beamer named overlay ranges with beanover

Jérôme Laurens

v0.2 2022/10/05

Abstract

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

Contents

1 Minimal example 2

2 Named slide ranges 2

2.1 Presentation 2

2.2 Definition 2

3 Named overlay specifications 3

3.1 Named slide ranges 3

3.2 Named list of slide ranges 4

4 `?(...)` expressions 4

5 Implementation 5

5.1 Package declarations 5

5.2 Local variables 5

5.3 Overlay specification 5

5.3.1 In slide range definitions 5

5.3.2 Regular expressions 7

5.3.3 Defining named slide ranges 10

5.3.4 Scanning named overlay specifications 13

5.3.5 Evaluation bricks 17

5.3.6 Evaluation 22

5.3.7 Reseting slide ranges 28

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 {beanover}
 3 \begin{document}
 4 \begin{frame}
 5 {\Large Frame \insertframenumber}
 6 {\Large Slide \insertslidenumber}
 7 \Beanover{
 8 A = 1:2,
 9 B = A.next:3,
10 C = B.next,
11 }
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 8, we declare a slide range named ‘A’, firsting at slide 1 and with length 2. On line 12, the new overlay specification `?(A.1)` stands for 1, on line 15, `?(A.2)` stands for 2 and on line 18, `?(A.3)` stands for 3. On line 9, we declare a second slide range named ‘B’, firsting after the 2 slides of ‘A’ namely 3. Its length is 3 meaning that its last side has number 5, thus each `?(B.last)` is replaced by 5. The next slide after time line ‘B’ has number 6 which is also the first slide of the third time line due to line 10.

2 Named slide ranges

2.1 Presentation

Within a frame, there are different slides that appear in turn. The main slide range covers all the slide numbers, from one to the total amount of slides. In general, a slide range is a range of positive integers identified by a unique name. The main practical interest is that time lines may be defined relative to one another. Moreover we can specify overlay specifications based on time lines. Finally we can have lists of slide ranges.

2.2 Definition

`\Beanover`

`\Beanover{`⟨*key--value list*⟩`}`

The keys are the slide ranges names, they are case sensitive and must contain no spaces nor '`/`' character. When the same key is used multiple times, only the last is taken into account. The possible values are the *range specifiers* ⟨*first*⟩, ⟨*first*⟩:⟨*length*⟩, ⟨*first*⟩::⟨*last*⟩, :⟨*length*⟩::⟨*last*⟩ where ⟨*first*⟩, ⟨*last*⟩ and ⟨*length*⟩ are algebraic expression involving any named overlay specification when an integer.

A comma separated list of such specifiers is also allowed, which results in a *list of named slide ranges.*

3 Named overlay specifications

3.1 Named slide ranges

For named slide ranges, the named overlay specifications are detailled in the tables below together with their replacement meaning value as beamer standard overlay specification.

syntax	**meaning**
⟨*name*⟩ `=` $[i,\ i+1,\ i+2, \ldots]$	
⟨**name**⟩`.1`	i
⟨**name**⟩`.2`	$i+1$
⟨**name**⟩`.`⟨**integer**⟩	$i+$ ⟨*integer*⟩ -1

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

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

When the slide range has been given a length, we also have

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

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

Using these specification on unfinite time lines is unsupported. Finally each time line has a dedicated cursor ⟨*name*⟩ that we can use and increment.

⟨***name***⟩ : use the position of the cursor

⟨***name***⟩+=⟨***integer***⟩ : advance the cursor by ⟨*integer*⟩ and use the new position

++⟨***name***⟩ : advance the cursor by 1 and use the new position

3.2 Named list of slide ranges

The declaration \Beanover{A=[⟨*spec*$_1$⟩,⟨*spec*$_2$⟩,...,⟨*spec*$_n$⟩]} is a convenient shortcut for \Beanover{A.1=⟨*spec*$_1$⟩, A.2=⟨*spec*$_2$⟩,..., A.n=⟨*spec*$_n$⟩}. The rule of the previous section can apply.

4 ?(...) expressions

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

The ⟨*queries*⟩ argument is a comma separated list of individual ⟨*query*⟩'s of next table.

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

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

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

```
⟨*package⟩
```

5 Implementation

Identify the internal prefix (LaTeX3 DocStrip convention).

```
⟨@@=beanover⟩
```

5.1 Package declarations

```
\NeedsTeXFormat{LaTeX2e}[2020/01/01]
\ProvidesExplPackage
  {beanover}
  {2022/10/05}
  {0.2}
  {Named overlay specifications for beamer}
```

5.2 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.

```
\group_begin:
\tl_clear_new:N  \l_a_tl
\tl_clear_new:N  \l_b_tl
\tl_clear_new:N  \l_c_tl
\tl_clear_new:N  \l_ans_tl
\seq_clear_new:N \l_ans_seq
\seq_clear_new:N \l_match_seq
\seq_clear_new:N \l_token_seq
\int_zero_new:N  \l_split_int
\seq_clear_new:N \l_split_seq
\int_zero_new:N  \l_depth_int
\tl_clear_new:N  \l_name_tl
\tl_clear_new:N  \l_group_tl
\tl_clear_new:N  \l_query_tl
\seq_clear_new:N \l_query_seq
\flag_clear_new:n { no_cursor }
\flag_clear_new:n { no_range }
\group_end:
```

5.3 Overlay specification

5.3.1 In slide range definitions

\g__beanover_prop ⟨*key*⟩–⟨*value*⟩ property list to store the slide ranges. The basic keys are, assuming ⟨*name*⟩ is a slide range identifier,

⟨***name***⟩/A for the first index

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

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

⟨***name***⟩/C for the cursor value, when used

⟨***name***⟩/C0 for initial value of the cursor (when reset)

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

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

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

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

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

```
\prop_new:N \g__beanover_prop
```

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

```
\__beanover_gput:nn
\__beanover_gput:nV
\__beanover_item:n
\__beanover_gremove:n
\__beanover_gclear:n
\__beanover_clear:
```

`\__beanover_gput:nn {`⟨*key*⟩`} {`⟨*value*⟩`}`
`\__beanover_item:n {`⟨*key*⟩`}`
`\__beanover_gremove:n {`⟨*key*⟩`}`
`\__beanover_gclear:n {`⟨*key*⟩`}`
`\__beanover_gclear:`

Convenient shortcuts to manage the storage.

```
\cs_new:Npn \__beanover_gput:nn {
  \prop_gput:Nnn \g__beanover_prop
}
\cs_new:Npn \__beanover_item:n {
  \prop_item:Nn \g__beanover_prop
}
\cs_new:Npn \__beanover_gremove:n {
  \prop_gremove:Nn \g__beanover_prop
}
\cs_new:Npn \__beanover_gclear:n #1 {
  \clist_map_inline:nn { A, L, Z, C, C0, /A, /L, /Z, /N } {
    \__beanover_gremove:n { #1 / ##1 }
  }
}
\cs_new:Npn \__beanover_gclear: {
  \prop_gclear:N \g__beanover_prop
}
\cs_generate_variant:Nn \__beanover_gput:nn { nV }
```

```
\__beanover_if_in_p:n ⋆
\__beanover_if_in:nTF ⋆
```

`\__beanover_if_in_p:n {`⟨*key*⟩`}`
`\__beanover_if_in:nTF {`⟨*key*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Convenient shortcuts to test for the existence of some key.

```
\prg_new_conditional:Npnn \__beanover_if_in:n #1 { p, T, F, TF } {
  \prop_if_in:NnTF \g__beanover_prop { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__beanover_get:nN*TF*

__beanover_get:nNTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Convenient shortcuts to retrieve the value with branching. Execute ⟨*true code*⟩ when the item is found, ⟨*false code*⟩ otherwise. In the latter case, the content of the ⟨*tl variable*⟩ is undefined.

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

Utility message.

```
\msg_new:nnn { __beanover } { :n } { #1 }
```

5.3.2 Regular expressions

\c__beanover_name_regex

The name of a slide range consists of a list of alphanumerical characters and underscore, but with no leading digit.

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

(*End definition for \c__beanover_name_regex.*)

\c__beanover_key_regex
\c__beanover_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__beanover_key_regex {
  \ur{c__beanover_name_regex} (?: \. \d+ )*
}
\regex_const:Nn \c__beanover_A_key_Z_regex {
  \A \ur{c__beanover_key_regex} \Z
}
```

(*End definition for \c__beanover_key_regex and \c__beanover_A_key_Z_regex.*)

\c__beanover_dotted_regex

A specifier is the name of a slide range possibly followed by attributes using a dot syntax. This is a poor man version to save computations.

```
\regex_const:Nn \c__beanover_dotted_regex {
  \A \ur{c__beanover_name_regex} (?: \. [^.]+ )* \Z
}
```

(*End definition for \c__beanover_dotted_regex.*)

\c__beanover_colons_regex

For ranges defined by a colon syntax.

```
\regex_const:Nn \c__beanover_colons_regex { :(:+) }
```

(*End definition for \c__beanover_colons_regex.*)

\c__beanover_A_cln_Z_regex

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

(*End definition for \c__beanover_A_cln_Z_regex.*)

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

- 2: ⟨*first*⟩

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

- 3: second optional colon

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

- 4: ⟨*length*⟩

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

- 5: standalone ⟨*first*⟩

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

\c__beanover_int_regex

A decimal integer with an eventual sign.

```
\regex_const:Nn \c__beanover_int_regex {
  (?:[-+]\s*)?[0-9]+
}
```

(*End definition for* \c__beanover_int_regex.)

\c__beanover_list_regex

A comma separated list between square brackets. Capture groups:

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

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

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

(*End definition for* \c__beanover_list_regex.)

\c__beanover_split_regex

Used to parse slide ranges overlay specifications. Next are the 7 capture groups. Group numbers are 1 based because it is used in splitting contexts where only capture groups are considered.

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

We first with '+=' instrussions[1].

- 1: ⟨*name*⟩ of a cursor

```
    ( \ur{c__beanover_name_regex} )
```

- 2: optionally followed by positive integers attributes

```
    ( (?: \. \d+ )* ) \s*
    \+= \s*
```

[1] At the same time an instruction and an expression... synonym of exprection

- 3: the poor man integer expression after '+='. When it contains no parenthesis, it is an algebraic expression involving integers and ⟨*key*⟩'s. Otherwise it firsts with a parenthesis and ends with the first parenthesis followed by a white space or the end of the text. This tricky definition allows quite any algebraic expression involving parenthesis. The problems arise when dealing with nested expressions.

```
      ( (?: \ur{c__beanover_int_regex} | \ur{c__beanover_key_regex} )
        (?: [+\-*/] (?: \d+ | \ur{c__beanover_key_regex}) )*
      | \( \S+ \) (?: \Z | \s )
      )
```

- 4: ⟨*name*⟩ of a slide range...

```
  | \+\+ ( \ur{c__beanover_name_regex} )
```

- 5: eventually followed by positive integer attributes.

```
    ( (?: \. \d+ )* )
```

- 6: ⟨*name*⟩ of a slide range...

```
  | ( \ur{c__beanover_name_regex} )
```

- 7: optionally followed by attributes. In the correct syntax nonnegative integer attributes must come first. Here they are allowed everywhere and there is below an explicit error management with a dedicated error message.

```
    ( (?: \. [[:alnum:]]+ )* )

  ) \s*
}
```

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

```
\regex_const:Nn \c__beanover_attr_regex {
```

- 1: ⟨*integer*⟩ attribute

```
    ( \ur{c__beanover_int_regex} )
```

- 2: the ⟨*length*⟩ attribute

```
  |  l(e)ngth
```

- 3: the ⟨*last*⟩ attribute

```
  |  l(a)st
```

- 4: the ⟨*next*⟩ attribute

```
  |  (n)ext
```

- 5: the ⟨*range*⟩ attribute

```
  |  (r)ange

}
\regex_const:Nn \c__beanover_attrs_regex {
  \. (?:
```

- 1: ⟨*integer*⟩ attribute

```
    ( \ur{c__beanover_int_regex} )
```

- 2: the ⟨*length*⟩ attribute

```
  |  l(e)ngth
```

- 3: the ⟨*last*⟩ attribute

```
  |  l(a)st
```

- 4: the ⟨*next*⟩ attribute

```
  |  (n)ext
```

- 5: the ⟨*range*⟩ attribute

```
  |  (r)ange
```

- 6: other attribute

```
  |  ([^.]+)
```

```
  ) \b
}
```

5.3.3 Defining named slide ranges

`\__beanover_error:n`

Prints an error message when a key only item is used.

```
\cs_new:Npn \__beanover_error:n #1 {
  \msg_fatal:nnn { __beanover } { :n } { Missing~value~for~#1 }
}
```

`\__beanover_parse:nn`

`\__beanover_parse:nn {⟨name⟩} {⟨definition⟩}`

Auxiliary function called within a group. ⟨*name*⟩ is the slide range name, ⟨*definition*⟩ is the corresponding definition.

`\l_match_seq` Local storage for the match result.

(*End definition for* `\l_match_seq`*. This variable is documented on page* ??.)

`\__beanover_range:nnnn`
`\__beanover_range:nVVV`

`\__beanover_l:nnn {⟨key⟩} {⟨first⟩} {⟨length⟩} {⟨last⟩}`

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

```
\cs_new:Npn \__beanover_range:nnnn #1 #2 #3 #4 {
  \__beanover_gclear:n { #1 }
  \tl_if_empty:nF { #2 }{
    \__beanover_gput:nn { #1/A } { #2 }
  }
  \tl_if_empty:nF { #3 }{
    \__beanover_gput:nn { #1/L } { #3 }
  }
  \tl_if_empty:nF { #4 }{
    \__beanover_gput:nn { #1/Z } { #4 }
  }
}
\cs_generate_variant:Nn \__beanover_range:nnnn { nVVV }
```

__beanover_l:nnn

__beanover_l:nnn {⟨*name*⟩} {⟨*first*⟩} {⟨*length*⟩}

Auxiliary function called within a group. The ⟨*first*⟩ defaults to 1 and the ⟨*length*⟩ may be empty. Set the keys {⟨*name*⟩}.1 and eventually {⟨*name*⟩}.l.

```
\cs_new:Npn \__beanover_l:nnn #1 #2 #3 {
  \__beanover_gclear:n { #1 }
  \tl_if_empty:nF { #2 } {
    \__beanover_gput:nn { #1/A } { #2 }
  }
  \tl_if_empty:nF { #3 } {
    \__beanover_gput:nn { #1/L } { #3 }
  }
}
```

__beanover_n:nnn

__beanover_n:nnn {⟨*name*⟩} {⟨*first*⟩} {⟨*last*⟩}

Auxiliary function called within a group. ⟨*first*⟩ and ⟨*last*⟩ are optional.

```
\cs_new:Npn \__beanover_n:nnn #1 #2 #3 {
  \__beanover_gclear:n { #1 }
  \tl_if_empty:nF { #2 } {
    \__beanover_gput:nn { #1/A } { #2 }
  }
  \tl_if_empty:nF { #3 } {
    \__beanover_gput:nn { #1/Z } { #3 }
  }
}
```

```
\cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
\cs_new:Npn \__beanover_parse:nn #1 #2 {
  \regex_match:NnTF \c__beanover_A_key_Z_regex { #1 } {
```

We got a valid key.

```
    \regex_extract_once:NnNTF \c__beanover_list_regex { #2 } \l_match_seq {
```

This is a list.

```
      \exp_args:NNx
      \seq_set_from_clist:Nn \l_match_seq {
        \seq_item:Nn \l_match_seq { 2 }
      }
      \seq_map_indexed_inline:Nn \l_match_seq {
        \group_begin:
        \__beanover_parse:nn { #1.##1 } { ##2 }
        \group_end:
      }
    } {
```

This is a single range.

```
      \tl_clear:N \l_a_tl
      \tl_clear:N \l_b_tl
      \tl_clear:N \l_c_tl
      \regex_split:nnN { :(:*) } { #2 } \l_split_seq
      \seq_pop_left:NNT \l_split_seq \l_a_tl {
        \seq_pop_left:NNT \l_split_seq \l_b_tl {
          \tl_if_empty:NTF \l_b_tl {
            \seq_pop_left:NN \l_split_seq \l_b_tl
            \seq_pop_left:NNT \l_split_seq \l_c_tl {
              \tl_if_empty:NTF \l_c_tl {
\msg_error:nnn { __beanover } { :n } { Invalid~range~expression(1):~#2 }
              } {
                \int_compare:nNnT { \tl_count:N \l_c_tl } > { 1 } {
\msg_error:nnn { __beanover } { :n } { Invalid~range~expression(2):~#2 }
                }
                \seq_pop_left:NN \l_split_seq \l_c_tl
                \seq_if_empty:NF \l_split_seq {
\msg_error:nnn { __beanover } { :n } { Invalid~range~expression(3):~#2 }
                }
              }
            }
          } {
            \int_compare:nNnT { \tl_count:N \l_b_tl } > { 1 } {
\msg_error:nnn { __beanover } { :n } { Invalid~range~expression(4):~#2 }
            }
            \seq_pop_left:NN \l_split_seq \l_c_tl
            \seq_pop_left:NNTF \l_split_seq \l_b_tl {
              \tl_if_empty:NTF \l_b_tl {
                \seq_pop_left:NN \l_split_seq \l_b_tl
                \seq_if_empty:NF \l_split_seq {
\msg_error:nnn { __beanover } { :n } { Invalid~range~expression(5):~#2 }
                }
              } {
\msg_error:nnn { __beanover } { :n } { Invalid~range~expression(6):~#2 }
              }
            } {
              \tl_clear:N \l_b_tl
            }
          }
        }
      }
      \__beanover_range:nVVV { #1 } \l_a_tl \l_b_tl \l_c_tl
    }
  } {
    \msg_error:nnn { __beanover } { :n } { Invalid~key:~#1 }
  }
}
```

\Beanover

\Beanover {⟨*key--value list*⟩}

The keys are the slide range specifiers. We do not accept key only items, they are managed by __beanover_error:n. ⟨*key–value*⟩ items are parsed by __beanover_parse:nn. A group is open.

```
\NewDocumentCommand \Beanover { m } {
  \group_begin:
  \keyval_parse:NNn \__beanover_error:n \__beanover_parse:nn { #1 }
  \group_end:
  \ignorespaces
}
```

5.3.4 Scanning named overlay specifications

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

`\beamer@masterdecode`

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

Preprocess ⟨*overlay specification*⟩ before beamer uses it.

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

(*End definition for* `\l_ans_tl`*. This variable is documented on page* **??**.)

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

```
\cs_set_eq:NN \__beanover_beamer@masterdecode \beamer@masterdecode
\cs_set:Npn \beamer@masterdecode #1 {
  \group_begin:
  \tl_clear:N \l_ans_tl
  \__beanover_scan:NNn \__beanover_eval:Nn \l_ans_tl { #1 }
  \exp_args:NNV
  \group_end:
  \__beanover_beamer@masterdecode \l_ans_tl
}
```

__beanover_scan:NNn

__beanover_scan:NNn ⟨*eval*⟩ ⟨*tl variable*⟩ {⟨*named overlay expression*⟩}

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 `\__beanover_eval:Nn`. A group is created to use local variables:

`\l_ans_tl`: is the token list that will be appended to ⟨*tl variable*⟩ on return.

`\l_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_depth_int`. *This variable is documented on page* **??**.)

`\l_query_tl` Storage for the overlay query expression to be evaluated.

(*End definition for* `\l_query_tl`. *This variable is documented on page* **??**.)

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

(*End definition for* `\l_token_seq`. *This variable is documented on page* **??**.)

`\l__beanover_ask_bool` Whether a loop may continue. Controls the continuation of the main loop that scans the tokens of the ⟨*named overlay expression*⟩ looking for a question mark.

```
\bool_new:N \l__beanover_ask_bool
```

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

`\l__beanover_query_bool` Whether a loop may continue. Controls the continuation of the secondary loop that scans the tokens of the ⟨*overlay expression*⟩ looking for an opening parenthesis follow the question mark. It then controls the loop looking for the balanced closing parenthesis.

```
\bool_new:N \l__beanover_query_bool
```

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

`\l_token_tl` Storage for just one token.

(*End definition for* `\l_token_tl`. *This variable is documented on page* **??**.)

```
\cs_new:Npn \__beanover_scan:NNn #1 #2 #3 {
  \group_begin:
  \tl_clear:N \l_ans_tl
  \int_zero:N \l_depth_int
  \seq_clear:N \l_token_seq
```

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

```
  \regex_split:nnN {} { #3 } \l_token_seq
```

Run the top level loop to scan for a '?':

```
  \bool_set_true:N  \l__beanover_ask_bool
  \bool_while_do:Nn \l__beanover_ask_bool {
    \seq_pop_left:NN \l_token_seq \l_token_tl
    \quark_if_no_value:NTF \l_token_tl {
```

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

```
      \bool_set_false:N \l__beanover_ask_bool
    } {
```

`\l_token_tl` contains a ‘normal’ token.

```
      \tl_if_eq:NnTF \l_token_tl { ? } {
```

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

```
        \bool_set_true:N \l__beanover_query_bool
        \bool_while_do:Nn \l__beanover_query_bool {
```

Get next token.

```
          \seq_pop_left:NN \l_token_seq \l_token_tl
          \quark_if_no_value:NTF \l_token_tl {
```

No opening parenthesis found, raise.

```
            \msg_fatal:nnx { __beanover } { :n } {Missing~'('%---)
              ~after~a~?:~#3}
          } {
            \tl_if_eq:NnT \l_token_tl { ( %)
            } {
```

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

```
              \int_incr:N \l_depth_int
```

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

```
              \tl_clear:N \l_query_tl
              \bool_while_do:Nn \l__beanover_query_bool {
```

Get next token.

```
                \seq_pop_left:NN \l_token_seq \l_token_tl
                \quark_if_no_value:NTF \l_token_tl {
```

We reached the end of the sequence and the token list with no closing ‘`)`’. We raise and end both bool while loops. As recovery we feed `\l_query_tl` with the missing ‘`)`’. `\l_depth_int` is 0 whenever `\l@@_query_bool` is false.

```
                  \msg_error:nnx { __beanover } { :n } {Missing~%(---
                    `)':~#3 }
                  \int_do_while:nNnn \l_depth_int > 1 {
                    \int_decr:N \l_depth_int
                    \tl_put_right:Nn \l_query_tl {%(---
                    )}
                  }
                  \int_zero:N \l_depth_int
                  \bool_set_false:N \l__beanover_query_bool
                  \bool_set_false:N \l__beanover_ask_bool
                } {
                  \tl_if_eq:NnTF \l_token_tl { ( %---)
                  } {
```

We found a ‘`(`’, increment the depth and append the token to `\l_query_tl`.

```
                    \int_incr:N \l_depth_int
                    \tl_put_right:NV \l_query_tl \l_token_tl
                  } {
```

This is not a ‘`(`’.

```
                    \tl_if_eq:NnTF \l_token_tl { %(
                      )
                    } {
```

We found a ‘)’, decrement the depth.

```
           \int_decr:N \l_depth_int
           \int_compare:nNnTF \l_depth_int = 0 {
```

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

```
 \exp_args:NNV #1 \l_ans_tl \l_query_tl
 \bool_set_false:N \l__beanover_query_bool
           } {
```

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

```
            \tl_put_right:NV \l_query_tl \l_token_tl
           }
```

Above ends the code for a positive depth.

```
          } {
```

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

```
           \tl_put_right:NV \l_query_tl \l_token_tl
          }
         }
        }
```

Above ends the code for Not a ‘(’

```
       }
      }
```

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

```
     }
```

Above ends the code for not a no value quark.

```
    }
```

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

If we reached the end of the token list, then end both the current loop and its containing loop.

```
    \quark_if_no_value:NT \l_token_tl {
     \bool_set_false:N \l__beanover_query_bool
     \bool_set_false:N \l__beanover_ask_bool
    }
   } {
```

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

```
    \tl_put_right:NV \l_ans_tl \l_token_tl
   }
```

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

```
  }
 }
```

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

```
 \exp_args:NNNV
 \group_end:
 \tl_put_right:Nn #2 \l_ans_tl
}
```

Each new frame has its own set of slide ranges, we clear the property list on entering a new frame environment.

```
\AddToHook
  { env/beamer@framepauses/before }
  { \prop_gclear:N \g__beanover_prop }
```

5.3.5 Evaluation bricks

__beanover_if_first_p:nN ⋆
__beanover_if_first:nN*TF* ⋆

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

Append the first 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 \__beanover_if_first:nN #1 #2 { p, T, F, TF } {
  \__beanover_if_in:nTF { #1//A } {
    \tl_put_right:Nx #2 { \__beanover_item:n { #1//A } }
    \prg_return_true:
  } {
    \group_begin:
    \tl_clear:N \l_ans_tl
    \__beanover_if_in:nTF { #1/A } {
      \__beanover_eval:Nx \l_ans_tl {
        \__beanover_item:n { #1/A }
      }
    } {
      \bool_if:nTF {
        \__beanover_if_in_p:n { #1/L } && \__beanover_if_in_p:n { #1/Z }
      } {
        \__beanover_eval:Nx \l_ans_tl {
          \__beanover_item:n { #1/Z } - ( \__beanover_item:n { #1/L } - 1 )
        }
      } {
        \__beanover_if_in:nT { #1/C } {
          \flag_raise:n { no_cursor }
          \__beanover_eval:Nx \l_ans_tl {
            \__beanover_item:n { #1/C }
          }
        }
      }
    }
    \tl_if_empty:NTF \l_ans_tl {
      \group_end:
      \prg_return_false:
    } {
      \__beanover_gput:nV { #1//A } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #2 \l_ans_tl
      \prg_return_true:
    }
  }
}
```

__beanover_first:nN
__beanover_first:VN

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

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

```
\cs_new:Npn \__beanover_first:nN #1 #2 {
  \__beanover_if_first:nNF { #1 } #2 {
    \msg_error:nnn { __beanover } { :n } { Range~with~no~first:~#1 }
  }
}
\cs_generate_variant:Nn \__beanover_first:nN { VN }
```

__beanover_if_length_p:nN ⋆
__beanover_if_length:nN*TF* ⋆

__beanover_length_p:nN {⟨*name*⟩} ⟨*tl variable*⟩
__beanover_if_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 \__beanover_if_length:nN #1 #2 { p, T, F, TF } {
  \__beanover_if_in:nTF { #1//L } {
    \tl_put_right:Nx #2 { \__beanover_item:n { #1//L } }
    \prg_return_true:
  } {
    \group_begin:
    \tl_clear:N \l_ans_tl
    \__beanover_if_in:nTF { #1/L } {
      \__beanover_eval:Nx \l_ans_tl {
        \__beanover_item:n { #1/L }
      }
    } {
      \bool_if:nT {
        \__beanover_if_in_p:n { #1/A } && \__beanover_if_in_p:n { #1/Z }
      } {
        \__beanover_eval:Nx \l_ans_tl {
          \__beanover_item:n { #1/Z } - \__beanover_item:n { #1/A } + 1
        }
      }
    }
    \tl_if_empty:NTF \l_ans_tl {
      \group_end:
      \prg_return_false:
    } {
      \__beanover_gput:nV { #1//L } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #2 \l_ans_tl
      \prg_return_true:
    }
  }
}
```

__beanover_length:nN
__beanover_length:VN

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

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

```
\cs_new:Npn \__beanover_length:nN #1 #2 {
  \__beanover_if_length:nNF { #1 } #2 {
    \msg_error:nnn { __beanover } { :n } { Range~with~no~length:~#1 }
  }
}
\cs_generate_variant:Nn \__beanover_length:nN { VN }
```

__beanover_if_last_p:nN ★
__beanover_if_last:nN*TF* ★

__beanover_if_last_p:nN {⟨*name*⟩} ⟨*tl variable*⟩
__beanover_if_last:nNTF {⟨*name*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

```
\prg_new_conditional:Npnn \__beanover_if_last:nN #1 #2 { p, T, F, TF } {
  \__beanover_if_in:nTF { #1//Z } {
    \tl_put_right:Nx #2 { \__beanover_item:n { #1//Z } }
    \prg_return_true:
  }  {
    \group_begin:
    \tl_clear:N \l_ans_tl
    \__beanover_if_in:nTF { #1/Z } {
      \__beanover_eval:Nx \l_ans_tl {
        \__beanover_item:n { #1/Z }
      }
    } {
      \__beanover_get:nNT { #1/A } \l_a_tl {
        \__beanover_get:nNT { #1/L } \l_b_tl {
          \__beanover_eval:Nx \l_ans_tl {
            \l_a_tl + \l_b_tl - 1
          }
        }
      }
    }
    \tl_if_empty:NTF \l_ans_tl {
      \group_end:
      \prg_return_false:
    } {
      \__beanover_gput:nV { #1//Z } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #2 \l_ans_tl
      \prg_return_true:
    }
  }
}
```

__beanover_last:nN
__beanover_last:VN

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

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

```
\cs_new:Npn \__beanover_last:nN #1 #2 {
  \__beanover_if_last:nNF { #1 } #2 {
    \msg_error:nnn { __beanover } { :n } { Range~with~no~last:~#1 }
  }
}
\cs_generate_variant:Nn \__beanover_last:nN { VN }
```

__beanover_if_next_p:nN ⋆
__beanover_if_next:nN*TF* ⋆

__beanover_if_next_p:nN {⟨*name*⟩} ⟨*tl variable*⟩
__beanover_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 \__beanover_if_next:nN #1 #2 { p, T, F, TF } {
  \__beanover_if_in:nTF { #1//N } {
    \tl_put_right:Nx #2 { \__beanover_item:n { #1//N } }
    \prg_return_true:
  } {
    \group_begin:
    \__beanover_get:nNTF { #1/Z } \l_ans_tl {
      \tl_put_right:Nn \l_ans_tl { +1 }
    } {
      \__beanover_get:nNT { #1/A } \l_a_tl {
        \__beanover_get:nNT { #1/L } \l_b_tl {
          \__beanover_eval:Nx \l_ans_tl {
            \l_a_tl + \l_b_tl
          }
        }
      }
    }
    \tl_if_empty:NTF \l_ans_tl {
      \group_end:
      \prg_return_false:
    } {
      \__beanover_gput:nV { #1//N } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #2 \l_ans_tl
      \prg_return_true:
    }
  }
}
```

__beanover_next:nN
__beanover_next:VN

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

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

```
\cs_new:Npn \__beanover_next:nN #1 #2 {
  \__beanover_if_next:nNF { #1 } #2 {
    \msg_error:nnn { __beanover } { :n } { Range~with~no~next:~#1 }
  }
}
\cs_generate_variant:Nn \__beanover_next:nN { VN }
```

__beanover_free_cursor:nN
__beanover_free_cursor:VN

__beanover_free_cursor:Nn {⟨*name*⟩} ⟨*tl variable*⟩

Append the value of the cursor associated to the {⟨*name*⟩} slide range to the right of ⟨*tl variable*⟩. There is no branching variant because, we always return some value, '1' by default.

```
\cs_new:Npn \__beanover_free_cursor:nN #1 #2 {
  \group_begin:
  \tl_clear:N \l_ans_tl
  \__beanover_get:nNF { #1/C } \l_ans_tl {
    \__beanover_if_first:nNF { #1 } \l_ans_tl {
      \__beanover_if_last:nNF { #1 } \l_ans_tl {
        \tl_set:Nn \l_ans_tl { 1 }
      }
    }
  }
  \__beanover_gput:nV { #1/C } \l_ans_tl
  \exp_args:NNNV
  \group_end:
  \tl_put_right:Nn #2 \l_ans_tl
}
\cs_generate_variant:Nn \__beanover_free_cursor:nN { VN }
```

__beanover_cursor:nN
__beanover_cursor:VN

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

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

```
\cs_new:Npn \__beanover_cursor:nN #1 #2 {
  \group_begin:
  \__beanover_free_cursor:nN { #1 } \l_ans_tl
```

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

```
  \tl_clear:N \l_a_tl
  \__beanover_if_first:nNT { #1 } \l_a_tl {
    \fp_compare:nNnT { \l_ans_tl } < { \l_a_tl } {
      \tl_set:NV \l_ans_tl \l_a_tl
    }
  }
```

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

```
  \tl_clear:N \l_a_tl
  \__beanover_if_last:nNT { #1 } \l_a_tl {
    \fp_compare:nNnT { \l_ans_tl } > { \l_a_tl } {
      \tl_set:NV \l_ans_tl \l_a_tl
    }
  }
  \exp_args:NNNV
  \group_end:
  \tl_set:Nn #2 \l_ans_tl
}
\cs_generate_variant:Nn \__beanover_cursor:nN { VN }
```

__beanover_incr:nn
__beanover_incr:nnN
__beanover_incr:VVN

__beanover_incr:nn {⟨*name*⟩} {⟨*offset*⟩}
__beanover_incr:nnN {⟨*name*⟩} {⟨*offset*⟩} ⟨*tl variable*⟩

Increment the cursor position accordingly. When requested, put the result in the ⟨*tl variable*⟩. The result will lay within the declared range.

```
\cs_new:Npn \__beanover_incr:nn #1 #2 {
  \group_begin:
  \tl_clear:N \l_a_tl
  \__beanover_free_cursor:nN { #1 }  \l_a_tl
  \tl_clear:N \l_ans_tl
  \__beanover_eval:Nx \l_ans_tl { \l_a_tl + ( #2 ) }
  \__beanover_gput:nV { #1/C } \l_ans_tl
  \group_end:
}
\cs_new:Npn \__beanover_incr:nnN #1 #2 #3 {
  \__beanover_incr:nn { #1 } { #2 }
  \__beanover_cursor:nN { #1 } #3
}
\cs_generate_variant:Nn \__beanover_incr:nnN { VVN }
```

5.3.6 Evaluation

`\__beanover_append:nN`
`\__beanover_append:VN`

`\__beanover_append:nN {⟨integer expression⟩} ⟨tl variable⟩`

Evaluates the ⟨*integer expression*⟩, replacing all the named specifications by their counterpart then put the result to the right of the ⟨*tl variable*⟩. Executed within a group. Local variables: `\l_ans_tl` for the content of ⟨*tl variable*⟩

`\l_split_seq` The sequence of queries and non queries.

(*End definition for* `\l_split_seq`*. This variable is documented on page* **??**.)

`\l_split_int` Is the index of the non queries, before all the catched groups.

(*End definition for* `\l_split_int`*. This variable is documented on page* **??**.)

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

(*End definition for* `\l_name_tl`*. This variable is documented on page* **??**.)

`\l__beanover_static_tl` Storage for the static values of named slide ranges.

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

`\l_group_tl` Storage for capture groups.

(*End definition for* `\l_group_tl`*. This variable is documented on page* **??**.)

```
\cs_new:Npn \__beanover_append:nN #1 #2 {
  \group_begin:
```

Local variables:

```
  \tl_clear:N  \l_ans_tl
  \int_zero:N  \l_split_int
  \seq_clear:N \l_split_seq
  \tl_clear:N  \l_name_tl
  \tl_clear:N  \l_group_tl
  \tl_clear:N  \l_a_tl
```

Implementation:

```
  \regex_split:NnN \c__beanover_split_regex { #1 } \l_split_seq
  \int_set:Nn \l_split_int { 1 }
  \tl_set:Nx \l_ans_tl { \seq_item:Nn \l_split_seq { \l_split_int } }
```

\switch:nTF

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

Helper function to locally set the \l_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_split_seq { \l_split_int + ##1 }
  }
  \tl_if_empty:NTF ##2 { ##4 } { ##3 }
}
```

Main loop.

```
\int_while_do:nNnn { \l_split_int } < { \seq_count:N \l_split_seq } {
      \switch:nNTF 1 \l_name_tl {
        \switch:nNTF 2 \l_a_tl {
```

Case ⟨*name*⟩.⟨*integer*⟩.

```
\group_begin:
\tl_clear:N \l_ans_tl
\exp_args:NV \__beanover_first:nN \l_name_tl \l_ans_tl
\tl_put_right:Nn \l_ans_tl { + ( \l_group_tl ) - 1 }
\exp_args:NNNx
\group_end:
\tl_put_right:Nn \l_ans_tl {
  \fp_to_int:n \l_ans_tl
          }
        } {
          \switch:nNTF 3 \l_a_tl {
```

Case ⟨*name*⟩.length.

```
            \__beanover_length:NV \l_ans_tl \l_name_tl
          } {
            \switch:nNTF 4 \l_a_tl {
```

Case ⟨*name*⟩.range. **conceptual problem with ‘::’**

```
\flag_if_raised:nT { no_range } {
  \msg_fatal:nnn { __beanover } { :n } {
    No~\l_name_tl.range available:~#1
  }
}
\__beanover_first:NV \l_ans_tl \l_name_tl
\tl_put_right:Nn \l_ans_tl { :: }
\__beanover_last:NV \l_ans_tl \l_name_tl
            } {
              \switch:nNTF 5 \l_a_tl {
```

Case ⟨*name*⟩.last.

```
\__beanover_last:NV \l_ans_tl \l_name_tl
              } {
                \switch:nNTF 6 \l_a_tl {
```

Case ⟨*name*⟩.next.

```
\__beanover_next:NV \l_ans_tl \l_name_tl
                } {
                  \switch:nNTF 7 \l_group_tl {
```

Case ⟨*name*⟩`.reset`.

```
  \flag_if_raised:nT { no_cursor } {
    \msg_fatal:nnn { __beanover } { :n } {
      No~\l_name_tl~cursor~available~inside~\cs{Beanover}:~#1
    }
  }
  \__beanover_reset:nV { 0 } \l_name_tl
                    } {
                      \switch:nNTF 8 \l_group_tl {
```

Case ⟨*name*⟩`.UNKNOWN`.

```
  \msg_fatal:nnn { __beanover } { :n } { Unknown~attribute~\l_group_tl:~#1 }
                      } { }
                    }
                  }
                }

              }
            }
          }
        } {
          \switch:nNTF { 12 } \l_name_tl {
            \flag_if_raised:nT { no_cursor } {
              \msg_fatal:nnn { __beanover } { :n } {
No~\l_name_tl~cursor~available~inside~\cs{Beanover}:~#1
              }
            }
            \switch:nNTF { 11 } \l_ans_tl {
```

Case `++`⟨*name*⟩.

```
              \exp_args:NV
              \__beanover_incr:nnN \l_name_tl 1 \l_ans_tl
            } {
```

Case ⟨*name*⟩.

```
              \__beanover_cursor:VN \l_name_tl \l_ans_tl
            }
          } {
        }
      }
  }
  \exp_args:NNNx
  \group_end:
  \tl_put_right:Nn #2 { \fp_to_int:n { \l_ans_tl } }
}
\cs_generate_variant:Nn \__beanover_append:nN { VN }
```

`\__beanover_eval_query:nN`

`\__beanover_eval_query:Nn {⟨overlay query⟩} ⟨seq variable⟩`

Evaluates the single ⟨*overlay query*⟩, which is expected to contain no comma. 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_a_tl` Storage for the first of a range.

(*End definition for* `\l_a_tl`*. This variable is documented on page* **??**.)

`\l_b_tl` Storage for the end of a range, or its length.

(*End definition for* `\l_b_tl`*. This variable is documented on page* **??**.)

```
\cs_new:Npn \__beanover_eval_query:nN #1 #2 {
  \regex_extract_once:NnNTF \c__beanover_A_cln_Z_regex {
    #1
  } \l_match_seq {
    \tl_clear:N \l_ans_tl
    \flag_clear:n { no_cursor }
    \flag_raise:n { no_range }
```

`\switch:nTF`

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

Helper function to locally set the `\l_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_split_seq { ##1 }
      }
      \tl_if_empty:NTF ##2 { ##4 } { ##3 }
    }
    \switch:nNTF 5 \l_a_tl {
```

Single expression

```
      \flag_clear:n { no_range }
      \__beanover_append:NV \l_ans_tl \l_a_tl
      \seq_put_right:NV #1 \l_ans_tl
    } {
      \switch:nNTF 2 \l_a_tl {
        \switch:nNTF 4 \l_b_tl {
          \switch:nNTF 3 \l_a_tl {
```

`⟨first⟩::⟨last⟩` range

```
            \__beanover_append:NV \l_ans_tl \l_a_tl
            \tl_put_right:Nn \l_ans_tl { - }
            \__beanover_append:NV \l_ans_tl \l_b_tl
            \seq_put_right:NV #1 \l_ans_tl
          } {
```

`⟨first⟩:⟨length⟩` range

```
            \__beanover_append:NV \l_ans_tl \l_a_tl
            \tl_put_right:Nx \l_ans_tl { - }
            \tl_put_right:Nx \l_a_tl { - ( \l_b_tl ) + 1}
```

```
      \__beanover_append:NV \l_ans_tl \l_b_tl
      \seq_put_right:NV #1 \l_ans_tl
     }
    } {
```

🗨 ⟨*first*⟩: an ⟨*first*⟩`::` range

```
     \__beanover_append:NV \l_ans_tl \l_a_tl
     \tl_put_right:Nn \l_ans_tl { - }
     \seq_put_right:NV #1 \l_ans_tl
    }
   } {
    \switch:nNTF 4 \l_b_tl {
     \switch:nNTF 3 \l_a_tl {
```

🗨 `::`⟨*last*⟩ range

```
      \tl_put_right:Nn \l_ans_tl { - }
      \__beanover_append:NV \l_ans_tl \l_a_tl
      \seq_put_right:NV #1 \l_ans_tl
     } {
\msg_error:nnx { __beanover } { :n } { Syntax~error(Missing~first):~#1 }
     }
    } {
```

🗨 `:` or `::` range

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

Error

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

`\__beanover_eval:Nn` `\__beanover_eval:Nn` ⟨*tl variable*⟩ `{`⟨*overlay queries*⟩`}`

Evaluates the ⟨*overlay queries*⟩, replacing all the named overlay specifications and integer expressions by their static counterparts, 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_query_seq` Storage for a sequence of ⟨*query*⟩'s obtained by splitting a comma separated list.

(*End definition for* `\l_query_seq`*. This variable is documented on page* **??**.)

`\l_ans_seq` Storage of the evaluated result.

(*End definition for* `\l_ans_seq`*. This variable is documented on page* **??**.)

`\c__beanover_comma_regex` Used to parse slide range overlay specifications.

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

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

No other variable is used.

```
\cs_new:Npn \__beanover_eval:Nn #1 #2 {
 \group_begin:
```

Local variables declaration

```
  \tl_clear:N  \l_a_tl
  \tl_clear:N  \l_b_tl
  \tl_clear:N  \l_ans_tl
  \seq_clear:N \l_ans_seq
  \seq_clear:N \l_query_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`.

```
  \__beanover_append:Nn \l_ans_tl { #2 }
  \exp_args:NNV
  \regex_split:NnN \c__beanover_comma_regex \l_ans_tl \l_query_seq
```

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

```
  \seq_map_tokens:Nn \l_query_seq {
    \__beanover_eval_query:Nn \l_ans_seq
  }
```

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

```
  \exp_args:NNNx
  \group_end:
  \tl_put_right:Nn #1 { \seq_use:Nn \l_ans_seq , }
}
\cs_generate_variant:Nn \__beanover_eval:Nn { NV, Nx }
```

`\BeanoverEval`

`\BeanoverEval [`⟨*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 `\__beanover_eval:Nn` within a group. `\l_ans_tl` is used to store the result.

```
\NewExpandableDocumentCommand \BeanoverEval { s o m } {
  \group_begin:
  \tl_clear:N \l_ans_tl
  \IfBooleanTF { #1 } {
    \flag_raise:n { no_cursor }
  } {
    \flag_clear:n { no_cursor }
  }
  \__beanover_eval:Nn \l_ans_tl { #3 }
  \IfValueTF { #2 } {
    \exp_args:NNNV
    \group_end:
    \tl_set:Nn #2 \l_ans_tl
  } {
    \exp_args:NV
    \group_end: \l_ans_tl
  }
}
```

5.3.7 Reseting slide ranges

\BeanoverReset

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

```
\NewDocumentCommand \BeanoverReset { O{1} m } {
  \__beanover_reset:nn { #1 } { #2 }
  \ignorespaces
}
```

Forwards to __beanover_reset:nn.

__beanover_reset:nn

__beanover_reset:nn {⟨*first value*⟩} {⟨*slide range name*⟩}

Reset the cursor to the given ⟨*first value*⟩ which defaults to 1. Clean the cached values also (not usefull).

```
\cs_new:Npn \__beanover_reset:nn #1 #2 {
  \__beanover_if_in:nTF { #2/1 } {
    \__beanover_gremove:n { #2 }
    \__beanover_gremove:n { #2//A }
    \__beanover_gremove:n { #2//L }
    \__beanover_gremove:n { #2//N }
    \__beanover_gremove:n { #2//Z }
    \__beanover_gput:nn { #2/c } { #1 }
  } {
    \msg_warning:nnn { __beanover } { :n } { Unknown~name:~#2 }
  }
}
\makeatother
\ExplSyntaxOff
⟨/package⟩
```