, Vn }
```

__bnvs_id_name_set:nNN*TF*

__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*⟩.

```
\prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
  \__bnvs_group_begin:
  \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
    #1
  } \l__bnvs_match_seq {
    \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
    \tl_if_empty:NTF #2 {
      \exp_args:NNNx
      \__bnvs_group_end:
      \tl_set:Nn #3 { \l__bnvs_id_last_tl #1 }
      \tl_set_eq:NN #2 \l__bnvs_id_last_tl
    } {
      \cs_set:Npn \:n ##1 {
        \__bnvs_group_end:
        \tl_set:Nn #2 { ##1 }
        \tl_set:Nn \l__bnvs_id_last_tl { ##1 }
      }
      \exp_args:NV
      \:n #2
      \tl_set:Nn #3 { #1 }
    }
    \prg_return_true:
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
```

```
\cs_new:Npn \__bnvs_parse:nn #1 #2 {
  \__bnvs_group_begin:
  \tl_set:Nn \l__bnvs_a_tl { #1 }
  \tl_put_left:NV \l__bnvs_a_tl \l__bnvs_root_tl
  \exp_args:NV
  \__bnvs_id_name_set:nNNTF \l__bnvs_a_tl \l__bnvs_id_tl \l__bnvs_name_tl {
    \regex_match:nnTF { \S } { #2 } {
      \peek_catcode_ignore_spaces:NTF \c_group_begin_token {
```

This is a comma separated list, go recursive.

```
        \__bnvs_group_begin:
        \tl_set:NV \l__bnvs_root_tl \l__bnvs_name_tl
        \int_set:Nn \l__bnvs_int { 0 }
        \cs_set:Npn \bnvs:nn ##1 ##2 \s_stop {
          \regex_match:nnT { \S } { ##2 } {
            \__bnvs_error:n { Unexpected~value~#2 }
          }
          \keyval_parse:nnn {
            \__bnvs_parse:n
          } {
            \__bnvs_parse:nn
          } { ##1 }
          \__bnvs_group_end:
        }
        \bnvs:nn
      } {
        \__bnvs_parse_record:Vn \l__bnvs_name_tl { #2 }
        \use_none_delimit_by_s_stop:w
      } #2 \s_stop
    } {
```

Empty value given: remove the reference.

```
      \exp_args:NV
      \__bnvs_gclear:n \l__bnvs_name_tl
      \exp_args:NV
      \__bnvs_n_gremove:n \l__bnvs_name_tl
    }
```

```
  } {
    \__bnvs_error:n { Invalid~key:~#2 }
  }
```

We export `\l__bnvs_id_tl`:

```
  \exp_args:NNNV
  \__bnvs_group_end:
  \tl_set:Nn \l__bnvs_id_last_tl \l__bnvs_id_last_tl
}
```

```
\cs_new:Npn \__bnvs_parse_prepare:N #1 {
  \tl_set:Nx #1 #1
  \bool_set_false:N \l__bnvs_parse_bool
  \bool_do_until:Nn \l__bnvs_parse_bool {
    \tl_if_in:NnTF #1 {%---[
    ]} {
      \regex_replace_all:nnNF { \[ ([^\]]*) \] } { { { \1 } } } #1 {
        \bool_set_true:N \l__bnvs_parse_bool
      }
    } {
      \bool_set_true:N \l__bnvs_parse_bool
    }
  }
  \tl_if_in:NnTF #1 {%---[
  ]} {
    \__bnvs_error:n { Unbalanced~%---[
    ]}
  } {
    \tl_if_in:NnT #1 { [%---]
    } {
      \__bnvs_error:n { Unbalanced~[ %---]
      }
    }
  }
}
```

`\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`.

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

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

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

At the top level, clear everything.

```
      \__bnvs_gclear:
    }
    \__bnvs_group_begin:
    \tl_clear:N \l__bnvs_root_tl
    \int_zero:N \l__bnvs_int
    \tl_set:Nn \l__bnvs_a_tl { #2 }
    \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, use the global definitions.

```
      \seq_if_empty:NF \g__bnvs_def_seq {
        \tl_put_left:Nx \l__bnvs_a_tl {
          \seq_use:Nn \g__bnvs_def_seq , ,
        }
      }
    }
    \__bnvs_parse_prepare:N \l__bnvs_a_tl
    \IfBooleanTF {#1} {
      \bool_set_false:N \l__bnvs_parse_bool
    } {
      \bool_set_true:N \l__bnvs_parse_bool
    }
    \exp_args:NnnV
    \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn } \l__bnvs_a_tl
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_set:Nn \l__bnvs_id_last_tl \l__bnvs_id_last_tl
    \ignorespaces
  }
}
```

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

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

6.8.5 Scanning named overlay specifications

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

`\beamer@frame`
`\beamer@masterdecode`

`\beamer@frame {`⟨*overlay specification*⟩`}`
`\beamer@masterdecode {`⟨*overlay specification*⟩`}`

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.

```
\cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
\cs_set:Npn \beamer@frame < #1 > {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
  \exp_args:NNNV
  \__bnvs_group_end:
  \__bnvs_beamer@frame < \l__bnvs_ans_tl >
}
\cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
\cs_set:Npn \beamer@masterdecode #1 {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
  \exp_args:NNV
  \__bnvs_group_end:
  \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
}
```

__bnvs_scan:nNN

`\__bnvs_scan:nNN {⟨named overlay expression⟩} ⟨eval⟩ ⟨tl variable⟩`

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

\l__bnvs_ans_tl The token list that will be appended to ⟨*tl variable*⟩ on return.

(*End definition for \l__bnvs_ans_tl.*)

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

(*End definition for \l__bnvs_int.*)

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

(*End definition for \l__bnvs_query_tl.*)

\l__bnvs_token_seq The ⟨*overlay expression*⟩ is split into the sequence of its tokens.

(*End definition for \l__bnvs_token_seq.*)

\l__bnvs_token_tl Storage for just one token.

(*End definition for \l__bnvs_token_tl.*)

```
\cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \seq_clear:N \l__bnvs_token_seq
```

Explode the ⟨*named overlay expression*⟩ into a list of tokens:

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

\scan_question:

`\scan_question:`

At top level state, scan the tokens of the ⟨*named overlay expression*⟩ looking for a '?' character.

```
  \cs_set:Npn \scan_question: {
    \seq_pop_left:NNT \l__bnvs_token_seq \l__bnvs_token_tl {
      \tl_if_eq:NnTF \l__bnvs_token_tl { ? } {
        \require_open:
      } {
        \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
        \scan_question:
    }
  }
}
```

\require_open: \require_open:

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

```
\cs_set:Npn \require_open: {
```

Get next token.

```
  \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
    \tl_if_eq:NnTF \l__bnvs_token_tl { ( %)
    } {
```

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

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

Record the forthcomming content in the `\l__bnvs_query_tl` variable, up to the next balancing ‘)‘.

```
      \tl_clear:N \l__bnvs_query_tl
      \require_close:
    } {
```

Ignore this token and loop.

```
      \require_open:
    }
  } {
```

End reached but no opening parenthesis found, raise.

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

\require_close: \require_close:

We found a ‘?(’, we record the forthcomming content in the `\l__bnvs_query_tl` variable, up to the next balancing ‘)‘.

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

Get next token.

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

We found a ‘(’, increment the depth and append the token to `\l__bnvs_query_tl`, then scan again for a).

```
      \int_incr:N \l__bnvs_int
      \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
      \require_close:
    } {
```

This is not a ‘(’.

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

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

```
          \int_decr:N \l__bnvs_int
          \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 `?`.

```
            \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
            \scan_question:
          } {
```

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.

```
            \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
            \require_close:
          }
        } {
```

The scanned token is not a ‘(’ nor a ‘)’, we append it as is to `\l__bnvs_query_tl` and look for a`)`.

```
          \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
          \require_close:
        }
      }
    } {
```

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 ‘)’.

```
      \__bnvs_error:x {Missing~%(---
        `)':~#1 }
      \tl_put_right:Nx \l__bnvs_query_tl {
        \prg_replicate:nn { \l__bnvs_int } {%(---
        )}
      }
      \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
    }
  }
```

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

```
  \scan_question:
  \exp_args:NNNV
  \__bnvs_group_end:
  \tl_put_right:Nn #3 \l__bnvs_ans_tl
}
```

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:NNN*TF*

`\__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.

```
\exp_args_generate:n { VVx }
\prg_new_conditional:Npnn \__bnvs_inp:NNN
    #1 #2 #3 { T, F, TF } {
  \__bnvs_group_begin:
  \exp_args:NNV
  \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

```
    \exp_args:Nx
    \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
      \tl_put_left:NV #2 { #1 }
    }
    \exp_args:NNnx
    \seq_set_split:Nnn \l__bnvs_split_seq . {
      \seq_item:Nn \l__bnvs_match_seq 4
    }
    \seq_remove_all:Nn \l__bnvs_split_seq { }
    \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
    \seq_if_empty:NTF \l__bnvs_split_seq {
```

No new integer path component is added.

```
      \cs_set:Npn \:nn ##1 ##2 {
        \__bnvs_group_end:
        \tl_set:Nn #1 { ##1 }
        \tl_set:Nn #2 { ##2 }
      }
      \exp_args:NVV \:nn #1 #2
    } {
```

Some new dotted path components are added.

```
      \cs_set:Npn \:nnn ##1 ##2 ##3 {
        \__bnvs_group_end:
        \tl_set:Nn #1 { ##1 }
        \tl_set:Nn #2 { ##2 }
        \seq_set_split:Nnn #3 . { ##3 }
        \seq_remove_all:Nn #3 { }
      }
      \exp_args:NVVx
      \:nnn #1 #2 {
        \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
      }
    }
    \prg_return_true:
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
```

__bnvs_resolve_n:NNN*TF*
__bnvs_resolve_n:TFF*TF*
__bnvs_resolve_x:NNN*TF*
__bnvs_resolve_x:TFF*TF*

__bnvs_resolve_x:TF {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_resolve_x:NNNTF ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_resolve_n:TF {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_resolve_n:NNNTF ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

When too many nested calls occurred, {⟨*false code*⟩} is executed directly otherwise {⟨*true code*⟩} will be executed once resolution has occurred. The ⟨*id tl var*⟩, ⟨*name tl var*⟩ and ⟨*path seq var*⟩ are meant to contain proper information on input and on output as well. On input, {⟨*id tl var*⟩} contains a frame id, {⟨*name tl var*⟩} contains a slide range name and {⟨*path seq var*⟩} contains the components of an integer path, possibly empty. On return, ⟨*id tl var*⟩ contains the frame id used, ⟨*name tl var*⟩ contains the resolved range name and ⟨*path seq var*⟩ contains the sequence of integer path components that could not be resolved.

To resolve a level of a named one slide specification ⟨*qualified name*⟩.⟨i_1⟩.⟨i_2⟩...⟨i_n⟩, we replace the shortest ⟨*qualified name*⟩.⟨i_1⟩.⟨i_2⟩...⟨i_k⟩ where 0≤k≤n by its definition ⟨*qualified name'*⟩.⟨j_1⟩...⟨j_p⟩ 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 ⟨*name tl var*⟩ 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 ⟨*name tl var*⟩ is replaced by this name, the ⟨*id tl var*⟩ and \l__bnvs_id_tl are updates accordingly and the ⟨*path seq var*⟩ is prepended with the integer path.

2. If ⟨*path seq var*⟩ is not empty, append to the right of ⟨*name tl var*⟩ 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 ⟨*path seq var*⟩ is empty, ⟨*name tl var*⟩ can contain anything.

```
\cs_new:Npn \__bnvs_resolve_x:TFF #1 #2 {
  \__bnvs_resolve_x:NNNTF
  \l__bnvs_id_tl
  \l__bnvs_name_tl
  \l__bnvs_path_seq {
    \seq_if_empty:NTF \l__bnvs_path_seq { #1 } { #2 }
  }
}
\prg_new_conditional:Npnn \__bnvs_resolve_x:NNN
    #1 #2 #3 { T, F, TF } {
```

```
  \__bnvs_group_begin:
```

Local variables:

- `\l__bnvs_a_tl` contains the name with a partial index path currently resolved.
- `\l__bnvs_a_seq` contains the index path components currently resolved.
- `\l__bnvs_b_tl` contains the resolution.
- `\l__bnvs_b_seq` contains the index path components to be resolved.

```
  \seq_set_eq:NN \l__bnvs_a_seq #3
  \seq_clear:N \l__bnvs_b_seq
  \cs_set:Npn \loop: {
    \__bnvs_call:TF {
      \tl_set_eq:NN \l__bnvs_a_tl #2
      \seq_if_empty:NTF \l__bnvs_a_seq {
        \__bnvs_get:nVNTF L \l__bnvs_a_tl \l__bnvs_b_tl {
          \cs_set:Nn \loop: { \return_true: }
        } {
          \resolve:F {
```

Unknown key ⟨`\l_a_tl`⟩`/A` or the value for key ⟨`\l_a_tl`⟩`/A` does not fit.

```
            \cs_set:Nn \loop: { \return_true: }
          }
        }
      } {
        \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
        \resolve:F {
          \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
            \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
          }
        }
      }
      \loop:
    } {
      \__bnvs_group_end:
      \prg_return_false:
    }
  }
  \cs_set:Npn \resolve:F ##1 {
    \__bnvs_get:nVNTF A \l__bnvs_a_tl \l__bnvs_b_tl {
      \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
        \tl_set_eq:NN #2 \l__bnvs_b_tl
        \seq_set_eq:NN #3 \l__bnvs_b_seq
        \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
        \seq_clear:N \l__bnvs_b_seq
      } {
        \seq_if_empty:NTF \l__bnvs_b_seq {
          \tl_set_eq:NN #2 \l__bnvs_b_tl
          \seq_clear:N #3
          \seq_clear:N \l__bnvs_a_seq
        } {
          ##1
        }
      }
    } {
    \__bnvs_get:nVNTF V \l__bnvs_a_tl \l__bnvs_b_tl {
      \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
        \tl_set_eq:NN #2 \l__bnvs_b_tl
        \seq_set_eq:NN #3 \l__bnvs_b_seq
        \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
        \seq_clear:N \l__bnvs_b_seq
      } {
        \seq_if_empty:NTF \l__bnvs_b_seq {
          \tl_set_eq:NN #2 \l__bnvs_b_tl
          \seq_clear:N #3
          \seq_clear:N \l__bnvs_a_seq
        } {
          ##1
        }
      }
    } { ##1 }
  }
  }
  \cs_set:Npn \return_true: {
    \seq_pop_left:NNTF #3 \l__bnvs_a_tl {
      \seq_if_empty:NTF #3 {
        \tl_clear:N \l__bnvs_b_tl
        \__bnvs_can_index:VTF #2 {
          \__bnvs_if_index:VVNTF #2 \l__bnvs_a_tl \l__bnvs_b_tl {
            \tl_set:NV #2 \l__bnvs_b_tl
          } {
            \tl_set:NV #2 \l__bnvs_a_tl
          }
        } {
          \tl_set:NV #2 \l__bnvs_a_tl
        }
      } {
        \__bnvs_error:x { Path~too~long:~#2.\l__bnvs_a_tl
          .\seq_use:Nn\l__bnvs_path_seq .}
      }
    } {
      \tl_clear:N \l__bnvs_b_tl
      \__bnvs_raw_value:VNT #2 \l__bnvs_b_tl {
        \tl_set:NV #2 \l__bnvs_b_tl
      }
    }
    \cs_set:Npn \:nnn ####1 ####2 ####3 {
      \__bnvs_group_end:
      \tl_set:Nn #1 { ####1 }
      \tl_set:Nn #2 { ####2 }
      \seq_set_split:Nnn #3 . { ####3 }
      \seq_remove_all:Nn #3 { }
    }
    \exp_args:NVVx
    \:nnn #1 #2 {
      \seq_use:Nn #3 .
    }
```

```
    \prg_return_true:
  }
  \loop:
}
```

```
\cs_new:Npn \__bnvs_resolve_n:TFF #1 #2 {
  \__bnvs_resolve_n:NNNTF
  \l__bnvs_id_tl
  \l__bnvs_name_tl
  \l__bnvs_path_seq {
    \seq_if_empty:NTF \l__bnvs_path_seq { #1 } { #2 }
  }
}
\prg_new_conditional:Npnn \__bnvs_resolve_n: { T, F, TF } {
  \__bnvs_resolve_n:NNNTF
  \l__bnvs_name_tl
  \l__bnvs_id_tl
  \l__bnvs_path_seq {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_new_conditional:Npnn \__bnvs_resolve_n_old:NNN
    #1 #2 #3 { T, F, TF } {
  \__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.

```
  \seq_set_eq:NN \l__bnvs_a_seq #3
  \seq_clear:N \l__bnvs_b_seq
  \cs_set:Npn \loop: {
    \__bnvs_call:TF {
      \tl_set_eq:NN \l__bnvs_a_tl #2
      \seq_if_empty:NTF \l__bnvs_a_seq {
        \__bnvs_get:nVNTF L \l__bnvs_a_tl \l__bnvs_b_tl {
          \cs_set:Npn \loop: { \return_true: }
        } {
          \seq_if_empty:NTF \l__bnvs_b_seq {
            \cs_set:Npn \loop: { \return_true: }
          } {
            \:F {
```

Unknown key `⟨\l_a_tl⟩/A` or the value for key `⟨\l_a_tl⟩/A` does not fit.

```
              \cs_set:Npn \loop: { \return_true: }
            }
          }
        }
```

```
      } {
        \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
        \:F {
          \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
            \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
          }
        }
      }
      \loop:
    } {
      \__bnvs_group_end:
      \prg_return_false:
    }
  }
  \cs_set:Npn \:F ##1 {
    \__bnvs_get:nVNTF A \l__bnvs_a_tl \l__bnvs_b_tl {
      \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
        \tl_set_eq:NN #2 \l__bnvs_b_tl
        \seq_set_eq:NN #3 \l__bnvs_b_seq
        \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
        \seq_clear:N \l__bnvs_b_seq
      } { ##1 }
    } { ##1 }
  }
  \cs_set:Npn \return_true: {
    \cs_set:Npn \:nnn ####1 ####2 ####3 {
      \__bnvs_group_end:
      \tl_set:Nn #1 { ####1 }
      \tl_set:Nn #2 { ####2 }
      \seq_set_split:Nnn #3 . { ####3 }
      \seq_remove_all:Nn #3 { }
    }
    \exp_args:NVVx
    \:nnn #1 #2 { \seq_use:Nn #3 . }
    \prg_return_true:
  }
  \loop:
}
\prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
    #1 #2 #3 { T, F, TF } {
  \__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.

```
  \tl_set_eq:NN \l__bnvs_id_tl #1
  \tl_set_eq:NN \l__bnvs_name_tl #2
  \seq_set_eq:NN \l__bnvs_path_seq #3
  \seq_set_eq:NN \l__bnvs_a_seq #3
  \seq_clear:N \l__bnvs_b_seq
```

```
\cs_set:Npn \loop: {
  \__bnvs_call:TF {
    \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_name_tl
    \seq_if_empty:NTF \l__bnvs_a_seq {
      \seq_if_empty:NTF \l__bnvs_b_seq {
        \group_end_return_true:
      } {
        \resolve:nF A {
          \resolve:nF V {
            \may_loop:
          }
        }
      }
    } {
      \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
      \resolve:nF A {
        \resolve:nF V {
          \may_loop:
        }
      }
    }
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
\cs_set:Npn \may_loop: {
   \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
     \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
     \loop:
  } {
     \group_end_return_true:
  }
}
\cs_set:Npn \resolve:nF ##1 ##2 {
  \__bnvs_get:nVNTF ##1 \l__bnvs_a_tl \l__bnvs_b_tl {
    \__bnvs_inp:NNNTF \l__bnvs_id_tl \l__bnvs_b_tl \l__bnvs_b_seq {
      \tl_set_eq:NN \l__bnvs_name_tl \l__bnvs_b_tl
      \seq_set_eq:NN \l__bnvs_path_seq \l__bnvs_b_seq
      \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
      \seq_clear:N \l__bnvs_b_seq
      \loop:
    } {
      \may_loop:
    }
  } {
    ##2
  }
}
\cs_set:Npn \group_end_return_true: {
  \cs_set:Npn \:nnn ####1 ####2 ####3 {
    \__bnvs_group_end:
    \tl_set:Nn #1 { ####1 }
    \tl_set:Nn #2 { ####2 }
```

```
      \seq_set_split:Nnn #3 . { ####3 }
      \seq_remove_all:Nn #3 { }
    }
    \exp_args:NVVx
    \:nnn \l__bnvs_id_tl \l__bnvs_name_tl { \seq_use:Nn \l__bnvs_path_seq . }
    \prg_return_true:
  }
  \loop:
}
```

`\__bnvs_resolve_n:NNN`*TF*
`\__bnvs_resolve_x:NNN`*TF*

`\__bnvs_resolve_n:NNNTF` ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
`\__bnvs_resolve_x:NNNTF` ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

When too many nested calls occurred, {⟨*false code*⟩} is executed directly. ⟨*id tl var*⟩, ⟨*name tl var*⟩ and ⟨*path seq var*⟩ are meant to contain proper information. On input, {⟨*id tl var*⟩} contains a frame id, {⟨*name tl var*⟩} contains a slide range name and {⟨*path seq var*⟩} contains the components of an integer path, possibly empty. On return, ⟨*id tl var*⟩ contains the frame id used, ⟨*name tl var*⟩ contains the resolved range name and ⟨*path seq var*⟩ contains the sequence of integer path components that could not be resolved.

To resolve a level of a named one slide specification ⟨*qualified name*⟩.$\langle c_1\rangle$.$\langle c_2\rangle$...$\langle c_n\rangle$, we replace the shortest ⟨*qualified name*⟩.$\langle c_1\rangle$.$\langle c_2\rangle$...$\langle c_k\rangle$ where 0≤k≤n by its definition ⟨*qualified name'*⟩.$\langle c'_1\rangle$...$\langle c'_p\rangle$ 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 ⟨*name tl var*⟩ 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 ⟨*name tl var*⟩ is replaced by this name, the ⟨*id tl var*⟩ and `\l__bnvs_id_tl` are updates accordingly and the ⟨*path seq var*⟩ is prepended with the integer path.

2. If ⟨*path seq var*⟩ is not empty, append to the right of ⟨*name tl var*⟩ 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 ⟨*path seq var*⟩ is empty, ⟨*name tl var*⟩ can contain anything, including an integer for example.

6.8.7 Evaluation bricks

We start by helpers.

`\__bnvs_round:nN`
`\__bnvs_round:N`

`\__bnvs_round:nN` {⟨*expression*⟩} ⟨*tl variable*⟩
`\__bnvs_round:N` ⟨*tl variable*⟩

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

```
\cs_new:Npn \__bnvs_round:nN #1 #2 {
  \tl_if_empty:nTF { #1 } {
    \tl_put_right:Nn #2 { 0 }
  } {
    \tl_put_right:Nx #2 { \fp_eval:n { round(#1) } }
  }
}
\cs_new:Npn \__bnvs_round:N #1 {
  \tl_if_empty:VTF #1 {
    \tl_set:Nn #1 { 0 }
  } {
    \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
  }
}
```

__bnvs_group_end_return_true:nnN
__bnvs_group_end_return_false:nn

__bnvs_group_end_return_true:nnN {⟨*subkey*⟩} {⟨*key*⟩} ⟨*tl variable*⟩
__bnvs_group_end_return_false:nn {⟨*subkey*⟩} {⟨*key*⟩}

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

```
\cs_set:Npn \__bnvs_group_end_return_true:nnN #1 #2 #3 {
  \tl_if_empty:NTF \l__bnvs_ans_tl {
    \__bnvs_group_end:
    \__bnvs_gremove_cache:nn { #1 } { #2 }
    \prg_return_false:
  } {
    \__bnvs_round:N \l__bnvs_ans_tl
    \__bnvs_gput_cache:nnV { #1 } { #2 } \l__bnvs_ans_tl
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #3 \l__bnvs_ans_tl
    \prg_return_true:
  }
}
\cs_set:Npn \__bnvs_group_end_return_false:nn #1 #2 {
  \__bnvs_group_end:
  \__bnvs_gremove_cache:nn { #1 } { #2 }
  \prg_return_false:
}
```

__bnvs_raw_first:nN*TF*
__bnvs_raw_first:(xN|VN)*TF*

__bnvs_raw_first:nNTF {⟨*name*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_get_cache:nnNTF A { #1 } #2 {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #2 #2
    \prg_return_true:
  } {
    \__bnvs_get:nnNTF A { #1 } \l__bnvs_a_tl {
      \tl_clear:N \l__bnvs_ans_tl
      \quark_if_nil:NTF \l__bnvs_a_tl {
        \__bnvs_gput:nnn A { #1 } { \q_no_value }
```

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

```
        \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
          \tl_put_right:Nn \l__bnvs_ans_tl { - }
          \tl_clear:N \l__bnvs_a_tl
          \__bnvs_raw_length:nNTF { #1 } \l__bnvs_a_tl {
            \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
            \tl_put_right:Nn \l__bnvs_ans_tl { + 1 }
            \__bnvs_group_end_return_true:nnN A { #1 } #2
          } {
            \__bnvs_error:n { Unavailable~length~for~#1~(\__bnvs_raw_first:nNTF/2) }
            \__bnvs_group_end_return_false:nn A { #1 }
          }
        } {
          \__bnvs_error:n { Unavailable~last~for~#1~(\__bnvs_raw_first:nNTF/1) }
          \__bnvs_group_end_return_false:nn A { #1 }
        }
      } {
        \quark_if_no_value:NTF \l__bnvs_a_tl {
          \__bnvs_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
            \__bnvs_group_end_return_true:nnN A { #1 } #2
          } {
            \__bnvs_group_end_return_false:nn A { #1 }
          }
        }
      }
    } {
      \__bnvs_group_end_return_false:nn A { #1 }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_raw_first:nN { VN, xN } { T, F, TF }
```

__bnvs_raw_length:nN*TF*

__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.

```
\prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_get_cache:nnNTF L { #1 } #2 {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #2 #2
    \prg_return_true:
  } {
    \__bnvs_get:nnNTF L { #1 } \l__bnvs_a_tl {
      \tl_clear:N \l__bnvs_ans_tl
      \quark_if_nil:NTF \l__bnvs_a_tl {
        \__bnvs_gput:nnn L { #1 } { \q_no_value }
```

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

```
        \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
          \tl_put_right:Nn \l__bnvs_ans_tl { - }
          \tl_clear:N \l__bnvs_a_tl
          \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
            \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
            \tl_put_right:Nn \l__bnvs_ans_tl { + 1 }
            \__bnvs_group_end_return_true:nnN L { #1 } #2
          } {
            \__bnvs_error:n { Unavailable~first~for~#1~(\__bnvs_raw_length:nNTF/2) }
            \__bnvs_group_end_return_false:nn L { #1 }
          }
        } {
          \__bnvs_error:n { Unavailable~last~for~#1~(\__bnvs_raw_length:nNTF/1) }
          \__bnvs_group_end_return_false:nn L { #1 }
        }
      } {
        \quark_if_no_value:NTF \l__bnvs_a_tl {
          \__bnvs_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
            \__bnvs_group_end_return_true:nnN L { #1 } #2
          } {
            \__bnvs_group_end_return_false:nn L { #1 }
          }
        }
      }
    } {
      \__bnvs_group_end_return_false:nn L { #1 }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_raw_length:nN { VN } { T, F, TF }
```

__bnvs_raw_last:nN*TF*

`\__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.

```
\prg_new_conditional:Npnn \__bnvs_raw_last:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_get_cache:nnNTF Z { #1 } #2 {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #2 #2
    \prg_return_true:
  }  {
    \__bnvs_get:nnNTF Z { #1 } \l__bnvs_a_tl {
      \tl_clear:N \l__bnvs_ans_tl
      \quark_if_nil:NTF \l__bnvs_a_tl {
        \__bnvs_gput:nnn Z { #1 } { \q_no_value }
```

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

```
        \tl_clear:N \l__bnvs_a_tl
        \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
          \tl_put_right:Nn \l__bnvs_ans_tl { + }
          \tl_clear:N \l__bnvs_b_tl
          \__bnvs_raw_length:nNTF { #1 } \l__bnvs_b_tl {
            \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_b_tl
            \tl_put_right:Nn \l__bnvs_ans_tl { - 1 }
            \__bnvs_group_end_return_true:nnN Z { #1 } #2
          } {
            \__bnvs_error:n { Unavailable~length~for~#1~(\__bnvs_raw_last:nNTF/1) }
            \__bnvs_group_end_return_false:nn Z { #1 }
          }
        } {
          \__bnvs_error:n { Unavailable~start~for~#1~(\__bnvs_raw_last:nNTF/1) }
          \__bnvs_group_end_return_false:nn Z { #1 }
        }
      } {
        \quark_if_no_value:NTF \l__bnvs_a_tl {
          \__bnvs_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
            \__bnvs_group_end_return_true:nnN Z { #1 } #2
          } {
            \__bnvs_group_end_return_false:nn Z { #1 }
          }
        }
      }
    } {
      \__bnvs_group_end_return_false:nn Z { #1 }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_raw_last:nN { VN } { T, F, TF }
```

__bnvs_if_range:nN*TF*

__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.

```
\prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_a_tl
  \tl_clear:N \l__bnvs_b_tl
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
    \int_compare:nNnT { \l__bnvs_a_tl } < 0 {
      \tl_set:Nn \l__bnvs_a_tl { 0 }
    }
    \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
```

Limited from above and below.

```
      \int_compare:nNnT { \l__bnvs_b_tl } < 0 {
        \tl_set:Nn \l__bnvs_b_tl { 0 }
      }
      \exp_args:NNNx
      \__bnvs_group_end:
      \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
      \prg_return_true:
    } {
```

Limited from below.

```
      \exp_args:NNNV
      \__bnvs_group_end:
      \tl_put_right:Nn #2 \l__bnvs_a_tl
      \tl_put_right:Nn #2 { - }
      \prg_return_true:
    }
  } {
    \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
```

Limited from above.

```
      \int_compare:nNnT { \l__bnvs_b_tl } < 0 {
        \tl_set:Nn \l__bnvs_b_tl { 0 }
      }
      \exp_args:NNNx
      \__bnvs_group_end:
      \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
      \prg_return_true:
    } {
      \__bnvs_raw_value:nNTF { #1 } \l__bnvs_b_tl {
```

Unlimited range.

```
        \exp_args:NNNx
        \__bnvs_group_end:
        \tl_put_right:Nn #2 { - }
        \prg_return_true:
      } {
        \__bnvs_group_end:
        \prg_return_false:
      }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_range:nN { VN } { T, F, TF }
```

__bnvs_if_previous:nN*TF*

__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.

```
\prg_new_conditional:Npnn \__bnvs_if_previous:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_get_cache:nnNTF P { #1 } #2 {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #2 #2
    \prg_return_true:
  } {
    \tl_clear:N \l__bnvs_ans_tl
    \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
      \tl_put_right:Nn \l__bnvs_ans_tl { -1 }
      \__bnvs_group_end_return_true:nnN P { #1 } #2
    } {
      \__bnvs_group_end_return_false:nn P { #1 }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_previous:nN { VN } { T, F, TF }
```

__bnvs_if_next:nN*TF*

__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.

```
\prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_get_cache:nnNTF N { #1 } #2 {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #2 #2
    \prg_return_true:
  } {
    \tl_clear:N \l__bnvs_ans_tl
    \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
      \tl_put_right:Nn \l__bnvs_ans_tl { +1 }
      \__bnvs_group_end_return_true:nnN P { #1 } #2
    } {
      \__bnvs_group_end_return_false:nn P { #1 }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_next:nN { VN } { T, F, TF }
```

__bnvs_raw_value:nN*TF*

__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.

```
\prg_new_conditional:Npnn \__bnvs_raw_value:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_get_cache:nnNTF V { #1 } #2 {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_put_right:Nn #2 #2
    \prg_return_true:
  } {
    \__bnvs_get:nnNTF V { #1 } \l__bnvs_a_tl {
      \tl_clear:N \l__bnvs_ans_tl
      \quark_if_nil:NTF \l__bnvs_a_tl {
        \__bnvs_gput:nnn V { #1 } { \q_no_value }
        \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
          \__bnvs_group_end_return_true:nnN V { #1 } #2
        } {
          \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
            \__bnvs_group_end_return_true:nnN V { #1 } #2
          } {
            \__bnvs_group_end_return_false:nn V { #1 }
          }
        }
      } {
        \quark_if_no_value:NTF \l__bnvs_a_tl {
          \__bnvs_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
            \__bnvs_group_end_return_true:nnN V { #1 } #2
          } {
            \__bnvs_group_end_return_false:nn V { #1 }
          }
        }
      }
    } {
      \__bnvs_group_end_return_false:nn V { #1 }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_raw_value:nN{ V } { T, F, TF }
```

`\__bnvs_can_index:nTF`
`\__bnvs_can_index:VTF`
`\__bnvs_if_index:nnNTF`
`\__bnvs_if_index:VVNTF`

`\__bnvs_can_index:nTF` {⟨*name*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\__bnvs_if_index:nnNTF` {⟨*name*⟩} {⟨*integer*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
`\__bnvs_can_index:nTF` {⟨*name*⟩} {⟨*integer*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\prg_new_conditional:Npnn \__bnvs_can_index:n #1 { p, T, F, TF } {
  \bool_if:nTF {
       \__bnvs_if_in_p:nn A { #1 }
    || \__bnvs_if_in_p:nn Z { #1 }
    || \__bnvs_if_in_p:nn V { #1 }
  } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_can_index:n { V } { p, T, F, TF }
\prg_new_conditional:Npnn \__bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
  \__bnvs_group_begin:
  \cs_set:Npn \group_end_return_true:n ##1 {
    \tl_put_right:Nn \l__bnvs_ans_tl { + #2 - 1 }
    \exp_args:NNV
    \__bnvs_group_end:
  \__bnvs_round:nN \l__bnvs_ans_tl #3
    \prg_return_true:
  }
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
```

Limited overlay set.

```
    \group_end_return_true:n { A }
  } {
    \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
```

Right limited overlay set.

```
      \group_end_return_true:n { Z }
    } {
      \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
```

Unlimited overlay set.

```
        \group_end_return_true:n { V }
      } {
        \__bnvs_group_end:
        \prg_return_false:
      }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_index:nnN { VVN } { T, F, TF }
```

__bnvs_if_n_value:nN*TF*
__bnvs_if_n_value:VN*TF*

__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. ⟨*false code*⟩ is never executed.

```
\prg_new_conditional:Npnn \__bnvs_if_n_value:nN #1 #2 { T, F, TF } {
  \__bnvs_n_get:nNF { #1 } #2 {
    \tl_set:Nn #2 { 1 }
    \__bnvs_n_gput:nn { #1 } { 1 }
  }
  \prg_return_true:
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_n_value:nN { VN } { T, F, TF }
```

__bnvs_if_n_index:nN*TF*
__bnvs_if_n_index:VN*TF*

__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.

```
\prg_new_conditional:Npnn \__bnvs_if_n_index:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_if_n_value:nNF { #1 } \l__bnvs_a_tl { }
  \exp_args:NNnV
  \__bnvs_group_end:
  \__bnvs_if_index:nnNTF { #1 } \l__bnvs_a_tl #2 {
    \prg_return_true:
  } {
    \__bnvs_group_begin:
    \__bnvs_raw_value:nNTF {#1} \l__bnvs_ans_tl {
      \tl_put_right:Nn \l__bnvs_ans_tl { + #2 - 1 }
      \exp_args:NNV
      \__bnvs_group_end:
      \__bnvs_round:Nn \l__bnvs_ans_tl
      \prg_return_true:
    } {
      \__bnvs_group_end:
      \prg_return_false:
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_n_index:nN { VN } { T, F, TF }
```

__bnvs_if_incr:nn*TF*
__bnvs_if_incr:nnN*TF*
__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.

```
\prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_if_append:nNTF { #2 } \l__bnvs_ans_tl {
    \int_compare:nNnTF \l__bnvs_ans_tl = 0 {
      \tl_clear:N \l__bnvs_ans_tl
      \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
        \__bnvs_group_end:
        \prg_return_true:
      } {
        \__bnvs_group_end:
```

```
        \prg_return_false:
      }
    } {
      \tl_put_right:Nn \l__bnvs_ans_tl { + }
      \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
        \__bnvs_round:N \l__bnvs_ans_tl
        \__bnvs_gput_cache:nnV V { #1 } \l__bnvs_ans_tl
        \__bnvs_group_end:
        \prg_return_true:
      } {
        \__bnvs_group_end:
        \prg_return_false:
      }
    }
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
\prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
  \__bnvs_if_incr:nnTF { #1 } { #2 } {
    \__bnvs_raw_value:nNTF { #1 } #3 {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }
```

__bnvs_if_n_incr:nn*TF*
__bnvs_if_n_incr:nnN*TF*
__bnvs_if_n_incr:(VnN|VVN)*TF*

__bnvs_if_n_incr:nnTF {⟨*name*⟩} {⟨*offset*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_if_n_incr:nnNTF {⟨*name*⟩} {⟨*offset*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\prg_new_conditional:Npnn \__bnvs_if_n_incr:nn #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_n_get:nNF { #1 } \l__bnvs_ans_tl {
    \tl_set:Nn \l__bnvs_ans_tl { 1 }
  }
  \tl_clear:N \l__bnvs_a_tl
  \__bnvs_if_append:nNTF { #2 } \l__bnvs_a_tl {
    \tl_put_right:Nn \l__bnvs_ans_tl { + }
    \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
    \__bnvs_round:N \l__bnvs_ans_tl
    \__bnvs_n_gput:nV { #1 } \l__bnvs_ans_tl
    \__bnvs_group_end:
```

```
    \prg_return_true:
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
\prg_new_conditional:Npnn \__bnvs_if_n_incr:nnN #1 #2 #3 { T, F, TF } {
  \__bnvs_if_n_incr:nnTF { #1 } { #2 } {
    \__bnvs_n_get:nNTF { #1 } #3 {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_n_incr:nnN { VnN, VVN } { T, F, TF }
```

__bnvs_if_post:nnN*TF*
__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.

```
\prg_new_conditional:Npnn \__bnvs_if_post:nnN #1 #2 #3 { T, F, TF } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
    \__bnvs_if_incr:nnTF { #1 } { #2 } {
      \exp_args:NNNV
      \__bnvs_group_end:
      \tl_put_right:Nn #3 \l__bnvs_ans_tl
      \prg_return_true:
    } {
      \__bnvs_group_end:
      \prg_return_false:
    }
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_post:nnN { VnN, VVN } { T, F, TF }
```

6.8.8 Evaluation

__bnvs_if_append:nN*TF*
__bnvs_if_append:(VN|xN)*TF*

__bnvs_if_append:nNTF {⟨*integer expression*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\l__bnvs_ans_tl To feed ⟨*tl variable*⟩ with.

(*End definition for* `\l__bnvs_ans_tl`.)

\l__bnvs_split_seq The sequence of catched 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 catched groups.

(*End definition for* `\l__bnvs_split_int`.)

```
\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 TEX group to define local functions and variables, sometimes another grouping level is used. The main TEX group is closed in the `\return_-...` functions.

```
\prg_new_conditional:Nnn \__bnvs_if_append:nN { T, F, TF } {
  \__bnvs_call:TF {
    \__bnvs_group_begin:
```

This TEX group is closed just before returning. Local variables:

```
    \int_zero:N  \l__bnvs_split_int
    \seq_clear:N \l__bnvs_split_seq
    \tl_clear:N  \l__bnvs_id_tl
    \tl_clear:N  \l__bnvs_name_tl
    \tl_clear:N  \l__bnvs_path_tl
    \tl_clear:N  \l__bnvs_group_tl
    \tl_clear:N  \l__bnvs_ans_tl
    \tl_clear:N  \l__bnvs_a_tl
```

Implementation:

```
    \regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq
    \int_set:Nn \l__bnvs_split_int { 1 }
    \tl_set:Nx \l__bnvs_ans_tl {
      \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
    }
```

\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.

```
    \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
      \tl_set:Nx ##2 {
        \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
      }
      \tl_if_empty:NTF ##2 {
        ##4 } {
        ##3
      }
    }
    \cs_set:Npn \fp_round: {
      \__bnvs_round:N \l__bnvs_ans_tl
    }
```

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

```
    \cs_set:Npn \group_end_return_false: {
      \cs_set:Npn \loop: {
        \__bnvs_group_end:
        \prg_return_false:
      }
    }
    \cs_set:Npn \group_end_return_false:x ##1 {
      \__bnvs_error:x { ##1 }
      \group_end_return_false:
    }
    \cs_set:Npn \resolve_n:T ##1 {
      \__bnvs_resolve_n:TFF {
        ##1
      } {
        \group_end_return_false:x { Too~many~dotted~components:~#1 }
      } {
        \group_end_return_false:x { Unknown~dotted~path:~#1 }
      }
    }
    \cs_set:Npn \resolve_x:T ##1 {
      \__bnvs_resolve_x:TFF {
        ##1
      } {
        \group_end_return_false:x { Too~many~dotted~components:~#1 }
      } {
        \group_end_return_false:x { Unknown~dotted~path:~#1 }
      }
    }
    \cs_set:Npn \:nn ##1 ##2 {
      \switch:nNTF { ##1 } \l__bnvs_id_tl { } {
        \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_last_tl
        \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
      }
      \switch:nNTF { ##2 } \l__bnvs_path_tl {
        \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
        \seq_remove_all:Nn \l__bnvs_path_seq { }
      } {
        \seq_clear:N \l__bnvs_path_seq
      }
    }
    \cs_set:cpn {.n?:TF} ##1 ##2 {
      \seq_get_right:NNTF \l__bnvs_path_seq \l__bnvs_b_tl {
        \exp_args:NV
        \str_if_eq:nnTF \l__bnvs_b_tl { n } {
          \seq_pop_right:NN \l__bnvs_path_seq \l__bnvs_b_tl
          ##1
        } { ##2 }
      } { ##2 }
    }
    \cs_set:cpn {...++n:} {
      \__bnvs_group_begin:
      \__bnvs_resolve_n:TFF {
        \tl_clear:N \l__bnvs_b_tl
        \__bnvs_if_n_incr:VnNTF \l__bnvs_name_tl { 1 } \l__bnvs_b_tl {
          \exp_args:NNNV
          \__bnvs_group_end:
          \tl_set:Nn \l__bnvs_b_tl \l__bnvs_b_tl
          \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
          \resolve_x:T {
            \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
          }
        } {
          \__bnvs_group_end:
        }
      } {
        \__bnvs_group_end:
        \group_end_return_false:x { Too~many~dotted~components:~#1 }
      } {
        \__bnvs_group_end:
        \group_end_return_false:
      }
    }
```

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.

```
    \cs_set:Npn \loop: {
      \int_compare:nNnTF {
        \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
      } {
        \switch:nNTF { 1 } \l__bnvs_name_tl {
          \:nn { 2 } { 3 }
          \use:c {.n?:TF} {
```

- Case `++...n`.

```
            \use:c { ...++n: }
          } {
```

- Case ++⟨*name*⟩⟨*integer path*⟩.

```
            \resolve_n:T {
              \tl_clear:N \l__bnvs_ans_tl
              \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
                \group_end_return_false:
              }
            }
          }
        } {
          \switch:nNTF 4 \l__bnvs_name_tl {
            \:nn { 5 } { 6 }
            \switch:nNTF 7 \l__bnvs_a_tl {
```

- Case ...++n.

```
              \use:c { ...++n: }
            } {
              \switch:nNTF 8 \l__bnvs_a_tl {
                \use:c { .n?:TF } {
```

- Case ....n+=⟨*integer*⟩.

```
\__bnvs_group_begin:
\__bnvs_resolve_n:TFF {
  \tl_clear:N \l__bnvs_b_tl
  \__bnvs_if_n_incr:VVNTF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_b_tl {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_set:Nn \l__bnvs_b_tl \l__bnvs_b_tl
    \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
    \resolve_x:T {
      \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
    }
  } {
    \__bnvs_group_end:
  }
} {
    \__bnvs_group_end:
    \group_end_return_false:x { Too~many~dotted~components:~#1 }
} {
  \__bnvs_group_end:
  \group_end_return_false:x { Unknown~dotted~path:~#1 }
}
                } {
```

- Case A+=⟨*integer*⟩.

```
\resolve_n:T {
  \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
    \group_end_return_false:
  }
}
                }
              } {
                \switch:nNTF 9 \l__bnvs_a_tl {
```

- Case ...++.

```
\resolve_n:T {
  \__bnvs_if_post:VnNF \l__bnvs_name_tl { 1 } \l__bnvs_ans_tl {
    \return_false:
  }
}
                } {
```

Only the path, branch according to the last component.

```
\seq_pop_right:NNTF \l__bnvs_path_seq \l__bnvs_b_tl {
  \exp_args:NV
  \str_case:nnF \l__bnvs_b_tl {
    { n } {
```

- Case ...n.

```
      \__bnvs_group_begin:
      \resolve_n:T {
        \exp_args:NNV
        \__bnvs_group_end:
        \__bnvs_if_n_value:nNTF \l__bnvs_name_tl \l__bnvs_b_tl {
          \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
          \resolve_x:T {
            \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
          }
        } {
\group_end_return_false:x { Undefined~dotted~path:~#1 }
        }
      }
    }
    { length } {
```

- Case ...length.

```
      \resolve_n:T {
        \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
          \group_end_return_false:
        }
      }
    }
    { last } {
```

- Case ...last.

```
      \resolve_n:T {
        \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
          \group_end_return_false:
        }
      }
    }
    { range } {
```

- Case ...range.

```
   \resolve_n:T {
    \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
     \cs_set_eq:NN \fp_round: \prg_do_nothing:
    } {
     \group_end_return_false:
    }
   }
  }
  { previous } {
```

- Case ...previous.

```
   \resolve_n:T {
    \__bnvs_if_previous:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
     \group_end_return_false:
    }
   }
  }
  { next } {
```

- Case ...next.

```
   \resolve_n:T {
    \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
     \group_end_return_false:
    }
   }
  }
 } {
```

- Case ...⟨*integer*⟩.

```
  \resolve_n:T {
   \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_b_tl \l__bnvs_ans_tl {
    \group_end_return_false:
   }
  }
 }
} {
```

- Case ....

```
 \resolve_n:T {
  \__bnvs_raw_value:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
   \group_end_return_false:
  }
 }
}
        }
       }
      }
     } {
```

No name. Unreachable code.

```
          }
        }
        \int_add:Nn \l__bnvs_split_int { 10 }
        \tl_put_right:Nx \l__bnvs_ans_tl {
          \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
        }
        \loop:
      } {
        \fp_round:
        \exp_args:NNNV
        \__bnvs_group_end:
        \tl_put_right:Nn #2 \l__bnvs_ans_tl
        \prg_return_true:
      }
    }
    \loop:
  } {
    \__bnvs_error:x { Too~many~calls:~ #1 }
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_append:nN { VN } { T, F, TF }
```

__bnvs_if_eval_query:nN*TF* `\__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.)

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

- 2: ⟨*first*⟩

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

- 3: second optional colon

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

- 4: ⟨*length*⟩

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

- 5: standalone ⟨*first*⟩

```
    | ( [^:]+ )
  ) \s* \Z
}
```

```
\prg_new_conditional:Npnn \__bnvs_if_eval_query:nN #1 #2 { T, F, TF } {
  \__bnvs_call_greset:
  \cs_set:Npn \return_true: {
    \prg_return_true:
  }
  \cs_set:Npn \return_false: {
    \prg_return_false:
  }
  \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
    #1
  } \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 emptyness of this variable.

```
\cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
  \tl_set:Nx ##2 {
    \seq_item:Nn \l__bnvs_match_seq { ##1 }
  }
  \tl_if_empty:NTF ##2 { ##4 } { ##3 }
}
\switch:nNTF 5 \l__bnvs_a_tl {
```

Single expression

```
  \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
    \return_true:
  } {
    \return_false:
  }
} {
  \switch:nNTF 2 \l__bnvs_a_tl {
    \switch:nNTF 4 \l__bnvs_b_tl {
      \switch:nNTF 3 \l__bnvs_c_tl {
```

⟨*first*⟩`::`⟨*last*⟩ range

```
        \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
          \tl_put_right:Nn #2 { - }
          \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
            \return_true:
          } {
            \return_false:
          }
        } {
          \return_false:
        }
      } {
```

⟨*first*⟩`:`⟨*length*⟩ range

```
        \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
          \tl_put_right:Nx #2 { - }
          \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1}
          \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
            \return_true:
          } {
            \return_false:
          }
        } {
          \return_false:
        }
      }
    } {
```

⟨*first*⟩`:` and ⟨*first*⟩`::` range

```
          \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
            \tl_put_right:Nn #2 { - }
            \return_true:
          } {
            \return_false:
          }
        }
      } {
        \switch:nNTF 4 \l__bnvs_b_tl {
          \switch:nNTF 3 \l__bnvs_c_tl {
```

🗨 ::⟨*last*⟩ range

```
            \tl_put_right:Nn #2 { - }
            \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
              \return_true:
            } {
              \return_false:
            }
          } {
            \__bnvs_error:x { Syntax~error(Missing~first):~#1 }
          }
        } {
```

🗨 : or :: range

```
          \seq_put_right:Nn #2 { - }
        }
      }
    }
  } {
```

Error

```
    \__bnvs_error:n { Syntax~error:~#1 }
    \return_false:
  }
}
```

__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.

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

(*End definition for* `\c__bnvs_comma_regex`.)
No other variable is used.

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

Local variables declaration

```
  \seq_clear:N \l__bnvs_query_seq
  \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`.

```
  \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.

```
  \seq_map_inline:Nn \l__bnvs_query_seq {
    \tl_clear:N \l__bnvs_ans_tl
    \__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {
      \seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl
    } {
      \seq_map_break:n {
        \__bnvs_fatal:n { Circular/Undefined~dependency~in~#1}
      }
    }
  }
```

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

```
  \exp_args:NNNx
  \__bnvs_group_end:
  \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
}
\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.

```
\NewDocumentCommand \BeanovesEval { o m } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_eval:nN { #2 } \l__bnvs_ans_tl
  \IfValueTF { #1 } {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_set:Nn #1 \l__bnvs_ans_tl
  } {
    \exp_args:NV
    \__bnvs_group_end: \l__bnvs_ans_tl
  }
}
```

6.8.9 Reseting counters

\BeanovesReset
\BeanovesReset*

\beanovesReset [⟨*first value*⟩] {⟨*key*⟩}
\beanovesReset* [⟨*first value*⟩] {⟨*key*⟩}

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

```
\NewDocumentCommand \BeanovesReset { s O{} m } {
  \__bnvs_id_name_set:nNNTF { #3 } \l__bnvs_id_tl \l__bnvs_name_tl {
    \IfBooleanTF { #1 } {
      \exp_args:NV \__bnvs_reset_all:nn
    } {
      \exp_args:NV \__bnvs_reset:nn
    }
    \l__bnvs_name_tl { #2 }
  } {
    \__bnvs_warning:n { Unknown~name:~#1 }
  }
  \ignorespaces
}
```

__bnvs_reset:nn
__bnvs_reset_all:nn

__bnvs_reset:nn {⟨*key*⟩} {⟨*first value*⟩}

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

```
\cs_new:Npn \__bnvs_reset_all:nn #1 #2 {
  \bool_if:nTF {
       \__bnvs_if_in_p:nn A { #1 }
    || \__bnvs_if_in_p:nn Z { #1 }
    || \__bnvs_if_in_p:nn V { #1 }
  } {
    \__bnvs_gremove_cache:nn A { #1 }
    \__bnvs_gremove_cache:nn L { #1 }
    \__bnvs_gremove_cache:nn Z { #1 }
    \__bnvs_gremove_cache:nn P { #1 }
    \__bnvs_gremove_cache:nn N { #1 }
    \__bnvs_gremove_cache:nn V { #1 }
    \tl_if_empty:nF { #2 } {
      \__bnvs_gput_cache:nnn V { #1 } { #2 }
    }
  } {
    \__bnvs_warning:n { Unknown~name:~#1 }
  }
}
\cs_new:Npn \__bnvs_reset:nn #1 #2 {
  \__bnvs_if_in:nnTF V { #1 } {
    \__bnvs_gremove_cache:nn V { #1 }
    \tl_if_empty:nF { #2 } {
      \__bnvs_gput_cache:nnn V { #1 } { #2 }
    }
  } {
    \__bnvs_warning:n { Unknown~name:~#1 }
  }
}
\makeatother
\ExplSyntaxOff
```