beamer named overlay specification with beanoves

Jérôme Laurens

v1.0   2022/10/28

Abstract

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

Contents

1 Minimal example

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

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

On line 4, we use the \Beanoves command to declare named slide ranges. On line 5, we declare a slide range named ‘A’, starting at slide 1 and with length 2. On line 12, the extended *named overlay specification* `?(A.1)` stands for 1, on line 15, `?(A.2)` stands for 2 whereas on line 18, `?(A.3)` stands for 3. On line 6, we declare a second slide range named ‘B’, starting after the 2 slides of ‘A’ namely 3. Its length is 3 meaning that its last slide number is 5, thus each `?(B.last)` is replaced by 5. The next slide number after slide range ‘B’ is 6 which is also the start of the third slide range due to line 7.

2 Named slide lists

2.1 Presentation

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

2.2 Defining named slide lists

In order to define named slide lists, we can either use the `\Beanoves` command below before a beamer frame environment, or use the `beanoves` option of this environment. The value of the `beanoves` option is similar to the argument of the `\Beanoves` commands, but the latter takes precedence on the former. This behaviour may be useful to input the very same source code into different frames and have different combinations of slides.

`beanoves`

```
beanoves = {
  ⟨name_1⟩=⟨spec_1⟩,
  ⟨name_2⟩=⟨spec_2⟩,
  ...,
  ⟨name_n⟩=⟨spec_n⟩,
}
```

`\Beanoves`

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

The keys $\langle \mathit{name}_i \rangle$ are the slide lists names, they are case sensitive and must contain no spaces nor ‘`/`’ character. In order to avoid name conflicts with floating point functions, it is suggested to let them contain at least an uppercase letter ot an underscore. When the same key is used multiple times, only the last one is taken into account. Possible values for $\langle \mathit{spec}_i \rangle$ are the *slide range specifiers* ⟨*first*⟩, ⟨*first*⟩:⟨*length*⟩, ⟨*first*⟩::⟨*last*⟩, :⟨*length*⟩::⟨*last*⟩ where ⟨*first*⟩, ⟨*length*⟩ and ⟨*last*⟩ are algebraic expression possibly involving any integer valued named overlay specifications defined below.

Also possible values are *slide list specifiers* which are comma separated list of *slide range specifiers* and *slide list specifier* between square brackets. The definition

```
⟨name⟩=[⟨spec_1⟩,⟨spec_2⟩,...,⟨spec_n⟩],
```

is a convenient shortcut for

$\langle name\rangle$.1=$\langle spec_1\rangle$,
$\langle name\rangle$.2=$\langle spec_2\rangle$,
...,
$\langle name\rangle$.n=$\langle spec_n\rangle$.

The rules above can apply individually to each
$\langle name\rangle$.i=$\langle spec_i\rangle$.

Moreover we can go deeper: the definition
$\langle name\rangle$=[[$\langle spec_{1.1}\rangle$, $\langle spec_{1.2}\rangle$],[[$\langle spec_{2.1}\rangle$, $\langle spec_{2.2}\rangle$]]

happens to be a convenient shortcut for
$\langle name\rangle$.1.1=$\langle spec_{1.1}\rangle$,
$\langle name\rangle$.1.2=$\langle spec_{1.2}\rangle$,
$\langle name\rangle$.2.1=$\langle spec_{2.1}\rangle$,
$\langle name\rangle$.2.2=$\langle spec_{2.2}\rangle$

and so on.

3 Named overlay specifications

3.1 Named slide ranges

When *slide range specifications* are used, the named overlay specifications are detailled in the tables below together with their replacement meaning value as beamer standard overlay specification.

$\langle name\rangle$ == [i, $i+1$, $i+2,\ldots$]	
syntax	**meaning**
$\langle$**name**$\rangle$.1	i
$\langle$**name**$\rangle$.2	$i+1$
$\langle$**name**$\rangle$.$\langle$**integer**$\rangle$	$i+\langle integer\rangle-1$

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

```
\Beanoves {
  A = 3:6,
}
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\ttfamily
\BeanovesEval(A.1) ==3,
\BeanovesEval(A.2) ==4,
\BeanovesEval(A.-1)==1,
\end{frame}
```

When the slide range has been given a length or an end, like in the frame example below, we also have

$\langle name\rangle$ == [i, $i+1,\ldots,$ j]			
syntax	**meaning**	**example**	**output**
$\langle$**name**$\rangle$.**length**	$j-i+1$	A.length	6
$\langle$**name**$\rangle$.**last**	j	A.last	8
$\langle$**name**$\rangle$.**next**	$j+1$	A.next	9
$\langle$**name**$\rangle$.**range**	i ''-'' j	A.range	3-8

```
\Beanoves {
  A = 3:6, % or equivalently A = 3::8 or A = :6::8,

}
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\ttfamily
\BeanovesEval(A.1)      == 3,
\BeanovesEval(A.length) == 6,
\BeanovesEval(A.last)   == 8,
\BeanovesEval(A.next)   == 9,
\BeanovesEval(A.range)  == 3-8,
\end{frame}
```

Using these specifications on unfinite named slide ranges is unsupported. Finally each named slide range has a dedicated counter ⟨*name*⟩`.n` which is some kind of variable that can be used and incremented[1].

⟨*name*⟩.n : use the position of the counter

⟨*name*⟩.n+=⟨*integer*⟩ : advance the counter by ⟨*integer*⟩ and use the new position

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

Notice that "`.n`" can generally be omitted.

3.2 Named slide lists

After the definition
⟨*name*⟩=[⟨*spec*$_1$⟩,⟨*spec*$_2$⟩,...,⟨*spec*$_n$⟩]
the rules of the previous section apply recursively to each individual declaration
⟨*name*⟩.*i*=⟨*spec*$_i$⟩.

4 ?(...) query expressions

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

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

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

[1] This is actually an experimental feature.

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

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

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

5 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.

5.1 Package declarations

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

5.2 logging

Utility message.

```
\msg_new:nnn { beanoves } { :n } { #1 }
\msg_new:nnn { beanoves } { :nn } { #1~(#2) }
```

5.3 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.

5.4 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_current_tl
\tl_set:Nn \l__bnvs_id_current_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
\int_new:N \g__bnvs_call_int
\int_new:N \l__bnvs_depth_int
\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_no_counter_bool
\bool_new:N \l__bnvs_no_range_bool
\bool_new:N \l__bnvs_in_frame_bool
\bool_set_false:N \l__bnvs_in_frame_bool
```

5.5 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:
  }
}
```

5.6 Overlay specification

5.6.1 In slide range definitions

\g__bnvs_prop ⟨*key*⟩–⟨*value*⟩ property list to store the named slide lists. The basic keys are, assuming ⟨*id*⟩!⟨*name*⟩ is a fully qualified slide list name,

⟨***id***⟩**!**⟨***name***⟩**/A** for the first index

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

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

⟨***id***⟩**!**⟨***name***⟩**/C** for the counter value, when used

⟨***id***⟩**!**⟨***name***⟩**/C0** for initial value of the counter (when reset)

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

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

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

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

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

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

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

(*End definition for* \g__bnvs_prop.)

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

```
\__bnvs_gput:nn {⟨key⟩} {⟨value⟩}
\__bnvs_gprovide:nn {⟨key⟩} {⟨value⟩}
\__bnvs_item:n {⟨key⟩}
\__bnvs_get:n {⟨key⟩} ⟨tl variable⟩
\__bnvs_gremove:n {⟨key⟩}
\__bnvs_gclear:n {⟨key⟩}
\__bnvs_gclear_cache:n {⟨key⟩}
\__bnvs_gclear:
```

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

```
\cs_new:Npn \__bnvs_gput:nn #1 #2 {
  \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
}
\cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
  \prop_if_in:NnF \g__bnvs_prop { #1 } {
    \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
  }
}
\cs_new:Npn \__bnvs_item:n {
  \prop_item:Nn \g__bnvs_prop
}
\cs_new:Npn \__bnvs_get:nN {
  \prop_get:NnN \g__bnvs_prop
}
\cs_new:Npn \__bnvs_gremove:n {
  \prop_gremove:Nn \g__bnvs_prop
}
\cs_new:Npn \__bnvs_gclear:n #1 {
  \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
    \__bnvs_gremove:n { #1 / ##1 }
  }
}
\cs_new:Npn \__bnvs_gclear_cache:n #1 {
  \clist_map_inline:nn { /A, /L, /Z, /N } {
    \__bnvs_gremove:n { #1 / ##1 }
  }
}
\cs_new:Npn \__bnvs_gclear: {
  \prop_gclear:N \g__bnvs_prop
}
\cs_generate_variant:Nn \__bnvs_gput:nn { nV }
\cs_generate_variant:Nn \__bnvs_gprovide:nn { nV }
```

```
\__bnvs_if_in_p:n ⋆
\__bnvs_if_in_p:V ⋆
\__bnvs_if_in:nTF ⋆
\__bnvs_if_in:VTF ⋆
```

```
\__bnvs_if_in_p:n {⟨key⟩}
\__bnvs_if_in:nTF {⟨key⟩} {⟨true code⟩} {⟨false code⟩}
```

Convenient shortcuts to test for the existence of some key, it makes the code more concise and readable.

```
\prg_new_conditional:Npnn \__bnvs_if_in:n #1 { p, T, F, TF } {
  \prop_if_in:NnTF \g__bnvs_prop { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn \__bnvs_if_in:n {V} { p, T, F, TF }
```

__bnvs_get:nN*TF*
__bnvs_get:nnN*TF*

__bnvs_get:nNTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_get:nnNTF {⟨*id*⟩} {⟨*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:nN #1 #2 { T, F, TF } {
  \prop_get:NnNTF \g__bnvs_prop { #1 } #2 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

5.6.2 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 {
  (?: \. [+-]? \d+ )*
}
```

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

\c__bnvs_key_regex
\c__bnvs_A_key_Z_regex

A key is the name of a slide range possibly followed by positive integer attributes using a dot syntax. The '`A_key_Z`' variant matches the whole string.

```
\regex_const:Nn \c__bnvs_key_regex {
  \ur{c__bnvs_id_regex} ?
  \ur{c__bnvs_name_regex}
  \ur{c__bnvs_path_regex}
}
\regex_const:Nn \c__bnvs_A_key_Z_regex {
```

2: slide ⟨*id*⟩

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

4: The range name

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

5: the path, if any.

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

```

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

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

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

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

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

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

Capture groups:

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

```
    ( [^\] %[---
    ]*? )
  \s* \] \Z
}
```

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

`\c__bnvs_split_regex` Used to parse slide list overlay specifications in queries. Next are the 10 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

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

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

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

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

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

- 3: optionally followed by an integer path

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

We continue with other expressions

- 4: qualified ⟨*name*⟩ of a slide range,
- 5: ⟨*id*⟩ of a slide range plus the exclamation mark (to manage void ⟨*id*⟩)

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

- 6: optionally followed by an integer path

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

Next comes another branching

```
    (?:
```

- 7: the ⟨*length*⟩ attribute

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

- 8: the ⟨*last*⟩ attribute

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

- 9: the ⟨*next*⟩ attribute

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

- 10: the ⟨*range*⟩ attribute

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

- 11: the ⟨*n*⟩ attribute

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

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

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

- 13: the post increment

```
      | (\+)\+ )?
    )?
  ) \s*
}
```

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

5.6.3 beamer.cls interface

Work in progress.

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

5.6.4 Defining named slide ranges

`\__bnvs_parse:Nnn`

`\__bnvs_parse:Nnn ⟨command⟩ {⟨key⟩} {⟨definition⟩}`

Auxiliary function called within a group. ⟨*key*⟩ is the slide range key, including eventually a dotted integer path and a slide identifier, ⟨*definition*⟩ is the corresponding definition. ⟨*command*⟩ is `\__bnvs_range:nVVV` at runtime.

`\l__bnvs_match_seq` Local storage for the match result.

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

`\__bnvs_range:nnnn`
`\__bnvs_range:nVVV`
`\__bnvs_range_alt:nnnn`
`\__bnvs_range_alt:nVVV`
`\__bnvs_range:Nnnnn`

`\__bnvs_range:nnnn {⟨key⟩} {⟨first⟩} {⟨length⟩} {⟨last⟩}`
`\__bnvs_range_alt:nnnn {⟨key⟩} {⟨first⟩} {⟨length⟩} {⟨last⟩}`
`\__bnvs_range:Nnnnn ⟨cmd⟩ {⟨key⟩} {⟨first⟩} {⟨length⟩} {⟨last⟩}`

Auxiliary function called within a group. Setup the model to define a range. The alt variant does not override an already existing value.

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

```
\cs_new:Npn \__bnvs_range:Nnnnn #1 #2 #3 #4 #5 {
  \tl_if_empty:nTF { #3 } {
    \tl_if_empty:nTF { #4 } {
      \tl_if_empty:nTF { #5 } {
        \msg_error:nnn { beanoves } { :n } { Not~a~range:~:~#2 }
      } {
        #1 { #2/Z } { #5 }
      }
    } {
      #1 { #2/L } { #4 }
      \tl_if_empty:nF { #5 } {
        #1 { #2/Z } { #5 }
        #1 { #2/A } { #2.last - (#2.length) + 1 }
      }
    }
  } {
    #1 { #2/A } { #3 }
    \tl_if_empty:nTF { #4 } {
      \tl_if_empty:nF { #5 } {
        #1 { #2/Z } { #5 }
        #1 { #2/L } { #2.last - (#2.1) + 1 }
      }
    } {
      #1 { #2/L } { #4 }
      #1 { #2/Z } { #2.1 + #2.length - 1 }
    }
  }
}
\cs_new:Npn \__bnvs_range:nnnn #1 {
  \__bnvs_gclear:n { #1 }
  \__bnvs_range:Nnnnn \__bnvs_gput:nn { #1 }
}
\cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
\cs_new:Npn \__bnvs_range_alt:nnnn #1 {
  \__bnvs_gclear_cache:n { #1 }
  \__bnvs_range:Nnnnn \__bnvs_gprovide:nn { #1 }
}
\cs_generate_variant:Nn \__bnvs_range_alt:nnnn { nVVV }
```

__bnvs_parse:Nn

__bnvs_parse:Nn ⟨*command*⟩ {⟨*key*⟩}

Define a hidden range, for which slides are never shown. This is useful to conditionally show or hide a sequence of slides.

```
\cs_new:Npn \__bnvs_parse:Nn #1 #2 {
  \__bnvs_group_begin:
  \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
    \exp_args:Nx \__bnvs_gput:nn { \l__bnvs_name_tl/ } { }
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
  } {
    \msg_error:nnn { beanoves } { :n } { Unexpected~key:~#2 }
    \__bnvs_group_end:
  }
}
```

__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 }
\cs_new:Npn \__bnvs_range_set:NNNn #1 #2 #3 #4 {
  \__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:NnN \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:

```
\msg_error:nnn { beanoves } { :n } { Invalid~range~expression(1):~#4 }
     } {
      \int_compare:nNnT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
\msg_error:nnn { beanoves } { :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 {
\msg_error:nnn { beanoves } { :n } { Invalid~range~expression(3):~#4 }
      }
     }
    }
   } {
```

This is a two colon range.

```
    \int_compare:nNnT { \tl_count:N \l__bnvs_b_tl } > { 1 } {
\msg_error:nnn { beanoves } { :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 {
\msg_error:nnn { beanoves } { :n } { Invalid~range~expression(5):~#4 }
      }
     } {
\msg_error:nnn { beanoves } { :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
  } {
\msg_error:nnn { beanoves } { :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
}
```

`\__bnvs_do_parse:Nnn`

`\__bnvs_do_parse:Nnn` ⟨*command*⟩ {⟨*full name*⟩}

Auxiliary function for `\__bnvs_parse:Nn`. ⟨*command*⟩ is `\__bnvs_range:nVVV` at runtime and must have signature `nVVV`.

```
\cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
\cs_new:Npn \__bnvs_do_parse:Nnn #1 #2 #3 {
```

This is not a list.

```
\__bnvs_range_set:NNNn \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl { #3 }
  #1 { #2 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
}
\cs_generate_variant:Nn \__bnvs_do_parse:Nnn { Nxn, Non }
```

`\__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_current_tl #1 }
      \tl_set_eq:NN #2 \l__bnvs_id_current_tl
    } {
      \cs_set:Npn \:n ##1 {
        \__bnvs_group_end:
        \tl_set:Nn #2 { ##1 }
        \tl_set:Nn \l__bnvs_id_current_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:Nnn #1 #2 #3 {
  \__bnvs_group_begin:
  \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
    \regex_extract_once:NnNTF \c__bnvs_list_regex {
      #3
    } \l__bnvs_match_seq {
```

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

```
      \exp_args:NNx
      \seq_set_from_clist:Nn \l__bnvs_match_seq {
        \seq_item:Nn \l__bnvs_match_seq { 2 }
      }
      \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
        \__bnvs_do_parse:Nxn #1  { \l__bnvs_name_tl.##1 } { ##2 }
      }
    } {
      \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
    }
  } {
    \msg_error:nnn { beanoves } { :n } { Invalid~key:~#2 }
  }
```

We export `\l__bnvs_id_tl`:

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

\Beanoves

\Beanoves {⟨*key--value list*⟩}

The keys are the slide range specifiers. When no value is provided, it defaults to 1. On the contrary, ⟨*key–value*⟩ items are parsed by __bnvs_parse:Nnn.

```
\NewDocumentCommand \Beanoves { sm } {
  \tl_if_eq:NnT \@currenvir { document } {
    \__bnvs_gclear:
  }
  \IfBooleanTF {#1} {
    \keyval_parse:nnn {
      \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
    } {
      \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
    }
  } {
    \keyval_parse:nnn {
      \__bnvs_parse:Nn \__bnvs_range:nVVV
    } {
      \__bnvs_parse:Nnn \__bnvs_range:nVVV
    }
  }
  { #2 }
  \ignorespaces
}
```

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

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

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

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

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

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

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

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

\l__bnvs_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_depth_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.

```
    \msg_fatal:nnx { beanoves } { :n } {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_depth_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_depth_int
          \int_compare:nNnTF \l__bnvs_depth_int = 0 {
```

The depth level has reached 0: we found our balancing parenthesis of the `?(...)` instruction. We can append the evaluated slide ranges token list to `\l_ans_tl` and 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 ‘`)`’.

```
      \msg_error:nnx { beanoves } { :n } {Missing~%(---
        `)':~#1 }
      \tl_put_right:Nx \l__bnvs_query_tl {
        \prg_replicate:nn { \l__bnvs_depth_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

5.6.6 Resolution

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

__bnvs_extract_key:NNN*TF*

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

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

```
\exp_args_generate:n { VVx }
\prg_new_conditional:Npnn \__bnvs_extract_key: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 integer 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 .
      }
```

```
⟨/!gubed⟩
% \end{bnvs.gobble}
%    \begin{macrocode}
    }
    \prg_return_true:
  } {
    \__bnvs_group_end:
    \prg_return_false:
  }
}
```

__bnvs_resolve:NNN*TF*

`\__bnvs_resolve: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 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 path, $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ is turned into $\langle name_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ where $\langle name_0\rangle.\langle i_1\rangle$ is $\langle name_1\rangle$, then $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$ where $\langle name_1\rangle.\langle i_2\rangle$ is $\langle name_2\rangle$... If the above rule does not apply, $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ may turn into $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$ when $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle$ is $\langle name_2\rangle$... The algorithm is not yet more clever. The resolution algorithm is quite straightforward:

1. If ⟨*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. None of the tl variables must be one of `\l_a_tl`, `\l_b_tl` or `\l_c_tl`. None of the seq variables must be one of `\l_a_seq`, `\l_b_seq`.

```
\prg_new_conditional:Npnn \__bnvs_resolve: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 {
        \exp_args:Nx
        \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
          \cs_set:Nn \loop: { \return_true: }
        } {
          \get_extract: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 . }
        \get_extract: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 \get_extract:F ##1 {
    \exp_args:Nx
    \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
      \__bnvs_extract_key: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:
}
```

`\__bnvs_resolve_n:NNNTF`*TF*

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

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

```
\prg_new_conditional:Npnn \__bnvs_resolve_n: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 {
        \exp_args:Nx
        \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
          \cs_set:Nn \loop: { \return_true: }
        } {
          \seq_if_empty:NTF \l__bnvs_b_seq {
            \cs_set:Nn \loop: { \return_true: }
          } {
            \get_extract: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 . }
        \get_extract: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 \get_extract:F ##1 {
    \exp_args:Nx
    \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
      \__bnvs_extract_key: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:
}
```

__bnvs_resolve:NNNNTF*TF*

`\__bnvs_resolve:NNNNTF` ⟨*cs:nn*⟩ ⟨*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. To resolve a path, $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ is turned into $\langle name_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ where $\langle name_0\rangle.\langle i_1\rangle$ is $\langle name_1\rangle$, then $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$ where $\langle name_1\rangle.\langle i_2\rangle$ is $\langle name_2\rangle$... If the above rule does not apply, $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ may turn into $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$ when $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle$ is $\langle name_2\rangle$... We try to match the longest sequence of components first. The algorithm is not yet more clever. In general, ⟨*cs:nn*⟩ is just `\use_i:nn` but for in place incrementation, we must resolve only when there is an integer path. See the implementation of the `\__bnvs_if_append:...` conditionals.

```
\prg_new_conditional:Npnn \__bnvs_resolve:NNNN
    #1 #2 #3 #4 { T, F, TF } {
  #1 {
    \__bnvs_group_begin:
```

`\l_a_tl` contains the name with a partial index path currently resolved. `\l_a_seq` contains the remaining index path components to be resolved. `\l_b_seq` contains the current index path components to be resolved.

```
    \tl_set_eq:NN \l__bnvs_a_tl #3
    \seq_set_eq:NN \l__bnvs_a_seq #4
    \tl_clear:N \l__bnvs_b_tl
    \seq_clear:N \l__bnvs_b_seq
    \cs_set:Npn \return_true: {
      \cs_set:Npn \:nnn ####1 ####2 ####3 {
        \__bnvs_group_end:
        \tl_set:Nn #2 { ####1 }
        \tl_set:Nn #3 { ####2 }
        \seq_set_split:Nnn #4 . { ####3 }
        \seq_remove_all:Nn #4 { }
      }
      \exp_args:NVVx
      \:nnn #2 #3 {
        \seq_use:Nn #4 .
      }
      \prg_return_true:
    }
    \cs_set:Npn \branch:n ##1 {
      \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
        \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_b_tl
        \tl_set:Nn \l__bnvs_a_tl { #3 . }
        \tl_put_right:Nx \l__bnvs_a_tl { \seq_use:Nn \l__bnvs_a_seq . }
      } {
        \cs_set_eq:NN \loop: \return_true:
      }
    }
    \cs_set:Npn \branch:FF ##1 ##2 {
      \exp_args:Nx
      \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
        \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
          \tl_set_eq:NN #3 \l__bnvs_b_tl
          \seq_set_eq:NN #4 \l__bnvs_b_seq
          \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
        } { ##1 }
      } { ##2 }
    }
    \cs_set:Npn \extract_key:F {
      \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
        \tl_set_eq:NN #3 \l__bnvs_b_tl
        \seq_set_eq:NN #4 \l__bnvs_b_seq
        \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
      }
    }
    \cs_set:Npn \loop: {
      \__bnvs_call:TF {
        \exp_args:Nx
        \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
```

If there is a length, no resolution occurs.

```
          \branch:n { 1 }
        } {
          \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
            \seq_clear:N \l__bnvs_b_seq
            \tl_set:Nn \l__bnvs_a_tl { #3 . }
            \tl_put_right:Nx \l__bnvs_a_tl {
              \seq_use:Nn \l__bnvs_a_seq . .
            }
            \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
            \branch:FF {
```

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

```
\seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
            } {
```

Unknown key ⟨`\l_a_tl`⟩/L.

```
\seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
            }
          } {
            \branch:FF {
              \cs_set_eq:NN \loop: \return_true:
            } {
              \cs_set:Npn \loop: {
                \__bnvs_group_end:
                \prg_return_false:
              }
            }
          }
        }
      } {
        \cs_set:Npn \loop: {
          \__bnvs_group_end:
          \prg_return_false:
        }
      }
      \loop:
    }
    \loop:
  } {
    \prg_return_true:
  }
}
\prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
    #1 #2 #3 #4 { T, F, TF } {
  #1 {
    \__bnvs_group_begin:
```

`\l_a_tl` contains the name with a partial index path to be resolved. `\l_a_seq` contains the remaining index path components to be resolved.

```
    \tl_set_eq:NN \l__bnvs_a_tl #3
    \seq_set_eq:NN \l__bnvs_a_seq #4
    \cs_set:Npn \return_true: {
      \cs_set:Npn \:nnn ####1 ####2 ####3 {
        \__bnvs_group_end:
        \tl_set:Nn #2 { ####1 }
        \tl_set:Nn #3 { ####2 }
        \seq_set_split:Nnn #4 . { ####3 }
        \seq_remove_all:Nn #4 { }
      }
      \exp_args:NVVx
      \:nnn #2 #3 {
        \seq_use:Nn #4 .
      }
      \prg_return_true:
    }
    \cs_set:Npn \branch:n ##1 {
      \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
        \tl_put_right:Nn \l__bnvs_a_tl { . }
        \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
      } {
        \cs_set_eq:NN \loop: \return_true:
      }
    }
    \cs_set:Npn \loop: {
      \__bnvs_call:TF {
        \exp_args:Nx
        \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
          \branch:n { 1 }
        } {
          \exp_args:Nx
          \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
            \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
              \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
              \tl_set_eq:NN #3 \l__bnvs_b_tl
              \seq_set_eq:NN #4 \l__bnvs_a_seq
            } {
              \branch:n { 2 }
            }
          } {
            \branch:n { 3 }
          }
        }
      } {
        \cs_set:Npn \loop: {
          \__bnvs_group_end:
          \prg_return_false:
        }
      }
      \loop:
    }
    \loop:
  } {
    \prg_return_true:
  }
}
```

5.6.7 Evaluation bricks

__bnvs_fp_round:nN
__bnvs_fp_round:N

__bnvs_fp_round:nN {⟨*expression*⟩} ⟨*tl variable*⟩
__bnvs_fp_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_fp_round:nN #1 #2 {
  \tl_if_empty:nTF { #1 } {
  } {
    \tl_put_right:Nx #2 {
      \fp_eval:n { round(#1) }
    }
  }
}
\cs_generate_variant:Nn \__bnvs_fp_round:nN { VN, xN }
\cs_new:Npn \__bnvs_fp_round:N #1 {
  \tl_if_empty:VTF #1 {
  } {
    \tl_set:Nx #1 {
      \fp_eval:n { round(#1) }
    }
  }
}
```

__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.

```
\cs_set:Npn \__bnvs_return_true:nnN #1 #2 #3 {
  \tl_if_empty:NTF \l__bnvs_ans_tl {
    \__bnvs_group_end:
    \__bnvs_gremove:n { #1//#2 }
    \prg_return_false:
  } {
    \__bnvs_fp_round:N \l__bnvs_ans_tl
    \__bnvs_gput:nV { #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_return_false:nn #1 #2 {
  \__bnvs_group_end:
  \__bnvs_gremove:n { #1//#2 }
  \prg_return_false:
}
\prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
  \__bnvs_if_in:nTF { #1//A } {
    \tl_put_right:Nx #2 { \__bnvs_item:n { #1//A } }
    \prg_return_true:
  } {
    \__bnvs_group_begin:
    \tl_clear:N \l__bnvs_ans_tl
    \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
      \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
        \__bnvs_return_true:nnN { #1 } A #2
      } {
        \__bnvs_return_false:nn { #1 } A
      }
    } {
      \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
        \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
          \__bnvs_if_append:xNTF {
            \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
          } \l__bnvs_ans_tl {
            \__bnvs_return_true:nnN { #1 } A #2
          } {
            \__bnvs_return_false:nn { #1 } A
          }
        } {
          \__bnvs_return_false:nn { #1 } A
        }
      } {
        \__bnvs_return_false:nn { #1 } A
      }
    }
  }
}
\prg_generate_conditional_variant:Nnn
    \__bnvs_raw_first:nN { VN, xN } { T, F, TF }
```

`\__bnvs_if_first:nN`*TF*

`\__bnvs_if_first:nNTF {`⟨*name*⟩`} `⟨*tl variable*⟩` {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Append the first index of the ⟨*name*⟩ slide range to the ⟨*tl variable*⟩. If no first index was explicitely given, use the counter when available and 1 hen not. Cache the result. Execute ⟨*true code*⟩ when there is a ⟨*first*⟩, ⟨*false code*⟩ otherwise.

```
\prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
  \__bnvs_raw_first:nNTF { #1 } #2 {
    \prg_return_true:
  } {
    \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {
      \bool_set_true:N \l_no_counter_bool
      \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
        \__bnvs_return_true:nnN { #1 } A #2
      } {
        \__bnvs_return_false:nn { #1 } A
      }
    } {
      \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
        \__bnvs_gput:nn { #1/A } { 1 }
        \tl_set:Nn #2 { 1 }
        \__bnvs_return_true:nnN { #1 } A #2
      } {
        \__bnvs_return_false:nn { #1 } A
      }
    }
  }
}
```

`\__bnvs_first:nN`
`\__bnvs_first:VN`

`\__bnvs_first:nN {`⟨*name*⟩`} `⟨*tl variable*⟩

Append the start of the ⟨*name*⟩ slide range to the ⟨*tl variable*⟩. Cache the result.

```
\cs_new:Npn \__bnvs_first:nN #1 #2 {
  \__bnvs_if_first:nNF { #1 } #2 {
    \msg_error:nnn { beanoves } { :n } { Range~with~no~first:~#1 }
  }
}
\cs_generate_variant:Nn \__bnvs_first:nN { VN }
```

__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_if_in:nTF { #1//L } {
    \tl_put_right:Nx #2 { \__bnvs_item:n { #1//L } }
    \prg_return_true:
  } {
    \__bnvs_gput:nn { #1//L } { 0 }
    \__bnvs_group_begin:
    \tl_clear:N \l__bnvs_ans_tl
    \__bnvs_if_in:nTF { #1/L } {
      \__bnvs_if_append:xNTF {
        \__bnvs_item:n { #1/L }
      } \l__bnvs_ans_tl {
        \__bnvs_return_true:nnN { #1 } L #2
      } {
        \__bnvs_return_false:nn { #1 } L
      }
    } {
      \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
        \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
          \__bnvs_if_append:xNTF {
            \l__bnvs_b_tl - (\l__bnvs_a_tl) + 1
          } \l__bnvs_ans_tl {
            \__bnvs_return_true:nnN { #1 } L #2
          } {
            \__bnvs_return_false:nn { #1 } L
          }
        } {
          \__bnvs_return_false:nn { #1 } L
        }
      } {
        \__bnvs_return_false:nn { #1 } L
      }
    }
  }
}
\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_if_in:nTF { #1//Z } {
    \tl_put_right:Nx #2 { \__bnvs_item:n { #1//Z } }
    \prg_return_true:
  }  {
    \__bnvs_gput:nn { #1//Z } { 0 }
    \__bnvs_group_begin:
    \tl_clear:N \l__bnvs_ans_tl
    \__bnvs_if_in:nTF { #1/Z } {
      \__bnvs_if_append:xNTF {
        \__bnvs_item:n { #1/Z }
      } \l__bnvs_ans_tl {
        \__bnvs_return_true:nnN { #1 } Z #2
      } {
        \__bnvs_return_false:nn { #1 } Z
      }
    } {
      \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
        \__bnvs_get:nNTF { #1/L } \l__bnvs_b_tl {
          \__bnvs_if_append:xNTF {
            \l__bnvs_a_tl + (\l__bnvs_b_tl) - 1
          } \l__bnvs_ans_tl {
            \__bnvs_return_true:nnN { #1 } Z #2
          } {
            \__bnvs_return_false:nn { #1 } Z
          }
        } {
          \__bnvs_return_false:nn { #1 } Z
        }
      } {
        \__bnvs_return_false:nn { #1 } Z
      }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_raw_last:nN { VN } { T, F, TF }
```

__bnvs_last:nN
__bnvs_last:VN

__bnvs_last:nN {⟨*name*⟩} ⟨*tl variable*⟩

Append the last index of the fully qualified ⟨*name*⟩ slide range to ⟨*tl variable*⟩

```
\cs_new:Npn \__bnvs_last:nN #1 #2 {
  \__bnvs_raw_last:nNF { #1 } #2 {
    \msg_error:nnn { beanoves } { :n } { Range~with~no~last:~#1 }
  }
}
\cs_generate_variant:Nn \__bnvs_last:nN { VN }
```

__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_if_in:nTF { #1//N } {
    \tl_put_right:Nx #2 { \__bnvs_item:n { #1//N } }
    \prg_return_true:
  } {
    \__bnvs_group_begin:
    \cs_set:Npn \__bnvs_return_true: {
      \tl_if_empty:NTF \l__bnvs_ans_tl {
        \__bnvs_group_end:
        \prg_return_false:
      } {
        \__bnvs_fp_round:N \l__bnvs_ans_tl
        \__bnvs_gput:nV { #1//N } \l__bnvs_ans_tl
        \exp_args:NNNV
        \__bnvs_group_end:
        \tl_put_right:Nn #2 \l__bnvs_ans_tl
        \prg_return_true:
      }
    }
    \cs_set:Npn \return_false: {
      \__bnvs_group_end:
      \prg_return_false:
    }
    \tl_clear:N \l__bnvs_a_tl
    \__bnvs_raw_last:nNTF { #1 } \l__bnvs_a_tl {
      \__bnvs_if_append:xNTF {
        \l__bnvs_a_tl + 1
      } \l__bnvs_ans_tl {
        \__bnvs_return_true:
      } {
        \return_false:
      }
    } {
      \return_false:
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_next:nN { VN } { T, F, TF }
```

`\__bnvs_next:nN`
`\__bnvs_next:VN`

`\__bnvs_next:nN` {⟨*name*⟩} ⟨*tl variable*⟩

Append the index after the ⟨*name*⟩ slide range to the ⟨*tl variable*⟩.

```
\cs_new:Npn \__bnvs_next:nN #1 #2 {
  \__bnvs_if_next:nNF { #1 } #2 {
    \msg_error:nnn { beanoves } { :n } { Range~with~no~next:~#1 }
  }
}
\cs_generate_variant:Nn \__bnvs_next:nN { VN }
```

__bnvs_if_index:nnN*TF*
__bnvs_if_index:VVN*TF*
__bnvs_if_index:nnnN*TF*

__bnvs_if_index:nnNTF {⟨*name*⟩} {⟨*integer*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\prg_new_conditional:Npnn \__bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
    \tl_put_right:Nn \l__bnvs_ans_tl { + (#2) - 1}
    \exp_args:NNV
    \__bnvs_group_end:
    \__bnvs_fp_round:nN \l__bnvs_ans_tl #3
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_index:nnN { VVN } { 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 } {
  \bool_if:NTF \l__bnvs_no_range_bool {
    \prg_return_false:
  } {
    \__bnvs_if_in:nTF { #1/ } {
      \tl_put_right:Nn { 0-0 }
    } {
      \__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 {
        \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
          \exp_args:NNNx
          \__bnvs_group_end:
          \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
          \prg_return_true:
        } {
          \exp_args:NNNx
          \__bnvs_group_end:
          \tl_put_right:Nn #2 { \l__bnvs_a_tl - }
          \prg_return_true:
        }
      } {
        \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
          \exp_args:NNNx
          \__bnvs_group_end:
          \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
          \prg_return_true:
        } {
          \__bnvs_group_end:
          \prg_return_false:
        }
      }
    }
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_range:nN { VN } { T, F, TF }
```

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

`\__bnvs_range:nN {`⟨*name*⟩`}` ⟨*tl variable*⟩

Append the range of the ⟨*name*⟩ slide range to the ⟨*tl variable*⟩.

```
\cs_new:Npn \__bnvs_range:nN #1 #2 {
  \__bnvs_if_range:nNF { #1 } #2 {
    \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
  }
}
\cs_generate_variant:Nn \__bnvs_range:nN { VN }
```

`\__bnvs_if_free_counter:nN`*TF*
`\__bnvs_if_free_counter:VN`*TF*

`\__bnvs_if_free_counter:nNTF {`⟨*name*⟩`}` ⟨*tl variable*⟩ `{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Set the ⟨*tl variable*⟩ to the value of the counter associated to the {⟨*name*⟩} slide range.

```
\prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
    \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
      \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
    }
  }
  \tl_if_empty:NTF \l__bnvs_ans_tl {
    \__bnvs_group_end:
    \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
      \__bnvs_gput:nn { #1/C } { 1 }
      \tl_set:Nn #2 { 1 }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_set:Nn #2 \l__bnvs_ans_tl
    \prg_return_true:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_free_counter:nN { VN } { T, F, TF }
```

__bnvs_if_counter:nN*TF*
__bnvs_if_counter:VN*TF*

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

Append the value of the counter associated to the {⟨*name*⟩} slide range to the right of ⟨*tl variable*⟩. The value always lays in between the range, whenever possible.

```
\prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {
  \__bnvs_group_begin:
  \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_ans_tl {
```

If there is a ⟨*first*⟩, use it to bound the result from below.

```
    \tl_clear:N \l__bnvs_a_tl
    \__bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
      \fp_compare:nNnT { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
        \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
      }
    }
```

If there is a ⟨*last*⟩, use it to bound the result from above.

```
    \tl_clear:N \l__bnvs_a_tl
    \__bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
      \fp_compare:nNnT { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
        \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
      }
    }
    \exp_args:NNV
    \__bnvs_group_end:
    \__bnvs_fp_round:nN \l__bnvs_ans_tl #2
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_counter: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_a_tl
  \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_a_tl {
    \tl_clear:N \l__bnvs_b_tl
    \__bnvs_if_append:xNTF { \l__bnvs_a_tl + (#2) } \l__bnvs_b_tl {
      \__bnvs_fp_round:N \l__bnvs_b_tl
      \__bnvs_gput:nV { #1/C } \l__bnvs_b_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_if_counter: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_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_if_counter:nNTF { #1 } #3 {
    \__bnvs_if_incr:nnTF { #1 } { #2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_post:nnN { VnN, VVN } { T, F, TF }
```

5.6.8 Evaluation

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

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

Evaluates the ⟨*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.

```
\prg_new_conditional:Npnn \__bnvs_if_append:nN #1 #2 { T, F, TF } {
  \__bnvs_call:TF {
    \__bnvs_group_begin:
```

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:nTF` `\switch:nTF {`⟨*capture group number*⟩`} {`⟨*black code*⟩`} {`⟨*white code*⟩`}`

Helper function to locally set the `\l__bnvs_group_tl` variable to the captured group ⟨*capture group number*⟩ and branch.

```
\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
  }
}
```

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

```
\cs_set:Npn \return_true: {
  \fp_round:
  \exp_args:NNNV
  \__bnvs_group_end:
  \tl_put_right:Nn #2 \l__bnvs_ans_tl
  \prg_return_true:
}
\cs_set:Npn \fp_round: {
  \__bnvs_fp_round:N \l__bnvs_ans_tl
}
\cs_set:Npn \return_false: {
  \__bnvs_group_end:
  \prg_return_false:
}
\cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {
  \switch:nNTF { ##2 } \l__bnvs_id_tl { } {
    \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_current_tl
    \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
  }
  \switch:nNTF { ##3 } \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
  }
  ##1 \l__bnvs_id_tl \l__bnvs_name_tl \l__bnvs_path_seq {
    \cs_set:Npn \: {
      ##4
    }
  } {
    \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
  }
  \:
}
\cs_set:Npn \:T ##1 {
  \seq_if_empty:NTF \l__bnvs_path_seq { ##1 } {
    \cs_set_eq:NN \loop: \return_false:
  }
```

```
    }
```

Main loop.

```
    \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 {
```

- Case ++⟨*name*⟩⟨*integer path*⟩.n.

```
       \:NnnT \__bnvs_resolve_n:NNNTF 2 3 {
        \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
         \cs_set_eq:NN \loop: \return_false:
        }
       }
      } {
       \switch:nNTF 4 \l__bnvs_name_tl {
```

- Cases ⟨*name*⟩⟨*integer path*⟩....

```
        \switch:nNTF 7 \l__bnvs_a_tl {
         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
          \:T {
           \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
            \cs_set_eq:NN \loop: \return_false:
           }
          }
         }
```

- Case ...length.

```
        } {
         \switch:nNTF 8 \l__bnvs_a_tl {
```

- Case ...last.

```
          \:NnnT \__bnvs_resolve:NNNTF 5 6 {
           \:T {
            \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
             \cs_set_eq:NN \loop: \return_false:
            }
           }
          }
```

```
         } {
          \switch:nNTF 9 \l__bnvs_a_tl {
```

- Case ...next.

```
                  \:NnnT \__bnvs_resolve:NNNTF 5 6 {
                    \:T {
                      \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
                        \cs_set_eq:NN \loop: \return_false:
                      }
                    }
                  }
                } {
                  \switch:nNTF { 10 } \l__bnvs_a_tl {
```

- Case ...range.

```
\:NnnT \__bnvs_resolve:NNNTF 5 6 {
  \:T {
    \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
      \cs_set_eq:NN \fp_round: \prg_do_nothing:
    } {
      \cs_set_eq:NN \loop: \return_false:
    }
  }
}
                  } {
```

- Case ...n.

```
                    \switch:nNTF { 12 } \l__bnvs_a_tl {
```

- Case ...+=⟨*integer*⟩.

```
\:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
  \:T {
    \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
      \cs_set_eq:NN \loop: \return_false:
    }
  }
}
                    } {
```

- Case ...n++.

```
                      \switch:nNTF { 13 } \l__bnvs_a_tl {
                        \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
                          \seq_if_empty:NTF \l__bnvs_path_seq {
\__bnvs_if_post:VnNF \l__bnvs_name_tl { 1 } \l__bnvs_ans_tl {
  \cs_set_eq:NN \loop: \return_false:
}
                          } {
\msg_error:nnx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
\cs_set_eq:NN \loop: \return_false:
                          }
                        }
                      } {
                        \switch:nNTF { 11 } \l__bnvs_a_tl {
```

- Case ...n++.

```
                          \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
                            \seq_if_empty:NTF \l__bnvs_path_seq {
\__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
  \cs_set_eq:NN \loop: \return_false:
}
                            } {
\seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
\seq_if_empty:NTF \l__bnvs_path_seq {
  \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
    \cs_set_eq:NN \loop: \return_false:
  }
} {
  \msg_error:nnx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
  \cs_set_eq:NN \loop: \return_false:
}
                            }
                          }
                        } {
                          \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
                            \seq_if_empty:NTF \l__bnvs_path_seq {
\__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
  \cs_set_eq:NN \loop: \return_false:
}
                            } {
\seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
\seq_if_empty:NTF \l__bnvs_path_seq {
  \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
    \cs_set_eq:NN \loop: \return_false:
  }
} {
  \msg_error:nnx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
  \cs_set_eq:NN \loop: \return_false:
}
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          } {
```

No name.

```
          }
        }
        \int_add:Nn \l__bnvs_split_int { 14 }
        \tl_put_right:Nx \l__bnvs_ans_tl {
          \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
        }
        \loop:
      } {
        \return_true:
      }
    }
    \loop:
  } {
    \msg_error:nnx { beanoves } { :n } { Too~many~calls:~ #1 }
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_append:nN { VN, xN } { 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 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:
\regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
  #1
} \l__bnvs_match_seq {
  \bool_set_false:N \l__bnvs_no_counter_bool
  \bool_set_false:N \l__bnvs_no_range_bool
```

\switch:nNTF

\switch:nNTF {⟨*capture group number*⟩} ⟨*tl variable*⟩ {⟨*black code*⟩} {⟨*white code*⟩}

Helper function to locally set the ⟨*tl variable*⟩ to the captured group ⟨*capture group number*⟩ and branch depending on the 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

```
    \bool_set_false:N \l__bnvs_no_range_bool
    \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
      \prg_return_true:
    } {
      \prg_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 {
              \prg_return_true:
            } {
              \prg_return_false:
            }
          } {
            \prg_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 {
              \prg_return_true:
            } {
              \prg_return_false:
            }
          } {
```

```
       \prg_return_false:
      }
     }
    } {
```

⟨*first*⟩: and ⟨*first*⟩:: range

```
     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
      \tl_put_right:Nn #2 { - }
      \prg_return_true:
     } {
      \prg_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 {
       \prg_return_true:
      } {
       \prg_return_false:
      }
     } {
\msg_error:nnx { beanoves } { :n } { Syntax~error(Missing~first):~#1 }
     }
    } {
```

: or :: range

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

Error

```
  \msg_error:nnn { beanoves } { :n } { Syntax~error:~#1 }
 }
}
```

__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 {
        \msg_fatal:nnn { beanoves } { :n } { Circular~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 { s o m } {
  \__bnvs_group_begin:
  \tl_clear:N \l__bnvs_ans_tl
  \IfBooleanTF { #1 } {
    \bool_set_true:N  \l__bnvs_no_counter_bool
  } {
    \bool_set_false:N \l__bnvs_no_counter_bool
  }
  \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
  \IfValueTF { #2 } {
    \exp_args:NNNV
    \__bnvs_group_end:
    \tl_set:Nn #2 \l__bnvs_ans_tl
  } {
    \exp_args:NV
    \__bnvs_group_end: \l__bnvs_ans_tl
  }
}
```

5.6.9 Reseting slide ranges

`\BeanovesReset`

`\beanovesReset [`⟨*first value*⟩`] {`⟨*Slide range name*⟩`}`

```
\NewDocumentCommand \BeanovesReset { O{1} m } {
  \__bnvs_reset:nn { #1 } { #2 }
  \ignorespaces
}
```

Forwards to `\__bnvs_reset:nn`.

`\__bnvs_reset:nn`

`\__bnvs_reset:nn {`⟨*first value*⟩`} {`⟨*slide range name*⟩`}`

Reset the counter to the given ⟨*first value*⟩. Clean the cached values also.

```
\cs_new:Npn \__bnvs_reset:nn #1 #2 {
  \bool_if:nTF {
    \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
  } {
    \__bnvs_gremove:n { #2/C }
    \__bnvs_gremove:n { #2//A }
    \__bnvs_gremove:n { #2//L }
    \__bnvs_gremove:n { #2//Z }
    \__bnvs_gremove:n { #2//N }
    \__bnvs_gput:nn { #2/C0 } { #1 }
  } {
    \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
  }
}
\makeatother
\ExplSyntaxOff
```