beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2024/01/11

Abstract

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

Contents

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 = 1:3,
6      B = A.last::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.range)> {Only on slide 3 to 5}\\
14 \visible<?(C.1)> {Only on slide 6}\\
15 \visible<?(A.2)> {Only on slide 2}\\
16 \visible<?(B.2:B.last)> {Only on slide 4 to 5}\\
17 \visible<?(C.2)> {Only on slide 7}\\
18 \visible<?(A.next)-> {From slide 3}\\
19 \visible<?(B.3:B.last)> {Only on slide 5}\\
20 \visible<?(C.3)> {Only on slide 8}\\
21 \end{frame}
22 \end{document}
```

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

2 Named overlay sets

2.1 Presentation

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

2.2 Named overlay reference

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

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

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

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

⟨*frame id*⟩`!`⟨*short name*⟩⟨*dotted path*⟩ : denoted by *qualified full names* like `X!A.1` and `Y!C.2`.

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

The *dotted path* is a string `.`⟨*component*$_1$⟩`.`⟨*component*$_2$⟩`....`⟨*component*$_n$⟩, where each ⟨*component*$_i$⟩ denotes either an integer, eventually signed, or a ⟨*short name*⟩. The *dotted path* can be empty for which `n` is 0.

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

2.3 Defining named overlay sets

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

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

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

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

2.3.1 Basic case

In the possible values for ⟨*spec*⟩ hereafter, ⟨*value*⟩, ⟨*first*⟩, ⟨*length*⟩ and ⟨*last*⟩ are algebraic expression (with algebraic operators +, −, ...) possibly involving any *named overlay reference* defined above.

⟨*value*⟩**,** the simple *value specifiers* for the whole signed integers set. If only the ⟨*key*⟩ is provided, the ⟨*value*⟩ defaults to 1.

⟨*first*⟩**`:` and** ⟨*first*⟩**`::`,** for the infinite range of signed integers starting at and including ⟨*first*⟩.

`:`⟨*last*⟩**,** for the infinite range of signed integers ending at and including ⟨*last*⟩.

⟨*first*⟩**`:`**⟨*last*⟩**,** ⟨*first*⟩**`::`**⟨*length*⟩**,** **`:`**⟨*last*⟩**`::`**⟨*length*⟩**,** **`::`**⟨*length*⟩**`:`**⟨*last*⟩**,** are variants for the finite range of signed integers starting at and including ⟨*first*⟩, ending at and including ⟨*last*⟩. At least one of ⟨*first*⟩ or ⟨*last*⟩ must be provided. We always have ⟨*first*⟩ + ⟨*length*⟩ = ⟨*last*⟩ + 1.

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside LaTeX environments, each new call cumulates with the previous one. Notice that the argument of this function can contain macros: they will be exhaustively expanded at resolution time[1].

2.3.2 List specifiers

Also possible values are *list specifiers* which are comma separated lists of ⟨*path*⟩=⟨*spec*⟩ definitions. The definition

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

is a convenient shortcut for

⟨*ref*⟩.⟨*path*$_1$⟩=⟨*spec*$_1$⟩,
⟨*ref*⟩.⟨*path*$_2$⟩=⟨*spec*$_2$⟩,
...,
⟨*ref*⟩.⟨*path*$_n$⟩=⟨*spec*$_n$⟩.

The rules above can apply individually to each line.

To support an array like syntax, we can omit the ⟨*path*⟩ key and only give the ⟨*spec*⟩ value. The first missing ⟨*path*⟩ key is replaced by 1, the second by 2, and so on.

2.3.3 `.n` specifiers

⟨*ref*⟩`.n=`⟨*value*⟩ is used to set the value of the index counter defined below.

3 Named overlay resolution

Turning a *named overlay reference* into the static integer it represents, as when above `<?(A.1)>` was replaced by 1, is denoted by *named overlay resolution* or simply *resolution*. This section is devoted to *resolution rules* depending on the definition of the named overlay set. Here ⟨*i*⟩ denotes a signed integer whereas ⟨*first*⟩, ⟨*last*⟩ and ⟨*length*⟩ stand for integers, or integer valued algebraic expressions.

3.1 Simple definitions

⟨*ref*⟩ `=` ⟨*value*⟩ For an unlimited range

reference	resolution
⟨*ref*⟩`.1`	⟨*value*⟩
⟨*ref*⟩`.2`	⟨*value*⟩ + 1
⟨*ref*⟩.⟨*i*⟩	⟨*value*⟩ + ⟨*i*⟩ − 1

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

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

[1] Precision is needed for the exact time when the expansion occurs.

Notice that ⟨*ref*⟩`.previous` and ⟨*ref*⟩`.0` are sometimes synonyms.

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

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

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

For a range limited from both above and below:

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

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

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

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

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.

3.2 Counters

Each named overlay set defined has a dedicated value counter which is some kind of variable that can be used and incremented. A standalone ⟨*ref*⟩ *named value reference* is resolved into the position of this value counter. For each frame, this variable is initialized to the first available amongst ⟨*value*⟩, ⟨*name*⟩`.first` or ⟨*name*⟩`.last`. If none is available, an error is raised.

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

⟨*name*⟩`+=`⟨*integer expression*⟩, resolve ⟨*integer expression*⟩ into ⟨*integer*⟩, advance the value counter by ⟨*integer*⟩ and use the new position. Here ⟨*integer expression*⟩ is the longest character sequence with no space[2].

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

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

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

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

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

Here again, ⟨*integer expression*⟩ denotes the longest character sequence with no space.

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

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

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

These counters are reset to their default value for each new frame, which is 1 for the ⟨*name*⟩`.n` counter, and whichever ⟨*name*⟩`.first` or ⟨*name*⟩`.last` is defined for the ⟨*name*⟩ counter.

3.3 Dotted paths

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

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

```
\Beanoves {
  A = 3,
  B = 3,
  B.3 = 0,
}
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\ttfamily
\BeanovesEval[see](A.1) == 3,
\BeanovesEval[see](A.3) == 5,
\BeanovesEval[see](B.1) == 3,
\BeanovesEval[see](B.3) == 0,
\end{frame}
```

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

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

3.4 Frame id

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

In order to set the *frame id* of the current frame to ⟨*frame id*⟩, use the new `beanoves id` option of the beamer frame environment.

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

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

4 ?(...) query expressions

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

The ⟨*queries*⟩ argument is a comma separated list of individual ⟨*query*⟩'s from next table. Sometimes, using ⟨*name*⟩`.range` is not allowed because the resolution would be interpreted as an algebraic difference instead of a beamer range. If it is not possible, an error is raised.

query	resolution	limitation
⟨*start expr*⟩	⟨*start*⟩	
⟨*start expr*⟩`:`	⟨*start*⟩ `-`	no ⟨*name*⟩`.range`
⟨*start expr*⟩`:`⟨*end expr*⟩	⟨*start*⟩ `-` ⟨*end*⟩	no ⟨*name*⟩`.range`
`::`⟨*length expr*⟩`:`⟨*end expr*⟩	⟨*start*⟩ `-` ⟨*end*⟩	no ⟨*name*⟩`.range`
`:`⟨*end expr*⟩	`-` ⟨*end*⟩	no ⟨*name*⟩`.range`
`:`	`-`	
⟨*start expr*⟩`::`	⟨*start*⟩ `-`	no ⟨*name*⟩`.range`
⟨*start expr*⟩`::`⟨*length expr*⟩	⟨*start*⟩ `-` ⟨*end*⟩	no ⟨*name*⟩`.range`
`:`⟨*end expr*⟩`::`⟨*length expr*⟩	⟨*start*⟩ `-` ⟨*end*⟩	no ⟨*name*⟩`.range`
`::`	`-`	

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

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

5 Support

See https://github.com/jlaurens/beanoves. One can report issues.

6 Implementation

Identify the internal prefix (LaTeX3 DocStrip convention, unused).

```
1 ⟨@@=bnvs⟩
```

Reserved namespace: identifiers containing the case insensitive string `beanoves` or containing the case insensitive string `bnvs` delimited by two non characters.

6.1 Package declarations

```
2 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
3 \ProvidesExplPackage
4   {beanoves}
5   {2024/01/11}
6   {1.0}
7   {Named overlay specifications for beamer}
```

6.2 Facility layer: definitions and naming

In order to make the code shorter and easier to read, we add a layer over LaTeX3. The `c` and `v` argument specifiers take a slightly different meaning when used in a function which name contains with `bnvs` or `BNVS`. Where LaTeX3 would transform `l__bnvs_ref_tl` into `\l__bnvs_ref_tl`, `bnvs` will directly transform `ref` into `\l__bnvs_ref_tl`. The type of the local variable used depends on the context and may be `seq` or `int` for example. There are however a pair of exceptions mentionned below. For a better reading experience, '`ref`' will generally stand for `\l__bnvs_ref_tl`, whereas '`path` sequence' will generally stand for `\l__bnvs_path_seq`. Other similar shortcuts are used as well.

Functions with `BNVS` in their names are management functions. They belong to a deeper layer and do not contain any logic specific to the `beanoves` package.

`\BNVS:c` `\BNVS:c {⟨cs core name⟩}`
`\BNVS_l:cn` `\BNVS_l:cn {⟨local variable core name⟩} {⟨ type ⟩}`
`\BNVS_g:cn` `\BNVS_g:cn {⟨global variable core name⟩} {⟨ type ⟩}`

These are naming functions.

```
\cs_new:Npn \BNVS:c    #1    { __bnvs_#1     }
\cs_new:Npn \BNVS_l:cn #1 #2 { l__bnvs_#1_#2 }
\cs_new:Npn \BNVS_g:cn #1 #2 { g__bnvs_#1_#2 }
```

`\BNVS_use_raw:c` `\BNVS_use_raw:c {⟨cs name⟩}`
`\BNVS_use_raw:Nc` `\BNVS_use_raw:Nc ⟨function⟩ {⟨cs name⟩}`
`\BNVS_use_raw:nc` `\BNVS_use_raw:nc {⟨tokens⟩} {⟨cs name⟩}`
`\BNVS_use:c` `\BNVS_use:c {⟨cs core⟩}`
`\BNVS_use:Nc` `\BNVS_use:Nc ⟨function⟩ {⟨cs core⟩}`
`\BNVS_use:nc` `\BNVS_use:nc {⟨tokens⟩} {⟨cs core⟩}`

`\BNVS_use_raw:c` is a wrapper over `\use:c`. possibly prepended with some code. It needs 3 expansion steps just like `\BNVS_use:c`. The other are used to expand `\use:c` enough before usage by ⟨*function*⟩ or ⟨*tokens*⟩. The first argument of ⟨*function*⟩ has type `N`. The next token after ⟨*tokens*⟩ will have type `N` too. ⟨*cs name*⟩ is a full cs name whereas ⟨*cs core*⟩ will be prepended with the appropriate prefix.

```
\cs_new:Npn \BNVS_use_raw:N #1 { #1 }
\cs_new:Npn \BNVS_use_raw:c #1 {
  \exp_last_unbraced:No
  \BNVS_use_raw:N { \cs:w #1 \cs_end: }
}
\cs_new:Npn \BNVS_use:c #1 {
  \BNVS_use_raw:c { \BNVS:c { #1 } }
}
\cs_new:Npn \BNVS_use_raw:NN #1 #2 {
  #1 #2
}
\cs_new:Npn \BNVS_use_raw:nN #1 #2 {
  #1 #2
}
\cs_new:Npn \BNVS_use_raw:Nc #1 #2 {
  \exp_last_unbraced:NNo
  \BNVS_use_raw:NN #1 { \cs:w #2 \cs_end: }
}
```

```
\cs_new:Npn \BNVS_use_raw:nc #1 #2 {
  \exp_last_unbraced:Nno
  \BNVS_use_raw:nN { #1 } { \cs:w #2 \cs_end: }
}
\cs_new:Npn \BNVS_use:Nc #1 #2 {
  \BNVS_use_raw:Nc #1 { \BNVS:c { #2 } }
}
\cs_new:Npn \BNVS_use:nc #1 #2 {
  \BNVS_use_raw:nc { #1 } { \BNVS:c { #2 } }
}

\cs_new:Npn \BNVS_log:n #1 { }
\cs_generate_variant:Nn \BNVS_log:n { x }
\cs_new:Npn \BNVS_DEBUG_on: {
  \cs_set:Npn \BNVS_DEBUG_log:n { \BNVS_log:n }
}
\cs_new:Npn \BNVS_DEBUG_off: {
  \cs_set:Npn \BNVS_DEBUG_log:n { \use_none:n }
}
\BNVS_DEBUG_off:
```

`\BNVS_new:cpn`
`\BNVS_set:cpn`

`\BNVS_new:cpn` is like `\cs_new:cpn` except that the name argument is tagged for beanoves package. Similarly for `\BNVS_set:cpn`.

```
\cs_new:Npn \BNVS_new:cpn #1 {
  \cs_new:cpn { \BNVS:c { #1 } }
}
\cs_new:Npn \BNVS_set:cpn #1 {
  \cs_set:cpn { \BNVS:c { #1 } }
}

\cs_generate_variant:Nn \cs_generate_variant:Nn { c }
\cs_new:Npn \BNVS_generate_variant:cn #1 {
  \cs_generate_variant:cn { \BNVS:c { #1 } }
}
```

6.3 logging

Utility messaging.

```
\msg_new:nnn { beanoves } { :n } { #1 }
\msg_new:nnn { beanoves } { :nn } { #1~(#2) }
\BNVS_new:cpn { warning:n } {
  \msg_warning:nnn { beanoves } { :n }
}
\BNVS_generate_variant:cn { warning:n } { x }
\cs_new:Npn \BNVS_error:n {
  \msg_error:nnn { beanoves } { :n }
}
\cs_new:Npn \BNVS_error:x {
  \msg_error:nnx { beanoves } { :n }
}
\cs_new:Npn \BNVS_fatal:n {
  \msg_fatal:nnn { beanoves } { :n }
```

```
}
\cs_new:Npn \BNVS_fatal:x {
  \msg_fatal:nnx { beanoves } { :n }
}
```

6.4 Facility layer: Variables

\BNVS_N_new:c
\BNVS_v_new:c

`\BNVS_N_new:n {`⟨*type*⟩`}`

Creates typed utility functions, see usage below. Undefined when no longer used. ⟨*type*⟩ is one of `tl`, `seq`...

```
\cs_new:Npn \BNVS_N_new:c #1 {
  \cs_new:cpn { BNVS_#1:c } ##1 {
    l \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
  }
  \cs_new:cpn { BNVS_#1_new:c } ##1 {
    \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
  }
  \cs_new:cpn { BNVS_#1_use:c } ##1 {
    \use:c { \use:c { BNVS_#1:c } { ##1 } }
  }
  \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
    \BNVS_use_raw:Nc
      ##1 { \use:c { BNVS_#1:c } { ##2 } }
  }
  \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
    \BNVS_use_raw:nc
      { ##1 } { \use:c { BNVS_#1:c } { ##2 } }
  }
}
\cs_new:Npn \BNVS_v_new:c #1 {
  \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
    \BNVS_use_raw:nc
      { \exp_args:NV ##1 }
      { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
  }
  \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
    \BNVS_use_raw:nc
      { \exp_args:NnV \use:n { ##1 } }
      { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
  }
}
\BNVS_N_new:c { bool }
\BNVS_N_new:c { int }
\BNVS_v_new:c { int }
\BNVS_N_new:c { tl }
\BNVS_v_new:c { tl }
\BNVS_N_new:c { str }
\BNVS_v_new:c { str }
\BNVS_N_new:c { seq }
\BNVS_v_new:c { seq }
\cs_undefine:N \BNVS_N_new:c
```

\BNVS_use:Ncn \BNVS_use:Ncn ⟨*function*⟩ {⟨*core name*⟩} {⟨*type*⟩}

```
\cs_new:Npn \BNVS_use:Ncn #1 #2 #3 {
  \BNVS_use_raw:c { BNVS_#3_use:Nc }   #1   { #2 }
}
\cs_new:Npn \BNVS_use:ncn #1 #2 #3 {
  \BNVS_use_raw:c { BNVS_#3_use:nc } { #1 } { #2 }
}
\cs_new:Npn \BNVS_use:Nvn #1 #2 #3 {
  \BNVS_use_raw:c { BNVS_#3_use:Nv }   #1   { #2 }
}
\cs_new:Npn \BNVS_use:nvn #1 #2 #3 {
  \BNVS_use_raw:c { BNVS_#3_use:nv } { #1 } { #2 }
}
\cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:Ncn   #1   { #2 } { #3 }
  }
}
\cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:ncn { #1 } { #2 } { #3 }
  }
}
\cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:Nvn   #1   { #2 } { #3 }
  }
}
\cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:nvn { #1 } { #2 } { #3 }
  }
}
\cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
  \BNVS_use:ncn {
    \BNVS_use:Ncncn   #1   { #2 } { #3 } { #4 } { #5 }
  }
}
\cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
  \BNVS_use:ncn {
    \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
  }
}
```

\BNVS_new_c:cn \BNVS_new_c:nc {⟨*type*⟩} {⟨*core name*⟩}

```
\cs_new:Npn \BNVS_new_c:nc #1 #2 {
  \BNVS_new:cpn { #1_#2:c } {
    \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
  }
}
\cs_new:Npn \BNVS_new_cn:nc #1 #2 {
  \BNVS_new:cpn { #1_#2:cn } ##1 {
```

```
    \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
  }
}
\cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
  \BNVS_new:cpn { #2:cnn } ##1 {
    \BNVS_use:Ncn { #3 } { ##1 } { #1 }
  }
}
\cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cnn:ncN { #1 } { #1_#2 }
  } { #1_#2:Nnn }
}
\cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
  \BNVS_new:cpn { #2:cnv } ##1 ##2 {
    \BNVS_tl_use:nv {
      \BNVS_use:Ncn #3 { ##1 } { #1 } {  ##2 }
    }
  }
}
\cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cnv:ncN { #1 } { #1_#2 }
  } { #1_#2:Nnn }
}
\cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
  \BNVS_new:cpn { #2:cnx } ##1 ##2 {
    \exp_args:Nnx \use:n {
      \BNVS_use:Ncn #3 { ##1 } { #1 } {  ##2 }
    }
  }
}
\cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cnx:ncN { #1 } { #1_#2 }
  } { #1_#2:Nnn }
}
\cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
  \BNVS_new:cpn { #2:cc } ##1 ##2 {
    \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
  }
}
\cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cc:ncNn { #1 } { #1_#2 }
  } { #1_#2:NN }
}
\cs_new:Npn \BNVS_new_cc:nc #1 #2 {
  \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
}
\cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
  \BNVS_new:cpn { #2:cn } ##1 {
    \BNVS_use:Ncn #3 { ##1 } { #1 }
  }
```

```
}
\cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cn:ncNn { #1 } { #1_#2 }
  } { #1_#2:Nn }
}
\cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
  \BNVS_new:cpn { #2:cv } ##1 ##2 {
    \BNVS_use:nvn {
      \BNVS_use:Ncn #3 { ##1 } { #1 }
    } { ##2 } { #4 }
  }
}
\cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cv:ncNn { #1 } { #1_#2 }
  } { #1_#2:Nn }
}
\cs_new:Npn \BNVS_new_cv:nc #1 #2 {
  \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
}
\cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
  \BNVS_use_raw:Nc   #1   { \BNVS_l:cn { #2 } { #3 } }
}
\cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
  \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
}
\cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
  \BNVS_use_raw:Nc   #1   { \BNVS_g:cn { #2 } { #3 } }
}
\cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
  \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
}
\cs_new:Npn \BNVS_g_prop_use:Nc #1 #2 {
  \BNVS_use_raw:Nc   #1   { \BNVS_g:cn { #2 } { prop } }
}
\cs_new:Npn \BNVS_g_prop_use:nc #1 #2 {
  \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { prop } }
}
\cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
  \BNVS_use:ncncncn { \exp_args:NVVV #1 }
    { #2 } { tl } { #3 } { tl } { #4 } { tl }
}
```

\BNVS_new_conditional:cpnn `\BNVS_new_conditional:cpnn {`⟨*core name*⟩`} ⟨`*parameter*`⟩ {`⟨*conditions*⟩`} {`⟨*code*⟩`}`

```
\cs_generate_variant:Nn \prg_new_conditional:Npnn { c }
\cs_new:Npn \BNVS_new_conditional:cpnn #1 {
  \prg_new_conditional:cpnn { \BNVS:c { #1 } }
}
\cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
\cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
  \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
}
```

```
\cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
    \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vn:cNnn { #1 }
  } { #1:nn TF } { #2 }
}
\cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
    \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vc:cnn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vc:cNnn { #1 }
  } { #1:ncTF }
}
\cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
    \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vc:cNn { #1 }
  } { #1:ncTF }
}
```

6.4.1 Regex

```
\cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
  \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
}
```

__bnvs_match_once:Nn*TF*
__bnvs_match_once:Nv*TF*
__bnvs_match_once:nn*TF*
__bnvs_regex_split:cn*TF*

__bnvs_match_once:NnTF ⟨*regex variable*⟩ {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_match_once:nnTF {⟨*regex*⟩} {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_regex_split:cncTF {⟨*regex core*⟩} {⟨*expression*⟩} ⟨*seq core*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_regex_split:cnTF {⟨*regex core*⟩} {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

- `\regex_match_once:NnNTF` with the `match` sequence as N argument
- `\regex_match_once:nnNTF` with the `match` sequence as N argument
- `\regex_split:NnNTF` with the `split` sequence as last N argument

```
\BNVS_new_conditional:cpnn { match_once:Ncn } #1 #2 #3 { T, F, TF } {
  \BNVS_use:ncn {
    \regex_extract_once:NnNTF #1 { #3 }
  } { #2 } { seq } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { match_once:Nn } #1 #2 { T, F, TF } {
  \BNVS_use:ncn {
    \regex_extract_once:NnNTF #1 { #2 }
  } { match } { seq } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { match_once:Ncv } #1 #2 #3 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_tl_use:nv {
      \regex_extract_once:NnNTF #1
    } { #3 }
  } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { match_once:Nv } #1 #2 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_tl_use:nv {
      \regex_extract_once:NnNTF #1
    } { #2 }
  } { match } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
```

```
}
\BNVS_new_conditional:cpnn { match_once:nn } #1 #2 { T, F, TF } {
  \BNVS_seq_use:nc {
    \regex_extract_once:nnNTF { #1 } { #2 }
  } { match } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { regex_split:cnc } #1 #2 #3 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
  } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { regex_split:cn } #1 #2 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
  } { split } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.4.2 Token lists

__bnvs_tl_clear:c	__bnvs_tl_clear:c {⟨*core key tl*⟩}
__bnvs_tl_use:c	__bnvs_tl_use:c {⟨*core*⟩}
__bnvs_tl_set_eq:cc	__bnvs_tl_count:c {⟨*core*⟩}
__bnvs_tl_set:cn	__bnvs_tl_set_eq:cc {⟨*lhs core name*⟩} {⟨*rhs core name*⟩}
__bnvs_tl_set:(cv|cx)	__bnvs_tl_set:cn {⟨*core*⟩} {⟨*tl*⟩}
__bnvs_tl_put_left:cn	__bnvs_tl_set:cv {⟨*core*⟩} {⟨*value core name*⟩}
__bnvs_tl_put_right:cn	__bnvs_tl_put_left:cn {⟨*core*⟩} {⟨*tl*⟩}
__bnvs_tl_put_right:(cx|cv)	__bnvs_tl_put_right:cn {⟨*core*⟩} {⟨*tl*⟩}
	__bnvs_tl_put_right:cv {⟨*core*⟩} {⟨*value core name*⟩}

These are shortcuts to

- \tl_clear:c {l__bnvs_⟨*core*⟩_tl}
- \tl_use:c {l__bnvs_⟨*core*⟩_tl}
- \tl_set_eq:cc {l__bnvs_⟨*lhs core*⟩_tl}{l__bnvs_⟨*rhs core*⟩_tl}
- \tl_set:cv {l__bnvs_⟨*core*⟩_tl}{l__bnvs_⟨*value core*⟩_tl}
- \tl_set:cx {l__bnvs_⟨*core*⟩_tl}{⟨*tl*⟩}
- \tl_put_left:cn {l__bnvs_⟨*core*⟩_tl}{⟨*tl*⟩}
- \tl_put_right:cn {l__bnvs_⟨*core*⟩_tl}{⟨*tl*⟩}
- \tl_put_right:cv {l__bnvs_⟨*core*⟩_tl}{l__bnvs_⟨*value core*⟩_tl}

\BNVS_new_conditional_vnc:cn \BNVS_new_conditional_vnc:cn {⟨*core*⟩} {⟨*conditions*⟩}

⟨*function*⟩ is the test function with signature ...:nncTF. ⟨*core*⟩:nncTF is used for testing.

```
\cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
    \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vnc:cNn { #1 }
  } { #1:nncTF }
}
```

\BNVS_new_conditional_vnc:cn \BNVS_new_conditional_vnc:cn {⟨*core*⟩} {⟨*conditions*⟩}

Forwards to \BNVS_new_conditional_vnc:cNn with \⟨*core*⟩:nncTF as function argument. Used for testing.

```
\cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:Nv #2 { ##1 }
    } { ##2 } { ##3 } { ##4 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvnc:cNn { #1 }
  } { #1:nnncTF }
}
\cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:nv {
        \BNVS_tl_use:Nv #2 { ##1 }
      } { ##2 }
    } { ##3 } { ##4 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvvc:cNn { #1 }
  } { #1:nnncTF }
}
\cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:Nv #2 { ##1 }
    } { ##2 } { ##3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvc:cNn { #1 }
  } { #1:nncTF }
}
\cs_new:Npn \BNVS_new_tl_c:c {
  \BNVS_new_c:nc { tl }
}
```

```
\BNVS_new_tl_c:c { clear }
\BNVS_new_tl_c:c { use }
\BNVS_new_tl_c:c { count }

\BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
  \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
}
\cs_new:Npn \BNVS_new_tl_cn:c {
  \BNVS_new_cn:nc { tl }
}
\cs_new:Npn \BNVS_new_tl_cv:c #1 {
  \BNVS_new_cv:ncn { tl } { #1 } { tl }
}
\BNVS_new_tl_cn:c { set }
\BNVS_new_tl_cv:c { set }
\BNVS_new:cpn { tl_set:cx } {
  \exp_args:Nnx \__bnvs_tl_set:cn
}
\BNVS_new_tl_cn:c { put_right }
\BNVS_new_tl_cv:c { put_right }
% \BNVS_generate_variant:cn { tl_put_right:cn } { cx }
\BNVS_new:cpn { tl_put_right:cx } {
  \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
}
\BNVS_new_tl_cn:c { put_left }
\BNVS_new_tl_cv:c { put_left }
% \BNVS_generate_variant:cn { tl_put_left:cn } { cx }
\BNVS_new:cpn { tl_put_left:cx } {
  \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
}
```

`\__bnvs_tl_if_empty:c`*TF* `\__bnvs_tl_if_empty:cTF` {⟨*core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_tl_if_blank:v`*TF* `\__bnvs_tl_if_blank:vTF` {⟨*core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_tl_if_eq:cn`*TF* `\__bnvs_tl_if_eq:cnTF` {⟨*core*⟩} {⟨*tl*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

- `\tl_if_empty:cTF` {`l__bnvs_`⟨*core*⟩`_tl`} {⟨*yes code*⟩} {⟨*no code*⟩}
- `\tl_if_eq:cnTF` {`l__bnvs_`⟨*core*⟩`_tl`}{⟨*tl*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

```
\cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
    \BNVS_use:Ncn #3 { ##1 } { #1 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
  } { #1_#2:NTF }
```

```
}
\BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }
\BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
  \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
    \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
  } { #1_#2:NnTF }
}
\BNVS_new_conditional_cn:ncn { tl } { if_eq } { T, F, TF }
\cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:cv } ##1 ##2 { #4 } {
    \BNVS_use:nvn {
      \BNVS_use:Ncn #3 { ##1 } { #1 }
    } { ##2 } { #1 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
  } { #1_#2:NnTF }
}
\BNVS_new_conditional_cv:ncn { tl } { if_eq } { T, F, TF }
```

6.4.3 Strings

__bnvs_str_if_eq:vn*TF* __bnvs_str_if_eq:vnTF {⟨*core*⟩} {⟨*tl*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

- \str_if_eq:ccTF {l__bnvs_⟨*core*⟩_tl}{⟨*yes code*⟩} {⟨*no code*⟩}

```
\cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:vn } ##1 ##2 { #4 } {
    \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
```

```
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
  } { #1_#2:nnTF }
}
\BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }
\cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
    \BNVS_use:nvn {
      \BNVS_use:Nvn #3 { ##1 } { #1 }
    }  { ##2 } { #1 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
  } { #1_#2:nnTF }
}
\BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }
```

6.4.4 Sequences

__bnvs_seq_count:c	__bnvs_seq_new:c {⟨*core*⟩}
__bnvs_seq_clear:c	__bnvs_seq_count:c {⟨*core*⟩}
__bnvs_seq_set_eq:cc	__bnvs_seq_clear:c {⟨*core*⟩}
__bnvs_seq_use:cn	__bnvs_seq_set_eq:cc {⟨$core_1$⟩} {⟨$core_2$⟩}
__bnvs_seq_item:cn	__bnvs_seq_use:cn {⟨*core*⟩} {⟨*separator*⟩}
__bnvs_seq_remove_all:cn	__bnvs_seq_item:cn {⟨*core*⟩} {⟨*integer expression*⟩}
__bnvs_seq_put_left:cv	__bnvs_seq_remove_all:cn {⟨*core*⟩} {⟨*tl*⟩}
__bnvs_seq_put_right:cn	__bnvs_seq_put_right:cn {⟨*seq core*⟩} {⟨*tl*⟩}
__bnvs_seq_put_right:cv	__bnvs_seq_put_right:cv {⟨*seq core*⟩} {⟨*tl core*⟩}
__bnvs_seq_set_split:cnn	__bnvs_seq_set_split:cnn {⟨*seq core*⟩} {⟨*tl*⟩} {⟨*separator*⟩}
__bnvs_seq_set_split:(cnv|cnx)	__bnvs_seq_pop_left:cc {⟨$core_1$⟩} {⟨$core_2$⟩}
__bnvs_seq_pop_left:cc	

These are shortcuts to

- \seq_set_eq:cc {l__bnvs_⟨$core_1$⟩_seq} {l__bnvs_⟨$core_2$⟩_seq}
- \seq_count:c {l__bnvs_⟨*core*⟩_seq}
- \seq_use:cn {l__bnvs_⟨*core*⟩_seq}{⟨*separator*⟩}
- \seq_item:cn {l__bnvs_⟨*core*⟩_seq}{⟨*integer expression*⟩}
- \seq_remove_all:cn {l__bnvs_⟨*core*⟩_seq}{⟨*tl*⟩}
- __bnvs_seq_clear:c {l__bnvs_⟨*core*⟩_seq}
- \seq_put_right:cv {l__bnvs_⟨*seq core*⟩_seq} {l__bnvs_⟨*tl core*⟩_tl}
- \seq_set_split:cnn{l__bnvs_⟨*seq core*⟩_seq}{l__bnvs_⟨*tl core*⟩_tl}{⟨*tl*⟩}

```
\BNVS_new_c:nc   { seq } { count }
\BNVS_new_c:nc   { seq } { clear }
\BNVS_new_cn:nc  { seq } { use }
\BNVS_new_cn:nc  { seq } { item }
\BNVS_new_cn:nc  { seq } { remove_all }
\BNVS_new_cn:nc  { seq } { map_inline }
\BNVS_new_cc:nc  { seq } { set_eq }
\BNVS_new_cv:ncn { seq } { put_left  } { tl }
\BNVS_new_cn:ncn { seq } { put_right } { tl }
\BNVS_new_cv:ncn { seq } { put_right } { tl }
\BNVS_new_cnn:nc { seq } { set_split }
\BNVS_new_cnv:nc { seq } { set_split }
\BNVS_new_cnx:nc { seq } { set_split }
\BNVS_new_cc:ncn { seq } { pop_left  } { tl }
\BNVS_new_cc:ncn { seq } { pop_right } { tl }
```

__bnvs_seq_if_empty:c*TF*	\ __bnvs_seq_if_empty:cTF {⟨*seq core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_seq_get_right:cc*TF*	\ __bnvs_seq_get_right:ccTF {⟨*seq core*⟩} {⟨*tl core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_seq_pop_left:cc*TF*	
__bnvs_seq_pop_right:cc*TF*	

```
\cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #1_#2:cc } ##1 ##2 { #4 } {
    \BNVS_use:ncncn {
      \BNVS_use_raw:c { #1_#2:NNTF }
    } { ##1 } { #1 } { ##2 } { #3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
\BNVS_new_conditional_cc:ncnn
  { seq } { get_right } { tl } { T, F, TF }
\BNVS_new_conditional_cc:ncnn
  { seq } { pop_left  } { tl } { T, F, TF }
\BNVS_new_conditional_cc:ncnn
  { seq } { pop_right } { tl } { T, F, TF }
```

6.4.5 Integers

__bnvs_int_new:c __bnvs_int_use:c __bnvs_int_zero:c __bnvs_int_inc:c __bnvs_int_decr:c __bnvs_int_set:cn __bnvs_int_set:cv

__bnvs_int_new:c {⟨*core*⟩}
__bnvs_int_use:c {⟨*core*⟩}
__bnvs_int_incr:c {⟨*core*⟩}
__bnvs_int_decr:c {⟨*core*⟩}
__bnvs_int_set:cn {⟨*core*⟩} {⟨*value*⟩}

These are shortcuts to

- \int_new:c {l__bnvs_⟨*core*⟩_int}
- \int_use:c {l__bnvs_⟨*core*⟩_int}
- \int_incr:c {l__bnvs_⟨*core*⟩_int}
- \int_idecr:c {l__bnvs_⟨*core*⟩_int}
- \int_set:cn {l__bnvs_⟨*core*⟩_int} ⟨*value*⟩

```
\BNVS_new_c:nc   { int } { new  }
\BNVS_new_c:nc   { int } { use  }
\BNVS_new_c:nc   { int } { zero }
\BNVS_new_c:nc   { int } { incr }
\BNVS_new_c:nc   { int } { decr }
\BNVS_new_cn:nc  { int } { set  }
\BNVS_new_cv:ncn { int } { set  } { int }
```

6.4.6 Prop

__bnvs_prop_get:Nnc*TF*

```
\BNVS_new_conditional:cpnn { prop_get:Nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_use:ncn {
    \prop_get:NnNTF #1 { #2 }
  } { #3 } { tl } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.5 Debug 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. In particular, we have aliases for `\group_begin:` and `\group_end:` to allow the display of supplemental informations while debugging.

6.6 Debug messages

6.7 Variable facilities

6.8 Testing facilities

6.9 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. The number of variables used has not been optimized, nor the TEX groups used. Optimization often goes against readability.

```
\tl_new:N \l__bnvs_id_last_tl
\tl_set:Nn \l__bnvs_id_last_tl { ?! }
\tl_new:N \l__bnvs_a_tl
\tl_new:N \l__bnvs_b_tl
\tl_new:N \l__bnvs_c_tl
\tl_new:N \l__bnvs_V_tl
\tl_new:N \l__bnvs_A_tl
\tl_new:N \l__bnvs_L_tl
\tl_new:N \l__bnvs_Z_tl
\tl_new:N \l__bnvs_ans_tl
\tl_new:N \l__bnvs_Q_name_tl
\tl_new:N \l__bnvs_FQ_name_tl
\tl_new:N \l__bnvs_key_tl
\tl_new:N \l__bnvs_key_base_tl
\tl_new:N \l__bnvs_ref_tl
\tl_new:N \l__bnvs_ref_base_tl
\tl_new:N \l__bnvs_id_tl
\tl_new:N \l__bnvs_n_tl
\tl_new:N \l__bnvs_path_tl
\tl_new:N \l__bnvs_group_tl
\tl_new:N \l__bnvs_scan_tl
\tl_new:N \l__bnvs_query_tl
```

```
\tl_new:N \l__bnvs_token_tl
\tl_new:N \l__bnvs_root_tl
\tl_new:N \l__bnvs_n_incr_tl
\tl_new:N \l__bnvs_incr_tl
\tl_new:N \l__bnvs_post_tl
\tl_new:N \l__bnvs_suffix_tl
\int_new:N \g__bnvs_call_int
\int_new:N \l__bnvs_int
\int_new:N \l__bnvs_i_int
\seq_new:N \g__bnvs_def_seq
\seq_new:N \l__bnvs_a_seq
\seq_new:N \l__bnvs_b_seq
\seq_new:N \l__bnvs_ans_seq
\seq_new:N \l__bnvs_match_seq
\seq_new:N \l__bnvs_split_seq
\seq_new:N \l__bnvs_path_seq
\seq_new:N \l__bnvs_path_base_seq
\seq_new:N \l__bnvs_query_seq
\seq_new:N \l__bnvs_token_seq
\bool_new:N \l__bnvs_in_frame_bool
\bool_set_false:N \l__bnvs_in_frame_bool
\bool_new:N \l__bnvs_parse_bool
```

In order to implement the provide feature, we add getters and setters

```
\bool_new:N \l__bnvs_provide_bool
\BNVS_new:cpn { provide_on: } {
  \bool_set_true:N \l__bnvs_provide_bool
}
\BNVS_new:cpn { provide_off: } {
  \bool_set_false:N \l__bnvs_provide_bool
}
\__bnvs_provide_off:
```

`\__bnvs_if_provide:TF` `\__bnvs_if_provide:TF {⟨yes code⟩} {⟨no code⟩}`

Execute ⟨*yes code*⟩ when in provide mode (see `\Beanoves*{...}`), ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { if_provide: } { p, T, F, TF } {
  \bool_if:NTF \l__bnvs_provide_bool {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.10 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 of 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 {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Decrement the `\g__bnvs_call_int` counter globally and execute ⟨*yes code*⟩ if we have not reached 0, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { call: } { T, F, TF } {
  \int_gdecr:N \g__bnvs_call_int
  \int_compare:nNnTF \g__bnvs_call_int > 0 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.11 Overlay specification

6.12 Basic functions

\g__bnvs_prop ⟨*key*⟩–⟨*integer spec*⟩ property list to store the named overlay sets. The keys are constructed from fully qualified names denoted as ⟨*FQ name*⟩.

⟨*FQ name*⟩`/V` for the value

⟨*FQ name*⟩`/A` for the first index

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

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

The implementation is private, in particular, keys may change in future versions. They are exposed here for informational purposes only.

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

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

__bnvs_gput:nnn
__bnvs_gput:(nvn|nnv)
__bnvs_item:nn
__bnvs_gremove:nn
__bnvs_gclear:n
__bnvs_gclear:v
__bnvs_gclear:

`\__bnvs_gput:nnn {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`} {`⟨*integer spec*⟩`}`
`\__bnvs_item:nn {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`}`
`\__bnvs_gremove:nn {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`}`
`\__bnvs_gclear:n {`⟨*FQ name*⟩`}`
`\__bnvs_gclear:`

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over LaTeX3 eponym functions. The key used in `\g__bnvs_prop` is ⟨*FQ name*⟩`/`⟨*subkey*⟩. In practice, ⟨*subkey*⟩ is one of `V`, `A`, `L`, `Z`. `fq` means "fully qualified".

```
\BNVS_new:cpn { gput:nnn } #1 #2 {
  \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
}
```

```
\BNVS_new:cpn { gput:nvn } #1 {
  \BNVS_tl_use:nv {
    \__bnvs_gput:nnn { #1 }
  }
}
\BNVS_new:cpn { gput:nnv } #1 #2 {
  \BNVS_tl_use:nv {
    \__bnvs_gput:nnn { #1 } { #2 }
  }
}
\BNVS_new:cpn { item:nn } #1 #2 {
  \prop_item:Nn \g__bnvs_prop { #2 / #1 }
}
\BNVS_new:cpn { gremove:nn } #1 #2 {
  \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
}
\BNVS_new:cpn { gclear:n } #1 {
  \clist_map_inline:nn { V, A, Z, L } {
    \__bnvs_gremove:nn { ##1 } { #1 }
  }
  \__bnvs_cache_gclear:n { #1 }
}
\BNVS_new:cpn { gclear: } {
  \prop_gclear:N \g__bnvs_prop
}
\BNVS_generate_variant:cn { gclear:n } { V }
\BNVS_new:cpn { gclear:v } {
   \BNVS_tl_use:Nc \__bnvs_gclear:V
}
```

__bnvs_if_in_p:nn ⋆ __bnvs_if_in_p:nn {⟨*subkey*⟩} {⟨*FQ name*⟩}
__bnvs_if_in:nn*TF* ⋆ __bnvs_if_in:nnTF {⟨*subkey*⟩} {⟨*FQ name*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_in_p:n ⋆ __bnvs_if_in_p:n {⟨*FQ name*⟩}
__bnvs_if_in:n*TF* ⋆ __bnvs_if_in:nTF {⟨*FQ name*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Convenient shortcuts to test for the existence of ⟨*FQ name*⟩/⟨*subkey*⟩, it makes the code more concise and readable. The version with no ⟨*subkey*⟩ is the or combination for keys `V`, `A` and `Z`.

```
\BNVS_new_conditional:cpnn { if_in:nn } #1 #2 { p, T, F, TF } {
  \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { if_in:n } #1 { p, T, F, TF } {
  \bool_if:nTF {
       \__bnvs_if_in_p:nn V { #1 }
    || \__bnvs_if_in_p:nn A { #1 }
    || \__bnvs_if_in_p:nn Z { #1 }
  } {
    \prg_return_true:
  } {
```

```
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { if_in:v } #1 { p, T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_if_in:nTF { #1 }
    { \prg_return_true: } { \prg_return_false: }
}
```

__bnvs_gprovide:nnnT `\__bnvs_gprovide:nnnT {⟨subkey⟩} {⟨FQ name⟩} {⟨value⟩} {⟨yes precode⟩}`

Execute ⟨*yes precode*⟩ before providing.

```
\BNVS_new:cpn { gprovide:nnnT } #1 #2 #3 #4 {
  \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
    #4
    \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
  }
}
```

__bnvs_get:nnc*TF* `\__bnvs_get:nncTF {⟨subkey⟩} {⟨FQ name⟩} {⟨tl core⟩} {⟨yes code⟩} {⟨no code⟩}`

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute ⟨*yes code*⟩ when the item is found, ⟨*no code*⟩ otherwise. In the latter case, the content of the ⟨*tl core*⟩ variable is undefined, on resolution only. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```
\BNVS_new_conditional:cpnn { get:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nc {
    \prop_get:NnNTF \g__bnvs_prop { #2 / #1 }
  } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { get:nvc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \__bnvs_get:nncTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.13 Functions with cache

\g__bnvs_cache_prop ⟨*key*⟩–⟨*value*⟩ property list to store the named overlay sets. Other keys are eventually used to cache results when some attributes are defined from other slide ranges.

⟨*FQ name*⟩`/V` for the cached static value of the value

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

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

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

⟨*FQ name*⟩`/P` for the cached static value of the previous index

⟨*FQ 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_cache_prop
```

(*End of definition for* `\g__bnvs_cache_prop`.)

`\__bnvs_cache_gput:nnn`	`\__bnvs_cache_gput:nnn {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`} {`⟨*value*⟩`}`
`\__bnvs_cache_gput:(nnv|nvn)`	`\__bnvs_cache_item:nn {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`}`
`\__bnvs_cache_item:nn`	`\__bnvs_cache_gremove:nn {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`}`
`\__bnvs_cache_gremove:nn`	`\__bnvs_cache_gclear:n {`⟨*FQ name*⟩`}`
`\__bnvs_cache_gclear:n`	`\__bnvs_cache_gclear:`
`\__bnvs_cache_gclear:`	

Wrapper over the functions above for ⟨*FQ name*⟩`/`⟨*subkey*⟩.

```
\BNVS_new:cpn { cache_gput:nnn } #1 #2 {
  \prop_gput:Nnn \g__bnvs_cache_prop { #2 / #1 }
}
\cs_generate_variant:Nn \__bnvs_cache_gput:nnn { nV, nnV }
\BNVS_new:cpn { cache_gput:nvn } #1 {
  \BNVS_tl_use:nc {
    \__bnvs_cache_gput:nVn { #1 }
  }
}
\BNVS_new:cpn { cache_gput:nnv } #1 #2 {
  \BNVS_tl_use:nc {
    \__bnvs_cache_gput:nnV { #1 } { #2 }
  }
}
\BNVS_new:cpn { cache_item:nn } #1 #2 {
  \prop_item:Nn \g__bnvs_cache_prop { #2 / #1 }
}
\BNVS_new:cpn { cache_gremove:nn } #1 #2 {
  \prop_gremove:Nn \g__bnvs_cache_prop { #2 / #1 }
}
\BNVS_new:cpn { cache_gclear:n } #1 {
  \clist_map_inline:nn { V, A, Z, L, P, N } {
    \prop_gremove:Nn \g__bnvs_cache_prop { #1 / ##1 }
  }
}
\BNVS_new:cpn { cache_gclear: } {
  \prop_gclear:N \g__bnvs_cache_prop
}
```

`\__bnvs_cache_if_in_p:nn` ⋆	`\__bnvs_cache_if_in_p:n {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`}`
`\__bnvs_cache_if_in:nn`*TF* ⋆	`\__bnvs_cache_if_in:nTF {`⟨*subkey*⟩`} {`⟨*FQ name*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

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

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

__bnvs_cache_get:nnc*TF* `\__bnvs_cache_get:nncTF {⟨subkey⟩} {⟨FQ name⟩} {⟨tl core⟩} {⟨yes code⟩} {⟨no code⟩}`

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute ⟨*yes code*⟩ when the item is found, ⟨*no code*⟩ otherwise. In the latter case, the content of the ⟨*tl core*⟩ variable is undefined. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```
\BNVS_new_conditional:cpnn { cache_get:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nc {
    \prop_get:NnNTF \g__bnvs_cache_prop { #2 / #1 }
  } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.13.1 Implicit value counter

The implicit value counter is local to the current frame. It is defined at the global level because changes made at any depth must be made at the frame depth. If the frame were a closure, this counter would belong to that closure. When used for the first time, it either defaults to the first index or last index.

\g__bnvs_v_prop ⟨*key*⟩–⟨*value*⟩ property list to store the contents or the named value counters. The keys are qualified names ⟨*id*⟩!⟨*short name*⟩ denoted as ⟨*Q name*⟩.

```
\prop_new:N \g__bnvs_v_prop
```

(*End of definition for* `\g__bnvs_v_prop`.)

__bnvs_v_gput:nn
__bnvs_v_gput:(nV|Vn)
__bnvs_v_item:n
__bnvs_v_gremove:n
__bnvs_v_gclear:

```
\__bnvs_v_gput:nn {⟨Q name⟩} {⟨value⟩}
\__bnvs_v_item:n {⟨Q name⟩}
\__bnvs_v_gremove:n {⟨Q name⟩}
\__bnvs_v_gclear:
```

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

```
\BNVS_new:cpn { v_gput:nn } {
  \prop_gput:Nnn \g__bnvs_v_prop
}
\BNVS_new:cpn { v_gput:nv } #1 {
  \BNVS_tl_use:nv {
    \__bnvs_v_gput:nn { #1 }
  }
}
```

```
\BNVS_new:cpn { v_item:n } #1 {
  \prop_item:Nn \g__bnvs_v_prop { #1 }
}
\BNVS_new:cpn { v_gremove:n } {
  \prop_gremove:Nn \g__bnvs_v_prop
}
\BNVS_new:cpn { v_gclear: } {
  \prop_gclear:N \g__bnvs_v_prop
}
```

`\__bnvs_v_if_in_p:n` ⋆ `\__bnvs_v_if_in_p:n {⟨Q name⟩}`
`\__bnvs_v_if_in:nTF` ⋆ `\__bnvs_v_if_in:nTF {⟨Q name⟩} {⟨yes code⟩} {⟨no code⟩}`

Convenient shortcuts to test for the existence of the ⟨*key*⟩ value counter.

```
\BNVS_new_conditional:cpnn { v_if_in:n } #1 { p, T, F, TF } {
  \prop_if_in:NnTF \g__bnvs_v_prop { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

`\__bnvs_v_get:ncTF` `\__bnvs_v_get:ncTF {⟨Q name⟩} ⟨tl core⟩ {⟨yes code⟩} {⟨no code⟩}`

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute ⟨*yes code*⟩ when the item is found, ⟨*no code*⟩ otherwise. In the latter case, the content of the ⟨*tl core*⟩ variable is undefined. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```
\BNVS_new_conditional:cpnn { v_get:nc } #1 #2 { T, F, TF } {
  \BNVS_tl_use:nc {
    \prop_get:NnNTF \g__bnvs_v_prop { #1 }
  } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

`\__bnvs_v_greset:nnTF` `\__bnvs_v_greset:nnTF {⟨Q name⟩} {⟨initial value⟩} {⟨yes code⟩} {⟨no code⟩}`
`\__bnvs_v_greset:vnTF` `\__bnvs_greset_all:nnTF {⟨Q name⟩} {⟨initial value⟩} {⟨yes code⟩} {⟨no code⟩}`
`\__bnvs_greset_all:nnTF`
`\__bnvs_greset_all:vnTF`

The key must include the frame id. Reset the value counter to the given ⟨*initial value*⟩. The `..._all` variant also cleans the cached values. If the ⟨*Q name*⟩ is known, ⟨*yes code*⟩ is executed, otherwise ⟨*no code*⟩ is executed.

```
\BNVS_new_conditional:cpnn { v_greset:nn } #1 #2 { T, F, TF } {
  \__bnvs_v_if_in:nTF { #1 } {
```

```
    \__bnvs_v_gremove:n { #1 }
    \tl_if_empty:nF { #2 } {
      \__bnvs_v_gput:nn { #1 } { #2 }
    }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { v_greset:vn } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_v_greset:nnTF { #1 } { #2 }
    { \prg_return_true: } { \prg_return_false: }
}
\BNVS_new_conditional:cpnn { greset_all:nn } #1 #2 { T, F, TF } {
  \__bnvs_if_in:nTF { #1 } {
    \BNVS_begin:
    \clist_map_inline:nn { V, A, Z, L } {
      \__bnvs_get:nncT { ##1 } { #1 } { a } {
        \__bnvs_quark_if_nil:cT { a } {
          \__bnvs_cache_get:nncTF { ##1 } { #1 } { a } {
            \__bnvs_gput:nnv { ##1 } { #1 } { a }
          } {
            \__bnvs_gput:nnn { ##1 } { #1 } { 1 }
          }
        }
      }
    }
    \BNVS_end:
    \__bnvs_cache_gclear:n { #1 }
    \__bnvs_v_greset:nnT { #1 } { #2 } {}
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { greset_all:vn } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_greset_all:nnTF { #1 } { #2 }
    { \prg_return_true: } { \prg_return_false: }
}
```

`\__bnvs_gclear_all:n` `\__bnvs_gclear_all:n {⟨FQ name⟩}`
`\__bnvs_gclear_all:` `\__bnvs_gclear_all:`

Convenient shortcuts to clear all the storage, for the given fully qualified name in the first case.

```
\BNVS_new:cpn { gclear_all: } {
  \__bnvs_gclear:
  \__bnvs_cache_gclear:
  \__bnvs_n_gclear:
  \__bnvs_v_gclear:
}
\BNVS_new:cpn { gclear_all:n } #1 {
  \__bnvs_gclear:n { #1 }
```

```
  \__bnvs_cache_gclear:n { #1 }
  \__bnvs_n_gremove:n { #1 }
  \__bnvs_v_gremove:n { #1 }
}
```

6.13.2 Implicit index counter

The implicit index counter is also local to the current frame. It is defined at the global level because changes made at any depth must be made at the frame depth. When used for the first time, it defaults to 1.

\g__bnvs_n_prop ⟨*key*⟩–⟨*value*⟩ property list to store the contents of the named index counters. The keys are qualified names ⟨*id*⟩!⟨*short name*⟩.

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

(*End of definition for* \g__bnvs_n_prop.)

__bnvs_n_gput:nn
__bnvs_n_gput:(nv|vn)
__bnvs_n_gprovide:nn
__bnvs_n_item:n
__bnvs_n_gremove:n
__bnvs_n_gremove:v
__bnvs_n_gclear:

```
\__bnvs_n_gput:nn {⟨Q name⟩} {⟨value⟩}
\__bnvs_n_item:n {⟨Q name⟩}
\__bnvs_n_gremove:n {⟨Q name⟩}
\__bnvs_n_gclear:
```

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

```
\BNVS_new:cpn { n_gput:nn } {
  \prop_gput:Nnn \g__bnvs_n_prop
}
\cs_generate_variant:Nn \__bnvs_n_gput:nn { nV }
\BNVS_new:cpn { n_gput:nv } #1 {
  \BNVS_tl_use:nc {
    \__bnvs_n_gput:nV { #1 }
  }
}
\BNVS_new:cpn { n_gprovide:nn } #1 #2 {
  \prop_if_in:NnF \g__bnvs_n_prop { #1 } {
    \prop_gput:Nnn \g__bnvs_n_prop { #1 } { #2 }
  }
}
\BNVS_new:cpn { n_item:n } #1 {
  \prop_item:Nn \g__bnvs_n_prop { #1 }
}
\BNVS_new:cpn { n_gremove:n } {
  \prop_gremove:Nn \g__bnvs_n_prop
}
\BNVS_generate_variant:cn { n_gremove:n } { V }
\BNVS_new:cpn { n_gremove:v } {
  \BNVS_tl_use:nc {
    \__bnvs_n_gremove:V
  }
}
\BNVS_new:cpn { n_gclear: } {
  \prop_gclear:N \g__bnvs_n_prop
```

```
}
\cs_generate_variant:Nn \__bnvs_n_gremove:n { V }
```

__bnvs_n_if_in_p:n ⋆ __bnvs_n_if_in_p:nn {⟨*Q name*⟩}
__bnvs_n_if_in:n*TF* ⋆ __bnvs_n_if_in:nTF {⟨*Q name*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Convenient shortcuts to test for the existence of the ⟨*Q name*⟩ value counter.

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

__bnvs_n_get:nc*TF* __bnvs_n_get:ncTF {⟨*Q name*⟩} ⟨*tl core*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute ⟨*yes code*⟩ when the item is found, ⟨*no code*⟩ otherwise. In the latter case, the content of the ⟨*tl core*⟩ variable is undefined. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```
\prg_new_conditional:Npnn \__bnvs_n_get:nc #1 #2 { T, F, TF } {
  \__bnvs_prop_get:NncTF \g__bnvs_n_prop { #1 } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.13.3 Regular expressions

\c__bnvs_key_regex This regular expressioin is used for both short names and dot path components. The short name of an overlay set consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

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

(*End of definition for* `\c__bnvs_key_regex`.)

\c__bnvs_id_regex The frame identifier 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_key_regex} | [?] )? !
}
```

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

\c__bnvs_path_regex A sequence of .⟨*positive integer*⟩ or .⟨*short name*⟩ items representing a path.

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

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

`\c__bnvs_A_FQ_name_Z_regex` A fully qualified name is the qualified name of an overlay set possibly followed by a dotted path. Matches the whole string.

(*End of definition for* `\c__bnvs_A_FQ_name_Z_regex`.)

```
\regex_const:Nn \c__bnvs_A_FQ_name_Z_regex {
```

1: The range name including the frame ⟨*id*⟩ and exclamation mark if any

2: frame ⟨*id*⟩ including the exclamation mark

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

3: the path, if any.

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

`\c__bnvs_A_FQ_name_n_Z_regex` A key is the name of an overlay set possibly followed by a dotted path. Matches the whole string. Catch the ending `.n`.

(*End of definition for* `\c__bnvs_A_FQ_name_n_Z_regex`.)

```
\regex_const:Nn \c__bnvs_A_FQ_name_n_Z_regex {
```

1: The full match

2: The fully qualified name including the frame ⟨*id*⟩ and exclamation mark if any, the dotted path but excluding the trailing `.n` (this is `\c__bnvs_path_regex` with a trailing `?`).

3: frame ⟨*id*⟩ including the exclamation mark

```
      \A ( ( \ur{c__bnvs_id_regex} ? )
      \ur{c__bnvs_key_regex}
      (?: \. \ur{c__bnvs_key_regex} | \. [-+]? \d+ )*? )
```

4: the last `.n` component if any.

```
      ( \. n )? \Z
    }
```

`\c__bnvs_colons_regex` For ranges defined by a colon syntax. One catching group for more than one colon.

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

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

\c__bnvs_split_regex Used to parse slide list overlay specifications in queries. Next are the 9 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[3].

```
      \+\+
```

- 1: ⟨*qualified name*⟩ of an overlay set
- 2: ⟨*id*⟩ of a an overlay set including the exclamation mark

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

- 3: optionally followed by a dotted path

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

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

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

- 6: optionally followed by a dotted path

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

We continue with other expressions

- 7: the ⟨*++n*⟩ attribute

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

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

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

- 9: the post increment

```
      | (\+)\+


    )?
  ) \s*
}
```

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

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

6.13.4 beamer.cls interface

Work in progress.

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

6.13.5 Defining named slide ranges

`\__bnvs_range_set:cccnTF` `\__bnvs_range_set:cccnTF {⟨core first⟩} {⟨core end⟩} {⟨core length⟩} {⟨tl⟩} {⟨yes code⟩} {⟨no code⟩}`

Parse ⟨*tl*⟩ as a range according to `\c__bnvs_colons_regex` and set the variables accordingly. ⟨*tl*⟩ is expected to only contain colons and integers.

```
\BNVS_new_conditional:cpnn { split_pop_left:c } #1 { T, F, TF } {
  \__bnvs_seq_pop_left:ccTF { split } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\exp_args_generate:n { VVV }
\BNVS_new_conditional:cpnn { range_set:cccn } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_tl_clear:c { a }
  \__bnvs_tl_clear:c { b }
  \__bnvs_tl_clear:c { c }
  \__bnvs_regex_split:cnTF { colons } { #4 } {
    \__bnvs_seq_pop_left:ccT { split } { a } {
```

`a` may contain the ⟨*start*⟩.

```
      \__bnvs_seq_pop_left:ccT { split } { b } {
        \__bnvs_tl_if_empty:cTF { b } {
```

This is a one colon range.

```
          \__bnvs_split_pop_left:cTF { b } {
```

`b` may contain the ⟨*end*⟩.

```
            \__bnvs_seq_pop_left:ccT { split } { c } {
              \__bnvs_tl_if_empty:cTF { c } {
```

A `::` was expected:

```
                \BNVS_error:n { Invalid~range~expression(1):~#4 }
              } {
                \int_compare:nNnT { \__bnvs_tl_count:c { c } } > { 1 } {
                  \BNVS_error:n { Invalid~range~expression(2):~#4 }
                }
                \__bnvs_split_pop_left:cTF { c } {
```

`\l__bnvs_c_tl` may contain the ⟨*length*⟩.

```
                  \__bnvs_seq_if_empty:cF { split } {
                    \BNVS_error:n { Invalid~range~expression(3):~#4 }
                  }
                } {
                  \BNVS_error:n { Internal~error }
                }
              }
            }
          } {
          }
        } {
```

This is a two colon range component.

```
          \int_compare:nNnT { \__bnvs_tl_count:c { b } } > { 1 } {
            \BNVS_error:n { Invalid~range~expression(4):~#4 }
          }
          \__bnvs_seq_pop_left:ccT { split } { c } {
```

`c` contains the ⟨*length*⟩.

```
            \__bnvs_split_pop_left:cTF { b } {
              \__bnvs_tl_if_empty:cTF { b } {
                \__bnvs_seq_pop_left:cc { split } { b }
```

`b` may contain the ⟨*end*⟩.

```
                \__bnvs_seq_if_empty:cF { split } {
                  \BNVS_error:n { Invalid~range~expression(5):~#4 }
                }
              } {
                \BNVS_error:n { Invalid~range~expression(6):~#4 }
              }
            } {
              \__bnvs_tl_clear:c { b }
            }
          }
        }
      }
    }
```

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

```
    \cs_set:Npn \BNVS_next: { }
    \__bnvs_tl_if_empty:cT { a } {
      \__bnvs_tl_if_empty:cT { b } {
        \__bnvs_tl_if_empty:cT { c } {
          \cs_set:Npn \BNVS_next: {
            \BNVS_error:n { Invalid~range~expression(7):~#3 }
```

```
          }
        }
      }
    }
    \BNVS_next:
    \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
      \BNVS_end:
      \__bnvs_tl_set:cn { #1 } { ##1 }
      \__bnvs_tl_set:cn { #2 } { ##2 }
      \__bnvs_tl_set:cn { #3 } { ##3 }
    }
    \BNVS_exp_args:Nvvv \BNVS:nnn { a } { b } { c }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

__bnvs_range:nnnn
__bnvs_range:nvvv

__bnvs_range:nnnn {⟨*key*⟩} {⟨*start*⟩} {⟨*end*⟩} {⟨*length*⟩}

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

```
\BNVS_new:cpn { range:nnnn } #1 {
  \__bnvs_if_provide:TF {
    \__bnvs_if_in:nnTF A { #1 } {
      \use_none:nnn
    } {
      \__bnvs_if_in:nnTF Z { #1 } {
        \use_none:nnn
      } {
        \__bnvs_if_in:nnTF L { #1 } {
          \use_none:nnn
        } {
          \__bnvs_do_range:nnnn { #1 }
        }
      }
    }
  } {
    \__bnvs_do_range:nnnn { #1 }
  }
}
\BNVS_new:cpn { range:nvvv } #1 #2 #3 #4 {
  \BNVS_tl_use:nv {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:nv {
        \BNVS_use:c { range:nnnn } { #1 }
      } { #2 }
    } { #3 }
  } { #4 }
}
```

__bnvs_parse_record:n
__bnvs_parse_record:v
__bnvs_parse_record:nn
__bnvs_parse_record:(xn|vn)
__bnvs_n_parse_record:n
__bnvs_n_parse_record:v
__bnvs_n_parse_record:nn
__bnvs_n_parse_record:(xn|vn)

__bnvs_parse_record:n {⟨*Q?F name*⟩}
__bnvs_parse_record:nn {⟨*Q?F name*⟩} {⟨*value*⟩}
__bnvs_n_parse_record:n {⟨*Q?F name*⟩}
__bnvs_n_parse_record:nn {⟨*Q?F name*⟩} {⟨*value*⟩}

Auxiliary function for `\__bnvs_parse:n` and `\__bnvs_parse:nn` below. If ⟨*value*⟩ does not correspond to a range, the `V` key is used. The `_n` variant concerns the index counter. This is a bottleneck.

```
\BNVS_new:cpn { parse_record:n } #1 {
  \__bnvs_if_provide:TF {
    \__bnvs_gprovide:nnnT V { #1 } { 1 } {
      \__bnvs_gclear:n { #1 }
    }
  } {
    \__bnvs_gclear:n { #1 }
    \__bnvs_gput:nnn V { #1 } { 1 }
  }
}
\cs_generate_variant:Nn \__bnvs_parse_record:n { V }
\BNVS_new:cpn { parse_record:v } {
  \BNVS_tl_use:nc {
    \__bnvs_parse_record:V
  }
}
\BNVS_new:cpn { parse_record:nn } #1 #2 {
  \__bnvs_range_set:cccnTF { a } { b } { c } { #2 } {
    \__bnvs_range:nvvv { #1 } { a } { b } { c }
  } {
    \__bnvs_if_provide:TF {
      \__bnvs_gprovide:nnnT V { #1 } { #2 } {
        \__bnvs_gclear_all:n { #1 }
      }
    } {
      \__bnvs_gclear_all:n { #1 }
      \__bnvs_gput:nnn V { #1 } { #2 }
    }
  }
}
\cs_generate_variant:Nn \__bnvs_parse_record:nn { x, V }
\BNVS_new:cpn { parse_record:vn } {
  \BNVS_tl_use:nc {
    \__bnvs_parse_record:Vn
  }
}
\BNVS_new:cpn { n_parse_record:n } #1 {
  \bool_if:NTF \l__bnvs_n_provide_bool {
    \__bnvs_n_gprovide:nn
  } {
    \__bnvs_n_gput:nn
```

```
  }
  { #1 } { 1 }
}
\cs_generate_variant:Nn \__bnvs_n_parse_record:n { V }
\BNVS_new:cpn { n_parse_record:v } {
  \BNVS_tl_use:nc {
    \__bnvs_n_parse_record:V
  }
}
\BNVS_new:cpn { n_parse_record:nn } #1 #2 {
  \__bnvs_range_set:cccnTF { a } { b } { c } { #2 } {
    \BNVS_error:n { Unexpected~range:~#2 }
  } {
    \__bnvs_if_provide:TF {
      \__bnvs_n_gprovide:nn { #1 } { #2 }
    } {
      \__bnvs_n_gput:nn { #1 } { #2 }
    }
  }
}
\cs_generate_variant:Nn \__bnvs_n_parse_record:nn { x, V }
\BNVS_new:cpn { n_parse_record:vn } {
  \BNVS_tl_use:Nc \__bnvs_n_parse_record:Vn
}
```

__bnvs_id_name_n_get:n*TF*
__bnvs_id_name_n_get:v*TF*

__bnvs_id_name_n_set:nTF {⟨*ref*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

If ⟨*ref*⟩ is a fully qualified name, put the frame id it defines into `id` and the fully qualified name into `key`, then execute ⟨*yes code*⟩. The `n tl` variable is empty except when ⟨*ref*⟩ ends with `.n`. Otherwise execute ⟨*no code*⟩. If ⟨*ref*⟩ is only a qualified name, put it in `key`, prepended with `id_last`, and set `id` to this value as well.

```
\BNVS_new:cpn { id_name_n_end_export: } {
  \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
    \BNVS_end:
    \__bnvs_tl_set:cn { id } { ##1 }
    \__bnvs_tl_set:cn { key } { ##2 }
    \__bnvs_tl_set:cn { n } { ##3 }
  }
  \__bnvs_tl_if_empty:cTF { id } {
    \BNVS_exp_args:Nvvv
    \BNVS:nnn { id_last } { key } { n }
    \__bnvs_tl_put_left:cv { key } { id_last }
  } {
    \BNVS_exp_args:Nvvv
    \BNVS:nnn { id } { key } { n }
    \__bnvs_tl_set:cv { id_last } { id }
  }
}
\BNVS_new_conditional:cpnn { id_name_n_get:n } #1 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_match_once:NnTF \c__bnvs_A_FQ_name_n_Z_regex { #1 } {
    \__bnvs_match_pop_left:cTF { key } {
      \__bnvs_match_pop_left:cTF { key } {
```

```
        \__bnvs_match_pop_left:cTF { id } {
          \__bnvs_match_pop_left:cTF { n } {
            \__bnvs_id_name_n_end_export:
            \prg_return_true:
          } {
            \BNVS_end:
            \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/n }
            \prg_return_false:
          }
        } {
          \BNVS_end:
          \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/id }
          \prg_return_false:
        }
      } {
        \BNVS_end:
        \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/name }
        \prg_return_false:
      }
    } {
      \BNVS_end:
      \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/n }
      \prg_return_false:
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { id_name_n_get:v } #1 { T, F, TF } {
  \BNVS_tl_use:nv { \BNVS_use:c { id_name_n_get:nTF } } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

`\__bnvs_parse:n` `\__bnvs_parse:n {⟨F/Q name⟩}`
`\__bnvs_parse:nn` `\__bnvs_parse:nn {⟨F/Q name⟩} {⟨definition⟩}`

Auxiliary functions called within a group by `\keyval_parse:nnn`. ⟨*F/Q name*⟩ is the overlay (eventually fully) qualified name, including eventually a dotted path and a frame identifier, ⟨*definition*⟩ is the corresponding definition.

`\l__bnvs_match_seq` Local storage for the match result.

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

```
\BNVS_new:cpn { parse:n } #1 {
  \peek_remove_spaces:n {
    \peek_catcode:NTF \c_group_begin_token {
      \__bnvs_tl_if_empty:cTF { root } {
        \BNVS_error:n { Unexpected~list~at~top~level. }
      } {
        \BNVS_begin:
```

```
        \__bnvs_int_incr:c { i }
        \__bnvs_tl_put_right:cx { root } { \__bnvs_int_use:c { i } . }
        \cs_set:Npn \bnvs:w ####1 ####2 \s_stop {
          \regex_match:nnT { \S* } { ####2 } {
            \BNVS_error:n { Unexpected~####2 }
          }
          \keyval_parse:nnn {
            \__bnvs_parse:n
          } {
            \__bnvs_parse:nn
          } { ####1 }
          \BNVS_end:
        }
        \bnvs:w #1 \s_stop
      }
    } {
      \__bnvs_tl_if_empty:cTF { root } {
        \__bnvs_id_name_n_get:nTF { #1 } {
          \__bnvs_tl_if_empty:cTF { n } {
            \__bnvs_parse_record:v
          } {
            \__bnvs_n_parse_record:v
          }
          { key }
        } {
          \BNVS_error:n { Unexpected~key:~#1 }
        }
      } {
        \__bnvs_int_incr:c { i }
        \__bnvs_tl_if_empty:cTF { n } {
          \__bnvs_parse_record:xn
        } {
          \__bnvs_n_parse_record:xn
        } {
          \__bnvs_tl_use:c { root } \__bnvs_int_use:c { i }
        } { #1 }
      }
    }
  }
}
\BNVS_new:cpn { do_range:nnnn } #1 #2 #3 #4 {
    \__bnvs_gclear_all:n { #1 }
  \tl_if_empty:nTF { #4 } {
    \tl_if_empty:nTF { #2 } {
      \tl_if_empty:nTF { #3 } {
        \BNVS_error:n { Not~a~range:~:~#1 }
      } {
        \__bnvs_gput:nnn Z { #1 } { #3 }
        \__bnvs_gput:nnn V { #1 } { \q_nil }
      }
    } {
      \__bnvs_gput:nnn A { #1 } { #2 }
      \__bnvs_gput:nnn V { #1 } { \q_nil }
      \tl_if_empty:nF { #3 } {
```

```
        \__bnvs_gput:nnn Z { #1 } { #3 }
        \__bnvs_gput:nnn L { #1 } { \q_nil }
      }
    }
  } {
    \tl_if_empty:nTF { #2 } {
      \__bnvs_gput:nnn L { #1 } { #4 }
      \tl_if_empty:nF { #3 } {
        \__bnvs_gput:nnn Z { #1 } { #3 }
        \__bnvs_gput:nnn A { #1 } { \q_nil }
        \__bnvs_gput:nnn V { #1 } { \q_nil }
      }
    } {
      \__bnvs_gput:nnn A { #1 } { #2 }
      \__bnvs_gput:nnn L { #1 } { #4 }
      \__bnvs_gput:nnn Z { #1 } { \q_nil }
      \__bnvs_gput:nnn V { #1 } { \q_nil }
    }
  }
}
\cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
  \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
    { #4 }
}
\cs_new:Npn \BNVS_end_tl_set:cv #1 {
  \BNVS_tl_use:nv {
    \BNVS_end: \__bnvs_tl_set:cn { #1 }
  }
}
\BNVS_new:cpn { parse:nn } #1 #2 {
  \BNVS_begin:
  \__bnvs_tl_set:cn { a } { #1 }
```

We prepend the argument with `root`, in case we are recursive.

```
  \__bnvs_tl_put_left:cv { a } { root }
  \__bnvs_id_name_n_get:vTF { a } {
    \regex_match:nnTF { \S } { #2 } {
      \peek_remove_spaces:n {
        \peek_catcode:NTF \c_group_begin_token {
```

The value is a comma separated list, we warn about an unexpected `.n` suffix, if any.

```
          \__bnvs_tl_if_empty:cF { n } {
\__bnvs_warning:n { Ignoring~unexpected~suffix~.n:~#1 }
          }
```

We go recursive opening a new TEX group. The `root` contains the common part that will prefix the subkeys.

```
          \BNVS_begin:
          \__bnvs_gput:nvn V { key } { \q_nil }
          \__bnvs_tl_set:cv { root } { key }
          \__bnvs_tl_put_right:cn { root } { . }
          \__bnvs_int_set:cn { i } { 0 }
```

```
          \cs_set:Npn \BNVS:w ##1 ##2 \s_stop {
            \regex_match:nnT { \S } { ##2 } {
              \BNVS_error:n { Unexpected~value~#2 }
            }
            \keyval_parse:nnn {
              \__bnvs_parse:n
            } {
              \__bnvs_parse:nn
            } { ##1 }
            \BNVS_end:
          }
          \BNVS:w
        } {
```

Next character is not a group begin token.

```
          \__bnvs_tl_if_empty:cTF { n } {
            \__bnvs_parse_record:vn
          } {
            \__bnvs_n_parse_record:vn
          }
          { key } { #2 }
          \use_none_delimit_by_s_stop:w
        }
      }
      #2 \s_stop
    } {
```

Empty value given: remove the reference.

```
      \__bnvs_tl_if_empty:cTF { n } {
        \__bnvs_gclear:v
      } {
        \__bnvs_n_gremove:v
      }
      { key }
    }
  } {
    \BNVS_error:n { Invalid~key:~#2 }
  }
```

We export `\l__bnvs_id_last_tl`:

```
  \BNVS_end_tl_set:cv { id_last } { id_last }
}

\BNVS_new:cpn { parse_prepare:N } #1 {
  \tl_set:Nx #1 #1
  \bool_set_false:N \l__bnvs_parse_bool
  \bool_do_until:Nn \l__bnvs_parse_bool {
    \tl_if_in:NnTF #1 {%---[
    ]} {
      \regex_replace_all:nnNF { \[ ([^\]%---)
      ]*%---[(
      ) \] } { { { \1 } } } #1 {
        \bool_set_true:N \l__bnvs_parse_bool
      }
    } {
```

```
      \bool_set_true:N \l__bnvs_parse_bool
    }
  }
  \tl_if_in:NnTF #1 {%---[
  ]} {
    \BNVS_error:n { Unbalanced~%---[
    ]}
  } {
    \tl_if_in:NnT #1 { [%---]
    } {
      \BNVS_error:n { Unbalanced~[ %---]
      }
    }
  }
}
```

\Beanoves `\Beanoves {⟨key-value list⟩}`

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

```
\cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_tl_put_right:cn { #1 }
  } { #2 }
}
\cs_new:Npn \BNVS_end_v_gput:nv #1 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_v_gput:nn { #1 }
  }
}
\NewDocumentCommand \Beanoves { sm } {
  \tl_if_empty:NTF \@currenvir {
```

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

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

At the top level, clear everything.

```
      \__bnvs_gclear:
    }
    \BNVS_begin:
    \__bnvs_tl_clear:c { root }
    \__bnvs_int_zero:c { i }
    \__bnvs_tl_set:cn { a } { #2 }
    \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, use the global definitions.

```
      \seq_if_empty:NF \g__bnvs_def_seq {
        \__bnvs_tl_put_left:cx { a } {
          \seq_use:Nn \g__bnvs_def_seq , ,
        }
      }
    }
    \__bnvs_parse_prepare:N \l__bnvs_a_tl
    \IfBooleanTF {#1} {
      \__bnvs_provide_on:
    } {
      \__bnvs_provide_off:
    }
    \BNVS_tl_use:nv {
      \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn }
    } { a }
    \BNVS_end_tl_set:cv { id_last } { id_last }
    \ignorespaces
  }
}
```

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

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

6.13.6 Scanning named overlay specifications

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

`\__bnvs@frame` `\__bnvs@frame {`⟨*overlay specification*⟩`}`
`\__bnvs@masterdecode` `\__bnvs@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 of definition for* `\l__bnvs_ans_tl`.)

Save the original macros `\beamer@frame` and `\beamer@masterdecode` then override them to properly preprocess the argument. We start by defining the overloads.

```
\makeatletter
\cs_set:Npn \__bnvs@frame < #1 > {
  \BNVS_begin:
  \__bnvs_tl_clear:c { ans }
  \__bnvs_scan:nNc { #1 } \__bnvs_eval:nc { ans }
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_saved@frame <
  } { ans } >
}
\cs_set:Npn \__bnvs@masterdecode #1 {
  \BNVS_begin:
  \__bnvs_tl_clear:c { ans }
  \__bnvs_scan:nNc { #1 } \__bnvs_eval:nc { ans }
```

```
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_saved@masterdecode
  } { ans }
}
\cs_new:Npn \BeanovesOff {
  \cs_set_eq:NN \beamer@frame \__bnvs_saved@frame
  \cs_set_eq:NN \beamer@masterdecode \__bnvs_saved@masterdecode
}
\cs_new:Npn \BeanovesOn {
  \cs_set_eq:NN \beamer@frame \__bnvs@frame
  \cs_set_eq:NN \beamer@masterdecode \__bnvs@masterdecode
}
\AddToHook{begindocument/before}{
  \cs_if_exist:NTF \beamer@frame {
    \cs_set_eq:NN \__bnvs_saved@frame \beamer@frame
    \cs_set_eq:NN \__bnvs_saved@masterdecode \beamer@masterdecode
  } {
    \cs_set:Npn \__bnvs_saved@frame < #1 > {
      \BNVS_error:n {Missing~package~beamer}
    }
    \cs_set:Npn \__bnvs_saved@masterdecode < #1 > {
      \BNVS_error:n {Missing~package~beamer}
    }
  }
  \BeanovesOn
}
\makeatother
```

`\__bnvs_scan:nNc` `\__bnvs_scan:nNc {⟨named overlay expression⟩} ⟨eval⟩ ⟨tl core⟩`

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 of definition for* `\l__bnvs_ans_tl`.)

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

(*End of definition for* `\l__bnvs_int`.)

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

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

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

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

`\l__bnvs_token_tl` Storage for just one token.

(*End of definition for* `\l__bnvs_token_tl`.)

Next are helpers.

__bnvs_scan_question:T `\__bnvs_scan_question:T {⟨code⟩}`

At top level state, scan the tokens of the ⟨*named overlay expression*⟩ looking for a '?' character. If a '?(...)' is found, then the ⟨*code*⟩ is executed.

```
\BNVS_new:cpn { scan_question:T } #1 {
  \__bnvs_seq_pop_left:ccT { token } { token } {
    \__bnvs_tl_if_eq:cnTF { token } { ? } {
      \__bnvs_scan_require_open:
      #1
    } {
      \__bnvs_tl_put_right:cv { ans } { token }
    }
    }
    \__bnvs_scan_question:T { #1 }
  }
}
```

__bnvs_scan_require_open: `\__bnvs_scan_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.

```
\BNVS_new:cpn { scan_require_open: } {
```

Get next token.

```
  \__bnvs_seq_pop_left:ccTF { token } { token } {
    \tl_if_eq:NnTF \l__bnvs_token_tl { ( %)
    } {
```

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

```
      \__bnvs_int_set:cn { } { 1 }
```

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

```
      \__bnvs_tl_clear:c { query }
      \__bnvs_scan_require_close:
    } {
```

Ignore this token and loop.

```
      \__bnvs_scan_require_open:
    }
  } {
```

End reached but no opening parenthesis found, raise.

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

__bnvs_scan_require_close: __bnvs_scan_require_close:

We found a '?(', we record the forthcomming content in the query variable, up to the next balancing ')'.

```
\BNVS_new:cpn { scan_require_close: } {
```

Get next token.

```
  \__bnvs_seq_pop_left:ccTF { token } { token } {
    \__bnvs_tl_if_eq:cnTF { token } { ( %---)
    } {
```

We found a '(', increment the depth and append the token to query, then scan again for a).

```
      \__bnvs_int_incr:c { }
      \__bnvs_tl_put_right:cv { query } { token }
      \__bnvs_scan_require_close:
    } {
```

This is not a '('.

```
      \__bnvs_tl_if_eq:cnTF { token } { %(---
        )
      } {
```

We found a balancing ')', we decrement and test the depth.

```
        \__bnvs_int_decr:c { }
        \int_compare:nNnTF { \__bnvs_int_use:c { } } = 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 ans and look for the next ?.

```
        } {
```

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

```
          \__bnvs_tl_put_right:cv { query } { token }
          \__bnvs_scan_require_close:
        }
      } {
```

The scanned token is not a '(' nor a ')', we append it as is to query and look for a balancing).

```
        \__bnvs_tl_put_right:cv { query } { token }
        \__bnvs_scan_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 query with the missing ')'.

```
    \BNVS_error:x { Missing~%(---
      `)' }
    \__bnvs_tl_put_right:cx { query } {
      \prg_replicate:nn { \l__bnvs_int } {%(---
      )}
    }
  }
}

\BNVS_new:cpn { scan:nNc } #1 #2 #3 {
  \BNVS_begin:
  \BNVS_set:cpn { fatal:x } ##1 {
    \msg_fatal:nnx { beanoves } { :n }
      { \tl_to_str:n { #1 }:~##1}
  }
  \BNVS_set:cpn { error:x } ##1 {
    \msg_error:nnx { beanoves } { :n }
      { \tl_to_str:n { #1 }:~##1}
  }
  \__bnvs_tl_set:cn { scan } { #1 }
  \__bnvs_tl_clear:c { ans }
  \__bnvs_seq_clear:c { token }
```

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

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

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

```
  \__bnvs_scan_question:T {
    \BNVS_tl_use:Nv #2 { query } { ans }
  }
  \BNVS_end_tl_put_right:cv { #3 } { ans }
}
```

6.13.7 Resolution

Given a name, a frame id 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_kip:cccTF` `\__bnvs_kip:cccTF {⟨key⟩} {⟨id⟩} {⟨path⟩} {⟨yes code⟩} {⟨no code⟩}`

Auxiliary function. On input, the ⟨*key*⟩ tl variable contains a set name whereas the ⟨*id*⟩ tl variable contains a frame id. If ⟨*key*⟩ tl variable contents is a recorded key, on return, ⟨*key*⟩ tl variable contains the resolved name, ⟨*id*⟩ tl variable contains the used frame id, ⟨*path*⟩ seq variable is prepended with new dotted path components, ⟨*yes code*⟩ is executed, otherwise ⟨*no code*⟩ is executed.

```
\exp_args_generate:n { VVx }
\quark_new:N \q__bnvs
\BNVS_new:cpn { end_kip_export_seq:nnnccc } #1 #2 #3 #4 #5 #6 {
  \BNVS_end:
  \tl_if_empty:nTF { #2 } {
    \__bnvs_tl_set:cn { #4 } { #1 }
    \__bnvs_tl_put_left:cv { #4 } { #5 }
```

```
  } {
    \__bnvs_tl_set:cn { #4 } { #1 }
    \__bnvs_tl_set:cn { #5 } { #2 }
  }
  \__bnvs_seq_set_split:cnn { #6 } { \q__bnvs } { #3 }
  \__bnvs_seq_remove_all:cn { #6 } { }
}
\BNVS_new:cpn { end_kip_export:ccc } {
  \exp_args:Nnnx \BNVS_tl_use:nv {
    \BNVS_tl_use:Nv \__bnvs_end_kip_export_seq:nnnccc { key }
  } { id } {
    \__bnvs_seq_use:cn { path } { \q__bnvs }
  }
}
\BNVS_new_conditional:cpnn { match_pop_kip: } { T, F, TF } {
  \__bnvs_match_pop_left:cTF { key } {
    \__bnvs_match_pop_left:cTF { key } {
      \__bnvs_match_pop_left:cTF { id } {
        \__bnvs_match_pop_left:cTF { path } {
          \__bnvs_seq_set_split:cnv { path } { . } { path }
          \__bnvs_seq_remove_all:cn { path } { }
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { kip:ccc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_match_once:NvTF \c__bnvs_A_FQ_name_Z_regex { #1 }  {
```

This is a correct key, update the path sequence accordingly.

```
    \__bnvs_match_pop_kip:TF {
      \__bnvs_end_kip_export:ccc { #1 } { #2 } { #3 }
      \prg_return_true:
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

__bnvs_kip_n_path_resolve:TF __bnvs_kip_n_path_resolve:TF {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_kip_x_path_resolve:TF __bnvs_kip_x_path_resolve:TF {⟨*yes code*⟩} {⟨*no code*⟩}

{⟨*yes code*⟩} will be executed once resolution has occurred, {⟨*no code*⟩} otherwise. The key and id variables as well as the path sequence are meant to contain proper information on input and on output as well. On input, \l__bnvs_key_tl contains a slide range name, \l__bnvs_id_tl contains a frame id and \l__bnvs_path_seq contains the components of an integer path, possibly empty. On return, the variable \l__bnvs_key_tl contains the resolved range name, \l__bnvs_id_tl contains the frame id used and \l__bnvs_path_seq contains the sequence of integer path components that could not be resolved.

To resolve one level of a named one slide specification like $\langle name\rangle.\langle i_1\rangle...\langle i_n\rangle$, we replace the shortest $\langle name\rangle.\langle i_1\rangle...\langle i_k\rangle$ where $0 \leq k \leq n$ by its definition $\langle name'\rangle.\langle j_1\rangle...\langle j_p\rangle$ if any. The __bnvs_resolve_?:NNNTF function uses this one level resolution as many times as possible, but no more than a predefined limit to catch circular references that would lead to an infinite loop.

1. If \l__bnvs_key_tl 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 \l__bnvs_key_tl is replaced by this name, the \l__bnvs_id_tl 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 \l__bnvs_key_tl after a separating dot, all its left elements but the last one and loop. Otherwise return.

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

   In the _x variant, the resolution is driven one step further: if ⟨*path seq var*⟩ is empty, ⟨*name tl var*⟩ can contain anything, including an integer for example.

__bnvs_kip_x_path_resolve:TFF __bnvs_kip_x_path_resolve:TFF {⟨*yes code*⟩} {⟨*no code 1*⟩} {⟨*no code 2*⟩}

```
\BNVS_new:cpn { kip_x_path_resolve:TFF } #1 #2 {
  \__bnvs_kip_x_path_resolve:TF {
    \__bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
  }
}
```

Local variables:

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

```
\BNVS_new:cpn { end_kip_export: } {
  \exp_args:Nnnx
  \BNVS_tl_use:nv {
```

```
    \BNVS_tl_use:Nv \__bnvs_end_kip_export_seq:nnnccc { key }
  } { id } {
    \__bnvs_seq_use:cn { path } { \q__bnvs }
  } { key } { id } { path }
}
\BNVS_new:cpn { seq_merge:cc } #1 #2 {
  \__bnvs_seq_if_empty:cF { #2 } {
    \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
      \__bnvs_seq_use:cn { #1 } { \q__bnvs }
      \exp_not:n { \q__bnvs }
      \__bnvs_seq_use:cn { #2 } { \q__bnvs }
    }
    \__bnvs_seq_remove_all:cn { #1 } { }
  }
}
\BNVS_new:cpn { kip_x_path_resolve:nFF } #1 #2 #3 {
  \__bnvs_get:nvcTF #1 { a } { b } {
    \__bnvs_kip:cccTF { b } { id } { path } {
      \__bnvs_tl_set_eq:cc { key } { b }
      \__bnvs_seq_merge:cc { path } { b }
      \__bnvs_seq_clear:c { b }
      \__bnvs_seq_set_eq:cc { a } { path }
      \__bnvs_kip_x_path_resolve_loop_or_end_return:
    } {
      \__bnvs_seq_if_empty:cTF { b } {
        \__bnvs_tl_set_eq:cc { key } { b }
        \__bnvs_seq_clear:c { path }
        \__bnvs_seq_clear:c { a }
        \__bnvs_kip_x_path_resolve_loop_or_end_return:
      } {
        #2
      }
    }
  } {
    #3
  }
}
\BNVS_new:cpn { kip_x_path_resolve_VAL_loop_or_end_return:F } #1 {
  \__bnvs_kip_x_path_resolve:nFF V { #1 } {
    \__bnvs_kip_x_path_resolve:nFF A { #1 } {
      \__bnvs_kip_x_path_resolve:nFF L { #1 } { #1 }
    }
  }
}
\BNVS_new:cpn { kip_x_path_resolve_end_return_true: } {
  \__bnvs_seq_pop_left:ccTF { path } { a } {
    \__bnvs_seq_if_empty:cTF { path } {
      \__bnvs_tl_clear:c { b }
      \__bnvs_index_can:vTF { key } {
        \__bnvs_index_append:vvcTF { key } { a } { b } {
          \__bnvs_tl_set:cv { key } { b }
        } {
          \__bnvs_tl_set:cv { key } { a }
        }
```

```
      } {
        \__bnvs_tl_set:cv { key } { a }
      }
    } {
      \BNVS_error:x { Path~too~long~.\BNVS_tl_use:c { a }
        .\__bnvs_seq_use:cn { path } . }
    }
  } {
    \__bnvs_value_resolve:vcT { key } { key } {}
  }
  \__bnvs_end_kip_export:
  \prg_return_true:
}
\BNVS_new_conditional:cpnn { kip_x_path_resolve: } { T, F, TF } {
  \BNVS_begin:
  \__bnvs_seq_set_eq:cc { a } { path }
  \__bnvs_seq_clear:c { b }
  \__bnvs_kip_x_path_resolve_loop_or_end_return:
}
\BNVS_new:cpn { kip_x_path_resolve_loop_or_end_return: } {
  \__bnvs_call:TF {
    \__bnvs_tl_set_eq:cc { a } { key }
    \__bnvs_seq_if_empty:cTF { a } {
      \__bnvs_kip_x_path_resolve_VAL_loop_or_end_return:F {
        \__bnvs_kip_x_path_resolve_end_return_true:
      }
    } {
      \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { a } . }
      \__bnvs_kip_x_path_resolve_VAL_loop_or_end_return:F {
        \__bnvs_seq_pop_right:ccT { a } { c } {
          \__bnvs_seq_put_left:cv { b } { c }
        }
        \__bnvs_kip_x_path_resolve_loop_or_end_return:
      }
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new:cpn { kip_n_path_resolve_or_end_return:nF } #1 #2 {
  \__bnvs_get:nvcTF { #1 } { a } { b } {
    \__bnvs_kip:cccTF { b } { id } { path } {
      \__bnvs_tl_set_eq:cc { key } { b }
      \__bnvs_seq_merge:cc { path } { b }
      \__bnvs_seq_set_eq:cc { a } { path }
      \__bnvs_seq_clear:c { b }
      \__bnvs_kip_n_path_resolve_loop_or_end_return:
    } {
      \__bnvs_seq_pop_right:ccTF { a } { c } {
        \__bnvs_seq_put_left:cv { b } { c }
        \__bnvs_kip_n_path_resolve_loop_or_end_return:
      } {
```

```
        \__bnvs_kip_n_path_resolve_end_return_true:
      }
    }
  } {
    #2
  }
}
\BNVS_new:cpn { kip_n_path_resolve_VAL_loop_or_end_return: } {
  \__bnvs_kip_n_path_resolve_or_end_return:nF V {
    \__bnvs_kip_n_path_resolve_or_end_return:nF A {
      \__bnvs_kip_n_path_resolve_or_end_return:nF L {
        \__bnvs_seq_pop_right:ccTF { a } { c } {
          \__bnvs_seq_put_left:cv { b } { c }
          \__bnvs_kip_n_path_resolve_loop_or_end_return:
        } {
          \__bnvs_kip_n_path_resolve_end_return_true:
        }
      }
    }
  }
}
\BNVS_new:cpn { kip_n_path_resolve_end_return_false: } {
  \BNVS_end:
  \prg_return_false:
}
\BNVS_new:cpn { kip_n_path_resolve_end_return_true: } {
  \__bnvs_end_kip_export:
  \prg_return_true:
}
```

__bnvs_kip_n_path_resolve_loop_or_end_return:

Loop to resolve the path.

```
\BNVS_new:cpn { kip_n_path_resolve_loop_or_end_return: } {
  \__bnvs_call:TF {
    \__bnvs_tl_set_eq:cc { a } { key }
    \__bnvs_seq_if_empty:cTF { a } {
      \__bnvs_seq_if_empty:cTF { b } {
        \__bnvs_kip_n_path_resolve_end_return_true:
      } {
        \__bnvs_kip_n_path_resolve_VAL_loop_or_end_return:
      }
    } {
      \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { a } . }
      \__bnvs_kip_n_path_resolve_VAL_loop_or_end_return:
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

`__bnvs_kip_n_path_resolve:` This is the entry point to resolve the path. Local variables:

- `\...key_tl`, `\...id_tl`, `\...path_seq` contain the resolution.
- `...a_tl` contains the name with a partial index path currently resolved.
- `\...a_seq` contains the dotted path components to be resolved. It equals `\...path_seq` at the beginning
- `\...b_seq` is used as well. Initially empty.

```
\BNVS_new_conditional:cpnn { kip_n_path_resolve: } { T, F, TF } {
  \BNVS_begin:
  \__bnvs_seq_set_eq:cc { a } { path }
  \__bnvs_seq_clear:c { b }
  \__bnvs_kip_n_path_resolve_loop_or_end_return:
}
```

6.13.8 Evaluation bricks

We start by helpers.

`\__bnvs_round_ans:n` `\__bnvs_round:c` ⟨*tl core name*⟩
`\__bnvs_round:c` `\__bnvs_round_ans:`
`\__bnvs_round_ans:` `\__bnvs_round_ans:n` {⟨*expression*⟩}

The first function replaces the variable content with its rounded floating point evaluation. The second function replaces `ans` tl variable content with its rounded floating point evaluation. The last function appends to the `ans` tl variable the rounded floating point evaluation of the argument.

```
\BNVS_new:cpn { round_ans:n } #1 {
  \tl_if_empty:nTF { #1 } {
    \__bnvs_tl_put_right:cn { ans } { 0 }
  } {
    \__bnvs_tl_put_right:cx { ans } { \fp_eval:n { round(#1) } }
  }
}
\BNVS_new:cpn { round:N } #1 {
  \tl_if_empty:NTF #1 {
    \tl_set:Nn #1 { 0 }
  } {
    \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
  }
}
\BNVS_new:cpn { round:c } {
  \BNVS_tl_use:Nc \__bnvs_round:N
}
```

`\BNVS_end_return_false:` `\BNVS_end_return_false:x` `\__bnvs_end_return_false:`
`\__bnvs_end_return_false:x` {⟨*message*⟩}

End a group and calls `\prg_return_false:`. The message is for debugging only.

```
\cs_new:Npn \BNVS_end_return_false: {
  \BNVS_end:
  \prg_return_false:
}
\cs_new:Npn \BNVS_end_return_false:x #1 {
  \BNVS_error:x { #1 }
  \BNVS_end_return_false:
}
```

__bnvs_value_resolve:nc*TF* __bnvs_value_resolve:ncTF {⟨*key*⟩} ⟨*tl core*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_value_resolve:vc*TF* __bnvs_value_append:ncTF {⟨*key*⟩} ⟨*tl core*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_value_append:nc*TF*
__bnvs_value_append:(xc|vc)*TF*

Resolve the content of the ⟨*key*⟩ value counter into the ⟨*tl variable*⟩ or append this value to the right of the variable. Execute ⟨*yes code*⟩ when there is a ⟨*value*⟩, ⟨*no code*⟩ otherwise. Inside the ⟨*no code*⟩ branch, the content of the ⟨*tl variable*⟩ is undefined. Implementation detail: we return the first in the cache for subkey V and in the general prop for subkey V. Once we have found a value, we feed the previous items such that the next search stops at the first item. The cache contains an integer which is the computed value from the general prop. A group is created while appending but not while resolving.

```
\BNVS_new:cpn { value_resolve_return:nnnT } #1 #2 #3 #4 {
  \__bnvs_tl_if_empty:cTF { #3 } {
    \prg_return_false:
  } {
    \__bnvs_cache_gput:nnv V { #2 } { #3 }
    #4
    \prg_return_true:
  }
}
\BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
  \BNVS_tl_use:Nc \quark_if_nil:NTF { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
  \BNVS_tl_use:Nc \quark_if_no_value:NTF { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { value_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_cache_get:nncTF V { #1 } { #2 } {
    \prg_return_true:
  } {
    \__bnvs_get:nncTF V { #1 } { #2 } {
      \__bnvs_quark_if_nil:cTF { #2 } {
```

We can retrieve the value from either the first or last index.

```
        \__bnvs_gput:nnn V { #1 } { \q_no_value }
        \__bnvs_first_resolve:ncTF { #1 } { #2 } {
          \__bnvs_value_resolve_return:nnnT A { #1 } { #2 } {
            \__bnvs_gput:nnn V { #1 } { \q_nil }
          }
        } {
          \__bnvs_last_resolve:ncTF { #1 } { #2 } {
            \__bnvs_value_resolve_return:nnnT Z { #1 } { #2 } {
              \__bnvs_gput:nnn V { #1 } { \q_nil }
            }
          } {
            \__bnvs_gput:nnn V { #1 } { \q_nil }
            \prg_return_false:
          }
        }
      } {
        \__bnvs_quark_if_no_value:cTF { #2 } {
          \BNVS_fatal:n {Circular~definition:~#1}
        } {
```

Possible recursive call.

```
          \__bnvs_if_resolve:vcTF { #2 } { #2 } {
            \__bnvs_value_resolve_return:nnnT V { #1 } { #2 } {
              \__bnvs_gput:nnn V { #1 } { \q_nil }
            }
          } {
            \__bnvs_gput:nnn V { #1 } { \q_nil }
            \prg_return_false:
          }
        }
      }
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional:cpnn { value_resolve:vc } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_value_resolve:ncTF { #1 } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { end_put_right:vc } #1 #2 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_tl_put_right:cn { #2 }
  } { #1 }
}
\BNVS_new_conditional:cpnn { value_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_value_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
```

```
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_true:
  }
}
\BNVS_new_conditional_vc:cn { value_append } { T, F, TF }
```

cTF:nnnnvalueFIRST2222

__bnvs_first_resolve:nc*TF* __bnvs_first_resolve:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_first_resolve:(xc|vc)*TF* __bnvs_first_append:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_first_append:nc*TF*
__bnvs_first_append:(xc|vc)*TF*

Resolve the first index of the ⟨*key*⟩ slide range into the ⟨*tl variable*⟩ or append the first index of the ⟨*key*⟩ slide range to the ⟨*tl variable*⟩. If no resolution occurs the content of the ⟨*tl variable*⟩ is undefined in the first case and unmodified in the second. Cache the result. Execute ⟨*yes code*⟩ when there is a ⟨*first*⟩, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { first_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_cache_get:nncTF A { #1 } { #2 } {
    \prg_return_true:
  } {
    \__bnvs_get:nncTF A { #1 } { #2 } {
      \__bnvs_quark_if_nil:cTF { #2 } {
        \__bnvs_gput:nnn A { #1 } { \q_no_value }
```

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

```
        \__bnvs_last_resolve:ncTF { #1 } { #2 } {
          \__bnvs_tl_put_right:cn { #2 } { - }
          \__bnvs_length_append:ncTF { #1 } { #2 } {
            \__bnvs_tl_put_right:cn { #2 } { + 1 }
            \__bnvs_round:c { #2 }
            \__bnvs_tl_if_empty:cTF { #2 } {
              \__bnvs_gput:nnn A { #1 } { \q_nil }
              \prg_return_false:
            } {
              \__bnvs_gput:nnn A { #1 } { \q_nil }
              \__bnvs_cache_gput:nnv A { #1 } { #2 }
              \prg_return_true:
            }
          } {
            \BNVS_error:n {
Unavailable~length~for~#1~(\token_to_str:N\__bnvs_first_resolve:ncTF/2) }
            \__bnvs_gput:nnn A { #1 } { \q_nil }
            \prg_return_false:
          }
        } {
          \BNVS_error:n {
Unavailable~last~for~#1~(\token_to_str:N\__bnvs_first_resolve:ncTF/1) }
          \__bnvs_gput:nnn A { #1 } { \q_nil }
          \prg_return_false:
        }
      } {
```

```
        \__bnvs_quark_if_no_value:cTF { a } {
          \BNVS_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_resolve:vcTF { #2 } { #2 } {
            \__bnvs_cache_gput:nnv A { #1 } { #2 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      }
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vc:cn { first_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { first_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_first_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_last_resolve:nc*TF* __bnvs_last_resolve:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_last_append:nc*TF* __bnvs_last_append:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the last index of the fully qualified ⟨*key*⟩ range into or to the right of the right of the ⟨*tl variable*⟩, when possible. Execute ⟨*yes code*⟩ when a last index was given, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { last_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_cache_get:nncTF Z { #1 } { #2 } {
    \prg_return_true:
  }  {
    \__bnvs_get:nncTF Z { #1 } { #2 } {
      \__bnvs_quark_if_nil:cTF { #2 } {
        \__bnvs_gput:nnn Z { #1 } { \q_no_value }
```

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

```
        \__bnvs_first_resolve:ncTF { #1 } { #2 } {
          \__bnvs_tl_put_right:cn { #2 } { + }
          \__bnvs_length_append:ncTF { #1 } { #2 } {
            \__bnvs_tl_put_right:cn { #2 } { - 1 }
            \__bnvs_round:c { #2 }
            \__bnvs_cache_gput:nnv Z { #1 } { #2 }
            \__bnvs_gput:nnn Z { #1 } { \q_nil }
            \prg_return_true:
          } {
            \BNVS_error:x {
 Unavailable~length~for~#1~(\token_to_str:N \__bnvs_last_resolve:ncTF/1) }
```

```
            \__bnvs_gput:nnn Z { #1 } { \q_nil }
            \prg_return_false:
          }
        } {
          \BNVS_error:x {
Unavailable~first~for~#1~(\token_to_str:N \__bnvs_last_resolve:ncTF/1) }
          \__bnvs_gput:nnn Z { #1 } { \q_nil }
          \prg_return_false:
        }
      } {
        \__bnvs_quark_if_no_value:cTF { #2 } {
          \BNVS_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_resolve:vcTF { #2 } { #2 } {
            \__bnvs_cache_gput:nnv Z { #1 } { #2 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      }
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vc:cn { last_resolve } { T, F, TF }
\prg_new_conditional:Npnn \__bnvs_last_append:nc #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_last_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { last_append } { T, F, TF }
```

`\__bnvs_length_resolve:nc`*TF* `\__bnvs_length_resolve:ncTF` {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_length_append:nc`*TF* `\__bnvs_length_append:ncTF` {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the length of the ⟨*key*⟩ slide range into ⟨*tl variable*⟩, or append the length of the ⟨*key*⟩ slide range to ⟨*tl variable*⟩. Execute ⟨*yes code*⟩ when there is a ⟨*length*⟩, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { length_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_cache_get:nncTF L { #1 } { #2 } {
    \prg_return_true:
  } {
    \__bnvs_get:nncTF L { #1 } { #2 } {
      \__bnvs_quark_if_nil:cTF { #2 } {
        \__bnvs_gput:nnn L { #1 } { \q_no_value }
```

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

```
        \__bnvs_last_resolve:ncTF { #1 } { #2 } {
          \__bnvs_tl_put_right:cn { #2 } { - }
          \__bnvs_first_append:ncTF { #1 } { #2 } {
            \__bnvs_tl_put_right:cn { #2 } { + 1 }
            \__bnvs_round:c { #2 }
            \__bnvs_gput:nnn L { #1 } { \q_nil }
            \__bnvs_cache_gput:nnv L { #1 } { #2 }
            \prg_return_true:
          } {
            \BNVS_error:n {
Unavailable~first~for~#1~(\__bnvs_length_resolve:ncTF/2) }
            \return_false:
          }
        } {
          \BNVS_error:n {
Unavailable~last~for~#1~(\__bnvs_length_resolve:ncTF/1) }
          \return_false:
        }
      } {
        \__bnvs_quark_if_no_value:cTF { #2 } {
          \BNVS_fatal:n {Circular~definition:~#1}
        } {
          \__bnvs_if_resolve:vcTF { #2 } { #2 } {
            \__bnvs_cache_gput:nnv L { #1 } { #2 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      }
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vc:cn { length_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { length_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_length_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { length_append } { T, F, TF }
```

__bnvs_range_resolve:nc*TF* __bnvs_range_resolve:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_range_append:nc*TF* __bnvs_range_append:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the range of the ⟨*key*⟩ slide range into the ⟨*tl variable*⟩ or append this range to the ⟨*tl variable*⟩. Execute ⟨*yes code*⟩ when there is a ⟨*range*⟩, ⟨*no code*⟩ otherwise, in that latter case the content the ⟨*tl variable*⟩ is undefined on resolution only.

```
\BNVS_new_conditional:cpnn { range_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_first_resolve:ncTF { #1 } { a } {
    \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
      \__bnvs_tl_set:cn { a } { 0 }
    }
    \__bnvs_last_resolve:ncTF { #1 } { b } {
```

Limited from above and below.

```
      \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
        \__bnvs_tl_set:cn { b } { 0 }
      }
      \__bnvs_tl_put_right:cn { a } { - }
      \__bnvs_tl_put_right:cv { a } { b }
      \BNVS_end_tl_put_right:cv { #2 } { a }
      \prg_return_true:
    } {
```

Limited from below.

```
      \BNVS_end_tl_put_right:cv { #2 } { a }
      \__bnvs_tl_put_right:cn { #2 } { - }
      \prg_return_true:
    }
  } {
    \__bnvs_last_resolve:ncTF { #1 } { b } {
```

Limited from above.

```
      \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
        \__bnvs_tl_set:cn { b } { 0 }
      }
      \__bnvs_tl_put_left:cn { b } { - }
      \BNVS_end_tl_put_right:cv { #2 } { b }
      \prg_return_true:
    } {
      \__bnvs_value_resolve:ncTF { #1 } { b } {
      \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
        \__bnvs_tl_set:cn { b } { 0 }
      }
```

Unlimited range.

```
        \BNVS_end_tl_put_right:cv { #2 } { b }
        \__bnvs_tl_put_right:cn { #2 } { - }
        \prg_return_true:
      } {
        \BNVS_end:
        \prg_return_false:
      }
    }
  }
}
\BNVS_new_conditional_vc:cn { range_append } { T, F, TF }
\BNVS_new_conditional:cpnn { range_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_tl_clear:c { #2 }
  \__bnvs_range_append:ncTF { #1 } { #2 } {
```

```
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { range_resolve } { T, F, TF }
```

`\__bnvs_previous_resolve:nc`*TF* `\__bnvs_previous_append:ncTF {`⟨*key*⟩`} ⟨`*tl variable*⟩ `{`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_previous_append:nc`*TF*

Resolve the index after the ⟨*key*⟩ slide range into the ⟨*tl variable*⟩, or append this index to the variable. Execute ⟨*yes code*⟩ when there is a ⟨*next*⟩ index, ⟨*no code*⟩ otherwise. In the latter case, the ⟨*tl variable*⟩ is undefined on resolution only.

```
\BNVS_new_conditional:cpnn { previous_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_cache_get:nncTF P { #1 } { #2 } {
    \prg_return_true:
  } {
    \__bnvs_first_resolve:ncTF { #1 } { #2 } {
      \__bnvs_tl_put_right:cn { #2 } { -1 }
      \__bnvs_round:c { #2 }
      \__bnvs_cache_gput:nnv P { #1 } { #2 }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vc:cn { previous_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { previous_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_previous_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { previous_append } { T, F, TF }
```

`\__bnvs_next_resolve:nc`*TF* `\__bnvs_next_resolve:ncTF {`⟨*key*⟩`} ⟨`*tl variable*⟩ `{`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_next_append:nc`*TF* `\__bnvs_next_append:ncTF {`⟨*key*⟩`} ⟨`*tl variable*⟩ `{`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Resolve the index after the ⟨*key*⟩ slide range into the ⟨*tl variable*⟩, or append this index to this variable. Execute ⟨*yes code*⟩ when there is a ⟨*next*⟩ index, ⟨*no code*⟩ otherwise. In the latter case, the content of the ⟨*tl variable*⟩ is undefined, on resolution only.

```
\BNVS_new_conditional:cpnn { next_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_cache_get:nncTF N { #1 } { #2 } {
```

```
    \prg_return_true:
  } {
    \__bnvs_last_resolve:ncTF { #1 } { #2 } {
      \__bnvs_tl_put_right:cn { #2 } { +1 }
      \__bnvs_round:c { #2 }
      \__bnvs_cache_gput:nnv N { #1 } { #2 }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vc:cn { next_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { next_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_next_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_true:
  }
}
\BNVS_new_conditional_vc:cn { next_append } { T, F, TF }
```

__bnvs_v_resolve:nc*TF* __bnvs_v_resolve:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_v_append:nc*TF* __bnvs_v_append:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the value of the ⟨*key*⟩ overlay set into the ⟨*tl variable*⟩ or append this value to the right of this variable. Execute ⟨*yes code*⟩ when there is a ⟨*value*⟩, ⟨*no code*⟩ otherwise. In the latter case, the content of the ⟨*tl variable*⟩ is undefined, on resolution only.

```
\BNVS_new_conditional:cpnn { v_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_v_get:ncTF { #1 } { #2 } {
    \__bnvs_quark_if_no_value:cTF { #2 } {
      \BNVS_fatal:n {Circular~definition:~#1}
      \prg_return_false:
    } {
      \prg_return_true:
    }
  } {
    \__bnvs_v_gput:nn { #1 } { \q_no_value }
    \__bnvs_value_resolve:ncTF { #1 } { #2 } {
      \__bnvs_v_gput:nv { #1 } { #2 }
      \prg_return_true:
    } {
      \__bnvs_first_resolve:ncTF { #1 } { #2 } {
        \__bnvs_v_gput:nv { #1 } { #2 }
        \prg_return_true:
      } {
        \__bnvs_last_resolve:ncTF { #1 } { #2 } {
```

```
        \__bnvs_v_gput:nv { #1 } { #2 }
          \prg_return_true:
        } {
          \__bnvs_v_gremove:n { #1 }
          \prg_return_false:
        }
      }
    }
  }
}
\BNVS_new_conditional_vc:cn { v_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { v_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_v_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { v_append } { T, F, TF }
```

__bnvs_index_can:n*TF* __bnvs_index_can:nTF {⟨*key*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_index_can:v*TF* __bnvs_index_resolve:nncTF {⟨*key*⟩} {⟨*integer*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩}
__bnvs_index_resolve:nnc*TF* {⟨*no code*⟩}
__bnvs_index_resolve:vvc*TF* __bnvs_index_append:nncTF {⟨*key*⟩} {⟨*integer*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩}
__bnvs_index_append:nnc*TF* {⟨*no code*⟩}
__bnvs_index_append:vvc*TF*

Resolve the index associated to the ⟨*key*⟩ and ⟨*integer*⟩ slide range into the ⟨*tl variable*⟩ or append this index to the right of this variable. When ⟨*integer*⟩ is 1, this is the first index, when ⟨*integer*⟩ is 2, this is the second index, and so on. When ⟨*integer*⟩ is 0, this is the index, before the first one, and so on. If the computation is possible, ⟨*yes code*⟩ is executed, otherwise ⟨*no code*⟩ is executed. In the latter case, the content of the ⟨*tl variable*⟩ is undefined, on resolution only. The computation may fail when too many recursion calls are made.

```
\BNVS_new_conditional:cpnn { index_can:n } #1 { p, T, F, TF } {
  \bool_if:nTF {
       \__bnvs_if_in_p:nn V { #1 }
    || \__bnvs_if_in_p:nn A { #1 }
    || \__bnvs_if_in_p:nn Z { #1 }
  } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { index_can:v } #1 { p, T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_index_can:nTF { #1 } {
    \prg_return_true:
  } {
```

```
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { index_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \exp_args:Nx \__bnvs_value_resolve:ncTF { #1.#2 } { #3 } {
      \prg_return_true:
  } {
    \__bnvs_first_resolve:ncTF { #1 } { #3 } {
      \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
      \__bnvs_round:c { #3 }
      \prg_return_true:
```

Limited overlay set.

```
    } {
      \__bnvs_last_resolve:ncTF { #1 } { #3 } {
        \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
        \__bnvs_round:c { #3 }
        \prg_return_true:
      } {
        \__bnvs_value_resolve:ncTF { #1 } { #3 } {
          \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
          \__bnvs_round:c { #3 }
          \prg_return_true:
        } {
        \prg_return_false:
        }
      }
  }
  }
}
\BNVS_new_conditional:cpnn { index_resolve:nvc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \__bnvs_index_resolve:nncTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { index_resolve:vvc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \BNVS_tl_use:Nv \__bnvs_index_resolve:nncTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { index_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_index_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
```

```
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { index_append:vvc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \BNVS_tl_use:Nv \__bnvs_index_append:nncTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.13.9 Index counter

__bnvs_n_resolve:nc*TF*
__bnvs_n_append:nc*TF*
__bnvs_n_append:Vc*TF*

__bnvs_n_resolve:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Evaluate the n counter associated to the {⟨*key*⟩} overlay set into ⟨*tl variable*⟩. Initialize this counter to 1 on the first use. ⟨*no code*⟩ is never executed.

```
\BNVS_new_conditional:cpnn { n_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_n_get:ncF { #1 } { #2 } {
    \__bnvs_tl_set:cn { #2 } { 1 }
    \__bnvs_n_gput:nn { #1 } { 1 }
  }
  \prg_return_true:
}
\BNVS_new_conditional:cpnn { n_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_n_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { n_append } { T, F, TF }
```

__bnvs_n_index_resolve:nc*TF*
__bnvs_n_index_append:nc*TF*
__bnvs_n_index_resolve:nnc*TF*
__bnvs_n_index_append:nnc*TF*

__bnvs_n_index_resolve:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_index_append:ncTF {⟨*key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_index_resolve:nncTF {⟨*key*⟩} {⟨*base key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_index_append:nncTF {⟨*key*⟩} {⟨*base key*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the index for the value of the n counter associated to the {⟨*key*⟩} overlay set into the ⟨*tl variable*⟩ or append this value the right of this variable. Initialize this counter to 1 on the first use. If the computation is possible, ⟨*yes code*⟩ is executed, otherwise ⟨*no code*⟩ is executed. In the latter case, the content of the ⟨*tl variable*⟩ is undefined on resolution only.

```
\BNVS_new_conditional:cpnn { n_index_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_n_resolve:ncTF { #1 } { #2 } {
    \__bnvs_index_resolve:nvcTF { #1 } { #2 } { #2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { n_index_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_n_resolve:ncTF { #1 } { #3 } {
    \__bnvs_tl_put_left:cn { #3 } { #2. }
    \__bnvs_if_resolve:vcTF { #3 } { #3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { n_index_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_n_index_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { n_index_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_n_index_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { n_index_append } { T, F, TF }
\BNVS_new_conditional_vvc:cn { n_index_append } { T, F, TF }
```

6.13.10 Value counter

__bnvs_v_incr_resolve:nnc*TF*
__bnvs_v_incr_append:nnc*TF*
__bnvs_v_incr_append:(VnN|VVN)*TF*

__bnvs_v_incr_append:nnTF {⟨*key*⟩} {⟨*offset*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_v_incr_resolve:nncTF {⟨*key*⟩} {⟨*offset*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_v_incr_append:nncTF {⟨*key*⟩} {⟨*offset*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Increment the value counter position accordingly. When requested, put the result in the ⟨*tl variable*⟩. In the second version, the result will lay within the declared range.

```
\BNVS_new_conditional:cpnn { v_incr_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve:ncTF { #2 } { #3 } {
    \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
      \__bnvs_v_resolve:ncTF { #1 } { #3 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    } {
      \__bnvs_tl_put_right:cn { #3 } { + }
      \__bnvs_v_append:ncTF { #1 } { #3 } {
        \__bnvs_round:c { #3 }
        \__bnvs_v_gput:nv { #1 } { #3 }
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vnc:cn { v_incr_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { v_incr_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_v_incr_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vnc:cn { v_incr_append } { T, F, TF }
\BNVS_new_conditional_vvc:cn { v_incr_append } { T, F, TF }
\BNVS_new_conditional:cpnn { v_post_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_v_resolve:ncTF { #1 } { #3 } {
    \BNVS_begin:
    \__bnvs_if_resolve:ncTF { #2 } { a } {
      \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
        \BNVS_end:
```

```
        \prg_return_true:
      } {
        \__bnvs_tl_put_right:cn { a } { + }
        \__bnvs_tl_put_right:cv { a } { #3 }
        \__bnvs_round:c { a }
        \BNVS_end_v_gput:nv { #1 } { a }
        \prg_return_true:
      }
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
      \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { v_post_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { v_post_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_v_post_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \prg_return_true:
  }
}
\BNVS_new_conditional_vnc:cn { v_post_append } { T, F, TF }
\BNVS_new_conditional_vvc:cn { v_post_append } { T, F, TF }
```

__bnvs_n_incr_resolve:nnnc*TF*	__bnvs_n_incr_resolve:nncTF {⟨*key*⟩} {⟨*base key*⟩} {⟨*offset*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_incr_resolve:vvnc*TF*	
__bnvs_n_incr_resolve:nnc*TF*	__bnvs_n_incr_resolve:nncTF {⟨*key*⟩} {⟨*offset*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_incr_resolve:vvc*TF*	
__bnvs_n_incr_append:nnnc*TF*	__bnvs_n_incr_append:nnncTF {⟨*key*⟩} {⟨*base key*⟩} {⟨*offset*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_incr_append:nnc*TF*	
__bnvs_n_incr_append:(vnc|vvc)*TF*	__bnvs_n_incr_append:nncTF {⟨*key*⟩} {⟨*offset*⟩} ⟨*tl core name*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_n_post_resolve:nnc*TF*	
__bnvs_n_post_append:nnc*TF*	

Increment the implicit n counter accordingly. When requested, put the resulting index in the variable with ⟨*tl core name*⟩.

```
\BNVS_new_conditional:cpnn { n_incr_resolve:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_resolve:ncTF { #3 } { #4 } {
   \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
      \__bnvs_n_resolve:ncTF { #1 } { #4 } {
        \__bnvs_index_resolve:nvcTF { #1 } { #4 } { #4 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
```

```
        \prg_return_false:
      }
    } {
      \__bnvs_tl_put_right:cn { #4 } { + }
      \__bnvs_n_append:ncTF { #1 } { #4 } {
        \__bnvs_round:c { #4 }
        \__bnvs_n_gput:nv { #1 } { #4 }
        \__bnvs_index_resolve:nvcTF { #2 } { #4 } { #4 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { n_incr_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve:ncTF { #2 } { #3 } {
   \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
      \__bnvs_n_resolve:ncTF { #1 } { #3 } {
        \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    } {
      \__bnvs_tl_put_right:cn { #3 } { + }
      \__bnvs_n_append:ncTF { #1 } { #3 } {
        \__bnvs_round:c { #3 }
        \__bnvs_n_gput:nv { #1 } { #3 }
        \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vnc:cn { n_incr_resolve } { T, F, TF }
\BNVS_new_conditional_vvc:cn { n_incr_resolve } { T, F, TF }
\BNVS_new_conditional:cpnn { n_incr_append:nnnc } #1 #2 #3 #4 { T, F, TF } {
```

```
  \BNVS_begin:
  \__bnvs_n_incr_resolve:nnncTF { #1 } { #2 } { #3 } { #4 }{
    \BNVS_end_tl_put_right:cv { #4 } { #4 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvnc:cn { n_incr_append } { T, F, TF }
\BNVS_new_conditional_vvvc:cn { n_incr_append } { T, F, TF }
\BNVS_new_conditional:cpnn { n_incr_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_n_incr_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vnc:cn { n_incr_append } { T, F, TF }
\BNVS_new_conditional_vvc:cn { n_incr_append } { T, F, TF }
```

`\__bnvs_v_post_resolve:nnc`*TF*
`\__bnvs_v_post_resolve:vvc`*TF*
`\__bnvs_v_post_append:nnc`*TF*
`\__bnvs_v_post_append:(vnN|vvN)`*TF*

`\__bnvs_v_post_resolve:nncTF` {⟨*key*⟩} {⟨*offset*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_v_post_append:nncTF` {⟨*key*⟩} {⟨*offset*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the value of the free counter for the given ⟨*key*⟩ into the ⟨*tl variable*⟩ then increment this free counter position accordingly. The append version, appends the value to the right of the ⟨*tl variable*⟩. The content of the ⟨*tl variable*⟩ is undefined while in the {⟨*no code*⟩} branch and on resolution only.

```
\BNVS_new_conditional:cpnn { n_post_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_n_resolve:ncTF { #1 } { #3 } {
    \BNVS_begin:
    \__bnvs_if_resolve:ncTF { #2 } { #3 } {
      \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
        \BNVS_end:
        \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \__bnvs_tl_put_right:cn { #3 } { + }
        \__bnvs_n_append:ncTF { #1 } { #3 } {
          \__bnvs_round:c { #3 }
          \__bnvs_n_gput:nv { #1 } { #3 }
          \BNVS_end:
          \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
```

```
            \prg_return_true:
          } {
            \prg_return_false:
          }
        } {
          \BNVS_end:
          \prg_return_false:
        }
      }
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { n_post_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_n_post_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vnc:cn { n_post_append } { T, F, TF }
\BNVS_new_conditional_vvc:cn { n_post_append } { T, F, TF }
```

6.13.11 Evaluation

`\__bnvs_round_ans:` `\__bnvs_rslv_round:`

Helper function to round the `\l__bnvs_ans_tl` variable. For ranges only, this will be set to `\prg_do_nothing` because we do not want to interpret the `-` sign as a minus operator.

```
\BNVS_set:cpn { round_ans: } {
  \__bnvs_round:c { ans }
}
```

6.13.12 Functions for the resolution

They manily start with `\__bnvs_if_resolve_`

`\__bnvs_if_resolve_end_return_false:n` `\__bnvs_if_resolve_end_return_false:n {`⟨*message*⟩`}`

Close one TEX group, display a message and return false.

`\__bnvs_path_resolve_n:TFF` `\__bnvs_path_resolve_n:TFF {`⟨*yes code*⟩`} {`⟨*no code 1*⟩`} {`⟨*no code 2*⟩`}`

```
\BNVS_new:cpn { path_resolve_n:TFF } #1 #2 {
  \__bnvs_kip_n_path_resolve:TF {
    \__bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
  }
}
```

__bnvs_path_resolve_n:T `\__bnvs_path_resolve_n:T {⟨yes code⟩}`

Resolve the path and execute ⟨*yes code*⟩ on success.

```
\BNVS_new:cpn { if_resolve_end_return_false:n } #1 {
  \BNVS_end:
  \prg_return_false:
}
\BNVS_new:cpn { path_resolve_n:T } #1 {
  \__bnvs_path_resolve_n:TFF {
    #1
  } {
    \__bnvs_if_resolve_end_return_false:n {
      Too~many~dotted~components
    }
  } {
    \__bnvs_if_resolve_end_return_false:n {
      Unknown~dotted~path
    }
  }
}
\BNVS_set:cpn { resolve_x:T } #1 {
  \__bnvs_kip_x_path_resolve:TFF {
    #1
  } {
    \__bnvs_if_resolve_end_return_false:n {
      Too~many~dotted~components
    }
  } {
    \__bnvs_if_resolve_end_return_false:n { Unknown~dotted~path }
  }
}
```

__bnvs_path_suffix:n*TF* `\__bnvs_path_suffix:nTF {⟨tl⟩} {⟨yes code⟩} {⟨no code⟩}`

If the last item of `\l__bnvs_path_seq` is ⟨*suffix*⟩, then execute ⟨*yes code*⟩ otherwise execute ⟨*no code*⟩. The suffix is `n` in the second case.

```
\BNVS_new_conditional:cpnn { path_pop_right_n:c } #1 { T, F, TF } {
  \__bnvs_seq_pop_right:ccTF { path } { #1 }
    { \prg_return_true: } { \prg_return_false: }
}
```

__bnvs_if_resolve_pop_kip:TTF
__bnvs_if_resolve_pop_complete_white:T
__bnvs_if_resolve_pop_complete_black:T

__bnvs_if_resolve_pop_kip:TTF {⟨*blank code*⟩} {⟨*black code*⟩}
{⟨*end code*⟩}
__bnvs_if_resolve_pop_complete_white:T {⟨*blank code*⟩}
__bnvs_if_resolve_pop_complete_black:T {⟨*black code*⟩}

For `\__bnvs_if_resolve_pop_kip:TTF`. If the `split` sequence is empty, execute ⟨*end code*⟩. Otherwise pops the 3 heading items of the `split` sequence into the three `tl` variables `key`, `id`, `path`. If `key` is blank then execute ⟨*blank code*⟩, otherwise execute ⟨*black code*⟩.

For `\__bnvs_if_resolve_pop_complete_white:T`: pops the three heading items of the `split` sequence into the three variables `n_incr`, `incr`, `post`. Then execute ⟨*blank code*⟩.

For `\__bnvs_if_resolve_pop_complete_black:T`: pops the six heading items of the `split` sequence then execute ⟨*blank code*⟩.

```
\BNVS_new:cpn { if_resolve_pop_kip_complete: } {
  \__bnvs_tl_if_blank:vT { id } {
    \__bnvs_tl_put_left:cv { key } { id_last }
    \__bnvs_tl_set:cv { id } { id_last }
  }
  \__bnvs_tl_if_blank:vTF { path } {
    \__bnvs_seq_clear:c { path }
  } {
    \__bnvs_seq_set_split:cnv { path } { . } { path }
    \__bnvs_seq_remove_all:cn { path } { }
  }
  \__bnvs_tl_set_eq:cc { key_base } { key }
  \__bnvs_seq_set_eq:cc { path_base } { path }
}
\BNVS_new:cpn { if_resolve_pop_kip:TTF } #1 #2 #3 {
  \__bnvs_split_pop_left:cTF { key } {
    \__bnvs_split_pop_left:cTF { id } {
      \__bnvs_split_pop_left:cTF { path } {
        \__bnvs_tl_if_blank:vTF { key } {
```

The first 3 capture groups are empty, and the 3 next ones are expected to contain the expected information.

```
          #1
        } {
          \__bnvs_if_resolve_pop_kip_complete:
          #2
        }
      } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_kip:TTF/2 }
      }
    } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_kip:TTF/1 }
    }
  } { #3 }
}
```

__bnvs_if_resolve_pop_complete:nNT __bnvs_if_resolve_pop_kip:FFTF {⟨*empty key code*⟩} {⟨*no id code*⟩} {⟨*yes code*⟩} {⟨*no capture code*⟩}
__bnvs_if_resolve_pop_complete:nNT {⟨*tl*⟩} ⟨*tl var*⟩ {⟨*yes code*⟩}

⟨*tl*⟩ and ⟨*tl var*⟩ are the arguments of the `\__bnvs_if_resolve:nc` conditionals. conditional variants.

`\__bnvs_if_resolve_pop_kip:FFTF` locally sets the `key`, `id` and `path` `tl` variables to the 3 heading items of the split sequence, which correspond to the 3 eponym capture groups. If no capture group is available, ⟨*no capture code*⟩ is executed. If the capture group for the key is empty, then ⟨*empty key code*⟩ is executed. If there is no capture group for the id, then ⟨*no id code*⟩ is executed. Otherwise ⟨*yes code*⟩ is executed.

`\__bnvs_rslv_pop_end:T` locally sets the three `tl` variables `n_incr`, `incr` and `post` to the three heading items of the split sequence, which correspond to the last 3 eponym capture groups.

```
\BNVS_new:cpn { if_resolve_pop_complete_white:T } #1 {
  \__bnvs_split_pop_left:cTF { n_incr } {
    \__bnvs_split_pop_left:cTF { incr } {
      \__bnvs_split_pop_left:cTF { post } {
        #1
      } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/3 }
      }
    } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/2 }
    }
  } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/1 }
  }
}
\BNVS_new:cpn { if_resolve_pop_complete_black:T } #1 {
  \__bnvs_split_pop_left:cTF { a } {
    \__bnvs_split_pop_left:cTF { a } {
      \__bnvs_split_pop_left:cTF { a } {
        \__bnvs_split_pop_left:cTF { a } {
          \__bnvs_split_pop_left:cTF { a } {
            \__bnvs_split_pop_left:cTF { a } {
              #1
            } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/6 }
            }
          } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/5 }
          }
        } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/4 }
        }
      } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/3 }
      }
    } {
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/2 }
    }
  } {
```

```
\__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/1 }
  }
}
```

__bnvs_if_resolve:nc*TF*
__bnvs_if_resolve:vc*TF*
__bnvs_if_append:nc*TF*
__bnvs_if_append:(vc|xc)*TF*

__bnvs_if_append:ncTF {⟨*expression*⟩} ⟨*tl variable*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Evaluates the ⟨*expression*⟩, replacing all the named overlay specifications by their static counterpart then put the rounded result in ⟨*tl variable*⟩ when resolving or to the right of the ⟨*tl variable*⟩ when appending. Executed within a group. Heavily used by __bnvs_query_eval:nc, where ⟨*integer expression*⟩ was initially enclosed inside '?(...)'. Local variables:

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

(*End of definition for* \l__bnvs_ans_tl.)

\l__bnvs_split_seq The sequence of catched query groups and non queries.

(*End of definition for* \l__bnvs_split_seq.)

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

(*End of definition for* \l__bnvs_split_int.)

```
\BNVS_int_new:c { split }
```

\l__bnvs_key_tl Storage for split sequence items that represent names.

(*End of definition for* \l__bnvs_key_tl.)

\l__bnvs_path_tl Storage for split sequence items that represent integer paths.

(*End of definition for* \l__bnvs_path_tl.)

Catch circular definitions. Open a main TEX group to define local functions and variables, sometimes another grouping level is used. The main TEX group is closed in the various \...end_return... functions.

```
\BNVS_new:cpn { kip_x_path_resolve_or_end_return_false:nT } #1 #2 {
  \__bnvs_kip_x_path_resolve:TFF {
    #2
  } {
    \BNVS_end_return_false:x { Too~many~dotted~components:~#1 }
  } {
    \BNVS_end_return_false:x { Unknown~dotted~path:~#1 }
  }
}
\BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

```
\BNVS_new:cpn { end_unreachable_return_false:n } #1 {
  \BNVS_error:x { UNREACHABLE/#1 }
  \BNVS_end:
  \prg_return_false:
}
\BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_call:TF {
    \BNVS_begin:
    \BNVS_set:cpn { if_resolve_warning:n } ##1 {
      \__bnvs_warning:n { #1:~##1 }
      \BNVS_set:cpn { if_resolve_warning:n } {
        \use_none:n
      }
    }
```

This TEX group will be closed just before returning. Implementation:

```
  \__bnvs_regex_split:cnTF { split } { #1 } {
```

The leftmost item is not a special item: we start feeding `\l__bnvs_ans_tl` with it.

```
      \BNVS_set:cpn { if_resolve_end_return_true: } {
```

Normal and unique end of the loop.

```
        \__bnvs_if_resolve_round_ans:
        \BNVS_end_tl_set:cv { #2 } { ans }
        \prg_return_true:
      }
      \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round_ans: }
      \__bnvs_tl_clear:c { ans }
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_tl_clear:c { ans }
      \__bnvs_round_ans:n { #1 }
      \BNVS_end_tl_set:cv { #2 } { ans }
      \prg_return_true:
    }
  } {
    \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { if_append:vc } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_if_append:ncTF { #1 } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional:cpnn { if_resolve:vc } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_if_resolve:ncTF { #1 } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

Next functions are helpers for the `\__bnvs_if_resolve:nc` conditional variants. When present, their two first arguments ⟨*tl*⟩ and ⟨*tl var*⟩ are exactly the ones given to the variants.

`\__bnvs_if_resolve_loop_or_end_return:` `\__bnvs_if_resolve_loop_or_end_return:`

May call itself at the end.

```
\BNVS_new:cpn { if_resolve_loop_or_end_return: } {
  \__bnvs_split_pop_left:cTF { a } {
    \__bnvs_tl_put_right:cv { ans } { a }
    \__bnvs_if_resolve_pop_kip:TTF {
      \__bnvs_if_resolve_pop_kip:TTF {
\__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/3 }
      } {
        \__bnvs_if_resolve_pop_complete_white:T {
          \__bnvs_tl_if_blank:vTF { n_incr } {
            \__bnvs_tl_if_blank:vTF { incr } {
              \__bnvs_tl_if_blank:vTF { post } {
                \__bnvs_if_resolve_value_loop_or_end_return_true:F {
```

Only the dotted path, branch according to the last component.

```
                  \__bnvs_seq_pop_right:ccTF { path } { a } {
                    \BNVS_tl_use:Nv \str_case:nnF { a } {
{ n         } { \BNVS_use:c { if_resolve_loop_or_end_return[.n]: } }
{ length    } { \BNVS_use:c { if_resolve_loop_or_end_return[.length]: } }
{ last      } { \BNVS_use:c { if_resolve_loop_or_end_return[.last]:  } }
{ range     } { \BNVS_use:c { if_resolve_loop_or_end_return[.range]: } }
{ previous  } { \BNVS_use:c { if_resolve_loop_or_end_return[.previous]: } }
{ next      } { \BNVS_use:c { if_resolve_loop_or_end_return[.next]:  } }
{ reset     } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset]: } }
{ reset_all } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset_all]: } }
                    } {
\BNVS_use:c { if_resolve_loop_or_end_return[...<integer>]: }
                    }
                  } {
\BNVS_use:c { if_resolve_loop_or_end_return[...]: }
                  }
                }
              } {
\BNVS_use:c { if_resolve_loop_or_end_return[...++]: }
              }
            } {
              \__bnvs_path_suffix:nTF { n } {
\BNVS_use:c { if_resolve_loop_or_end_return[...n+=...]: }
              } {
\BNVS_use:c { if_resolve_loop_or_end_return[...+=...]: }
              }
            }
          } {
\BNVS_use:c { if_resolve_loop_or_end_return[...++n]: }
          }
        }
      } {
% split sequence empty
```

```
\__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/2 }
      }
    } {
      \__bnvs_if_resolve_pop_complete_black:T {
        \__bnvs_path_suffix:nTF { n } {
\BNVS_use:c { if_resolve_loop_or_end_return[++...n]: }
        } {
\BNVS_use:c { if_resolve_loop_or_end_return[++...]: }
        }
      }
    } {
      \__bnvs_if_resolve_end_return_true:
    }
  } {
\__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/1 }
  }
}
\BNVS_set:cpn { if_resolve_value_loop_or_end_return_true:F } #1 {
  \__bnvs_tl_set:cx { a } {
    \BNVS_tl_use:c { key } \BNVS_tl_use:c { path }
  }
  \__bnvs_v_resolve:vcTF { a } { a } {
    \__bnvs_tl_put_right:cv { ans } { a }
    \__bnvs_if_resolve_loop_or_end_return:
  } {
    \__bnvs_value_resolve:vcTF { a } { a } {
      \__bnvs_tl_put_right:cv { ans } { a }
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      #1
    }
  }
}
\BNVS_new:cpn { end_return_error:n } #1 {
      \BNVS_error:n { #1 }
      \BNVS_end:
      \prg_return_false:
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.n]: } {
```

- Case `...n`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_base_resolve_n:
    \__bnvs_n_index_append:vvcTF { key } { key_base } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_end_return_error:n {
        Undefined~dotted~path
      }
    }
  }
}
```

```
\BNVS_new_conditional:cpnn { path_suffix:n } #1 { T, F, TF } {
  \__bnvs_seq_get_right:ccTF { path } { a } {
    \__bnvs_tl_if_eq:cnTF { a } { #1 } {
      \__bnvs_seq_pop_right:ccT { path } { a } { }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.length]: } {
```

- Case `...length`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_length_append:vcTF { key } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~length }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.last]: } {
```

- Case `...last`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_last_append:vcTF { key } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \BNVS_end_return_false:x { NO~last }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.range]: } {
```

- Case `...range`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_range_append:vcTF { key } { ans } {
      \BNVS_set:cpn { if_resolve_round_ans: } { \prg_do_nothing: }
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~range }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.previous]: } {
```

- Case `...previous`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_previous_append:vcTF { key } { ans } {
```

```
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~previous }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.next]: } {
```

- Case `...next`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_next_append:vcTF { key } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~next }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.reset]: } {
```

- Case `...reset`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_v_greset:vnT { key } { } { }
    \__bnvs_value_append:vcTF { key } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~reset }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[.reset_all]: } {
```

- Case `...reset_all`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_greset_all:vnT { key } { } { }
    \__bnvs_value_append:vcTF { key } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~reset }
    }
  }
}
\BNVS_set:cpn { if_resolve_loop_or_end_return[...<integer>]: } {
```

- Case `...`⟨*integer*⟩.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_index_append:vvcTF { key } { a } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~integer }
    }
  }
}
\BNVS_set:cpn { if_resolve_loop_or_end_return[...]: } {
```

- Case `....`

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_value_append:vcTF { key } { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~value }
    }
  }
}
\BNVS_set:cpn { if_resolve_loop_or_end_return[...++]: } {
```

- Case `...++`.

```
  \__bnvs_path_suffix:nTF { reset } {
    \__bnvs_path_resolve_n:T {
      \__bnvs_v_greset:vnT { key } { } { }
      \__bnvs_v_post_append:vncTF { key } { 1 }  { ans } {
        \__bnvs_if_resolve_loop_or_end_return:
      } {
        \__bnvs_if_resolve_end_return_false:n { NO~post }
      }
    }
  } {
    \__bnvs_path_suffix:nTF { reset_all } {
      \__bnvs_path_resolve_n:T {
        \__bnvs_greset_all:vnT { key } { } { }
        \__bnvs_v_post_append:vncTF { key } { 1 }  { ans } {
          \__bnvs_if_resolve_loop_or_end_return:
        } {
          \__bnvs_if_resolve_end_return_false:n { NO~post }
        }
      }
    } {
      \__bnvs_path_resolve_n:T {
        \__bnvs_v_post_append:vncTF { key } { 1 }  { ans } {
          \__bnvs_if_resolve_loop_or_end_return:
        } {
          \__bnvs_if_resolve_end_return_false:n { NO~post }
        }
      }
    }
  }
}
\BNVS_set:cpn { if_resolve_loop_or_end_return[...n+=...]: } {
```

- Case `....n+=`⟨*integer*⟩.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_base_resolve_n:
    \__bnvs_n_incr_append:vvvcTF { key } { key_base }  { incr }  { ans } {
%    \begin{macrocode}
```

```
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n {
        NO~n~incrementation
      }
    }
  }
}
\BNVS_set:cpn { if_resolve_loop_or_end_return[...+=...]: } {
```

- Case `A+=`⟨*integer*⟩.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_v_incr_append:vvcTF { key }  { incr }  { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n {
        NO~incremented~value
      }
    }
  }
}
\BNVS_new:cpn { base_resolve_n: } {
  \__bnvs_seq_if_empty:cF { path_base } {
    \__bnvs_seq_pop_right:cc { path_base } { a }
    \__bnvs_seq_if_empty:cF { path_base } {
      \__bnvs_tl_put_right:cx { key_base } {
        . \__bnvs_seq_use:cn { path_base } { . }
      }
    }
  }
}
\BNVS_new:cpn { base_resolve: } {
  \__bnvs_seq_if_empty:cF { path_base } {
    \__bnvs_tl_put_right:cx { key_base } {
      . \__bnvs_seq_use:cn { path_base } { . }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[...++n]: } {
```

- Case `...++n`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_base_resolve:
    \__bnvs_n_incr_append:vvncTF { key } { key_base } { 1 }  { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~...++n }
    }
  }
}
\BNVS_set:cpn { if_resolve_loop_or_end_return[++...n]: } {
```

- Case `++...n`.

```
  \__bnvs_path_resolve_n:T {
    \__bnvs_base_resolve_n:
    \__bnvs_n_incr_append:vvncTF { key } { key_base } { 1 }  { ans } {
      \__bnvs_if_resolve_loop_or_end_return:
    } {
      \__bnvs_if_resolve_end_return_false:n { NO~++...n }
    }
  }
}
\BNVS_new:cpn { if_resolve_loop_or_end_return[++...]: } {
```

- Case `++...`.

```
  \__bnvs_path_suffix:nTF { reset } {
    \__bnvs_path_resolve_n:T {
      \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
        \__bnvs_v_greset:vnT { key } { } { }
        \__bnvs_if_resolve_loop_or_end_return:
      } {
        \__bnvs_v_greset:vnT { key } { } { }
        \__bnvs_if_resolve_end_return_false:n { No~increment }
      }
    }
  } {
    \__bnvs_path_suffix:nTF { reset_all } {
      \__bnvs_path_resolve_n:T {
        \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
          \__bnvs_greset_all:vnT { key } { } { }
          \__bnvs_if_resolve_loop_or_end_return:
        } {
          \__bnvs_greset_all:vnT { key } { } { }
          \__bnvs_if_resolve_end_return_false:n { No~increment }
        }
      }
    } {
      \__bnvs_path_resolve_n:T {
        \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
          \__bnvs_if_resolve_loop_or_end_return:
        } {
          \__bnvs_if_resolve_end_return_false:n { No~increment }
        }
      }
    }
  }
}
```

__bnvs_query_eval:nc*TF* `\__bnvs_query_eval:ncTF {⟨overlay query⟩} {⟨tl core⟩} {⟨yes code⟩} {⟨no 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 `\l__bnvs_ans_tl`. Ranges are supported with the colon syntax. This is executed within a local TEX group managed by the caller. Below are local variables and constants.

`\l__bnvs_V_tl` Storage for a single value out of a range.

(*End of definition for* `\l__bnvs_V_tl`.)

`\l__bnvs_A_tl` Storage for the first component of a range.

(*End of definition for* `\l__bnvs_A_tl`.)

`\l__bnvs_Z_tl` Storage for the last component of a range.

(*End of definition for* `\l__bnvs_Z_tl`.)

`\l__bnvs_L_tl` Storage for the length component of a range.

(*End of definition for* `\l__bnvs_L_tl`.)

`\c__bnvs_A_cln_Z_regex` Used to parse named overlay specifications. V, A:Z, A::L on one side, :Z, :Z::L and ::L:Z on the other sides. Next are the capture groups. The first one is for the whole match.

(*End of definition for* `\c__bnvs_A_cln_Z_regex`.)

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

- 2 → V

```
    ( [^:]+? )
```

- 3, 4, 5 → A : Z? or A :: L?

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

- 6, 7 → ::(L:Z)?

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

- 8, 9 → :(Z::L)?

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


\BNVS_set:cpn { query_eval_end_return_true: } {
  \BNVS_end:
  \prg_return_true:
}
\BNVS_new:cpn { query_eval_end_return_false: } {
  \BNVS_end:
  \prg_return_false:
```

```
}
\BNVS_new:cpn { query_eval_end_return_false:n } #1 {
  \BNVS_end:
  \prg_return_false:
}
\BNVS_new:cpn { query_eval_error_end_return_false:n } #1 {
  \BNVS_error:x { #1 }
  \__bnvs_query_eval_end_return_false:
}
\BNVS_new:cpn { query_eval_unreachable: } {
  \__bnvs_query_eval_error_end_return_false:n { UNREACHABLE }
}
\BNVS_new:cpn { if_blank:cTF } #1 {
  \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
}
\BNVS_new_conditional:cpnn { match_pop_left:c } #1 { T, F, TF } {
  \BNVS_tl_use:nc {
    \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
  } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

`\__bnvs_query_eval_match_branch:`*TF* `\__bnvs_query_eval_match_branch:TF {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Called by `\__bnvs_query_eval:ncTF` that just filled `\l__bnvs_match_seq` after the `c__bnvs_A_cln_Z_regex`. Puts the proper items of `\l__bnvs_match_seq` into the variables `\l__bnvs_V_tl`, `\l__bnvs_A_tl`, `\l__bnvs_Z_tl`, `\l__bnvs_L_tl` then branches accordingly on one of the returning `\__bnvs_query_eval_return[`⟨*description*⟩`]:` functions. All these functions properly set the `\l__bnvs_ans_tl` variable and they end with either `\prg_return_true:` or `\prg_return_false:`. This is used only once but is not inlined for readability.

```
\BNVS_new_conditional:cpnn { query_eval_match_branch: } { T, F, TF } {
```

At start, we ignore the whole match.

```
  \__bnvs_match_pop_left:cT V {
    \__bnvs_match_pop_left:cT V {
      \__bnvs_if_blank:cTF V {
        \__bnvs_match_pop_left:cT A {
          \__bnvs_match_pop_left:cT Z {
            \__bnvs_match_pop_left:cT L {
              \__bnvs_if_blank:cTF A {
                \__bnvs_match_pop_left:cT L {
                  \__bnvs_match_pop_left:cT Z {
                    \__bnvs_if_blank:cTF L {
                      \__bnvs_match_pop_left:cT Z {
                        \__bnvs_match_pop_left:cT L {
                          \__bnvs_if_blank:cTF L {
                            \BNVS_use:c { query_eval_return[:Z]: }
                          } {
                            \BNVS_use:c { query_eval_return[:Z::L]: }
```

```
                          }
                        }
                      }
                    } {
                      \__bnvs_if_blank:cTF Z {
\__bnvs_query_eval_error_end_return_false:n { Missing~first~or~last }
                      } {
                        \BNVS_use:c { query_eval_return[:Z::L]: }
                      }
                    }
                  }
                } {
                  \__bnvs_if_blank:cTF Z {
                    \__bnvs_if_blank:cTF L {
                      \BNVS_use:c { query_eval_return[A:]: }
                    } {
                      \BNVS_use:c { query_eval_return[A::L]: }
                    }
                  } {
                    \__bnvs_if_blank:cTF L {
                      \BNVS_use:c { query_eval_return[A:Z]: }
                    } {
```

Logically unreachable code, the regular expression does not match this.

```
                      \__bnvs_query_eval_unreachable:
                    }
                  }
                }
              }
            }
          }
        } {
          \BNVS_use:c { query_eval_return[V]: }
        }
      }
    }
  }
}
\BNVS_new:cpn { query_eval_return[V]: } {
```

Single value

```
  \__bnvs_if_resolve:vcTF { V } { ans } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { query_eval_return[A:Z]: } {
```

$\langle first \rangle$:$\langle last \rangle$ range

```
  \__bnvs_if_resolve:vcTF { A } { ans } {
    \__bnvs_tl_put_right:cn { ans } { - }
    \__bnvs_if_append:vcTF { Z } { ans } {
      \prg_return_true:
    } {
```

```
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { query_eval_return[A::L]: } {
```

⟨*first*⟩::⟨*length*⟩ range

```
  \__bnvs_if_resolve:vcTF { A } { A } {
    \__bnvs_if_resolve:vcTF { L } { ans } {
      \__bnvs_tl_put_right:cn { ans } { + }
      \__bnvs_tl_put_right:cv { ans } { A }
      \__bnvs_tl_put_right:cn { ans } { -1 }
      \__bnvs_round_ans:
      \__bnvs_tl_put_left:cn { ans } { - }
      \__bnvs_tl_put_left:cv { ans } { A }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { query_eval_return[A:]: } {
```

⟨*first*⟩: and ⟨*first*⟩:: range

```
  \__bnvs_if_resolve:vcTF { A } { ans } {
    \__bnvs_tl_put_right:cn { ans } { - }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { query_eval_return[:Z::L]: } {
```

:⟨*last*⟩::⟨*length*⟩ or ::⟨*length*⟩:⟨*last*⟩ range

```
  \__bnvs_if_resolve:vcTF { Z } { Z } {
    \__bnvs_if_resolve:vcTF { L } { ans } {
      \__bnvs_tl_put_left:cn  { ans } { 1-}
      \__bnvs_tl_put_right:cn { ans } { + }
      \__bnvs_tl_put_right:cv { ans } { Z }
      \__bnvs_round_ans:
      \__bnvs_tl_put_right:cn { ans } { - }
      \__bnvs_tl_put_right:cv { ans } { Z }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\BNVS_new:cpn { query_eval_return[:]: } {
```

: or :: range

```
    \__bnvs_tl_set:cn { ans } { - }
    \prg_return_true:
  }
  \BNVS_new:cpn { query_eval_return[:Z]: } {
```

:⟨*last*⟩ range

```
    \__bnvs_tl_set:cn { ans } { - }
    \__bnvs_if_append:vcTF { Z } { ans } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
```

__bnvs_query_eval:nc*TF* __bnvs_query_eval:ncTF {⟨*query*⟩} {⟨*tl core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Evaluate only one query.

```
  \BNVS_new_conditional:cpnn { query_eval:nc } #1 #2 { T, F, TF } {
    \__bnvs_call_greset:
    \__bnvs_match_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
      \BNVS_begin:
      \__bnvs_query_eval_match_branch:TF {
        \BNVS_end_tl_set:cv { #2 } { ans }
        \prg_return_true:
      } {
        \BNVS_end:
        \prg_return_false:
      }
    } {
```

Error

```
      \BNVS_error:n { Syntax~error:~#1 }
      \prg_return_false:
    }
  }
```

__bnvs_eval:nc `\__bnvs_eval:nc {⟨overlay query list⟩} {⟨core tl name⟩}`

This is called by the *named overlay specifications* scanner. Evaluates the comma separated ⟨*overlay query list*⟩, replacing all the individual named overlay specifications and integer expressions by their static counterparts by calling `\__bnvs_query_eval:nc`, then append the result to the right of the ⟨*core tl name*⟩ 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 of definition for* `\l__bnvs_query_seq`.)

\l__bnvs_ans_seq Storage for the evaluated result.

(*End of 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 of definition for* `\c__bnvs_comma_regex`.)

No other variable is used.

```
\BNVS_new:cpn { eval:nc } #1 #2 {
  \BNVS_begin:
```

Local variables cleared

```
  \__bnvs_seq_clear:c { ans }
```

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 justed cleared above.

```
  \__bnvs_seq_map_inline:cn { query } {
    \__bnvs_tl_clear:c { ans }
    \__bnvs_query_eval:ncTF { ##1 } { ans } {
      \__bnvs_seq_put_right:cv { ans } { ans }
    } {
      \seq_map_break:n {
        \BNVS_error:n { Circular/Undefined~dependency~in~#1}
      }
    }
  }
```

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

```
  \exp_args:NNnx
  \BNVS_end:
  \__bnvs_tl_put_right:cn { #2 } { \__bnvs_seq_use:cn { ans } , }
}
```

\BeanovesEval `\BeanovesEval [⟨setup⟩] {⟨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. `\...ans_tl` is used locally to store the result.

The optional ⟨*setup*⟩ is a key–value list. The value for `in:N` key is the `tl` variable where the evaluation is stored. If the `see` key is provided, the result is typeset.

```
\NewDocumentCommand \BeanovesEval { O{} m } {
  \BNVS_begin:
  \keys_define:nn { BeanovesEval } {
    in:N .tl_set:N = \l__bnvs_eval_in_tl,
    in:N .initial:n = { },
    see .bool_set:N = \l__bnvs_eval_see_bool,
    see .default:n = true,
    see .initial:n = false,
  }
  \keys_set:nn { BeanovesEval } { #1 }
  \__bnvs_tl_clear:c { ans }
  \__bnvs_eval:nc { #2 } { ans }
  \__bnvs_tl_if_empty:cTF { eval_in } {
    \bool_if:nTF { \l__bnvs_eval_see_bool } {
      \BNVS_tl_use:Nv \BNVS_end: { ans }
    } {
      \BNVS_end:
    }
  } {
    \bool_if:nTF { \l__bnvs_eval_see_bool } {
      \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
        \BNVS_end:
        \tl_set:Nn ##1 { ##2 }
        ##2
      }
      \BNVS_tl_use:nv {
        \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_eval_in_tl
      } { ans }
    } {
      \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
        \BNVS_end:
        \tl_set:Nn ##1 { ##2 }
      }
      \BNVS_tl_use:nv {
        \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_eval_in_tl
      } { ans }
    }
  }
}
```

6.13.13 Reseting counters

\BeanovesReset `\BeanovesReset [⟨first value⟩] {⟨ref⟩}`
\BeanovesReset* `\BeanovesReset* [⟨first value⟩] {⟨ref⟩}`

Forwards to `\__bnvs_v_greset:nnF` or `\__bnvs_greset_all:nnF` when starred.

```
\NewDocumentCommand \BeanovesReset { s O{} m } {
  \__bnvs_id_name_n_get:nTF { #3 } {
    \BNVS_tl_use:nv {
      \IfBooleanTF { #1 } {
        \__bnvs_greset_all:nnF
      } {
        \__bnvs_v_greset:nnF
      }
    } { key } { #2 } {
%      \__bnvs_warning:n { Unknown~name:~#3 }
    }
  } {
    \__bnvs_warning:n { Bad~name:~#3 }
  }
  \ignorespaces
}

\ExplSyntaxOff
```