

beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2023/01/07

Abstract

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

Contents

1	Minimal example	2
2	Named overlay sets	3
2.1	Presentation	3
2.2	Named overlay reference	3
2.3	Defining named overlay sets	4
2.3.1	Basic case	4
2.3.2	List specifiers	5
2.3.3	.n specifiers	5
3	Named overlay resolution	5
3.1	Simple definitions	5
3.2	Counters	6
3.3	Dotted paths	7
3.4	Frame id	8
4	?(...) query expressions	8
5	Support	9

6	Implementation	9
6.1	Package declarations	9
6.2	Facility layer: definitions and naming	9
6.3	logging	11
6.4	Facility layer: Variables	11
6.4.1	Regex	16
6.4.2	Token lists	18
6.4.3	Strings	21
6.4.4	Sequences	22
6.4.5	Integers	24
6.4.6	Prop	24
6.5	Debug facilities	24
6.6	Debug messages	25
6.7	Variable facilities	25
6.8	Testing facilities	25
6.9	Local variables	25
6.10	Infinite loop management	26
6.11	Overlay specification	27
6.12	Basic functions	27
6.13	Functions with cache	29
6.13.1	Implicit value counter	31
6.13.2	Implicit index counter	33
6.13.3	Regular expressions	35
6.13.4	beamer.cls interface	37
6.13.5	Defining named slide ranges	37
6.13.6	Scanning named overlay specifications	47
6.13.7	Resolution	51
6.13.8	Evaluation bricks	56
6.13.9	Index counter	68
6.13.10	Value counter	70
6.13.11	Evaluation	75
6.13.12	Functions for the resolution	75
6.13.13	Reseting counters	96

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

```

On line 4, we use the `\Beanoves` command to declare *named overlay sets*. On line 5, we declare an overlay set named ‘A’, which is a range starting at slide 1 and 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 sets is a range of integers covering all the slide numbers, from one to the total amount of slides. In general, an overlay set is a range of positive integers identified by a unique name. The main practical interest is that such sets may be defined relative to one another, we can even have lists of overlay sets. Finally, we can use these lists to build and organize `beamer` overlay specifications logically.

2.2 Named overlay reference

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

`<short name>` : like `A` and `C`,

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

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

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

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

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

The mapping from *named overlay references* to integers is defined at the global T_EX level to allow its use in $\backslash \text{begin}\{\text{frame}\} \langle \dots \rangle$ 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 $\backslash \text{Beanoves}$ 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 $\backslash \text{Beanoves}$ 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	$\text{beanoves} = \{ \langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle, \langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle, \dots, \langle \text{ref}_n \rangle = \langle \text{spec}_n \rangle \}$
-----------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$\backslash \text{Beanoves}$	$\backslash \text{Beanoves}\{ \langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle, \langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle, \dots, \langle \text{ref}_n \rangle = \langle \text{spec}_n \rangle \}$
------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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

2.3.1 Basic case

In the possible values for $\langle \text{spec} \rangle$ below, $\langle \text{value} \rangle$, $\langle \text{first} \rangle$, $\langle \text{length} \rangle$ and $\langle \text{last} \rangle$ are algebraic expression possibly involving any *named overlay reference* defined above.

$\langle \text{value} \rangle$, the simple *value specifiers* for the whole signed integers set. If only the $\langle \text{key} \rangle$ is provided, the $\langle \text{value} \rangle$ defaults to 1.

$\langle \text{first} \rangle$: and $\langle \text{first} \rangle ::$, for the infinite range of signed integers starting at and including $\langle \text{first} \rangle$.

$:\langle \text{last} \rangle$, for the infinite range of signed integers ending at and including $\langle \text{last} \rangle$.

$\langle \text{first} \rangle : \langle \text{last} \rangle$, $\langle \text{first} \rangle :: \langle \text{length} \rangle$, $:\langle \text{last} \rangle :: \langle \text{length} \rangle$, $:: \langle \text{length} \rangle : \langle \text{last} \rangle$, are variants for the finite range of signed integers starting at and including $\langle \text{first} \rangle$, ending at and including $\langle \text{last} \rangle$. At least one of $\langle \text{first} \rangle$ or $\langle \text{last} \rangle$ must be provided. We always have $\langle \text{first} \rangle + \langle \text{length} \rangle = \langle \text{last} \rangle + 1$.

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

2.3.2 List specifiers

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

$\langle key \rangle = \{ \langle ref_1 \rangle = \langle spec_1 \rangle, \langle ref_2 \rangle = \langle spec_2 \rangle, \dots, \langle ref_n \rangle = \langle spec_n \rangle \}$

is a convenient shortcut for

$\langle key \rangle . \langle ref_1 \rangle = \langle spec_1 \rangle,$

$\langle key \rangle . \langle ref_2 \rangle = \langle spec_2 \rangle,$

$\dots,$

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

The rules above can apply individually to each line.

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

2.3.3 .n specifiers

$\langle key \rangle . n = \langle value \rangle$ 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 $\langle ?(A.1) \rangle$ 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 $\langle i \rangle$ denotes an integer whereas $\langle first \rangle$, $\langle last \rangle$ and $\langle length \rangle$ stand for integers, or integer valued expressions.

3.1 Simple definitions

$\langle key \rangle = \langle value \rangle$ For an unlimited range

reference	resolution
$\langle key \rangle . 1$	$\langle value \rangle$
$\langle key \rangle . 2$	$\langle value \rangle + 1$
$\langle key \rangle . \langle i \rangle$	$\langle value \rangle + \langle i \rangle - 1$

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

reference	resolution
$\langle key \rangle . 1$	$\langle first \rangle$
$\langle key \rangle . 2$	$\langle first \rangle + 1$
$\langle key \rangle . \langle i \rangle$	$\langle first \rangle + \langle i \rangle - 1$
$\langle key \rangle . \text{previous}$	$\langle first \rangle - 1$

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

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

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

For a range limited from both above and below:

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

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

For example

```

1 \Beanoves {
2   A = 3:8, % or equivalently A = 3::6, A = ::6:8 and A = :8::6
3 }
4 \begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
5 \ttfamily
6 \BeanovesEval[see](A.1)      == 3,
7 \BeanovesEval[see](A.-1)    == 1,
8 \BeanovesEval[see](A.previous) == 2,
9 \BeanovesEval[see](A.last)   == 8,
10 \BeanovesEval[see](A.next)   == 9,
11 \BeanovesEval[see](A.length) == 6,
12 \BeanovesEval[see](A.range)  == 3-8,
13 \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 simple $\langle key \rangle$ *named value reference* is resolved into the position of this value counter. For each frame, this variable is initialized to the first available amongst $\langle value \rangle$, $\langle key \rangle.first$ or $\langle key \rangle.last$. If none is available, an error is raised.

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

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

$\langle key \rangle += \langle integer\ expression \rangle$, resolve $\langle integer\ expression \rangle$ into $\langle integer \rangle$, advance the value counter by $\langle integer \rangle$ and use the new position. Here $\langle integer\ expression \rangle$ is the longest character sequence with no space¹.

$++\langle key \rangle$, advance the value counter for $\langle key \rangle$ by 1 and use the new position.

$\langle key \rangle ++$, use the actual position and advance the value counter for $\langle key \rangle$ by 1.

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

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

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

$\langle key \rangle.++n, ++\langle key \rangle.n$, advance the implicit index counter associate to $\langle key \rangle$ by 1 and use the resolution of $\langle key \rangle.n$,

$\langle key \rangle.n ++$, use the resolution of $\langle key \rangle.n$ and increment the implicit index counter associate to $\langle key \rangle$ by 1.

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

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

3.3 Dotted paths

$\langle key \rangle.\langle i \rangle = \langle spec \rangle$, All the preceding rules are overridden by this particular one and $\langle key \rangle.\langle i \rangle$ resolves to the resolution of $\langle spec \rangle$.

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

```

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

```

¹The parser for algebraic expression is very rudimentary.

Without line 3, A.3 would be evaluated to 5.

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

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

3.4 Frame id

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

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

```
beanoves id  $\langle frame\ id \rangle$ ,
```

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

4 ?(...) query expressions

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

The $\langle queries \rangle$ argument is a comma separated list of individual $\langle query \rangle$'s of next table. Sometimes, using $\langle key \rangle . range$ 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
$\langle start\ expr \rangle$	$\langle start \rangle$	
$\langle start\ expr \rangle :$	$\langle start \rangle -$	no $\langle key \rangle$.range
$\langle start\ expr \rangle : \langle end\ expr \rangle$	$\langle start \rangle - \langle end \rangle$	no $\langle key \rangle$.range
$:: \langle length\ expr \rangle : \langle end\ expr \rangle$	$\langle start \rangle - \langle end \rangle$	no $\langle key \rangle$.range
$: \langle end\ expr \rangle$	$- \langle end \rangle$	no $\langle key \rangle$.range
$:$	$-$	
$\langle start\ expr \rangle ::$	$\langle start \rangle -$	no $\langle key \rangle$.range
$\langle start\ expr \rangle :: \langle length\ expr \rangle$	$\langle start \rangle - \langle end \rangle$	no $\langle key \rangle$.range
$: \langle end\ expr \rangle :: \langle length\ expr \rangle$	$\langle start \rangle - \langle end \rangle$	no $\langle key \rangle$.range
$::$	$-$	

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

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

5 Support

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

6 Implementation

Identify the internal prefix (L^AT_EX3 DocStrip convention, unused).

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

Reserved namespace: identifiers containing the case insensitive string `beanoves` or 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   {2023/01/07}
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 L^AT_EX3. The `c` and `v` argument specifiers take a different meaning when used in a function which name contains with `bnvs` or `BNVS`. Where L^AT_EX3 would transform `l__bnvs_key_t1` into `\l__bnvs_key_t1`, `bnvs` will directly transform `name` into `\l__bnvs_key_t1`. 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, ‘`key`’ will generally stand for `\l__bnvs_key_t1`, 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 `beanoves` specific logic.

\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 >}
	\BNVS_g_prop:c {<global prop core name>}

These are naming functions.

```

8 \cs_new:Npn \BNVS:c #1 { __bnvs_#1 }
9 \cs_new:Npn \BNVS_l:cn #1 #2 { l__bnvs_#1_#2 }
10 \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 twice 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.

```

11 \cs_new:Npn \BNVS_use_raw:N #1 { #1 }
12 \cs_new:Npn \BNVS_use_raw:c #1 {
13   \exp_args:NNo
14   \exp_last_unbraced:No
15   \BNVS_use_raw:N { \use:c { #1 } }
16 }
17 \cs_new:Npn \BNVS_use:c #1 {
18   \BNVS_use_raw:c { \BNVS:c { #1 } }
19 }
20 \cs_new:Npn \BNVS_use_raw:NN #1 #2 {
21   #1 #2
22 }
23 \cs_new:Npn \BNVS_use_raw:nN #1 #2 {
24   #1 #2
25 }
26 \cs_new:Npn \BNVS_use_raw:Nc #1 #2 {
27   \exp_args:NNNo
28   \exp_last_unbraced:NNNo
29   \BNVS_use_raw:NN #1 { \use:c { #2 } }
30 }
31 \exp_args_generate:n { NNno }
32 \cs_new:Npn \BNVS_use_raw:nc #1 #2 {
33   \exp_args:NNno
34   \exp_last_unbraced:Nno
35   \BNVS_use_raw:nN { #1 } { \use:c { #2 } }
36 }
37 \cs_new:Npn \BNVS_use:Nc #1 #2 {
38   \BNVS_use_raw:Nc #1 { \BNVS:c { #2 } }
39 }
40 \cs_new:Npn \BNVS_use:nc #1 #2 {
41   \BNVS_use_raw:nc { #1 } { \BNVS:c { #2 } }
42 }

```

```

\BNVS_new:cpn
\BNVS_set:cpn
\BNVS_use:c

```

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

```

43 \cs_new:Npn \BNVS_log:n #1 { }
44 \cs_generate_variant:Nn \BNVS_log:n { x }
45 \cs_new:Npn \BNVS_DEBUG_on: {
46   \cs_set:Npn \BNVS_DEBUG_log:n { \BNVS_log:n }
47 }
48 \cs_new:Npn \BNVS_DEBUG_off: {
49   \cs_set:Npn \BNVS_DEBUG_log:n { \use_none:n }
50 }
51 \BNVS_DEBUG_off:
52 \cs_new:Npn \BNVS_new:cpn #1 {
53   \cs_new:cpn { \BNVS:c { #1 } }
54 }
55 \cs_new:Npn \BNVS_set:cpn #1 {
56   \cs_set:cpn { \BNVS:c { #1 } }
57 }
58 \cs_generate_variant:Nn \cs_generate_variant:Nn { c }
59 \cs_new:Npn \BNVS_generate_variant:cn #1 {
60   \cs_generate_variant:cn { \BNVS:c { #1 } }
61 }

```

6.3 logging

Utility message.

```

62 \msg_new:nnn { beanoves } { :n } { #1 }
63 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
64 \BNVS_new:cpn { warning:n } {
65   \msg_warning:nnn { beanoves } { :n }
66 }
67 \BNVS_generate_variant:cn { warning:n } { x }
68 \BNVS_new:cpn { error:n } {
69   \msg_error:nnn { beanoves } { :n }
70 }
71 \BNVS_new:cpn { error:x } {
72   \msg_error:nnx { beanoves } { :n }
73 }
74 \BNVS_new:cpn { fatal:n } {
75   \msg_fatal:nnn { beanoves } { :n }
76 }
77 \BNVS_new:cpn { fatal:x } {
78   \msg_fatal:nnx { beanoves } { :n }
79 }

```

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`...

```

80 \cs_new:Npn \BNVS_N_new:c #1 {
81   \cs_new:cpn { BNVS_#1:c } ##1 {
82     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
83   }
84   \cs_new:cpn { BNVS_#1_new:c } ##1 {
85     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
86   }
87   \cs_new:cpn { BNVS_#1_use:c } ##1 {
88     \use:c { \use:c { BNVS_#1:c } { ##1 } }
89   }
90   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
91     \BNVS_use_raw:Nc
92     ##1 { \use:c { BNVS_#1:c } { ##2 } }
93   }
94   \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
95     \BNVS_use_raw:nc
96     { ##1 } { \use:c { BNVS_#1:c } { ##2 } }
97   }
98 }
99 \cs_new:Npn \BNVS_v_new:c #1 {
100   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
101     \BNVS_use_raw:nc
102     { \exp_args:Nv ##1 }
103     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
104   }
105   \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
106     \BNVS_use_raw:nc
107     { \exp_args:NnV \use:n { ##1 } }
108     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
109   }
110 }
111 \BNVS_N_new:c { bool }
112 \BNVS_N_new:c { int }
113 \BNVS_v_new:c { int }
114 \BNVS_N_new:c { tl }
115 \BNVS_v_new:c { tl }
116 \BNVS_N_new:c { str }
117 \BNVS_v_new:c { str }
118 \BNVS_N_new:c { seq }
119 \BNVS_v_new:c { seq }
120 \cs_undefine:N \BNVS_N_new:c
121 \cs_new:Npn \BNVS_use:Ncn #1 #2 #3 {
122   \BNVS_use_raw:c { BNVS_#3_use:Nc } #1 { #2 }
123 }
124 \cs_new:Npn \BNVS_use:ncn #1 #2 #3 {
125   \BNVS_use_raw:c { BNVS_#3_use:nc } { #1 } { #2 }
126 }
127 \cs_new:Npn \BNVS_use:Nvn #1 #2 #3 {
128   \BNVS_use_raw:c { BNVS_#3_use:Nv } #1 { #2 }
129 }
130 \cs_new:Npn \BNVS_use:nvn #1 #2 #3 {
131   \BNVS_use_raw:c { BNVS_#3_use:nv } { #1 } { #2 }
132 }

```

```

133 \cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
134   \BNVS_use:ncn {
135     \BNVS_use:Ncn    #1    { #2 } { #3 }
136   }
137 }
138 \cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
139   \BNVS_use:ncn {
140     \BNVS_use:ncn { #1 } { #2 } { #3 }
141   }
142 }
143 \cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
144   \BNVS_use:ncn {
145     \BNVS_use:Nvn    #1    { #2 } { #3 }
146   }
147 }
148 \cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
149   \BNVS_use:ncn {
150     \BNVS_use:nvn { #1 } { #2 } { #3 }
151   }
152 }
153 \cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
154   \BNVS_use:ncn {
155     \BNVS_use:Ncncn    #1    { #2 } { #3 } { #4 } { #5 }
156   }
157 }
158 \cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
159   \BNVS_use:ncn {
160     \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
161   }
162 }
163 \cs_new:Npn \BNVS_new_c:nc #1 #2 {
164   \BNVS_new:cpn { #1_#2:c } {
165     \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
166   }
167 }
168 \cs_new:Npn \BNVS_new_cn:nc #1 #2 {
169   \BNVS_new:cpn { #1_#2:cn } ##1 {
170     \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
171   }
172 }
173 \cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
174   \BNVS_new:cpn { #2:cnn } ##1 {
175     \BNVS_use:Ncn { #3 } { ##1 } { #1 }
176   }
177 }
178 \cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
179   \BNVS_use_raw:nc {
180     \BNVS_new_cnn:ncN { #1 } { #1_#2 }
181   } { #1_#2:Nnn }
182 }
183 \cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
184   \BNVS_new:cpn { #2:cnv } ##1 ##2 {
185     \BNVS_tl_use:nv {
186       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }

```

```

187     }
188 }
189 }
190 \cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
191   \BNVS_use_raw:nc {
192     \BNVS_new_cnv:ncN { #1 } { #1_#2 }
193   } { #1_#2:Nnn }
194 }
195 \cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
196   \BNVS_new_cpn { #2:cnx } ##1 ##2 {
197     \exp_args:Nnx \use:n {
198       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
199     }
200   }
201 }
202 \cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
203   \BNVS_use_raw:nc {
204     \BNVS_new_cnx:ncN { #1 } { #1_#2 }
205   } { #1_#2:Nnn }
206 }
207 \cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
208   \BNVS_new_cpn { #2:cc } ##1 ##2 {
209     \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
210   }
211 }
212 \cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
213   \BNVS_use_raw:nc {
214     \BNVS_new_cc:ncNn { #1 } { #1_#2 }
215   } { #1_#2:NN }
216 }
217 \cs_new:Npn \BNVS_new_cc:nc #1 #2 {
218   \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
219 }
220 \cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
221   \BNVS_new_cpn { #2:cn } ##1 {
222     \BNVS_use:Ncn #3 { ##1 } { #1 }
223   }
224 }
225 \cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
226   \BNVS_use_raw:nc {
227     \BNVS_new_cn:ncNn { #1 } { #1_#2 }
228   } { #1_#2:Nn }
229 }
230 \cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
231   \BNVS_new_cpn { #2:cv } ##1 ##2 {
232     \BNVS_use:nvn {
233       \BNVS_use:Ncn #3 { ##1 } { #1 }
234     } { ##2 } { #4 }
235   }
236 }
237 \cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
238   \BNVS_use_raw:nc {
239     \BNVS_new_cv:ncNn { #1 } { #1_#2 }
240   } { #1_#2:Nn }

```

```

241 }
242 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
243   \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
244 }
245 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
246   \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
247 }
248 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
249   \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
250 }
251 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
252   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
253 }
254 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
255   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
256 }
257 \cs_new:Npn \BNVS_g_prop_use:Nc #1 #2 {
258   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { prop } }
259 }
260 \cs_new:Npn \BNVS_g_prop_use:nc #1 #2 {
261   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { prop } }
262 }
263 \cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
264   \BNVS_use:ncncncn { \exp_args:NVVV #1 }
265   { #2 } { t1 } { #3 } { t1 } { #4 } { t1 }
266 }
267 \cs_generate_variant:Nn \prg_new_conditional:Npnn { c }
268 \cs_new:Npn \BNVS_new_conditional:cpnn #1 {
269   \prg_new_conditional:cpnn { \BNVS:c { #1 } }
270 }
271 \cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
272 \cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
273   \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
274 }
275 \cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
276   \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
277     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
278       \prg_return_true:
279     } {
280       \prg_return_false:
281     }
282   }
283 }
284 \cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
285   \BNVS_use:nc {
286     \BNVS_new_conditional_vn:cNnn { #1 }
287   } { #1:nn TF } { #2 }
288 }
289 \cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
290   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
291     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
292       \prg_return_true:
293     } {

```

```

294     \prg_return_false:
295   }
296 }
297 }
298 \cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
299   \BNVS_use:nc {
300     \BNVS_new_conditional_vc:cNn { #1 }
301   } { #1:ncTF }
302 }
303 \cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
304   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
305     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
306       \prg_return_true:
307     } {
308       \prg_return_false:
309     }
310   }
311 }
312 \cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
313   \BNVS_use:nc {
314     \BNVS_new_conditional_vc:cNn { #1 }
315   } { #1:ncTF }
316 }

```

6.4.1 Regex

```

317 \cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
318   \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
319 }

```

<pre> __bnvs_match_once:NnTF __bnvs_match_once:NvTF __bnvs_match_once:nnTF __bnvs_regex_split:cnTF </pre>	<pre> __bnvs_match_once:NnTF <regex variable> {<expression>} {<yes code>} {< >}no code __bnvs_match_once:nnTF {<regex>} {<expression>} {<yes code>} {< >}no code __bnvs_regex_split:cnTF <regex core> {<expression>} <seq core> {<yes code>} {< >}no code __bnvs_regex_split:cnTF <regex core> {<expression>} {<yes code>} {< >}no code </pre>
---------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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

```

320 \BNVS_new_conditional:cpnn { match_once:Nn } #1 #2 { T, F, TF } {
321   \BNVS_use:ncn {
322     \regex_extract_once:NnNTF #1 { #2 }
323   } { match } { seq } {
324     \prg_return_true:
325   } {
326     \prg_return_false:
327   }
328 }

```



```

329 \BNVS_new_conditional:cpnn { match_once:Nv } #1 #2 { T, F, TF } {
330   \BNVS_seq_use:nc {
331     \BNVS_tl_use:nv {
332       \regex_extract_once:NnNTF #1
333     } { #2 }
334   } { match } {
335     \prg_return_true:
336   } {
337     \prg_return_false:
338   }
339 }
340 \BNVS_new_conditional:cpnn { match_once:nn } #1 #2 { T, F, TF } {
341   \BNVS_seq_use:nc {
342     \regex_extract_once:nnNTF { #1 } { #2 }
343   } { match } {
344     \prg_return_true:
345   } {
346     \prg_return_false:
347   }
348 }
349 \BNVS_new_conditional:cpnn { regex_split:cnc } #1 #2 #3 { T, F, TF } {
350   \BNVS_seq_use:nc {
351     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
352   } { #3 } {
353     \prg_return_true:
354   } {
355     \prg_return_false:
356   }
357 }
358 \BNVS_new_conditional:cpnn { regex_split:cn } #1 #2 { T, F, TF } {
359   \BNVS_seq_use:nc {
360     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
361   } { split } {
362     \prg_return_true:
363   } {
364     \prg_return_false:
365   }
366 }

```

6.4.2 Token lists

<code>__bnvs_tl_clear:c</code>	<code>__bnvs_tl_clear:c {\core key_t l}</code>
<code>__bnvs_tl_use:c</code>	<code>__bnvs_tl_use:c {\core}</code>
<code>__bnvs_tl_set_eq:cc</code>	<code>__bnvs_tl_count:c {\core}</code>
<code>__bnvs_tl_set:cn</code>	<code>__bnvs_tl_set_eq:cc {\lhs core name} {\rhs core name}</code>
<code>__bnvs_tl_set:(cv cx)</code>	<code>__bnvs_tl_set:cn {\core} {\tl}</code>
<code>__bnvs_tl_put_left:cn</code>	<code>__bnvs_tl_set:cv {\core} {\value core name}</code>
<code>__bnvs_tl_put_right:cn</code>	<code>__bnvs_tl_put_left:cn {\core} {\tl}</code>
<code>__bnvs_tl_put_right:(cx cv)</code>	<code>__bnvs_tl_put_right:cn {\core} {\tl}</code>
	<code>__bnvs_tl_put_right:cv {\core} {\value core name}</code>

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}`

```

367 \cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
368   \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
369     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
370       \prg_return_true:
371     } {
372       \prg_return_false:
373     }
374   }
375 }
376 \cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
377   \BNVS_use:nc {
378     \BNVS_new_conditional_vnc:cNn { #1 }
379   } { #1:nncTF }
380 }
381 \cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
382   \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
383     \BNVS_tl_use:nv {
384       \BNVS_tl_use:Nv #2 { ##1 }
385     } { ##2 } { ##3 } { ##4 } {
386       \prg_return_true:
387     } {
388       \prg_return_false:
389     }
390   }
391 }
```

```

392 \cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
393   \BNVS_use:nc {
394     \BNVS_new_conditional_vvnc:cNn { #1 }
395   } { #1:nncTF }
396 }
397 \cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
398   \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
399     \BNVS_tl_use:nv {
400       \BNVS_tl_use:nv {
401         \BNVS_tl_use:Nv #2 { ##1 }
402       } { ##2 }
403     } { ##3 } { ##4 } {
404       \prg_return_true:
405     } {
406       \prg_return_false:
407     }
408   }
409 }
410 \cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
411   \BNVS_use:nc {
412     \BNVS_new_conditional_vvvc:cNn { #1 }
413   } { #1:nncTF }
414 }
415 \cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
416   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
417     \BNVS_tl_use:nv {
418       \BNVS_tl_use:Nv #2 { ##1 }
419     } { ##2 } { ##3 } {
420       \prg_return_true:
421     } {
422       \prg_return_false:
423     }
424   }
425 }
426 \cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
427   \BNVS_use:nc {
428     \BNVS_new_conditional_vvc:cNn { #1 }
429   } { #1:nncTF }
430 }
431 \cs_new:Npn \BNVS_new_tl_c:c {
432   \BNVS_new_c:nc { tl }
433 }
434 \BNVS_new_tl_c:c { clear }
435 \BNVS_new_tl_c:c { use }
436 \BNVS_new_tl_c:c { count }
437
438 \BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
439   \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
440 }
441 \cs_new:Npn \BNVS_new_tl_cn:c {
442   \BNVS_new_cn:nc { tl }
443 }
444 \cs_new:Npn \BNVS_new_tl_cv:c #1 {
445   \BNVS_new_cv:ncn { tl } { #1 } { tl }

```

```

446 }
447 \BNVS_new_tl_cn:c { set }
448 \BNVS_new_tl_cv:c { set }
449 \BNVS_new:cpn { tl_set:cx } {
450   \exp_args:Nnx \__bnvs_tl_set:cn
451 }
452 \BNVS_new_tl_cn:c { put_right }
453 \BNVS_new_tl_cv:c { put_right }
454 % \BNVS_generate_variant:cn { tl_put_right:cn } { cx }
455 \BNVS_new:cpn { tl_put_right:cx } {
456   \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
457 }
458 \BNVS_new_tl_cn:c { put_left }
459 \BNVS_new_tl_cv:c { put_left }
460 % \BNVS_generate_variant:cn { tl_put_left:cn } { cx }
461 \BNVS_new:cpn { tl_put_left:cx } {
462   \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
463 }

```

$\backslash_bnvs_tl_if_empty:c\overline{TF}$ $\backslash_bnvs_tl_if_blank:v\overline{TF}$ $\backslash_bnvs_tl_if_eq:cn\overline{TF}$	$\backslash_bnvs_tl_if_empty:cTF \{ \langle core \rangle \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$ $\backslash_bnvs_tl_if_blank:vTF \{ \langle core \rangle \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$ $\backslash_bnvs_tl_if_eq:cnTF \{ \langle core \rangle \} \{ \langle tl \rangle \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$
--------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

These are shortcuts to

- $\backslash tl_if_empty:cTF \{ l_bnvs_ \langle core \rangle_tl \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$
- $\backslash tl_if_eq:cnTF \{ l_bnvs_ \langle core \rangle_tl \} \{ \langle tl \rangle \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$

```

464 \cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
465   \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
466     \BNVS_use:Ncn #3 { ##1 } { #1 } {
467       \prg_return_true:
468     } {
469       \prg_return_false:
470     }
471   }
472 }
473 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
474   \BNVS_use_raw:nc {
475     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
476   } { #1_#2:NTF }
477 }
478 \BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }
479 \BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
480   \BNVS_tl_use:Nv \tl_if_blank:NTF { #1 } {
481     \prg_return_true:
482   } {
483     \prg_return_false:
484   }
485 }
486 \cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
487   \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
488     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {

```

```

489     \prg_return_true:
490   } {
491     \prg_return_false:
492   }
493 }
494 }
495 \cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
496   \BNVS_use_raw:nc {
497     \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
498   } { #1_#2:NnTF }
499 }
500 \BNVS_new_conditional_cn:ncn { t1 } { if_eq } { T, F, TF }
501 \cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
502   \BNVS_new_conditional_cpnn { #2:cv } ##1 ##2 { #4 } {
503     \BNVS_use:nvn {
504       \BNVS_use:Ncn #3 { ##1 } { #1 }
505     } { ##2 } { #1 } {
506       \prg_return_true:
507     } {
508       \prg_return_false:
509     }
510   }
511 }
512 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
513   \BNVS_use_raw:nc {
514     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
515   } { #1_#2:NnTF }
516 }
517 \BNVS_new_conditional_cv:ncn { t1 } { if_eq } { T, F, TF }

```

6.4.3 Strings

_bnvs_str_if_eq:vnTF _bnvs_str_if_eq:vnTF {<core>} {<t1>} {<yes code>} {<no code>}

These are shortcuts to

- \t1_if_empty:ctf {l_bnvs_<core>t1}{<yes code>} {<no code>}

```

518 \cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
519   \BNVS_new_conditional_cpnn { #2:vn } ##1 ##2 { #4 } {
520     \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
521       \prg_return_true:
522     } {
523       \prg_return_false:
524     }
525   }
526 }
527 \cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
528   \BNVS_use_raw:nc {
529     \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
530   } { #1_#2:nnTF }
531 }
532 \BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }

```

```

533 \cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
534   \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
535     \BNVS_use:nvn {
536       \BNVS_use:Nvn #3 { ##1 } { #1 }
537     } { ##2 } { #1 } {
538       \prg_return_true:
539     } {
540       \prg_return_false:
541     }
542   }
543 }
544 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
545   \BNVS_use_raw:nc {
546     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
547   } { #1_#2:nnTF }
548 }
549 \BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }

```

6.4.4 Sequences

<hr/>	
__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	
<hr/>	

These are shortcuts to

- \seq_set_eq:cc {l__bnvs_<core₁>_seq} {l__bnvs_<core₂>_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>}

```

550 \BNVS_new_c:nc { seq } { count }
551 \BNVS_new_c:nc { seq } { clear }
552 \BNVS_new_cn:nc { seq } { use }

```

```

553 \BNVS_new_cn:nc { seq } { item }
554 \BNVS_new_cn:nc { seq } { remove_all }
555 \BNVS_new_cn:nc { seq } { map_inline }
556 \BNVS_new_cc:nc { seq } { set_eq }
557 \BNVS_new_cv:ncn { seq } { put_left } { tl }
558 \BNVS_new_cn:ncn { seq } { put_right } { tl }
559 \BNVS_new_cv:ncn { seq } { put_right } { tl }
560 \BNVS_new_cnn:nc { seq } { set_split }
561 \BNVS_new_cnv:nc { seq } { set_split }
562 \BNVS_new_cnx:nc { seq } { set_split }
563 \BNVS_new_cc:ncn { seq } { pop_left } { tl }
564 \BNVS_new_cc:ncn { seq } { pop_right } { tl }

```

<pre> \underline{_bnvs_seq_if_empty:cTF} \underline{_bnvs_seq_get_right:ccTF} \underline{_bnvs_seq_pop_left:ccTF} \underline{_bnvs_seq_pop_right:ccTF} </pre>	<pre> __bnvs_seq_if_empty:cTF {<seq core name>} {<yes code>} {<no code>} __bnvs_seq_get_right:ccTF {<seq core name>} {<tl core name>} {<yes code>} {<no code>} </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

565 \cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
566   \BNVS_new_conditional:cpnn { #1_#2:cc } ##1 ##2 { #4 } {
567     \BNVS_use:ncncn {
568       \BNVS_use_raw:c { #1_#2:NNTF }
569     } { ##1 } { #1 } { ##2 } { #3 } {
570       \prg_return_true:
571     } {
572       \prg_return_false:
573     }
574   }
575 }
576 \BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
577 \BNVS_new_conditional_cc:ncnn
578 { seq } { get_right } { tl } { T, F, TF }
579 \BNVS_new_conditional_cc:ncnn
580 { seq } { pop_left } { tl } { T, F, TF }
581 \BNVS_new_conditional_cc:ncnn
582 { seq } { pop_right } { tl } { T, F, TF }

```

6.4.5 Integers

<code>__bnvs_int_new:c</code>	<code>__bnvs_int_new:c</code>	<code>{\core}</code>
<code>__bnvs_int_use:c</code>	<code>__bnvs_int_use:c</code>	<code>{\core}</code>
<code>__bnvs_int_inc:c</code>	<code>__bnvs_int_incr:c</code>	<code>{\core}</code>
<code>__bnvs_int_decr:c</code>	<code>__bnvs_int_decr:c</code>	<code>{\core}</code>
<code>__bnvs_int_set:cn</code>	<code>__bnvs_int_set:cn</code>	<code>{\core} {\value}</code>
<code>__bnvs_int_set:cv</code>		

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_idocr:c` `{l__bnvs_<core>_int}`
- `\int_set:cn` `{l__bnvs_<core>_int} <value>`

```

583 \BNVS_new_c:nc { int } { new }
584 \BNVS_new_c:nc { int } { use }
585 \BNVS_new_c:nc { int } { zero }
586 \BNVS_new_c:nc { int } { incr }
587 \BNVS_new_c:nc { int } { decr }
588 \BNVS_new_cn:nc { int } { set }
589 \BNVS_new_cv:ncn { int } { set } { int }

```

6.4.6 Prop

`__bnvs_prop_get:NncTF`

```

590 \BNVS_new_conditional:cpnn { prop_get:Nnc } #1 #2 #3 { T, F, TF } {
591   \BNVS_use:ncn {
592     \prop_get:NnNTF #1 { #2 }
593   } { #3 } { t1 } {
594     \prg_return_true:
595   } {
596     \prg_return_false:
597   }
598 }

```

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.

```
599 \tl_new:N \l__bnvs_id_last_tl
600 \tl_set:Nn \l__bnvs_id_last_tl { ?! }
601 \tl_new:N \l__bnvs_a_tl
602 \tl_new:N \l__bnvs_b_tl
603 \tl_new:N \l__bnvs_c_tl
604 \tl_new:N \l__bnvs_V_tl
605 \tl_new:N \l__bnvs_A_tl
606 \tl_new:N \l__bnvs_L_tl
607 \tl_new:N \l__bnvs_Z_tl
608 \tl_new:N \l__bnvs_ans_tl
609 \tl_new:N \l__bnvs_key_tl
610 \tl_new:N \l__bnvs_key_base_tl
611 \tl_new:N \l__bnvs_id_tl
612 \tl_new:N \l__bnvs_n_tl
613 \tl_new:N \l__bnvs_path_tl
614 \tl_new:N \l__bnvs_group_tl
615 \tl_new:N \l__bnvs_scan_tl
616 \tl_new:N \l__bnvs_query_tl
617 \tl_new:N \l__bnvs_token_tl
618 \tl_new:N \l__bnvs_root_tl
619 \tl_new:N \l__bnvs_n_incr_tl
620 \tl_new:N \l__bnvs_incr_tl
621 \tl_new:N \l__bnvs_post_tl
622 \tl_new:N \l__bnvs_suffix_tl
623 \int_new:N \g__bnvs_call_int
624 \int_new:N \l__bnvs_int
625 \seq_new:N \g__bnvs_def_seq
626 \seq_new:N \l__bnvs_a_seq
627 \seq_new:N \l__bnvs_b_seq
628 \seq_new:N \l__bnvs_ans_seq
629 \seq_new:N \l__bnvs_match_seq
630 \seq_new:N \l__bnvs_split_seq
631 \seq_new:N \l__bnvs_path_seq
632 \seq_new:N \l__bnvs_path_base_seq
633 \seq_new:N \l__bnvs_query_seq
634 \seq_new:N \l__bnvs_token_seq
635 \bool_new:N \l__bnvs_in_frame_bool
636 \bool_set_false:N \l__bnvs_in_frame_bool
637 \bool_new:N \l__bnvs_parse_bool
```

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

```

638 \bool_new:N \l__bnvs_provide_bool
639 \BNVS_new:cpn { provide_on: } {
640   \bool_set_true:N \l__bnvs_provide_bool
641 }
642 \BNVS_new:cpn { provide_off: } {
643   \bool_set_false:N \l__bnvs_provide_bool
644 }
645 \__bnvs_provide_off:

```

__bnvs_if_provide:TF

__bnvs_if_provide:TF {*<yes code>*} {*<no code>*}

Execute *<yes code>* when in provide mode, *<no code>* otherwise.

```

646 \BNVS_new_conditional:cpnn { if_provide: } { p, T, F, TF } {
647   \bool_if:NTF \l__bnvs_provide_bool {
648     \prg_return_true:
649   } {
650     \prg_return_false:
651   }
652 }

```

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 definition for \g__bnvs_call_int.)

```

653 \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.

```

654 \cs_set:Npn \__bnvs_call_greset: {
655   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
656 }

```

__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.

```

657 \BNVS_new_conditional:cpnn { call: } { T, F, TF } {
658   \int_gdecr:N \g__bnvs_call_int
659   \int_compare:nNnTF \g__bnvs_call_int > 0 {
660     \prg_return_true:
661   } {
662     \prg_return_false:
663   }
664 }

```

6.11 Overlay specification

6.12 Basic functions

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

$\langle id \rangle! \langle key \rangle / V$ for the value

$\langle id \rangle! \langle key \rangle / A$ for the first index

$\langle id \rangle! \langle key \rangle / L$ for the length when provided

$\langle id \rangle! \langle key \rangle / Z$ for the last index when provided

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

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

(End definition for `\g__bnvs_prop`.)

<code>__bnvs_gput:nnn</code>	<code>__bnvs_gput:nnn {<subkey>} {<key>} {<value>}</code>
<code>__bnvs_gput:nnn</code>	<code>__bnvs_item:nn {<subkey>} {<key>}</code>
<code>__bnvs_item:nn</code>	<code>__bnvs_gremove:nn {<subkey>} {<key>}</code>
<code>__bnvs_gremove:nn</code>	<code>__bnvs_gclear:n {<key>}</code>
<code>__bnvs_gclear:n</code>	<code>__bnvs_gclear:</code>
<code>__bnvs_gclear:v</code>	
<code>__bnvs_gclear:</code>	

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L^AT_EX3 eponym functions. The key argument is $\langle key \rangle / \langle subkey \rangle$.

```
666 \BNVS_new:cpn { gput:nnn } #1 #2 {
667   \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
668 }
669 \BNVS_new:cpn { gput:nnv } #1 #2 {
670   \BNVS_tl_use:nv {
671     \__bnvs_gput:nnn { #1 } { #2 }
672   }
673 }
674 \BNVS_new:cpn { item:nn } #1 #2 {
675   \prop_item:Nn \g__bnvs_prop { #2 / #1 }
676 }
677 \BNVS_new:cpn { gremove:nn } #1 #2 {
678   \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
679 }
680 \BNVS_new:cpn { gclear:n } #1 {
681   \clist_map_inline:nn { V, A, Z, L } {
682     \__bnvs_gremove:nn { ##1 } { #1 }
683   }
684   \__bnvs_cache_gclear:n { #1 }
685 }
686 \BNVS_new:cpn { gclear: } {
687   \prop_gclear:N \g__bnvs_prop
688 }
689 \BNVS_generate_variant:cn { gclear:n } { V }
690 \BNVS_new:cpn { gclear:v } {
691   \BNVS_tl_use:Nc \__bnvs_gclear:V
692 }
```

```

693 \__bnvs_if_in_p:nn * \__bnvs_if_in_p:nn {<subkey>} {<key>}
694 \__bnvs_if_in:nnTF * \__bnvs_if_in:nnTF {<subkey>} {<key>} {<yes code>} {<no code>}
695 \__bnvs_if_in_p:n * \__bnvs_if_in_p:n {<key>}
696 \__bnvs_if_in:nTF * \__bnvs_if_in:nTF {<key>} {<yes code>} {<no code>}

```

Convenient shortcuts to test for the existence of *<key>*/*<subkey>*, it makes the code more concise and readable. The version with no *<subkey>* is the or combination for keys V, A and Z.

```

693 \BNVS_new_conditional:cpnn { if_in:nn } #1 #2 { p, T, F, TF } {
694   \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
695     \prg_return_true:
696   } {
697     \prg_return_false:
698   }
699 }
700 \BNVS_new_conditional:cpnn { if_in:n } #1 { p, T, F, TF } {
701   \bool_if:nTF {
702     \__bnvs_if_in_p:nn V { #1 }
703     || \__bnvs_if_in_p:nn A { #1 }
704     || \__bnvs_if_in_p:nn Z { #1 }
705   } {
706     \prg_return_true:
707   } {
708     \prg_return_false:
709   }
710 }
711 \BNVS_new_conditional:cpnn { if_in:v } #1 { p, T, F, TF } {
712   \BNVS_tl_use:Nv \__bnvs_if_in:nTF { #1 }
713   { \prg_return_true: } { \prg_return_false: }
714 }

```

```

\__bnvs_gprovide:nnnT \__bnvs_gprovide:nnnT {<subkey>} {<key>} {<value>} {<true precode>}

```

Execute *<true precode>* before providing, or *<false precode>* before not providing.

```

715 \BNVS_new:cpn { gprovide:nnnT } #1 #2 #3 #4 {
716   \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
717     #4
718   \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
719 }
720 }

```

```

\__bnvs_get:nncTF \__bnvs_get:nncTF {<subkey>} {<key>} {<tl core name>} {<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 variable>* is undefined, on resolution only. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```

721 \BNVS_new_conditional:cpnn { get:nnc } #1 #2 #3 { T, F, TF } {
722   \BNVS_tl_use:nc {
723     \prop_get:NnNTF \g__bnvs_prop { #2 / #1 }
724   } { #3 } {

```

```

725     \prg_return_true:
726   } {
727     \prg_return_false:
728   }
729 }
730 \BNVS_new_conditional:cpnn { get:nvc } #1 #2 #3 { T, F, TF } {
731   \BNVS_tl_use:nv {
732     \__bnvs_get:nncTF { #1 }
733   } { #2 } { #3 } {
734     \prg_return_true:
735   } {
736     \prg_return_false:
737   }
738 }

```

6.13 Functions with cache

`\g__bnvs_prop` $\langle key \rangle$ – $\langle value \rangle$ 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.

$\langle id \rangle! \langle key \rangle/V$ for the cached static value of the value

$\langle id \rangle! \langle key \rangle/A$ for the cached static value of the first index

$\langle id \rangle! \langle key \rangle/L$ for the cached static value of the length

$\langle id \rangle! \langle key \rangle/Z$ for the cached static value of the last index

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

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

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

```

739 \prop_new:N \g__bnvs_cache_prop

```

(End definition for `\g__bnvs_prop`.)

<hr/>	<hr/>
<code>__bnvs_cache_gput:nnn</code>	<code>__bnvs_cache_gput:nnn {$\langle subkey \rangle$} {$\langle key \rangle$} {$\langle value \rangle$}</code>
<code>__bnvs_cache_gput:(nnv nvn)</code>	<code>__bnvs_cache_item:nn {$\langle subkey \rangle$} {$\langle key \rangle$}</code>
<code>__bnvs_cache_item:nn</code>	<code>__bnvs_cache_gremove:nn {$\langle subkey \rangle$} {$\langle key \rangle$}</code>
<code>__bnvs_cache_gremove:nn</code>	<code>__bnvs_cache_gclear:n {$\langle key \rangle$}</code>
<code>__bnvs_cache_gclear:n</code>	<code>__bnvs_cache_gclear:</code>
<code>__bnvs_cache_gclear:</code>	
<hr/>	<hr/>

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

```

740 \BNVS_new:cpn { cache_gput:nnn } #1 #2 {
741   \prop_gput:Nnn \g__bnvs_cache_prop { #2 / #1 }
742 }

```

```

743 \cs_generate_variant:Nn \__bnvs_cache_gput:nnn { nV, nnV }
744 \BNVS_new:cpn { cache_gput:nvn } #1 {
745   \BNVS_tl_use:nc {
746     \__bnvs_cache_gput:nVn { #1 }
747   }
748 }
749 \BNVS_new:cpn { cache_gput:nnv } #1 #2 {
750   \BNVS_tl_use:nc {
751     \__bnvs_cache_gput:nnV { #1 } { #2 }
752   }
753 }
754 \BNVS_new:cpn { cache_item:nn } #1 #2 {
755   \prop_item:Nn \g__bnvs_cache_prop { #2 / #1 }
756 }
757 \BNVS_new:cpn { cache_gremove:nn } #1 #2 {
758   \prop_gremove:Nn \g__bnvs_cache_prop { #2 / #1 }
759 }
760 \BNVS_new:cpn { cache_gclear:n } #1 {
761   \clist_map_inline:nn { V, A, Z, L, P, N } {
762     \prop_gremove:Nn \g__bnvs_cache_prop { #1 / ##1 }
763   }
764 }
765 \BNVS_new:cpn { cache_gclear: } {
766   \prop_gclear:N \g__bnvs_cache_prop
767 }

```

$\backslash_bnvs_cache_if_in_p:nn$ ★ $\backslash_bnvs_cache_if_in:nnTF$ ★	$\backslash_bnvs_cache_if_in_p:n$ {<subkey>} {<key>} $\backslash_bnvs_cache_if_in:nTF$ {<subkey>} {<key>} {<yes code>} {<no code>}
---------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------

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

```

768 \prg_new_conditional:Npnn \__bnvs_cache_if_in:nn #1 #2 { p, T, F, TF } {
769   \prop_if_in:NnTF \g__bnvs_cache_prop { #2 / #1 } {
770     \prg_return_true:
771   } {
772     \prg_return_false:
773   }
774 }

```

$\backslash_bnvs_cache_get:nncTF$	$\backslash_bnvs_cache_get:nncTF$ {<subkey>} {<key>} {<tl core name>} {<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 variable> is undefined. NB: the predicate won't work because $\backslash\prop_get:NnNTF$ is not expandable.

```

775 \BNVS_new_conditional:cpnn { cache_get:nnc } #1 #2 #3 { p, T, F, TF } {
776   \BNVS_tl_use:nc {
777     \prop_get:NnNTF \g__bnvs_cache_prop { #2 / #1 }
778   } { #3 } {
779     \prg_return_true:
780   } {

```

```

781     \prg_return_false:
782   }
783 }

```

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` $\langle key \rangle$ - $\langle value \rangle$ property list to store the contents or the named value counters. The keys are $\langle id \rangle!$ $\langle key \rangle$.

```

784 \prop_new:N \g__bnvs_v_prop

```

(End definition for `\g__bnvs_v_prop`.)

<pre> __bnvs_v_gput:nn __bnvs_v_gput:(nV Vn) __bnvs_v_item:n __bnvs_v_gremove:n __bnvs_v_gclear: </pre>	<pre> __bnvs_v_gput:nn {\langle key \rangle} {\langle value \rangle} __bnvs_v_item:n {\langle key \rangle} __bnvs_v_gremove:n {\langle key \rangle} __bnvs_v_gclear: </pre>
--------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L^AT_EX3 eponym functions.

```

785 \BNVS_new:cpn { v_gput:nn } {
786   \prop_gput:Nnn \g__bnvs_v_prop
787 }
788 \BNVS_new:cpn { v_gput:nv } #1 {
789   \BNVS_tl_use:nv {
790     \__bnvs_v_gput:nn { #1 }
791   }
792 }
793 \BNVS_new:cpn { v_item:n } #1 {
794   \prop_item:Nn \g__bnvs_v_prop { #1 }
795 }
796 \BNVS_new:cpn { v_gremove:n } {
797   \prop_gremove:Nn \g__bnvs_v_prop
798 }
799 \BNVS_new:cpn { v_gclear: } {
800   \prop_gclear:N \g__bnvs_v_prop
801 }

```

<pre> __bnvs_v_if_in_p:n ★ __bnvs_v_if_in:nTF ★ </pre>	<pre> __bnvs_v_if_in_p:n {\langle key \rangle} __bnvs_v_if_in:nTF {\langle key \rangle} {\langle yes code \rangle} {\langle no code \rangle} </pre>
----------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

Convenient shortcuts to test for the existence of the $\langle key \rangle$ value counter.

```

802 \BNVS_new_conditional:cpnn { v_if_in:n } #1 { p, T, F, TF } {
803   \prop_if_in:NnTF \g__bnvs_v_prop { #1 } {
804     \prg_return_true:
805   } {
806     \prg_return_false:
807   }
808 }

```

<u><code>_bnvs_v_get:ncTF</code></u>	<code>_bnvs_v_get:ncTF {<key>} <tl core name> {<yes code>} {<no code>}</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 variable>* is undefined. NB: the predicate won't work because `\prop_get:NnNTF` is not expandable.

```

809 \BNVS_new_conditional:cpnn { v_get:nc } #1 #2 { T, F, TF } {
810   \BNVS_tl_use:nc {
811     \prop_get:NnNTF \g__bnvs_v_prop { #1 }
812   } { #2 } {
813     \prg_return_true:
814   } {
815     \prg_return_false:
816   }
817 }
```

<u><code>_bnvs_v_greset:nnTF</code></u>	<code>_bnvs_v_greset:nnTF {<key>} {<initial value>} {<true code>} {<false code>}</code>
<u><code>_bnvs_v_greset:vnTF</code></u>	<code>_bnvs_v_greset:vnTF {<key>} {<initial value>} {<true code>} {<false code>}</code>
<u><code>_bnvs_greset_all:nn</code></u>	<code>_bnvs_greset_all:nnTF {<key>} {<initial value>} {<true code>} {<false code>}</code>

The key must include the frame id. Reset the value counter to the given *<initial value>*. The *_all* version also cleans the cached values. If the *<key>* is known, *<true code>* is executed, otherwise *<false code>* is executed.

```

818 \BNVS_new_conditional:cpnn { v_greset:nn } #1 #2 { T, F, TF } {
819   \_bnvs_v_if_in:nTF { #1 } {
820     \_bnvs_v_gremove:n { #1 }
821     \tl_if_empty:nF { #2 } {
822       \_bnvs_v_gput:nn { #1 } { #2 }
823     }
824     \prg_return_true:
825   } {
826     \prg_return_false:
827   }
828 }
829 \BNVS_new_conditional:cpnn { v_greset:vn } #1 #2 { T, F, TF } {
830   \BNVS_tl_use:Nv \_bnvs_v_greset:nnTF { #1 } { #2 }
831   { \prg_return_true: } { \prg_return_false: }
832 }
833 \BNVS_new_conditional:cpnn { greset_all:nn } #1 #2 { T, F, TF } {
834   \_bnvs_if_in:nTF { #1 } {
835     \BNVS_begin:
836     \clist_map_inline:nn { V, A, Z, L } {
837       \_bnvs_get:nncT { ##1 } { #1 } { a } {
838         \_bnvs_quark_if_nil:cT { a } {
839           \_bnvs_cache_get:nncTF { ##1 } { #1 } { a } {
840             \_bnvs_gput:nnv { ##1 } { #1 } { a }
841           } {
842             \_bnvs_gput:nnn { ##1 } { #1 } { 1 }
843           }
844         }
845       }
846     }
```



```

847 \BNVS_end:
848 \__bnvs_cache_gclear:n { #1 }
849 \__bnvs_v_greset:nnT { #1 } { #2 } {}
850 \prg_return_true:
851 } {
852 \prg_return_false:
853 }
854 }
855 \BNVS_new_conditional:cpnn { greset_all:vn } #1 #2 { T, F, TF } {
856 \BNVS_tl_use:Nv \__bnvs_greset_all:nnTF { #1 } { #2 }
857 { \prg_return_true: } { \prg_return_false: }
858 }

```

<code>__bnvs_gclear_all:n</code>	<code>__bnvs_gclear_all:n {<key>}</code>
<code>__bnvs_gclear_all:</code>	<code>__bnvs_gclear_all:</code>

Convenient shortcuts to clear all the storage, for the given key in the first case.

```

859 \BNVS_new:cpn { gclear_all: } {
860 \__bnvs_gclear:
861 \__bnvs_cache_gclear:
862 \__bnvs_n_gclear:
863 \__bnvs_v_gclear:
864 }
865 \BNVS_new:cpn { gclear_all:n } #1 {
866 \__bnvs_gclear:n { #1 }
867 \__bnvs_cache_gclear:n { #1 }
868 \__bnvs_n_gremove:n { #1 }
869 \__bnvs_v_gremove:n { #1 }
870 }

```

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` $\langle key \rangle$ – $\langle value \rangle$ property list to store the contents of the named index counters. The keys are $\langle id \rangle!$ $\langle key \rangle$.

```

871 \prop_new:N \g__bnvs_n_prop

```

(End definition for `\g__bnvs_n_prop`.)

<code>__bnvs_n_gput:nn</code>	<code>__bnvs_n_gput:nn {<key>} {<value>}</code>
<code>__bnvs_n_gput:(nv vn)</code>	<code>__bnvs_n_item:n {<key>}</code>
<code>__bnvs_n_gprovide:nn</code>	<code>__bnvs_n_gremove:n {<key>}</code>
<code>__bnvs_n_item:n</code>	<code>__bnvs_n_gclear:</code>
<code>__bnvs_n_gremove:n</code>	
<code>__bnvs_n_gremove:v</code>	
<code>__bnvs_n_gclear:</code>	

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L^AT_EX3 eponym functions.

```

872 \BNVS_new:cpn { n_gput:nn } {
873 \prop_gput:Nnn \g__bnvs_n_prop
874 }

```

```

875 \cs_generate_variant:Nn \__bnvs_n_gput:nn { nV }
876 \BNVS_new:cpn { n_gput:nv } #1 {
877   \BNVS_tl_use:nc {
878     \__bnvs_n_gput:nV { #1 }
879   }
880 }
881 \BNVS_new:cpn { n_gprovide:nn } #1 #2 {
882   \prop_if_in:NnF \g__bnvs_n_prop { #1 } {
883     \prop_gput:Nnn \g__bnvs_n_prop { #1 } { #2 }
884   }
885 }
886 \BNVS_new:cpn { n_item:n } #1 {
887   \prop_item:Nn \g__bnvs_n_prop { #1 }
888 }
889 \BNVS_new:cpn { n_gremove:n } {
890   \prop_gremove:Nn \g__bnvs_n_prop
891 }
892 \BNVS_generate_variant:cn { n_gremove:n } { V }
893 \BNVS_new:cpn { n_gremove:v } {
894   \BNVS_tl_use:nc {
895     \__bnvs_n_gremove:V
896   }
897 }
898 \BNVS_new:cpn { n_gclear: } {
899   \prop_gclear:N \g__bnvs_n_prop
900 }
901 \cs_generate_variant:Nn \__bnvs_n_gremove:n { V }

```

```

\__bnvs_n_if_in_p:n ★ \__bnvs_n_if_in_p:nn {<key>}
\__bnvs_n_if_in:nTF ★ \__bnvs_n_if_in:nTF {<key>} {<yes code>} {<no code>}

```

Convenient shortcuts to test for the existence of the $\langle key \rangle$ value counter.

```

902 \prg_new_conditional:Npnn \__bnvs_n_if_in:n #1 { p, T, F, TF } {
903   \prop_if_in:NnTF \g__bnvs_n_prop { #1 } {
904     \prg_return_true:
905   } {
906     \prg_return_false:
907   }
908 }

```

```

\__bnvs_n_get:ncTF \__bnvs_n_get:ncTF {<key>} <tl variable> {<yes code>} {<no code>}

```

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

```

909 \prg_new_conditional:Npnn \__bnvs_n_get:nc #1 #2 { T, F, TF } {
910   \__bnvs_prop_get:NnTF \g__bnvs_n_prop { #1 } { #2 } {
911     \prg_return_true:
912   } {
913     \prg_return_false:
914   }
915 }

```

6.13.3 Regular expressions

`\c__bnvs_name_regex` The name of a slide range consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```

916 \regex_const:Nn \c__bnvs_name_regex {
917   [[:alpha:]]_ [[:alnum:]]_*
918 }

```

(End definition for `\c__bnvs_name_regex`.)

`\c__bnvs_id_regex` The name of a slide range consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```

919 \regex_const:Nn \c__bnvs_id_regex {
920   (?: \ur{c__bnvs_name_regex} | [?] )? !
921 }

```

(End definition for `\c__bnvs_id_regex`.)

`\c__bnvs_path_regex` A sequence of *positive integer* items representing a path.

```

922 \regex_const:Nn \c__bnvs_path_regex {
923   (?: \. \ur{c__bnvs_name_regex} | \. [-+]? \d+ )*
924 }

```

(End definition for `\c__bnvs_path_regex`.)

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

(End definition for `\c__bnvs_A_key_Z_regex`.)

```

925 \regex_const:Nn \c__bnvs_A_key_Z_regex {
    1: The range name including the slide <id> and question mark if any
    2: slide <id> including the question mark
926   \A ( ( \ur{c__bnvs_id_regex} ? ) \ur{c__bnvs_name_regex} )
    3: the path, if any.
927   ( \ur{c__bnvs_path_regex} ) \Z
928 }

```

`\c__bnvs_TEST_A_key_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 definition for `\c__bnvs_TEST_A_key_n_Z_regex`.)

```

929 \regex_const:Nn \c__bnvs_TEST_A_key_n_Z_regex {
    1: The full match
    2: The overlay set name including the slide <id> and question mark if any, the dotted
       path but excluding the trailing .n
    3: slide <id> including the question mark

```

```

930      \A ( ( \ur{c__bnvs_id_regex} ? )
931      \ur{c__bnvs_name_regex}
932      (?: \. \ur{c__bnvs_name_regex} | \. [-+]? \d+ )*? )

```

4: the last .n component if any.

```

933      ( \. n )? \Z
934  }

```

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

```

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

```

(End definition for `\c__bnvs_colons_regex`.)

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

```

936  \regex_const:Nn \c__bnvs_split_regex {
937      \s* ( ? :

```

We start with ‘++’ instrussions².

```

938      \+\+

```

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

```

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

```

- 3: optionally followed by a dotted path

```

940      ( \ur{c__bnvs_path_regex} )

```

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

```

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

```

- 6: optionally followed by a dotted path

```

942      ( \ur{c__bnvs_path_regex} )

```

We continue with other expressions

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

```

943      (?: \. (\+)\+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.

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

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

- 9: the post increment

```
945 | (\+)\+
```

```
946 )?
```

```
947 ) \s*
```

```
948 }
```

(End definition for `\c__bnvs_split_regex`.)

6.13.4 beamer.cls interface

Work in progress.

```
949 \RequirePackage{keyval}
950 \define@key{beamerframe}{beanoves~id}[] {
951   \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }
952 }
953 \AddToHook{env/beamer@frameslide/before}{
954   \__bnvs_n_gclear:
955   \__bnvs_v_gclear:
956   \bool_set_true:N \l__bnvs_in_frame_bool
957 }
958 \AddToHook{env/beamer@frameslide/after}{
959   \bool_set_false:N \l__bnvs_in_frame_bool
960 }
```

6.13.5 Defining named slide ranges

```
\__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.

```
961 \BNVS_new_conditional:cpnn { split_pop_left:c } #1 { T, F, TF } {
962   \__bnvs_seq_pop_left:ccTF { split } { #1 } {
963     \prg_return_true:
964   } {
965     \prg_return_false:
966   }
967 }
968 \exp_args_generate:n { VVV }
969 \BNVS_new_conditional:cpnn { range_set:cccn } #1 #2 #3 #4 { T, F, TF } {
970   \BNVS_begin:
971   \__bnvs_tl_clear:c { a }
972   \__bnvs_tl_clear:c { b }
973   \__bnvs_tl_clear:c { c }
974   \__bnvs_regex_split:cnTF { colons } { #4 } {
975     \__bnvs_seq_pop_left:ccT { split } { a } {
```

a may contain the $\langle start \rangle$.

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

This is a one colon range.

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

b may contain the $\langle end \rangle$.

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

A :: was expected:

```
981     \__bnvs_error:n { Invalid-range-expression(1):~#4 }
982     } {
983     \int_compare:nNnT { \__bnvs_tl_count:c { c } } > { 1 } {
984     \__bnvs_error:n { Invalid-range-expression(2):~#4 }
985     }
986     \__bnvs_split_pop_left:cTF { c } {
```

\l__bnvs_c_tl may contain the $\langle length \rangle$.

```
987     \__bnvs_seq_if_empty:cF { split } {
988     \__bnvs_error:n { Invalid-range-expression(3):~#4 }
989     }
990     } {
991     \__bnvs_error:n { Internal~error }
992     }
993     }
994     }
995     } {
996     }
997     } {
```

This is a two colon range component.

```
998     \int_compare:nNnT { \__bnvs_tl_count:c { b } } > { 1 } {
999     \__bnvs_error:n { Invalid-range-expression(4):~#4 }
1000     }
1001     \__bnvs_seq_pop_left:ccT { split } { c } {
```

c contains the $\langle length \rangle$.

```
1002     \__bnvs_split_pop_left:cTF { b } {
1003     \__bnvs_tl_if_empty:cTF { b } {
1004     \__bnvs_seq_pop_left:cc { split } { b }
```

b may contain the $\langle end \rangle$.

```
1005     \__bnvs_seq_if_empty:cF { split } {
1006     \__bnvs_error:n { Invalid-range-expression(5):~#4 }
1007     }
1008     } {
1009     \__bnvs_error:n { Invalid-range-expression(6):~#4 }
1010     }
1011     } {
1012     \__bnvs_tl_clear:c { b }
1013     }
1014     }
1015     }
1016     }
1017     }
```

Providing both the $\langle start \rangle$, $\langle length \rangle$ and $\langle end \rangle$ of a range is not allowed, even if they happen to be consistent.

```

1018   \cs_set:Npn \BNVS_next: { }
1019   \__bnvs_tl_if_empty:cT { a } {
1020     \__bnvs_tl_if_empty:cT { b } {
1021       \__bnvs_tl_if_empty:cT { c } {
1022         \cs_set:Npn \BNVS_next: {
1023           \__bnvs_error:n { Invalid-range-expression(7):~#3 }
1024         }
1025       }
1026     }
1027   }
1028   \BNVS_next:
1029   \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1030     \BNVS_end:
1031     \__bnvs_tl_set:cn { #1 } { ##1 }
1032     \__bnvs_tl_set:cn { #2 } { ##2 }
1033     \__bnvs_tl_set:cn { #3 } { ##3 }
1034   }
1035   \BNVS_exp_args:Nvvv \BNVS:nnn { a } { b } { c }
1036   \prg_return_true:
1037 } {
1038   \BNVS_end:
1039   \prg_return_false:
1040 }
1041 }

```

$\backslash_bnvs_range:nnnn$
 $\backslash_bnvs_range:nvvv$

$\backslash_bnvs_range:nnnn$ $\{\langle key \rangle\}$ $\{\langle start \rangle\}$ $\{\langle end \rangle\}$ $\{\langle length \rangle\}$

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

```

1042 \BNVS_new:cpn { range:nnnn } #1 {
1043   \__bnvs_if_provide:TF {
1044     \__bnvs_if_in:nnTF A { #1 } {
1045       \use_none:nnn
1046     } {
1047       \__bnvs_if_in:nnTF Z { #1 } {
1048         \use_none:nnn
1049       } {
1050         \__bnvs_if_in:nnTF L { #1 } {
1051           \use_none:nnn
1052         } {
1053           \__bnvs_do_range:nnnn { #1 }
1054         }
1055       }
1056     }
1057   } {
1058     \__bnvs_do_range:nnnn { #1 }
1059   }
1060 }
1061 \BNVS_new:cpn { range:nvvv } #1 #2 #3 #4 {
1062   \BNVS_tl_use:nv {
1063     \BNVS_tl_use:nv {
1064       \BNVS_tl_use:nv {

```

```

1065         \BNVS_use:c { range:nnnn } { #1 }
1066     } { #2 }
1067 } { #3 }
1068 } { #4 }
1069 }

```

__bnvs_parse_record:n	__bnvs_parse_record:n {⟨full name⟩}
__bnvs_parse_record:v	__bnvs_parse_record:nn {⟨full name⟩} {⟨value⟩}
__bnvs_parse_record:nn	__bnvs_n_parse_record:n {⟨full name⟩}
__bnvs_parse_record:(xn vn)	__bnvs_n_parse_record:nn {⟨full name⟩} {⟨value⟩}
__bnvs_n_parse_record:n	
__bnvs_n_parse_record:v	
__bnvs_n_parse_record:nn	
__bnvs_n_parse_record:(xn vn)	

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.

```

1070 \BNVS_new:cpn { parse_record:n } #1 {
1071     \__bnvs_if_provide:TF {
1072         \__bnvs_gprovide:nnnT V { #1 } { 1 } {
1073             \__bnvs_gclear:n { #1 }
1074         }
1075     } {
1076         \__bnvs_gclear:n { #1 }
1077         \__bnvs_gput:nnn V { #1 } { 1 }
1078     }
1079 }
1080 \cs_generate_variant:Nn \__bnvs_parse_record:n { V }
1081 \BNVS_new:cpn { parse_record:v } {
1082     \BNVS_tl_use:nc {
1083         \__bnvs_parse_record:V
1084     }
1085 }
1086 \BNVS_new:cpn { parse_record:nn } #1 #2 {
1087     \__bnvs_range_set:cccnTF { a } { b } { c } { #2 } {
1088         \__bnvs_range:nvvv { #1 } { a } { b } { c }
1089     } {
1090         \__bnvs_if_provide:TF {
1091             \__bnvs_gprovide:nnnT V { #1 } { #2 } {
1092                 \__bnvs_gclear_all:n { #1 }
1093             }
1094         } {
1095             \__bnvs_gclear_all:n { #1 }
1096             \__bnvs_gput:nnn V { #1 } { #2 }
1097         }
1098     }
1099 }
1100 \cs_generate_variant:Nn \__bnvs_parse_record:nn { x, V }
1101 \BNVS_new:cpn { parse_record:vn } {
1102     \BNVS_tl_use:nc {
1103         \__bnvs_parse_record:Vn
1104     }

```



```

1105 }
1106 \BNVS_new:cpn { n_parse_record:n } #1 {
1107   \bool_if:NTF \l__bnvs_n_provide_bool {
1108     \__bnvs_n_gprovide:nn
1109   } {
1110     \__bnvs_n_gput:nn
1111   }
1112   { #1 } { 1 }
1113 }
1114 \cs_generate_variant:Nn \__bnvs_n_parse_record:n { V }
1115 \BNVS_new:cpn { n_parse_record:v } {
1116   \BNVS_tl_use:nc {
1117     \__bnvs_n_parse_record:V
1118   }
1119 }
1120 \BNVS_new:cpn { n_parse_record:nn } #1 #2 {
1121   \__bnvs_range_set:ccnTF { a } { b } { c } { #2 } {
1122     \__bnvs_error:n { Unexpected-range:~#2 }
1123   } {
1124     \__bnvs_if_provide:TF {
1125       \__bnvs_n_gprovide:nn { #1 } { #2 }
1126     } {
1127       \__bnvs_n_gput:nn { #1 } { #2 }
1128     }
1129   }
1130 }
1131 \cs_generate_variant:Nn \__bnvs_n_parse_record:nn { x, V }
1132 \BNVS_new:cpn { n_parse_record:vn } {
1133   \BNVS_tl_use:Nc \__bnvs_n_parse_record:Vn
1134 }

```

__bnvs_name_id_n_get:nTF
__bnvs_name_id_n_get:vTF

__bnvs_name_id_n_set:nTF {<key>} {< yes code>} {< no code>}

If the <key> is a key, put the name it defines into the **key** tl variable, the frame id in the **id** tl variable, then execute <yes code>. The **n** tl variable is empty except when <key> ends with .n. Otherwise execute <no code>. If <key> does not contain a frame id, then **key** is prepended with then **id_last** and **id** is set to this value as well.

```

1135 \BNVS_new:cpn { name_id_n_end_export: } {
1136   \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1137     \BNVS_end:
1138     \__bnvs_tl_set:cn { key } { ##1 }
1139     \__bnvs_tl_set:cn { id } { ##2 }
1140     \__bnvs_tl_set:cn { n } { ##3 }
1141   }
1142   \__bnvs_tl_if_empty:cTF { id } {
1143     \BNVS_exp_args:Nvvv
1144     \BNVS:nnn { key } { id_last } { n }
1145     \__bnvs_tl_put_left:cv { key } { id_last }
1146   } {
1147     \BNVS_exp_args:Nvvv
1148     \BNVS:nnn { key } { id } { n }
1149     \__bnvs_tl_set:cv { id_last } { id }
1150   }

```

```

1151 }
1152 \BNVS_new_conditional:cpnn { name_id_n_get:n } #1 { T, F, TF } {
1153   \BNVS_begin:
1154     \__bnvs_match_once:NnTF \c__bnvs_TEST_A_key_n_Z_regex { #1 } {
1155       \__bnvs_match_pop_left:cTF { key } {
1156         \__bnvs_match_pop_left:cTF { key } {
1157           \__bnvs_match_pop_left:cTF { id } {
1158             \__bnvs_match_pop_left:cTF { n } {
1159               \__bnvs_name_id_n_end_export:
1160               \prg_return_true:
1161             } {
1162               \BNVS_end:
1163               \__bnvs_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/n }
1164               \prg_return_false:
1165             }
1166           } {
1167             \BNVS_end:
1168             \__bnvs_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/id }
1169             \prg_return_false:
1170           }
1171         } {
1172           \BNVS_end:
1173           \__bnvs_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/name }
1174           \prg_return_false:
1175         }
1176       } {
1177         \BNVS_end:
1178         \__bnvs_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/n }
1179         \prg_return_false:
1180       }
1181     } {
1182       \BNVS_end:
1183       \prg_return_false:
1184     }
1185   }
1186 \BNVS_new_conditional:cpnn { name_id_n_get:v } #1 { T, F, TF } {
1187   \BNVS_tl_use:nv { \BNVS_use:c { name_id_n_get:nTF } } { #1 } {
1188     \prg_return_true:
1189   } {
1190     \prg_return_false:
1191   }
1192 }

```

__bnvs_parse:n	__bnvs_parse:n {<key>}
__bnvs_parse:nn	__bnvs_parse:nn {<key>} {<definition>}

Auxiliary functions called within a group by \keyval_parse:nnn. <key> is the overlay reference key, including eventually a dotted path and a frame identifier, <definition> is the corresponding definition.

\l__bnvs_match_seq Local storage for the match result.

(End definition for \l__bnvs_match_seq.)

```

1193 \BNVS_new:cpn { parse:n } #1 {

```

```

1194 \peek_remove_spaces:n {
1195   \peek_catcode:NTF \c_group_begin_token {
1196     \__bnvs_tl_if_empty:cTF { root } {
1197       \__bnvs_error:n { Unexpected~list~at~top~level. }
1198     }
1199     \BNVS_begin:
1200     \__bnvs_int_incr:c { }
1201     \__bnvs_tl_set:cx { root } { \__bnvs_int_use:c { } . }
1202     \cs_set:Npn \bnvs:nw #####1 #####2 \s_stop {
1203       \regex_match:nnT { \S* } { #####2 } {
1204         \__bnvs_error:n { Unexpected~#####2 }
1205       }
1206       \keyval_parse:nnn {
1207         \__bnvs_parse:n
1208       } {
1209         \__bnvs_parse:nn
1210       } { #####1 }
1211       \BNVS_end:
1212     }
1213     \bnvs:nw
1214   } {
1215     \__bnvs_tl_if_empty:cTF { root } {
1216       \__bnvs_name_id_n_get:nTF { #1 } {
1217         \__bnvs_tl_if_empty:cTF { n } {
1218           \__bnvs_parse_record:v
1219         } {
1220           \__bnvs_n_parse_record:v
1221         }
1222         { key }
1223       } {
1224         \__bnvs_error:n { Unexpected~key::~#1 }
1225       }
1226     } {
1227       \__bnvs_int_incr:c { }
1228       \__bnvs_tl_if_empty:cTF { n } {
1229         \__bnvs_parse_record:xn
1230       } {
1231         \__bnvs_n_parse_record:xn
1232       } {
1233         \__bnvs_tl_use:c { root } . \__bnvs_int_use:c { }
1234       } { #1 }
1235     }
1236     \use_none_delimit_by_s_stop:w
1237   }
1238 }
1239 #1 \s_stop
1240 }
1241 \BNVS_new:cpn { do_range:nnnn } #1 #2 #3 #4 {
1242   \__bnvs_gclear_all:n { #1 }
1243   \tl_if_empty:nTF { #4 } {
1244     \tl_if_empty:nTF { #2 } {
1245       \tl_if_empty:nTF { #3 } {
1246         \__bnvs_error:n { Not~a~range::~~#1 }
1247       } {

```

```

1248     \__bnvs_gput:nnn Z { #1 } { #3 }
1249     \__bnvs_gput:nnn V { #1 } { \q_nil }
1250   }
1251 } {
1252   \__bnvs_gput:nnn A { #1 } { #2 }
1253   \__bnvs_gput:nnn V { #1 } { \q_nil }
1254   \tl_if_empty:nF { #3 } {
1255     \__bnvs_gput:nnn Z { #1 } { #3 }
1256     \__bnvs_gput:nnn L { #1 } { \q_nil }
1257   }
1258 }
1259 } {
1260   \tl_if_empty:nTF { #2 } {
1261     \__bnvs_gput:nnn L { #1 } { #4 }
1262     \tl_if_empty:nF { #3 } {
1263       \__bnvs_gput:nnn Z { #1 } { #3 }
1264       \__bnvs_gput:nnn A { #1 } { \q_nil }
1265       \__bnvs_gput:nnn V { #1 } { \q_nil }
1266     }
1267   } {
1268     \__bnvs_gput:nnn A { #1 } { #2 }
1269     \__bnvs_gput:nnn L { #1 } { #4 }
1270     \__bnvs_gput:nnn Z { #1 } { \q_nil }
1271     \__bnvs_gput:nnn V { #1 } { \q_nil }
1272   }
1273 }
1274 }
1275 \cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
1276   \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
1277   { #4 }
1278 }
1279 \cs_new:Npn \BNVS_end_tl_set:cv #1 #2 {
1280   \BNVS_tl_use:nv {
1281     \BNVS_end: \__bnvs_tl_set:cn { #1 }
1282   } { #2 }
1283 }
1284 \BNVS_new:cpn { parse:nn } #1 #2 {
1285   \BNVS_begin:
1286   \__bnvs_tl_set:cn { a } { #1 }
1287   \__bnvs_tl_put_left:cv { a } { root }
1288   \__bnvs_name_id_n_get:vTF { a } {
1289     \regex_match:nnTF { \S } { #2 } {
1290       \peek_remove_spaces:n {
1291         \peek_catcode:NTF \c_group_begin_token {

```

The value is a comma separated list, go recursive. But before we warn about an unexpected .n suffix, if any.

```

1292     \__bnvs_tl_if_empty:cF { n } {
1293   \__bnvs_warning:n { Ignoring-unexpected-suffix~.n:~#1 }
1294   }
1295   \BNVS_begin:
1296   \__bnvs_tl_set:cv { root } { key }
1297   \int_set:Nn \l__bnvs_int { 0 }
1298   \cs_set:Npn \BNVS:nn ##1 ##2 \s_stop {

```

```

1299         \regex_match:nnT { \S } { ##2 } {
1300             \__bnvs_error:n { Unexpected~value~#2 }
1301         }
1302         \keyval_parse:nnn {
1303             \__bnvs_parse:n
1304         } {
1305             \__bnvs_parse:nn
1306         } { ##1 }
1307         \BNVS_end:
1308     }
1309     \BNVS:nn
1310 } {
1311     \__bnvs_tl_if_empty:cTF { n } {
1312         \__bnvs_parse_record:vn
1313     } {
1314         \__bnvs_n_parse_record:vn
1315     }
1316     { key } { #2 }
1317     \use_none_delimit_by_s_stop:w
1318 }
1319 }
1320 #2 \s_stop
1321 } {

```

Empty value given: remove the reference.

```

1322     \__bnvs_tl_if_empty:cTF { n } {
1323         \__bnvs_gclear:v
1324     } {
1325         \__bnvs_n_gremove:v
1326     }
1327     { key }
1328 }
1329 } {
1330     \__bnvs_error:n { Invalid~key:~#2 }
1331 }

```

We export \l__bnvs_id_last_tl:

```

1332     \BNVS_end_tl_set:cv { id_last } { id_last }
1333 }

1334 \BNVS_new:cpn { parse_prepare:N } #1 {
1335     \tl_set:Nx #1 #1
1336     \bool_set_false:N \l__bnvs_parse_bool
1337     \bool_do_until:Nn \l__bnvs_parse_bool {
1338         \tl_if_in:NnTF #1 {%---[
1339     ]} {
1340         \regex_replace_all:nnNF { \[ ([^\]]%---)
1341     ]*%---[(
1342     ) \] } { { { \1 } } } #1 {
1343         \bool_set_true:N \l__bnvs_parse_bool
1344     }
1345     } {
1346         \bool_set_true:N \l__bnvs_parse_bool
1347     }
1348 }

```

```

1349 \tl_if_in:NnTF #1 {%---[
1350 ]} {
1351   \__bnvs_error:n { Unbalanced~%---[
1352   ]}
1353 } {
1354   \tl_if_in:NnT #1 { [%---]
1355   } {
1356     \__bnvs_error:n { Unbalanced~[ %---]
1357     }
1358   }
1359 }
1360 }

```

\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`.

```

1361 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1362   \BNVS_tl_use:nv {
1363     \BNVS_end:
1364     \__bnvs_tl_put_right:cn { #1 }
1365   } { #2 }
1366 }
1367 \cs_new:Npn \BNVS_end_v_gput:nc #1 #2 {
1368   \BNVS_tl_use:nv {
1369     \BNVS_end:
1370     \__bnvs_v_gput:nn { #1 }
1371   } { #2 }
1372 }
1373 \NewDocumentCommand \Beanoves { sm } {
1374   \tl_if_empty:NtF \@currenvir {

```

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

```

1375   \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
1376 } {
1377   \tl_if_eq:NnT \@currenvir { document } {

```

At the top level, clear everything.

```

1378   \__bnvs_gclear:
1379 }
1380 \BNVS_begin:
1381 \__bnvs_tl_clear:c { root }
1382 \int_zero:N \l__bnvs_int
1383 \__bnvs_tl_set:cn { a } { #2 }
1384 \tl_if_eq:NnT \@currenvir { document } {

```

At the top level, use the global definitions.

```

1385   \seq_if_empty:NF \g__bnvs_def_seq {
1386     \__bnvs_tl_put_left:cx { a } {
1387       \seq_use:Nn \g__bnvs_def_seq , ,
1388     }
1389   }
1390 }
1391 \__bnvs_parse_prepare:N \l__bnvs_a_tl

```

```

1392 \IfBooleanTF {#1} {
1393   \__bnvs_provide_on:
1394 } {
1395   \__bnvs_provide_off:
1396 }
1397 \BNVS_tl_use:nv {
1398   \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn }
1399 } { a }
1400 \BNVS_end_tl_set:cv { id_last } { id_last }
1401 \ignorespaces
1402 }
1403 }

```

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

```

1404 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}

```

6.13.6 Scanning named overlay specifications

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

<code>\beamer@frame</code>	<code>\beamer@frame {<overlay specification>}</code>
<code>\beamer@masterdecode</code>	<code>\beamer@masterdecode {<overlay specification>}</code>

Preprocess `<overlay specification>` before `beamer` reads it.

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

(End definition for `\l__bnvs_ans_tl`.)

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

```

1405 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
1406 \cs_set:Npn \beamer@frame < #1 > {
1407   \BNVS_begin:
1408   \__bnvs_tl_clear:c { ans }
1409   \__bnvs_scan:nNc { #1 } \__bnvs_eval:nc { ans }
1410   \BNVS_tl_use:nv {
1411     \BNVS_end:
1412     \__bnvs_beamer@frame <
1413   } { ans } >
1414 }
1415 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
1416 \cs_set:Npn \beamer@masterdecode #1 {
1417   \BNVS_begin:
1418   \__bnvs_tl_clear:c { ans }
1419   \__bnvs_scan:nNc { #1 } \__bnvs_eval:nc { ans }
1420   \BNVS_tl_use:nv {
1421     \BNVS_end:
1422     \__bnvs_beamer@masterdecode
1423   } { ans }
1424 }

```

<u>__bnvs_scan:nNc</u>	<p>__bnvs_scan:nNc {<i>(named overlay expression)</i>} <i><eval></i> <i><tl core></i></p> <p>Scan the <i><named overlay expression></i> argument and feed the <i><tl variable></i> replacing <i>?(...)</i> instructions by their static counterpart with help from the <i><eval></i> function, which is __bnvs_eval:nN. A group is created to use local variables:</p>
\l__bnvs_ans_tl	<p>The token list that will be appended to <i><tl variable></i> on return.</p> <p>(End definition for \l__bnvs_ans_tl.)</p>
\l__bnvs_int	<p>Store the depth level in parenthesis grouping used when finding the proper closing parenthesis balancing the opening parenthesis that follows immediately a question mark in a <i>?(...)</i> instruction.</p> <p>(End definition for \l__bnvs_int.)</p>
\l__bnvs_query_tl	<p>Storage for the overlay query expression to be evaluated.</p> <p>(End definition for \l__bnvs_query_tl.)</p>
\l__bnvs_token_seq	<p>The <i><overlay expression></i> is split into the sequence of its tokens.</p> <p>(End definition for \l__bnvs_token_seq.)</p>
\l__bnvs_token_tl	<p>Storage for just one token.</p> <p>(End definition for \l__bnvs_token_tl.)</p> <p>Next are helpers.</p>

<u>__bnvs_scan_question:T</u>	<p>__bnvs_scan_question:T {<i><code></i>}</p> <p>At top level state, scan the tokens of the <i><named overlay expression></i> looking for a ‘?’ character. If a ‘?(...)’ is found, then the <i><code></i> is executed.</p> <pre> 1425 \BNVS_new:cpn { scan_question:T } #1 { 1426 __bnvs_seq_pop_left:ccT { token } { token } { 1427 __bnvs_tl_if_eq:cnTF { token } { ? } { 1428 __bnvs_scan_require_open: 1429 #1 1430 } { 1431 __bnvs_tl_put_right:cv { ans } { token } 1432 } 1433 } 1434 __bnvs_scan_question:T { #1 } 1435 } 1436 }</pre>
--------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<u>__bnvs_scan_require_open:</u>	<p>\require_open:</p> <p>We just found a ‘?’, we first gobble tokens until the next ‘(’, whatever they may be. In general, no tokens should be silently ignored.</p> <pre> 1437 \BNVS_new:cpn { scan_require_open: } {</pre> <p>Get next token.</p>
-----------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------


```

1438  \__bnvs_seq_pop_left:ccTF { token } { token } {
1439      \tl_if_eq:NnTF \l__bnvs_token_tl { ( %)
1440  } {

```

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

```

1441      \__bnvs_int_set:cn { } { 1 }

```

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

```

1442      \__bnvs_tl_clear:c { query }
1443      \__bnvs_scan_require_close:
1444  } {

```

Ignore this token and loop.

```

1445      \__bnvs_scan_require_open:
1446  }
1447  } {

```

End reached but no opening parenthesis found, raise.

```

1448      \__bnvs_fatal:x {Missing-'('%---)
1449      ~after~a~? }
1450  }
1451  }

```

```

\__bnvs_scan_require_close: \require_close:

```

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

```

1452 \BNVS_new:cpn { scan_require_close: } {

```

Get next token.

```

1453  \__bnvs_seq_pop_left:ccTF { token } { token } {
1454      \__bnvs_tl_if_eq:cnTF { token } { ( %---)
1455  } {

```

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

```

1456      \__bnvs_int_incr:c { }
1457      \__bnvs_tl_put_right:cv { query } { token }
1458      \__bnvs_scan_require_close:
1459  } {

```

This is not a ‘(’.

```

1460      \__bnvs_tl_if_eq:cnTF { token } { %(---)
1461  )
1462  } {

```

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

```

1463      \__bnvs_int_decr:c {}
1464      \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 ?.

```
1465         } {
```

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

```
1466         \__bnvs_tl_put_right:cv { query } { token }
1467         \__bnvs_scan_require_close:
1468     }
1469 } {
```

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

```
1470         \__bnvs_tl_put_right:cv { query } { token }
1471         \__bnvs_scan_require_close:
1472     }
1473 }
1474 } {
```

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 ‘)’.

```
1475     \__bnvs_error:x { Missing~%(---
1476         `)' }
1477     \__bnvs_tl_put_right:cx { query } {
1478         \prg_replicate:nn { \l__bnvs_int } {%(---
1479         )}
1480     }
1481 }
1482 }
```

```
1483 \BNVS_new:cpn { scan:nNc } #1 #2 #3 {
1484     \BNVS_begin:
1485     \BNVS_set:cpn { fatal:x } ##1 {
1486         \msg_fatal:nnx { beanoves } { :n }
1487         { \tl_to_str:n { #1 } :~##1}
1488     }
1489     \BNVS_set:cpn { error:x } ##1 {
1490         \msg_error:nnx { beanoves } { :n }
1491         { \tl_to_str:n { #1 } :~##1}
1492     }
1493     \__bnvs_tl_set:cn { scan } { #1 }
1494     \__bnvs_tl_clear:c { ans }
1495     \__bnvs_seq_clear:c { token }
```

Explode the *<named overlay expression>* into a list of individual tokens:

```
1496     \regex_split:nnN { } { #1 } \l__bnvs_token_seq
```

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

```
1497     \__bnvs_scan_question:T {
1498         \BNVS_tl_use:Nv #2 { query } { ans }
1499     }
1500     \BNVS_tl_use:nv {
1501         \BNVS_end:
1502         \__bnvs_tl_put_right:cn { #3 }
1503     } { ans }
1504 }
```

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.

```

1505 \exp_args_generate:n { VVx }
1506 \quark_new:N \q__bnvs
1507 \BNVS_new:cpn { end_kip_export_seq:nnccc } #1 #2 #3 #4 #5 #6 {
1508   \BNVS_end:
1509   \tl_if_empty:nTF { #2 } {
1510     \__bnvs_tl_set:cn { #4 } { #1 }
1511     \__bnvs_tl_put_left:cv { #4 } { #5 }
1512   } {
1513     \__bnvs_tl_set:cn { #4 } { #1 }
1514     \__bnvs_tl_set:cn { #5 } { #2 }
1515   }
1516   \__bnvs_seq_set_split:cn { #6 } { \q__bnvs } { #3 }
1517   \__bnvs_seq_remove_all:cn { #6 } { }
1518 }
1519 \BNVS_new:cpn { end_kip_export:ccc } {
1520   \exp_args:Nnnx \BNVS_tl_use:nv {
1521     \BNVS_tl_use:Nv \__bnvs_end_kip_export_seq:nnccc { key }
1522   } { id } {
1523     \__bnvs_seq_use:cn { path } { \q__bnvs }
1524   }
1525 }
1526 \BNVS_new_conditional:cpnn { match_pop_kip: } { T, F, TF } {
1527   \__bnvs_match_pop_left:cTF { key } {
1528     \__bnvs_match_pop_left:cTF { key } {
1529       \__bnvs_match_pop_left:cTF { id } {
1530         \__bnvs_match_pop_left:cTF { path } {
1531           \__bnvs_seq_set_split:cnv { path } { . } { path }
1532           \__bnvs_seq_remove_all:cn { path } { }
1533           \prg_return_true:
1534         } {
1535           \prg_return_false:
1536         }
1537       } {
1538         \prg_return_false:
1539       }
1540     } {
1541       \prg_return_false:
1542     }
1543   } {
1544     \prg_return_false:
1545   }

```

```

1546 }
1547 \BNVS_new_conditional:cpnn { kip:ccc } #1 #2 #3 { T, F, TF } {
1548   \BNVS_begin:
1549   \__bnvs_match_once:NvTF \c__bnvs_A_key_Z_regex { #1 } {

```

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

```

1550   \__bnvs_match_pop_kip:TF {
1551     \__bnvs_end_kip_export:ccc { #1 } { #2 } { #3 }
1552     \prg_return_true:
1553   } {
1554     \BNVS_end:
1555     \prg_return_false:
1556   }
1557 } {
1558   \BNVS_end:
1559   \prg_return_false:
1560 }
1561 }

```

```

\__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}

```

$\{\langle yes\ code\rangle\}$ will be executed once resolution has occurred, $\{\langle no\ code\rangle\}$ 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, $\backslash l_bnvs_key_tl$ contains a slide range name, $\backslash l_bnvs_id_tl$ contains a frame id and $\backslash l_bnvs_path_seq$ contains the components of an integer path, possibly empty. On return, the variable $\backslash l_bnvs_key_tl$ contains the resolved range name, $\backslash l_bnvs_id_tl$ contains the frame id used and $\backslash l_bnvs_path_seq$ contains the sequence of integer path components that could not be resolved.

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

1. If $\backslash l_bnvs_key_tl$ 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 $\backslash l_bnvs_key_tl$ is replaced by this name, the $\backslash l_bnvs_id_tl$ and $\backslash l_bnvs_id_tl$ are updates accordingly and the $\langle path\ seq\ var\rangle$ is prepended with the integer path.
2. If $\langle path\ seq\ var\rangle$ is not empty, append to the right of $\backslash l_bnvs_key_tl$ 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 $\langle path\ seq\ var\rangle$ is empty, $\langle name\ tl\ var\rangle$ 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}

```

```

1562 \BNVS_new:cpn { kip_x_path_resolve:TFF } #1 #2 {
1563   \__bnvs_kip_x_path_resolve:TF {
1564     \__bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
1565   }
1566 }

```

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.

```

1567 \BNVS_new:cpn { end_kip_export: } {
1568   \exp_args:Nnnx
1569   \BNVS_tl_use:nv {
1570     \BNVS_tl_use:Nv \__bnvs_end_kip_export_seq:nnccc { key }
1571   } { id } {
1572     \__bnvs_seq_use:cn { path } { \q__bnvs }
1573   } { key } { id } { path }
1574 }
1575 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
1576   \__bnvs_seq_if_empty:cF { #2 } {
1577     \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
1578       \__bnvs_seq_use:cn { #1 } { \q__bnvs }
1579       \exp_not:n { \q__bnvs }
1580       \__bnvs_seq_use:cn { #2 } { \q__bnvs }
1581     }
1582     \__bnvs_seq_remove_all:cn { #1 } { }
1583   }
1584 }
1585 \BNVS_new:cpn { kip_x_path_resolve:nFF } #1 #2 #3 {
1586   \__bnvs_get:nvcTF #1 { a } { b } {
1587     \__bnvs_kip:cccTF { b } { id } { path } {
1588       \__bnvs_tl_set_eq:cc { key } { b }
1589       \__bnvs_seq_merge:cc { path } { b }
1590       \__bnvs_seq_clear:c { b }
1591       \__bnvs_seq_set_eq:cc { a } { path }
1592       \__bnvs_kip_x_path_resolve_loop_or_end_return:
1593     } {
1594       \__bnvs_seq_if_empty:cTF { b } {
1595         \__bnvs_tl_set_eq:cc { key } { b }
1596         \__bnvs_seq_clear:c { path }
1597         \__bnvs_seq_clear:c { a }
1598         \__bnvs_kip_x_path_resolve_loop_or_end_return:
1599       } {
1600         #2
1601       }
1602     }
1603   } {
1604     #3
1605   }
1606 }

```

```

1607 \BNVS_new:cpn { kip_x_path_resolve_VAL_loop_or_end_return:F } #1 {
1608   \__bnvs_kip_x_path_resolve:nFF V { #1 } {
1609     \__bnvs_kip_x_path_resolve:nFF A { #1 } {
1610       \__bnvs_kip_x_path_resolve:nFF L { #1 } { #1 }
1611     }
1612   }
1613 }
1614 \BNVS_new:cpn { kip_x_path_resolve_end_return_true: } {
1615   \__bnvs_seq_pop_left:ccTF { path } { a } {
1616     \__bnvs_seq_if_empty:cTF { path } {
1617       \__bnvs_tl_clear:c { b }
1618       \__bnvs_index_can:vTF { key } {
1619         \__bnvs_index_append:vvTF { key } { a } { b } {
1620           \__bnvs_tl_set:cv { key } { b }
1621         } {
1622           \__bnvs_tl_set:cv { key } { a }
1623         }
1624       } {
1625         \__bnvs_tl_set:cv { key } { a }
1626       }
1627     } {
1628       \__bnvs_error:x { Path-too-long~.\BNVS_tl_use:c { a }
1629         .\__bnvs_seq_use:cn { path } . }
1630     }
1631   } {
1632     \__bnvs_value_resolve:vcT { key } { key } {}
1633   }
1634   \__bnvs_end_kip_export:
1635   \prg_return_true:
1636 }
1637 \BNVS_new_conditional:cpnn { kip_x_path_resolve: } { T, F, TF } {
1638   \BNVS_begin:
1639   \__bnvs_seq_set_eq:cc { a } { path }
1640   \__bnvs_seq_clear:c { b }
1641   \__bnvs_kip_x_path_resolve_loop_or_end_return:
1642 }
1643 \BNVS_new:cpn { kip_x_path_resolve_loop_or_end_return: } {
1644   \__bnvs_call:TF {
1645     \__bnvs_tl_set_eq:cc { a } { key }
1646     \__bnvs_seq_if_empty:cTF { a } {
1647       \__bnvs_kip_x_path_resolve_VAL_loop_or_end_return:F {
1648         \__bnvs_kip_x_path_resolve_end_return_true:
1649       }
1650     } {
1651       \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { a } . }
1652       \__bnvs_kip_x_path_resolve_VAL_loop_or_end_return:F {
1653         \__bnvs_seq_pop_right:ccT { a } { c } {
1654           \__bnvs_seq_put_left:cv { b } { c }
1655         }
1656       }
1657     }
1658   }
1659 } {
1660   \BNVS_end:

```

```

1661     \prg_return_false:
1662 }
1663 }

1664 \BNVS_new:cpn { kip_n_path_resolve_or_end_return:nF } #1 #2 {
1665   \__bnvs_get:nvcTF { #1 } { a } { b } {
1666     \__bnvs_kip:cccTF { b } { id } { path } {
1667       \__bnvs_tl_set_eq:cc { key } { b }
1668       \__bnvs_seq_merge:cc { path } { b }
1669       \__bnvs_seq_set_eq:cc { a } { path }
1670       \__bnvs_seq_clear:c { b }
1671       \__bnvs_kip_n_path_resolve_loop_or_end_return:
1672     } {
1673       \__bnvs_seq_pop_right:ccTF { a } { c } {
1674         \__bnvs_seq_put_left:cv { b } { c }
1675         \__bnvs_kip_n_path_resolve_loop_or_end_return:
1676       } {
1677         \__bnvs_kip_n_path_resolve_end_return_true:
1678       }
1679     } {
1680   } {
1681     #2
1682   }
1683 }

1684 \BNVS_new:cpn { kip_n_path_resolve_VAL_loop_or_end_return: } {
1685   \__bnvs_kip_n_path_resolve_or_end_return:nF V {
1686     \__bnvs_kip_n_path_resolve_or_end_return:nF A {
1687       \__bnvs_kip_n_path_resolve_or_end_return:nF L {
1688         \__bnvs_seq_pop_right:ccTF { a } { c } {
1689           \__bnvs_seq_put_left:cv { b } { c }
1690           \__bnvs_kip_n_path_resolve_loop_or_end_return:
1691         } {
1692           \__bnvs_kip_n_path_resolve_end_return_true:
1693         }
1694       }
1695     }
1696   }
1697 }

1698 \BNVS_new:cpn { kip_n_path_resolve_end_return_false: } {
1699   \BNVS_end:
1700   \prg_return_false:
1701 }

1702 \BNVS_new:cpn { kip_n_path_resolve_end_return_true: } {
1703   \__bnvs_end_kip_export:
1704   \prg_return_true:
1705 }

```

__bnvs_kip_n_path_resolve_loop_or_end_return:

Loop to resolve the path.

```

1706 \BNVS_new:cpn { kip_n_path_resolve_loop_or_end_return: } {
1707   \__bnvs_call:TF {

```

```

1708     \__bnvs_tl_set_eq:cc { a } { key }
1709     \__bnvs_seq_if_empty:cTF { a } {
1710         \__bnvs_seq_if_empty:cTF { b } {
1711             \__bnvs_kip_n_path_resolve_end_return_true:
1712         } {
1713             \__bnvs_kip_n_path_resolve_VAL_loop_or_end_return:
1714         }
1715     } {
1716         \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { a } . }
1717         \__bnvs_kip_n_path_resolve_VAL_loop_or_end_return:
1718     }
1719 } {
1720     \BNVS_end:
1721     \prg_return_false:
1722 }
1723 }

```

__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.

```

1724 \BNVS_new_conditional:cpnn { kip_n_path_resolve: } { T, F, TF } {
1725     \BNVS_begin:
1726     \__bnvs_seq_set_eq:cc { a } { path }
1727     \__bnvs_seq_clear:c { b }
1728     \__bnvs_kip_n_path_resolve_loop_or_end_return:
1729 }

```

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.

```

1730 \BNVS_new:cpn { round_ans:n } #1 {
1731     \tl_if_empty:nTF { #1 } {
1732         \__bnvs_tl_put_right:cn { ans } { 0 }

```



```

1733 } {
1734   \__bnvs_tl_put_right:cx { ans } { \fp_eval:n { round(#1) } }
1735 }
1736 }
1737 \BNVS_new:cpn { round:N } #1 {
1738   \tl_if_empty:NTF #1 {
1739     \tl_set:Nn #1 { 0 }
1740   } {
1741     \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
1742   }
1743 }
1744 \BNVS_new:cpn { round:c } {
1745   \BNVS_tl_use:Nc \__bnvs_round:N
1746 }

```

```

\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.

```

1747 \cs_new:Npn \BNVS_end_return_false: {
1748   \BNVS_end:
1749   \prg_return_false:
1750 }
1751 \cs_new:Npn \BNVS_end_return_false:x #1 {
1752   \__bnvs_error:x { #1 }
1753   \BNVS_end_return_false:
1754 }

```

```

\__bnvs_value_resolve:ncTF   \__bnvs_value_resolve:ncTF {<key>} <tl core> {<yes code>} {<no code>}
\__bnvs_value_resolve:vcTF   \__bnvs_value_append:ncTF {<key>} <tl core> {<yes code>} {<no code>}
\__bnvs_value_append:ncTF
\__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.

```

1755 \BNVS_new:cpn { value_resolve_return:nnnT } #1 #2 #3 #4 {
1756   \__bnvs_tl_if_empty:cTF { #3 } {
1757     \prg_return_false:
1758   } {
1759     \__bnvs_cache_gput:nnv V { #2 } { #3 }
1760     #4
1761     \prg_return_true:
1762   }
1763 }

```

```

1764 \BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
1765   \BNVS_tl_use:Nc \quark_if_nil:NTF { #1 } {
1766     \prg_return_true:
1767   } {
1768     \prg_return_false:
1769   }
1770 }
1771 \BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
1772   \BNVS_tl_use:Nc \quark_if_no_value:NTF { #1 } {
1773     \prg_return_true:
1774   } {
1775     \prg_return_false:
1776   }
1777 }
1778 \BNVS_new_conditional:cpnn { value_resolve:nc } #1 #2 { T, F, TF } {
1779   \__bnvs_cache_get:nncTF V { #1 } { #2 } {
1780     \prg_return_true:
1781   } {
1782     \__bnvs_get:nncTF V { #1 } { #2 } {
1783       \__bnvs_quark_if_nil:cTF { #2 } {

```

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

```

1784   \__bnvs_gput:nnn V { #1 } { \q_no_value }
1785   \__bnvs_first_resolve:ncTF { #1 } { #2 } {
1786     \__bnvs_value_resolve_return:nnnT A { #1 } { #2 } {
1787       \__bnvs_gput:nnn V { #1 } { \q_nil }
1788     }
1789   } {
1790     \__bnvs_last_resolve:ncTF { #1 } { #2 } {
1791       \__bnvs_value_resolve_return:nnnT Z { #1 } { #2 } {
1792         \__bnvs_gput:nnn V { #1 } { \q_nil }
1793       }
1794     } {
1795       \__bnvs_gput:nnn V { #1 } { \q_nil }
1796       \prg_return_false:
1797     }
1798   }
1799 } {
1800   \__bnvs_quark_if_no_value:cTF { #2 } {
1801     \__bnvs_fatal:n {Circular~definition:~#1}
1802   } {

```

Possible recursive call.

```

1803   \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1804     \__bnvs_value_resolve_return:nnnT V { #1 } { #2 } {
1805       \__bnvs_gput:nnn V { #1 } { \q_nil }
1806     }
1807   } {
1808     \__bnvs_gput:nnn V { #1 } { \q_nil }
1809     \prg_return_false:
1810   }
1811 }
1812 }
1813 } {

```

```

1814     \prg_return_false:
1815   }
1816 }
1817 }
1818 \BNVS_new_conditional:cpnn { value_resolve:vc } #1 #2 { T, F, TF } {
1819   \BNVS_tl_use:Nv \_bnvs_value_resolve:ncTF { #1 } { #2 } {
1820     \prg_return_true:
1821   } {
1822     \prg_return_false:
1823   }
1824 }
1825 \BNVS_new:cpn { end_put_right:vc } #1 #2 {
1826   \BNVS_tl_use:nv {
1827     \BNVS_end:
1828     \_bnvs_tl_put_right:cn { #2 }
1829   } { #1 }
1830 }
1831 \BNVS_new_conditional:cpnn { value_append:nc } #1 #2 { T, F, TF } {
1832   \BNVS_begin:
1833   \_bnvs_value_resolve:ncTF { #1 } { #2 } {
1834     \BNVS_end_tl_put_right:cv { #2 } { #2 }
1835     \prg_return_true:
1836   } {
1837     \BNVS_end:
1838     \prg_return_true:
1839   }
1840 }
1841 \BNVS_new_conditional_vc:cn { value_append } { T, F, TF }

```

cTF:nnnnvalueFIRST2222

```

\_bnvs_first_resolve:ncTF      \_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:ncTF
\_bnvs_first_append:(xc|vc)TF

```

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

```

1842 \BNVS_new_conditional:cpnn { first_resolve:nc } #1 #2 { T, F, TF } {
1843   \_bnvs_cache_get:nncTF A { #1 } { #2 } {
1844     \prg_return_true:
1845   } {
1846     \_bnvs_get:nncTF A { #1 } { #2 } {
1847       \_bnvs_quark_if_nil:cTF { #2 } {
1848         \_bnvs_gput:nnn A { #1 } { \q_no_value }

```

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

```

1849   \_bnvs_last_resolve:ncTF { #1 } { #2 } {
1850     \_bnvs_tl_put_right:cn { #2 } { - }
1851     \_bnvs_length_append:ncTF { #1 } { #2 } {
1852       \_bnvs_tl_put_right:cn { #2 } { + 1 }
1853       \_bnvs_round:c { #2 }
1854       \_bnvs_tl_if_empty:cTF { #2 } {

```

```

1855         \_bnvs_gput:nnn A { #1 } { \q_nil }
1856     \prg_return_false:
1857 } {
1858     \_bnvs_gput:nnn A { #1 } { \q_nil }
1859     \_bnvs_cache_gput:nnv A { #1 } { #2 }
1860     \prg_return_true:
1861 }
1862 } {
1863     \_bnvs_error:n {
1864 Unavailable~length~for~#1~(\token_to_str:N\_bnvs_first_resolve:ncTF/2) }
1865     \_bnvs_gput:nnn A { #1 } { \q_nil }
1866     \prg_return_false:
1867 }
1868 } {
1869     \_bnvs_error:n {
1870 Unavailable~last~for~#1~(\token_to_str:N\_bnvs_first_resolve:ncTF/1) }
1871     \_bnvs_gput:nnn A { #1 } { \q_nil }
1872     \prg_return_false:
1873 }
1874 } {
1875     \_bnvs_quark_if_no_value:cTF { a } {
1876         \_bnvs_fatal:n {Circular~definition:~#1}
1877     } {
1878         \_bnvs_if_resolve:vcTF { #2 } { #2 } {
1879             \_bnvs_cache_gput:nnv A { #1 } { #2 }
1880             \prg_return_true:
1881         } {
1882             \prg_return_false:
1883         }
1884     }
1885 }
1886 } {
1887     \prg_return_false:
1888 }
1889 }
1890 }
1891 \BNVS_new_conditional_vc:cn { first_resolve } { T, F, TF }
1892 \BNVS_new_conditional_cpnn { first_append:nc } #1 #2 { T, F, TF } {
1893     \BNVS_begin:
1894     \_bnvs_first_resolve:ncTF { #1 } { #2 } {
1895         \BNVS_end_tl_put_right:cv { #2 } { #2 }
1896         \prg_return_true:
1897     } {
1898         \prg_return_false:
1899     }
1900 }

```

<u>_bnvs_last_resolve:ncTF</u> <u>_bnvs_last_append:ncTF</u>	_bnvs_last_resolve:ncTF {<key>} <tl variable> {<yes code>} {<no code>} _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.

```

1901 \BNVS_new_conditional:cpnn { last_resolve:nc } #1 #2 { T, F, TF } {
1902   \__bnvs_cache_get:nncTF Z { #1 } { #2 } {
1903     \prg_return_true:
1904   } {
1905     \__bnvs_get:nncTF Z { #1 } { #2 } {
1906       \__bnvs_quark_if_nil:cTF { #2 } {
1907         \__bnvs_gput:nnn Z { #1 } { \q_no_value }

```

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

```

1908     \__bnvs_first_resolve:ncTF { #1 } { #2 } {
1909       \__bnvs_tl_put_right:cn { #2 } { + }
1910       \__bnvs_length_append:ncTF { #1 } { #2 } {
1911         \__bnvs_tl_put_right:cn { #2 } { - 1 }
1912         \__bnvs_round:c { #2 }
1913         \__bnvs_cache_gput:nnv Z { #1 } { #2 }
1914         \__bnvs_gput:nnn Z { #1 } { \q_nil }
1915         \prg_return_true:
1916       } {
1917         \__bnvs_error:x {
1918 Unavailable~length~for~#1~(\token_to_str:N \__bnvs_last_resolve:ncTF/1) }
1919         \__bnvs_gput:nnn Z { #1 } { \q_nil }
1920         \prg_return_false:
1921       }
1922     } {
1923       \__bnvs_error:x {
1924 Unavailable~first~for~#1~(\token_to_str:N \__bnvs_last_resolve:ncTF/1) }
1925       \__bnvs_gput:nnn Z { #1 } { \q_nil }
1926       \prg_return_false:
1927     }
1928   } {
1929     \__bnvs_quark_if_no_value:cTF { #2 } {
1930       \__bnvs_fatal:n {Circular~definition:~#1}
1931     } {
1932       \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1933         \__bnvs_cache_gput:nnv Z { #1 } { #2 }
1934         \prg_return_true:
1935       } {
1936         \prg_return_false:
1937       }
1938     }
1939   }
1940 } {
1941   \prg_return_false:
1942 }
1943 }
1944 }
1945 \BNVS_new_conditional_vc:cn { last_resolve } { T, F, TF }
1946 \prg_new_conditional:Npnn \__bnvs_last_append:nc #1 #2 { T, F, TF } {
1947   \BNVS_begin:
1948   \__bnvs_last_resolve:ncTF { #1 } { #2 } {
1949     \BNVS_end_tl_put_right:cv { #2 } { #2 }
1950     \prg_return_true:
1951   } {

```

```

1952     \BNVS_end:
1953     \prg_return_false:
1954   }
1955 }
1956 \BNVS_new_conditional_vc:cn { last_append } { T, F, TF }

```

```

\__bnvs_length_resolve:ncTF \__bnvs_length_resolve:ncTF {<key>} <tl variable> {<yes code>} {<no code>}
\__bnvs_length_append:ncTF \__bnvs_length_append:ncTF {<key>} <tl variable> {<yes code>} {<no code>}

```

Resolve the length of the $\langle key \rangle$ slide range into $\langle tl variable \rangle$, or append the length of the $\langle key \rangle$ slide range to $\langle tl variable \rangle$. Execute $\langle yes code \rangle$ when there is a $\langle length \rangle$, $\langle no code \rangle$ otherwise.

```

1957 \BNVS_new_conditional:cpnn { length_resolve:nc } #1 #2 { T, F, TF } {
1958   \__bnvs_cache_get:nncTF L { #1 } { #2 } {
1959     \prg_return_true:
1960   } {
1961     \__bnvs_get:nncTF L { #1 } { #2 } {
1962       \__bnvs_quark_if_nil:cTF { #2 } {
1963         \__bnvs_gput:nnn L { #1 } { \q_no_value }

```

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

```

1964   \__bnvs_last_resolve:ncTF { #1 } { #2 } {
1965     \__bnvs_tl_put_right:cn { #2 } { - }
1966     \__bnvs_first_append:ncTF { #1 } { #2 } {
1967       \__bnvs_tl_put_right:cn { #2 } { + 1 }
1968       \__bnvs_round:c { #2 }
1969       \__bnvs_gput:nnn L { #1 } { \q_nil }
1970       \__bnvs_cache_gput:nnv L { #1 } { #2 }
1971       \prg_return_true:
1972     } {
1973       \__bnvs_error:n {
1974         Unavailable~first~for~#1~(\__bnvs_length_resolve:ncTF/2) }
1975       \return_false:
1976     }
1977   } {
1978     \__bnvs_error:n {
1979       Unavailable~last~for~#1~(\__bnvs_length_resolve:ncTF/1) }
1980     \return_false:
1981   }
1982 } {
1983   \__bnvs_quark_if_no_value:cTF { #2 } {
1984     \__bnvs_fatal:n {Circular~definition:~#1}
1985   } {
1986     \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1987       \__bnvs_cache_gput:nnv L { #1 } { #2 }
1988       \prg_return_true:
1989     } {
1990       \prg_return_false:
1991     }
1992   }
1993 }
1994 } {

```

```

1995     \prg_return_false:
1996   }
1997 }
1998 }
1999 \BNVS_new_conditional_vc:cn { length_resolve } { T, F, TF }
2000 \BNVS_new_conditional:cpnn { length_append:nc } #1 #2 { T, F, TF } {
2001   \BNVS_begin:
2002   \__bnvs_length_resolve:ncTF { #1 } { #2 } {
2003     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2004     \prg_return_true:
2005   } {
2006     \prg_return_false:
2007   }
2008 }
2009 \BNVS_new_conditional_vc:cn { length_append } { T, F, TF }

```

`__bnvs_range_resolve:ncTF`
`__bnvs_range_append:ncTF`

`__bnvs_range_resolve:ncTF {<key>} <tl variable> {<yes code>} {<no code>}`
`__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.

```

2010 \BNVS_new_conditional:cpnn { range_append:nc } #1 #2 { T, F, TF } {
2011   \BNVS_begin:
2012   \__bnvs_first_resolve:ncTF { #1 } { a } {
2013     \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
2014       \__bnvs_tl_set:cn { a } { 0 }
2015     }
2016     \__bnvs_last_resolve:ncTF { #1 } { b } {

```

Limited from above and below.

```

2017     \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2018       \__bnvs_tl_set:cn { b } { 0 }
2019     }
2020     \__bnvs_tl_put_right:cn { a } { - }
2021     \__bnvs_tl_put_right:cv { a } { b }
2022     \BNVS_end_tl_put_right:cv { #2 } { a }
2023     \prg_return_true:
2024   } {

```

Limited from below.

```

2025     \BNVS_end_tl_put_right:cv { #2 } { a }
2026     \__bnvs_tl_put_right:cn { #2 } { - }
2027     \prg_return_true:
2028   }
2029 } {
2030   \__bnvs_last_resolve:ncTF { #1 } { b } {

```

Limited from above.

```

2031     \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2032       \__bnvs_tl_set:cn { b } { 0 }
2033     }
2034     \__bnvs_tl_put_left:cn { b } { - }
2035     \BNVS_end_tl_put_right:cv { #2 } { b }

```

```

2036     \prg_return_true:
2037   } {
2038     \__bnvs_value_resolve:ncTF { #1 } { b } {
2039       \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2040         \__bnvs_tl_set:cn { b } { 0 }
2041       }

```

Unlimited range.

```

2042     \BNVS_end_tl_put_right:cv { #2 } { b }
2043     \__bnvs_tl_put_right:cn { #2 } { - }
2044     \prg_return_true:
2045   } {
2046     \BNVS_end:
2047     \prg_return_false:
2048   }
2049 }
2050 }
2051 }
2052 \BNVS_new_conditional_vc:cn { range_append } { T, F, TF }
2053 \BNVS_new_conditional:cpnn { range_resolve:nc } #1 #2 { T, F, TF } {
2054   \__bnvs_tl_clear:c { #2 }
2055   \__bnvs_range_append:ncTF { #1 } { #2 } {
2056     \prg_return_true:
2057   } {
2058     \prg_return_false:
2059   }
2060 }
2061 \BNVS_new_conditional_vc:cn { range_resolve } { T, F, TF }

```

$\backslash_bnvs_previous_resolve:ncTF$ $\backslash_bnvs_previous_append:ncTF$	$\backslash_bnvs_previous_append:ncTF \{ \langle key \rangle \} \langle tl\ variable \rangle \{ \langle yes\ code \rangle \} \{ \langle no\ code \rangle \}$
-----------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

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

```

2062 \BNVS_new_conditional:cpnn { previous_resolve:nc } #1 #2 { T, F, TF } {
2063   \__bnvs_cache_get:nncTF P { #1 } { #2 } {
2064     \prg_return_true:
2065   } {
2066     \__bnvs_first_resolve:ncTF { #1 } { #2 } {
2067       \__bnvs_tl_put_right:cn { #2 } { -1 }
2068       \__bnvs_round:c { #2 }
2069       \__bnvs_cache_gput:nv P { #1 } { #2 }
2070       \prg_return_true:
2071     } {
2072       \prg_return_false:
2073     }
2074   }
2075 }
2076 \BNVS_new_conditional_vc:cn { previous_resolve } { T, F, TF }
2077 \BNVS_new_conditional:cpnn { previous_append:nc } #1 #2 { T, F, TF } {

```



```

2078 \BNVS_begin:
2079 \__bnvs_previous_resolve:ncTF { #1 } { #2 } {
2080   \BNVS_end_tl_put_right:cv { #2 } { #2 }
2081   \prg_return_true:
2082 } {
2083   \BNVS_end:
2084   \prg_return_false:
2085 }
2086 }
2087 \BNVS_new_conditional_vc:cn { previous_append } { T, F, TF }

```

```

\__bnvs_next_resolve:ncTF
\__bnvs_next_append:ncTF

```

```

\__bnvs_next_resolve:ncTF {<key>} <tl variable> {<yes code>} {<no code>}
\__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.

```

2088 \BNVS_new_conditional:cpnn { next_resolve:nc } #1 #2 { T, F, TF } {
2089   \__bnvs_cache_get:nncTF N { #1 } { #2 } {
2090     \prg_return_true:
2091   } {
2092     \__bnvs_last_resolve:ncTF { #1 } { #2 } {
2093       \__bnvs_tl_put_right:cn { #2 } { +1 }
2094       \__bnvs_round:c { #2 }
2095       \__bnvs_cache_gput:nv N { #1 } { #2 }
2096       \prg_return_true:
2097     } {
2098       \prg_return_false:
2099     }
2100   }
2101 }
2102 \BNVS_new_conditional_vc:cn { next_resolve } { T, F, TF }
2103 \BNVS_new_conditional:cpnn { next_append:nc } #1 #2 { T, F, TF } {
2104   \BNVS_begin:
2105   \__bnvs_next_resolve:ncTF { #1 } { #2 } {
2106     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2107     \prg_return_true:
2108   } {
2109     \BNVS_end:
2110     \prg_return_true:
2111   }
2112 }
2113 \BNVS_new_conditional_vc:cn { next_append } { T, F, TF }

```

```

\__bnvs_v_resolve:ncTF
\__bnvs_v_append:ncTF

```

```

\__bnvs_v_resolve:ncTF {<key>} <tl variable> {<yes code>} {<no code>}
\__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.

```

2114 \BNVS_new_conditional:cpnn { v_resolve:nc } #1 #2 { T, F, TF } {

```

```

2115 \__bnvs_v_get:ncTF { #1 } { #2 } {
2116   \__bnvs_quark_if_no_value:cTF { #2 } {
2117     \__bnvs_fatal:n {Circular~definition:~#1}
2118     \prg_return_false:
2119   } {
2120     \prg_return_true:
2121   }
2122 } {
2123   \__bnvs_v_gput:nn { #1 } { \q_no_value }
2124   \__bnvs_value_resolve:ncTF { #1 } { #2 } {
2125     \__bnvs_v_gput:nv { #1 } { #2 }
2126     \prg_return_true:
2127   } {
2128     \__bnvs_first_resolve:ncTF { #1 } { #2 } {
2129       \__bnvs_v_gput:nv { #1 } { #2 }
2130       \prg_return_true:
2131     } {
2132       \__bnvs_last_resolve:ncTF { #1 } { #2 } {
2133         \__bnvs_v_gput:nv { #1 } { #2 }
2134         \prg_return_true:
2135       } {
2136         \__bnvs_v_gremove:n { #1 }
2137         \prg_return_false:
2138       }
2139     }
2140   }
2141 }
2142 }
2143 \BNVS_new_conditional_vc:cn { v_resolve } { T, F, TF }
2144 \BNVS_new_conditional_cpnn { v_append:nc } #1 #2 { T, F, TF } {
2145   \BNVS_begin:
2146   \__bnvs_v_resolve:ncTF { #1 } { #2 } {
2147     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2148     \prg_return_true:
2149   } {
2150     \BNVS_end:
2151     \prg_return_false:
2152   }
2153 }
2154 \BNVS_new_conditional_vc:cn { v_append } { T, F, TF }

```

<code>__bnvs_index_can:nTF</code>	<code>__bnvs_index_can:nTF {<key>} {<yes code>} {<no code>}</code>
<code>__bnvs_index_can:vTF</code>	<code>__bnvs_index_resolve:nncTF {<key>} {<integer>} <tl core name> {<yes code>}</code>
<code>__bnvs_index_resolve:nncTF</code>	<code>{<no code>}</code>
<code>__bnvs_index_resolve:vvcTF</code>	<code>__bnvs_index_append:nncTF {<key>} {<integer>} <tl core name> {<yes code>}</code>
<code>__bnvs_index_append:nncTF</code>	<code>{<no code>}</code>
<code>__bnvs_index_append:vvcTF</code>	

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

```

2155 \BNVS_new_conditional:cpnn { index_can:n } #1 { p, T, F, TF } {
2156   \bool_if:nTF {
2157     \__bnvs_if_in_p:nn V { #1 }
2158     || \__bnvs_if_in_p:nn A { #1 }
2159     || \__bnvs_if_in_p:nn Z { #1 }
2160   } {
2161     \prg_return_true:
2162   } {
2163     \prg_return_false:
2164   }
2165 }
2166 \BNVS_new_conditional:cpnn { index_can:v } #1 { p, T, F, TF } {
2167   \BNVS_tl_use:Nv \__bnvs_index_can:nTF { #1 } {
2168     \prg_return_true:
2169   } {
2170     \prg_return_false:
2171   }
2172 }
2173 \BNVS_new_conditional:cpnn { index_resolve:nnc } #1 #2 #3 { T, F, TF } {
2174   \exp_args:Nx \__bnvs_value_resolve:ncTF { #1.#2 } { #3 } {
2175     \prg_return_true:
2176   } {
2177     \__bnvs_first_resolve:ncTF { #1 } { #3 } {
2178       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2179       \__bnvs_round:c { #3 }
2180       \prg_return_true:

```

Limited overlay set.

```

2181   } {
2182     \__bnvs_last_resolve:ncTF { #1 } { #3 } {
2183       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2184       \__bnvs_round:c { #3 }
2185       \prg_return_true:
2186     } {
2187       \__bnvs_value_resolve:ncTF { #1 } { #3 } {
2188         \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2189         \__bnvs_round:c { #3 }
2190         \prg_return_true:
2191       } {

```

```

2192         \prg_return_false:
2193     }
2194 }
2195 }
2196 }
2197 }
2198 \BNVS_new_conditional:cpnn { index_resolve:nvc } #1 #2 #3 { T, F, TF } {
2199     \BNVS_tl_use:nv {
2200         \__bnvs_index_resolve:nncTF { #1 }
2201     } { #2 } { #3 } {
2202         \prg_return_true:
2203     } {
2204         \prg_return_false:
2205     }
2206 }
2207 \BNVS_new_conditional:cpnn { index_resolve:vvc } #1 #2 #3 { T, F, TF } {
2208     \BNVS_tl_use:nv {
2209         \BNVS_tl_use:Nv \__bnvs_index_resolve:nncTF { #1 }
2210     } { #2 } { #3 } {
2211         \prg_return_true:
2212     } {
2213         \prg_return_false:
2214     }
2215 }
2216 \BNVS_new_conditional:cpnn { index_append:nnc } #1 #2 #3 { T, F, TF } {
2217     \BNVS_begin:
2218     \__bnvs_index_resolve:nncTF { #1 } { #2 } { #3 } {
2219         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2220         \prg_return_true:
2221     } {
2222         \BNVS_end:
2223         \prg_return_false:
2224     }
2225 }
2226 \BNVS_new_conditional:cpnn { index_append:vvc } #1 #2 #3 { T, F, TF } {
2227     \BNVS_tl_use:nv {
2228         \BNVS_tl_use:Nv \__bnvs_index_append:nncTF { #1 }
2229     } { #2 } { #3 } {
2230         \prg_return_true:
2231     } {
2232         \prg_return_false:
2233     }
2234 }

```

6.13.9 Index counter

$\underline{\underline{\backslash_bnvs_n_resolve:ncTF}}$ $\underline{\underline{\backslash_bnvs_n_append:ncTF}}$ $\underline{\underline{\backslash_bnvs_n_append:VcTF}}$	$\backslash_bnvs_n_resolve:ncTF \{ \langle key \rangle \} \langle tl \ variable \rangle \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$ Evaluate the n counter associated to the $\{ \langle key \rangle \}$ overlay set into $\langle tl \ variable \rangle$. Initialize this counter to 1 on the first use. $\langle no \ code \rangle$ is never executed.
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

2235 \BNVS_new_conditional:cpnn { n_resolve:nc } #1 #2 { T, F, TF } {

```

```

2236 \__bnvs_n_get:ncF { #1 } { #2 } {
2237   \__bnvs_tl_set:cn { #2 } { 1 }
2238   \__bnvs_n_gput:nn { #1 } { 1 }
2239 }
2240 \prg_return_true:
2241 }
2242 \BNVS_new_conditional:cpnn { n_append:nc } #1 #2 { T, F, TF } {
2243   \BNVS_begin:
2244     \__bnvs_n_resolve:ncTF { #1 } { #2 } {
2245       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2246       \prg_return_true:
2247     } {
2248       \BNVS_end:
2249       \prg_return_false:
2250     }
2251 }
2252 \BNVS_new_conditional_vc:cn { n_append } { T, F, TF }

```

<pre> __bnvs_n_index_resolve:ncTF __bnvs_n_index_append:ncTF __bnvs_n_index_resolve:nncTF __bnvs_n_index_append:nncTF </pre>	<pre> __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>} </pre>
----------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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.

```

2253 \BNVS_new_conditional:cpnn { n_index_resolve:nc } #1 #2 { T, F, TF } {
2254   \__bnvs_n_resolve:ncTF { #1 } { #2 } {
2255     \__bnvs_index_resolve:nvcTF { #1 } { #2 } { #2 } {
2256       \prg_return_true:
2257     } {
2258       \prg_return_false:
2259     }
2260   } {
2261     \prg_return_false:
2262   }
2263 }
2264 \BNVS_new_conditional:cpnn { n_index_resolve:nnc } #1 #2 #3 { T, F, TF } {
2265   \__bnvs_n_resolve:ncTF { #1 } { #3 } {
2266     \__bnvs_tl_put_left:cn { #3 } { #2. }
2267     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2268       \prg_return_true:
2269     } {
2270       \prg_return_false:
2271     }
2272   } {

```

```

2273     \prg_return_false:
2274 }
2275 }
2276 \BNVS_new_conditional:cpnn { n_index_append:nc } #1 #2 { T, F, TF } {
2277     \BNVS_begin:
2278     \__bnvs_n_index_resolve:ncTF { #1 } { #2 } {
2279         \BNVS_end_tl_put_right:cv { #2 } { #2 }
2280         \prg_return_true:
2281     } {
2282         \BNVS_end:
2283         \prg_return_false:
2284     }
2285 }
2286 \BNVS_new_conditional:cpnn { n_index_append:nnc } #1 #2 #3 { T, F, TF } {
2287     \BNVS_begin:
2288     \__bnvs_n_index_resolve:nncTF { #1 } { #2 } { #3 } {
2289         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2290         \prg_return_true:
2291     } {
2292         \BNVS_end:
2293         \prg_return_false:
2294     }
2295 }
2296 \BNVS_new_conditional_vc:cn { n_index_append } { T, F, TF }
2297 \BNVS_new_conditional_vvc:cn { n_index_append } { T, F, TF }

```

6.13.10 Value counter

<u>__bnvs_v_incr_resolve:nncTF</u>	__bnvs_v_incr_append:nnTF {<key>} {<offset>} {<yes code>} {<no code>}
<u>__bnvs_v_incr_append:nncTF</u>	__bnvs_v_incr_resolve:nncTF {<key>} {<offset>} {<tl core name>} {<yes code>} {<no code>}
<u>__bnvs_v_incr_append:(VnN VVN)TF</u>	__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.

```

2298 \BNVS_new_conditional:cpnn { v_incr_resolve:nnc } #1 #2 #3 { T, F, TF } {
2299     \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2300         \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2301             \__bnvs_v_resolve:ncTF { #1 } { #3 } {
2302                 \prg_return_true:
2303             } {
2304                 \prg_return_false:
2305             }
2306         } {
2307             \__bnvs_tl_put_right:cn { #3 } { + }
2308             \__bnvs_v_append:ncTF { #1 } { #3 } {
2309                 \__bnvs_round:c { #3 }
2310                 \__bnvs_v_gput:nv { #1 } { #3 }
2311             } \prg_return_true:
2312         } {

```

```

2313         \prg_return_false:
2314     }
2315 }
2316 } {
2317     \prg_return_false:
2318 }
2319 }
2320 \BNVS_new_conditional_vnc:cn { v_incr_resolve } { T, F, TF }
2321 \BNVS_new_conditional_cpnn { v_incr_append:nnc } #1 #2 #3 { T, F, TF } {
2322     \BNVS_begin:
2323     \__bnvs_v_incr_resolve:nncTF { #1 } { #2 } { #3 } {
2324         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2325         \prg_return_true:
2326     } {
2327         \prg_return_false:
2328     }
2329 }
2330 \BNVS_new_conditional_vnc:cn { v_incr_append } { T, F, TF }
2331 \BNVS_new_conditional_vvc:cn { v_incr_append } { T, F, TF }
2332 \BNVS_new_conditional_cpnn { v_post_resolve:nnc } #1 #2 #3 { T, F, TF } {
2333     \__bnvs_v_resolve:ncTF { #1 } { #3 } {
2334         \BNVS_begin:
2335         \__bnvs_if_resolve:ncTF { #2 } { a } {
2336             \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
2337                 \BNVS_end:
2338                 \prg_return_true:
2339             } {
2340                 \__bnvs_tl_put_right:cn { a } { + }
2341                 \__bnvs_tl_put_right:cv { a } { #3 }
2342                 \__bnvs_round:c { a }
2343                 \BNVS_end_v_gput:nc { #1 } { a }
2344                 \prg_return_true:
2345             }
2346         } {
2347             \BNVS_end:
2348             \prg_return_false:
2349         }
2350     } {
2351         \prg_return_false:
2352     }
2353 }
2354 \BNVS_new_conditional_vvc:cn { v_post_resolve } { T, F, TF }
2355 \BNVS_new_conditional_cpnn { v_post_append:nnc } #1 #2 #3 { T, F, TF } {
2356     \BNVS_begin:
2357     \__bnvs_v_post_resolve:nncTF { #1 } { #2 } { #3 } {
2358         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2359         \prg_return_true:
2360     } {
2361         \prg_return_true:
2362     }
2363 }
2364 \BNVS_new_conditional_vnc:cn { v_post_append } { T, F, TF }
2365 \BNVS_new_conditional_vvc:cn { v_post_append } { T, F, TF }

```

<code>_bnvs_n_incr_resolve:nncTF</code>	<code>_bnvs_n_incr_resolve:nncTF {<key>} {<base key>} {<offset>} {<tl core name>} {<yes code>} {<no code>}</code>
<code>_bnvs_n_incr_resolve:vnncTF</code>	
<code>_bnvs_n_incr_resolve:nncTF</code>	<code>_bnvs_n_incr_resolve:nncTF {<key>} {<offset>} {<tl core name>} {<yes code>} {<no code>}</code>
<code>_bnvs_n_incr_resolve:vvcTF</code>	
<code>_bnvs_n_incr_append:nnncTF</code>	<code>_bnvs_n_incr_append:nnncTF {<key>} {<base key>} {<offset>} {<tl core name>} {<yes code>} {<no code>}</code>
<code>_bnvs_n_incr_append:nncTF</code>	
<code>_bnvs_n_incr_append:(vnc vvc)TF</code>	<code>_bnvs_n_incr_append:nncTF {<key>} {<offset>} {<tl core name>} {<yes code>} {<no code>}</code>
<code>_bnvs_n_post_resolve:nncTF</code>	
<code>_bnvs_n_post_append:nncTF</code>	

Increment the implicit n counter accordingly. When requested, put the resulting index in the variable with `<tl core name>`.

```

2366 \BNVS_new_conditional:cpnn { n_incr_resolve:nnc } #1 #2 #3 #4 { T, F, TF } {
2367   \_bnvs_if_resolve:ncTF { #3 } { #4 } {
2368     \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
2369       \_bnvs_n_resolve:ncTF { #1 } { #4 } {
2370         \_bnvs_index_resolve:nvcTF { #1 } { #4 } { #4 } {
2371           \prg_return_true:
2372         } {
2373           \prg_return_false:
2374         }
2375       } {
2376         \prg_return_false:
2377       }
2378     } {
2379       \_bnvs_tl_put_right:cn { #4 } { + }
2380       \_bnvs_n_append:ncTF { #1 } { #4 } {
2381         \_bnvs_round:c { #4 }
2382         \_bnvs_n_gput:nv { #1 } { #4 }
2383         \_bnvs_index_resolve:nvcTF { #2 } { #4 } { #4 } {
2384           \prg_return_true:
2385         } {
2386           \prg_return_false:
2387         }
2388       } {
2389         \prg_return_false:
2390       }
2391     }
2392   } {
2393     \prg_return_false:
2394   }
2395 }
2396 \BNVS_new_conditional:cpnn { n_incr_resolve:nnc } #1 #2 #3 { T, F, TF } {
2397   \_bnvs_if_resolve:ncTF { #2 } { #3 } {
2398     \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2399       \_bnvs_n_resolve:ncTF { #1 } { #3 } {
2400         \_bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2401           \prg_return_true:
2402         } {
2403           \prg_return_false:
2404         }
2405       } {

```



```

2406         \prg_return_false:
2407     }
2408 } {
2409     \__bnvs_tl_put_right:cn { #3 } { + }
2410     \__bnvs_n_append:ncTF { #1 } { #3 } {
2411         \__bnvs_round:c { #3 }
2412         \__bnvs_n_gput:nv { #1 } { #3 }
2413         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2414             \prg_return_true:
2415         } {
2416             \prg_return_false:
2417         }
2418     } {
2419         \prg_return_false:
2420     }
2421 }
2422 } {
2423     \prg_return_false:
2424 }
2425 }
2426 \BNVS_new_conditional_vnc:cn { n_incr_resolve } { T, F, TF }
2427 \BNVS_new_conditional_vvc:cn { n_incr_resolve } { T, F, TF }
2428 \BNVS_new_conditional:cpnn { n_incr_append:nnnc } #1 #2 #3 #4 { T, F, TF } {
2429     \BNVS_begin:
2430     \__bnvs_n_incr_resolve:nnncTF { #1 } { #2 } { #3 } { #4 }{
2431         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2432         \prg_return_true:
2433     } {
2434         \BNVS_end:
2435         \prg_return_false:
2436     }
2437 }
2438 \BNVS_new_conditional_vvnc:cn { n_incr_append } { T, F, TF }
2439 \BNVS_new_conditional_vvvc:cn { n_incr_append } { T, F, TF }
2440 \BNVS_new_conditional:cpnn { n_incr_append:nnc } #1 #2 #3 { T, F, TF } {
2441     \BNVS_begin:
2442     \__bnvs_n_incr_resolve:nncTF { #1 } { #2 } { #3 } {
2443         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2444         \prg_return_true:
2445     } {
2446         \BNVS_end:
2447         \prg_return_false:
2448     }
2449 }
2450 \BNVS_new_conditional_vnc:cn { n_incr_append } { T, F, TF }
2451 \BNVS_new_conditional_vvc:cn { n_incr_append } { T, F, TF }

```

__bnvs_v_post_resolve:nncTF	__bnvs_v_post_resolve:nncTF {<key>} {<offset>} <tl variable> {<yes
__bnvs_v_post_resolve:vvcTF	code>} {<no code>}
__bnvs_v_post_append:nncTF	__bnvs_v_post_append:nncTF {<key>} {<offset>} <tl variable> {<yes
__bnvs_v_post_append:(vnN vvN)TF	code>} {<no code>}

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

```

2452 \BNVS_new_conditional:cpnn { n_post_resolve:nnc } #1 #2 #3 { T, F, TF } {
2453   \__bnvs_n_resolve:ncTF { #1 } { #3 } {
2454     \BNVS_begin:
2455     \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2456       \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2457         \BNVS_end:
2458         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2459           \prg_return_true:
2460         } {
2461           \prg_return_false:
2462         }
2463       } {
2464         \__bnvs_tl_put_right:cn { #3 } { + }
2465         \__bnvs_n_append:ncTF { #1 } { #3 } {
2466           \__bnvs_round:c { #3 }
2467           \__bnvs_n_gput:nv { #1 } { #3 }
2468         \BNVS_end:
2469         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2470           \prg_return_true:
2471         } {
2472           \prg_return_false:
2473         }
2474       } {
2475         \BNVS_end:
2476         \prg_return_false:
2477       }
2478     }
2479   } {
2480     \BNVS_end:
2481     \prg_return_false:
2482   }
2483 } {
2484   \prg_return_false:
2485 }
2486 }
2487 \BNVS_new_conditional:cpnn { n_post_append:nnc } #1 #2 #3 { T, F, TF } {
2488   \BNVS_begin:
2489   \__bnvs_n_post_resolve:nncTF { #1 } { #2 } { #3 } {
2490     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2491     \prg_return_true:
2492   } {
2493     \BNVS_end:
2494     \prg_return_false:
2495   }
2496 }
2497 \BNVS_new_conditional_vnc:cn { n_post_append } { T, F, TF }
2498 \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.

```
2499 \BNVS_set:cpn { round_ans: } {
2500   \_bnvs_round:c { ans }
2501 }
```

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 T_EX 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⟩}`

```
2502 \BNVS_new:cpn { path_resolve_n:TFF } #1 #2 {
2503   \_bnvs_kip_n_path_resolve:TF {
2504     \_bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
2505   }
2506 }
```

`_bnvs_path_resolve_n:T` `_bnvs_path_resolve_n:T {⟨yes code⟩}`

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

```
2507 \BNVS_new:cpn { if_resolve_end_return_false:n } #1 {
2508   \BNVS_end:
2509   \prg_return_false:
2510 }
2511 \BNVS_new:cpn { path_resolve_n:T } #1 {
2512   \_bnvs_path_resolve_n:TFF {
2513     #1
2514   } {
2515     \_bnvs_if_resolve_end_return_false:n {
2516       Too~many~dotted~components
2517     }
2518   } {
2519     \_bnvs_if_resolve_end_return_false:n {
2520       Unknown~dotted~path
2521     }
2522   }
2523 }
```

```

2524 \BNVS_set:cpn { resolve_x:T } #1 {
2525   \__bnvs_kip_x_path_resolve:TFF {
2526     #1
2527   } {
2528     \__bnvs_if_resolve_end_return_false:n {
2529       Too-many-dotted-components
2530     }
2531   } {
2532     \__bnvs_if_resolve_end_return_false:n { Unknown-dotted-path }
2533   }
2534 }

```

```

\__bnvs_path_suffix:nTF \__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.

```

2535 \BNVS_new_conditional:cpnn { path_pop_right_n:c } #1 { T, F, TF } {
2536   \__bnvs_seq_pop_right:ccTF { path } { #1 }
2537   { \prg_return_true: } { \prg_return_false: }
2538 }

```

<pre> __bnvs_if_resolve_pop_kip:TTF __bnvs_if_resolve_pop_complete_white:T __bnvs_if_resolve_pop_complete_black:T </pre>	<pre> __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>} </pre>
-----------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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>*.

```

2539 \BNVS_new:cpn { if_resolve_pop_kip_complete: } {
2540   \__bnvs_tl_if_blank:vT { id } {
2541     \__bnvs_tl_put_left:cv { key } { id_last }
2542     \__bnvs_tl_set:cv { id } { id_last }
2543   }
2544   \__bnvs_tl_if_blank:vTF { path } {
2545     \__bnvs_seq_clear:c { path }
2546   } {
2547     \__bnvs_seq_set_split:cnv { path } { . } { path }
2548     \__bnvs_seq_remove_all:cn { path } { }
2549   }
2550   \__bnvs_tl_set_eq:cc { key_base } { key }
2551   \__bnvs_seq_set_eq:cc { path_base } { path }
2552 }
2553 \BNVS_new:cpn { if_resolve_pop_kip:TTF } #1 #2 #3 {

```

```

2554 \__bnvs_split_pop_left:cTF { key } {
2555   \__bnvs_split_pop_left:cTF { id } {
2556     \__bnvs_split_pop_left:cTF { path } {
2557       \__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.

```

2558         #1
2559     } {
2560     \__bnvs_if_resolve_pop_kip_complete:
2561     #2
2562     }
2563   } {
2564   \__bnvs_end_unreachable_return_false:n { if_resolve_pop_kip:TTF/2 }
2565   }
2566   } {
2567   \__bnvs_end_unreachable_return_false:n { if_resolve_pop_kip:TTF/1 }
2568   }
2569   } { #3 }
2570 }

```

```

\__bnvs_if_resolve_pop_complete:nNT \__bnvs_if_resolve_pop_kip:FFTF {<empty key code>} {<no id code>}
{<true code>} {<no capture code>}
\__bnvs_if_resolve_pop_complete:nNT {<tl>} {<tl var>} {<true 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 *<true 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.

```

2571 \BNVS_new:cpn { if_resolve_pop_complete_white:T } #1 {
2572   \__bnvs_split_pop_left:cTF { n_incr } {
2573     \__bnvs_split_pop_left:cTF { incr } {
2574       \__bnvs_split_pop_left:cTF { post } {
2575         #1
2576       } {
2577       \__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/3 }
2578       }
2579     } {
2580     \__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/2 }
2581     }
2582   } {
2583   \__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/1 }
2584   }
2585 }
2586 \BNVS_new:cpn { if_resolve_pop_complete_black:T } #1 {
2587   \__bnvs_split_pop_left:cTF { a } {
2588     \__bnvs_split_pop_left:cTF { a } {

```

```

2589     \_bnvs_split_pop_left:cTF { a } {
2590         \_bnvs_split_pop_left:cTF { a } {
2591             \_bnvs_split_pop_left:cTF { a } {
2592                 \_bnvs_split_pop_left:cTF { a } {
2593                     #1
2594                 } {
2595     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/6 }
2596         }
2597     } {
2598     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/5 }
2599         }
2600     } {
2601     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/4 }
2602         }
2603     } {
2604     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/3 }
2605         }
2606     } {
2607     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/2 }
2608         }
2609     } {
2610     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/1 }
2611         }
2612 }

```

<u><code>__bnvs_if_resolve:ncTF</code></u>	<code>__bnvs_if_append:ncTF {⟨expression⟩} ⟨tl variable⟩ {⟨yes code⟩} {⟨no code⟩}</code>
<u><code>__bnvs_if_resolve:vcTF</code></u>	
<u><code>__bnvs_if_append:ncTF</code></u>	Evaluates the <i>⟨expression⟩</i> , replacing all the named overlay specifications by their static counterpart then put the rounded result in <i>⟨tl variable⟩</i> when resolving or to the right of the <i>⟨tl variable⟩</i> when appending. Executed within a group. Heavily used by <code>__bnvs_query_eval:nc</code> , where <i>⟨integer expression⟩</i> was initially enclosed inside ‘ <i>?(...)</i> ’. Local variables:
<u><code>__bnvs_if_append:(vc xc)TF</code></u>	
<code>\l__bnvs_ans_tl</code>	To feed <i>⟨tl variable⟩</i> with. (End definition for <code>\l__bnvs_ans_tl</code> .)
<code>\l__bnvs_split_seq</code>	The sequence of caught query groups and non queries. (End definition for <code>\l__bnvs_split_seq</code> .)
<code>\l__bnvs_split_int</code>	Is the index of the non queries, before all the caught groups. (End definition for <code>\l__bnvs_split_int</code> .)
	2613 <code>\BNVS_int_new:c { split }</code>
<code>\l__bnvs_key_tl</code>	Storage for <code>split</code> sequence items that represent names. (End definition for <code>\l__bnvs_key_tl</code> .)
<code>\l__bnvs_path_tl</code>	Storage for <code>split</code> sequence items that represent integer paths. (End definition for <code>\l__bnvs_path_tl</code> .)
	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 <code>\...end_return...</code> functions.
	2614 <code>\BNVS_new:cpn { kip_x_path_resolve_or_end_return_false:nt } #1 #2 {</code>
	2615 <code> __bnvs_kip_x_path_resolve:TFF {</code>
	2616 <code> #2</code>
	2617 <code> } {</code>
	2618 <code> \BNVS_end_return_false:x { Too-many-dotted-components:~#1 }</code>
	2619 <code> } {</code>
	2620 <code> \BNVS_end_return_false:x { Unknown-dotted-path:~#1 }</code>
	2621 <code> }</code>
	2622 <code>}</code>
	2623 <code>\BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { T, F, TF } {</code>
	2624 <code> \BNVS_begin:</code>
	2625 <code> __bnvs_if_resolve:ncTF { #1 } { #2 } {</code>
	2626 <code> \BNVS_end_tl_put_right:cv { #2 } { #2 }</code>
	2627 <code> ⟨*!final⟩</code>
	2628 <code> \BNVS_DEBUG_log_if_append_ncTF:nn { ... } { ...TRUE }</code>
	2629 <code> ⟨/!final⟩</code>
	2630 <code> \prg_return_true:</code>
	2631 <code> } {</code>
	2632 <code> \BNVS_end:</code>
	2633 <code> ⟨*!final⟩</code>
	2634 <code> \BNVS_DEBUG_log_if_append_ncTF:nn { ... } { ...FALSE }</code>
	2635 <code> ⟨/!final⟩</code>
	2636 <code> \prg_return_false:</code>
	2637 <code> }</code>

```

2638 }
2639 \BNVS_new:cpn { end_unreachable_return_false:n } #1 {
2640   \__bnvs_error:x { UNREACHABLE/#1 }
2641   \BNVS_end:
2642   \prg_return_false:
2643 }
2644 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { T, F, TF } {
2645   \__bnvs_call:TF {
2646     \BNVS_begin:
2647     \BNVS_set:cpn { if_resolve_warning:n } ##1 {
2648       \__bnvs_warning:n { #1:~##1 }
2649       \BNVS_set:cpn { if_resolve_warning:n } {
2650         \use_none:n
2651       }
2652     }

```

This \TeX group will be closed just before returning. Implementation:

```

2653   \__bnvs_regex_split:cnTF { split } { #1 } {

```

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

```

2654     \BNVS_set:cpn { if_resolve_end_return_true: } {

```

Normal and unique end of the loop.

```

2655       \__bnvs_if_resolve_round_ans:
2656       \BNVS_tl_use:nv {
2657         \BNVS_end:
2658         \__bnvs_tl_set:cn { #2 }
2659       } { ans }
2660       \prg_return_true:
2661     }
2662     \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round_ans: }
2663     \__bnvs_tl_clear:c { ans }
2664     \__bnvs_if_resolve_loop_or_end_return:
2665   } {
2666     \__bnvs_tl_clear:c { ans }
2667     \__bnvs_round_ans:n { #1 }
2668     \BNVS_end_tl_set:cv { #2 } { ans }
2669     \prg_return_true:
2670   }
2671 } {
2672   \__bnvs_error:n { TOO_MANY_NESTED_CALLS/Resolution }
2673   \prg_return_false:
2674 }
2675 }
2676 \BNVS_new_conditional:cpnn { if_append:vc } #1 #2 { T, F, TF } {
2677   \BNVS_tl_use:Nv \__bnvs_if_append:ncTF { #1 } { #2 } {
2678     \prg_return_true:
2679   } {
2680     \prg_return_false:
2681   }
2682 }
2683 \BNVS_new_conditional:cpnn { if_resolve:vc } #1 #2 { T, F, TF } {
2684   \BNVS_tl_use:Nv \__bnvs_if_resolve:ncTF { #1 } { #2 } {
2685     \prg_return_true:

```



```

2686 } {
2687   \prg_return_false:
2688 }
2689 }

```

Next functions are helpers for the `__bnvs_if_resolve:nc` conditional variants. When present, their two first arguments $\langle tl \rangle$ and $\langle tl\ var \rangle$ 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.

```

2690 \BNVS_new:cpn { if_resolve_loop_or_end_return: } {
2691   \__bnvs_split_pop_left:cTF { a } {
2692     \__bnvs_tl_put_right:cv { ans } { a }
2693     \__bnvs_if_resolve_pop_kip:TTF {
2694       \__bnvs_if_resolve_pop_kip:TTF {
2695         \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/3 }
2696       } {
2697         \__bnvs_if_resolve_pop_complete_white:T {
2698           \__bnvs_tl_if_blank:vTF { n_incr } {
2699             \__bnvs_tl_if_blank:vTF { incr } {
2700               \__bnvs_tl_if_blank:vTF { post } {
2701                 \__bnvs_if_resolve_value_loop_or_end_return_true:F {

```

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

```

2702           \__bnvs_seq_pop_right:ccTF { path } { a } {
2703             \BNVS_tl_use:Nv \str_case:nnF { a } {
2704 { n          } { \BNVS_use:c { if_resolve_loop_or_end_return[.n]: } }
2705 { length    } { \BNVS_use:c { if_resolve_loop_or_end_return[.length]: } }
2706 { last      } { \BNVS_use:c { if_resolve_loop_or_end_return[.last]: } }
2707 { range     } { \BNVS_use:c { if_resolve_loop_or_end_return[.range]: } }
2708 { previous  } { \BNVS_use:c { if_resolve_loop_or_end_return[.previous]: } }
2709 { next      } { \BNVS_use:c { if_resolve_loop_or_end_return[.next]: } }
2710 { reset     } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset]: } }
2711 { reset_all } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset_all]: } }
2712           } {
2713 \BNVS_use:c { if_resolve_loop_or_end_return[...<integer>]: }
2714           }
2715           } {
2716 \BNVS_use:c { if_resolve_loop_or_end_return[...]: }
2717           }
2718           }
2719           } {
2720 \BNVS_use:c { if_resolve_loop_or_end_return[...++]: }
2721           }
2722           } {
2723   \__bnvs_path_suffix:nTF { n } {
2724 \BNVS_use:c { if_resolve_loop_or_end_return[...n+=...]: }
2725           } {
2726 \BNVS_use:c { if_resolve_loop_or_end_return[...+=...]: }
2727           }
2728           }
2729           } {

```

```

2730 \BNVS_use:c { if_resolve_loop_or_end_return[...++n]: }
2731     }
2732     }
2733     } {
2734 % split sequence empty
2735 \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/2 }
2736     }
2737     } {
2738     \__bnvs_if_resolve_pop_complete_black:T {
2739     \__bnvs_path_suffix:nTF { n } {
2740 \BNVS_use:c { if_resolve_loop_or_end_return[+...n]: }
2741     } {
2742 \BNVS_use:c { if_resolve_loop_or_end_return[+...]: }
2743     }
2744     }
2745     } {
2746     \__bnvs_if_resolve_end_return_true:
2747     }
2748     } {
2749 \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/1 }
2750     }
2751 }
2752 \BNVS_set:cpn { if_resolve_value_loop_or_end_return_true:F } #1 {
2753     \__bnvs_tl_set:cx { a } {
2754     \BNVS_tl_use:c { key } \BNVS_tl_use:c { path }
2755     }
2756     \__bnvs_v_resolve:vcTF { a } { a } {
2757     \__bnvs_tl_put_right:cv { ans } { a }
2758     \__bnvs_if_resolve_loop_or_end_return:
2759     } {
2760     \__bnvs_value_resolve:vcTF { a } { a } {
2761     \__bnvs_tl_put_right:cv { ans } { a }
2762     \__bnvs_if_resolve_loop_or_end_return:
2763     } {
2764     #1
2765     }
2766     }
2767 }
2768 \BNVS_new:cpn { end_return_error:n } #1 {
2769     \__bnvs_error:n { #1 }
2770     \BNVS_end:
2771     \prg_return_false:
2772 }
2773 \BNVS_new:cpn { if_resolve_loop_or_end_return[.n]: } {

```

- Case ...n.

```

2774 \__bnvs_path_resolve_n:T {
2775     \__bnvs_base_resolve_n:
2776     \__bnvs_n_index_append:vcTF { key } { key_base } { ans } {
2777     \__bnvs_if_resolve_loop_or_end_return:
2778     } {
2779     \__bnvs_end_return_error:n {
2780     Undefined-dotted-path

```

```

2781     }
2782   }
2783 }
2784 }

2785 \BNVS_new_conditional:cpnn { path_suffix:n } #1 { T, F, TF } {
2786   \__bnvs_seq_get_right:ccTF { path } { a } {
2787     \__bnvs_tl_if_eq:cnTF { a } { #1 } {
2788       \__bnvs_seq_pop_right:ccT { path } { a } { }
2789       \prg_return_true:
2790     } {
2791       \prg_return_false:
2792     }
2793   } {
2794     \prg_return_false:
2795   }
2796 }
2797 \BNVS_new:cpn { if_resolve_loop_or_end_return[.length]: } {

```

- Case ...length.

```

2798   \__bnvs_path_resolve_n:T {
2799     \__bnvs_length_append:vcTF { key } { ans } {
2800 % \begin{bnvs.gobble}
2801 <*\final>
2802 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../length }
2803 </!\final>
2804 % \end{bnvs.gobble}
2805     \__bnvs_if_resolve_loop_or_end_return:
2806   } {
2807     \__bnvs_if_resolve_end_return_false:n { NO~length }
2808   }
2809 }
2810 }
2811 \BNVS_new:cpn { if_resolve_loop_or_end_return[.last]: } {

```

- Case ...last.

```

2812   \__bnvs_path_resolve_n:T {
2813     \__bnvs_last_append:vcTF { key } { ans } {
2814 % \begin{bnvs.gobble}
2815 <*\final>
2816 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../last }
2817 </!\final>
2818 % \end{bnvs.gobble}
2819     \__bnvs_if_resolve_loop_or_end_return:
2820   } {
2821     \BNVS_end_return_false:x { NO~last }
2822   }
2823 }
2824 }
2825 \BNVS_new:cpn { if_resolve_loop_or_end_return[.range]: } {

```

- Case ...range.

```

2826 \__bnvs_path_resolve_n:T {
2827   \__bnvs_range_append:vcTF { key } { ans } {
2828     \BNVS_set:cpn { if_resolve_round_ans: } { \prg_do_nothing: }
2829 % \begin{bnvs.gobble}
2830 <*\final>
2831 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../range }
2832 </!\final>
2833 % \end{bnvs.gobble}
2834   \__bnvs_if_resolve_loop_or_end_return:
2835   } {
2836     \__bnvs_if_resolve_end_return_false:n { NO~range }
2837   }
2838 }
2839 }
2840 \BNVS_new:cpn { if_resolve_loop_or_end_return[.previous]: } {

```

- Case ...previous.

```

2841 \__bnvs_path_resolve_n:T {
2842   \__bnvs_previous_append:vcTF { key } { ans } {
2843 % \begin{bnvs.gobble}
2844 <*\final>
2845 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../previous }
2846 </!\final>
2847 % \end{bnvs.gobble}
2848   \__bnvs_if_resolve_loop_or_end_return:
2849   } {
2850     \__bnvs_if_resolve_end_return_false:n { NO~previous }
2851   }
2852 }
2853 }
2854 \BNVS_new:cpn { if_resolve_loop_or_end_return[.next]: } {

```

- Case ...next.

```

2855 \__bnvs_path_resolve_n:T {
2856   \__bnvs_next_append:vcTF { key } { ans } {
2857 % \begin{bnvs.gobble}
2858 <*\final>
2859 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../next }
2860 </!\final>
2861 % \end{bnvs.gobble}
2862   \__bnvs_if_resolve_loop_or_end_return:
2863   } {
2864     \__bnvs_if_resolve_end_return_false:n { NO~next }
2865   }
2866 }
2867 }
2868 \BNVS_new:cpn { if_resolve_loop_or_end_return[.reset]: } {

```

- Case ...reset.

```

2869 \__bnvs_path_resolve_n:T {
2870   \__bnvs_v_greset:vnT { key } { } { }
2871   \__bnvs_value_append:vcTF { key } { ans } {

```

```

2872 % \begin{bnvs.gobble}
2873 <*\final>
2874 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../reset }
2875 </\final>
2876 % \end{bnvs.gobble}
2877     \__bnvs_if_resolve_loop_or_end_return:
2878     } {
2879     \__bnvs_if_resolve_end_return_false:n { NO~reset }
2880     }
2881 }
2882 }
2883 \BNVS_new:cpn { if_resolve_loop_or_end_return[.reset_all]: } {

    • Case ...reset_all.

2884     \__bnvs_path_resolve_n:T {
2885     \__bnvs_greset_all:vnT { key } { } { }
2886     \__bnvs_value_append:vcTF { key } { ans } {
2887 % \begin{bnvs.gobble}
2888 <*\final>
2889 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../reset }
2890 </\final>
2891 % \end{bnvs.gobble}
2892     \__bnvs_if_resolve_loop_or_end_return:
2893     } {
2894     \__bnvs_if_resolve_end_return_false:n { NO~reset }
2895     }
2896 }
2897 }
2898 \BNVS_set:cpn { if_resolve_loop_or_end_return[...<integer>]: } {

    • Case ...<integer>.

2899     \__bnvs_path_resolve_n:T {
2900     \__bnvs_index_append:vcTF { key } { a } { ans } {
2901 % \begin{bnvs.gobble}
2902 <*\final>
2903 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../<integer> }
2904 </\final>
2905 % \end{bnvs.gobble}
2906     \__bnvs_if_resolve_loop_or_end_return:
2907     } {
2908     \__bnvs_if_resolve_end_return_false:n { NO~integer }
2909     }
2910 }
2911 }
2912 \BNVS_set:cpn { if_resolve_loop_or_end_return[...]: } {

    • Case ....

2913     \__bnvs_path_resolve_n:T {
2914     \__bnvs_value_append:vcTF { key } { ans } {
2915 % \begin{bnvs.gobble}
2916 <*\final>
2917 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../... }
2918 </\final>

```

```

2919 % \end{bnvs.gobble}
2920   \__bnvs_if_resolve_loop_or_end_return:
2921   } {
2922     \__bnvs_if_resolve_end_return_false:n { NO~value }
2923   }
2924 }
2925 }
2926 \BNVS_set:cpn { if_resolve_loop_or_end_return[...++]: } {

• Case ...++.

2927 \__bnvs_path_suffix:nTF { reset } {
2928   \__bnvs_path_resolve_n:T {
2929     \__bnvs_v_greset:vnT { key } { } { }
2930     \__bnvs_v_post_append:vncTF { key } { 1 } { ans } {
2931       \__bnvs_if_resolve_loop_or_end_return:
2932     } {
2933       \__bnvs_if_resolve_end_return_false:n { NO~post }
2934     }
2935   }
2936 } {
2937   \__bnvs_path_suffix:nTF { reset_all } {
2938     \__bnvs_path_resolve_n:T {
2939       \__bnvs_greset_all:vnT { key } { } { }
2940       \__bnvs_v_post_append:vncTF { key } { 1 } { ans } {
2941         \__bnvs_if_resolve_loop_or_end_return:
2942       } {
2943         \__bnvs_if_resolve_end_return_false:n { NO~post }
2944       }
2945     }
2946 } {
2947   \__bnvs_path_resolve_n:T {
2948     \__bnvs_v_post_append:vncTF { key } { 1 } { ans } {
2949       \__bnvs_if_resolve_loop_or_end_return:
2950     } {
2951       \__bnvs_if_resolve_end_return_false:n { NO~post }
2952     }
2953   }
2954 }
2955 }
2956 }
2957 \BNVS_set:cpn { if_resolve_loop_or_end_return[...n+=...]: } {

• Case ....n+=integer.

2958 \__bnvs_path_resolve_n:T {
2959   \__bnvs_base_resolve_n:
2960   \__bnvs_n_incr_append:vvvcTF { key } { key_base } { incr } { ans } {
2961 % \begin{bnvs.gobble}
2962 <{*!final}
2963 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../...n+=... }
2964 </!final}
2965 % \end{bnvs.gobble}
2966   \__bnvs_if_resolve_loop_or_end_return:
2967   } {

```

```

2968     \__bnvs_if_resolve_end_return_false:n {
2969         NO~n~incrementation
2970     }
2971 }
2972 }
2973 }
2974 \BNVS_set:cpn { if_resolve_loop_or_end_return[...+=...]: } {
    • Case A+=<integer>.

2975     \__bnvs_path_resolve_n:T {
2976         \__bnvs_v_incr_append:vcTF { key } { incr } { ans } {
2977             % \begin{bnvs.gobble}
2978             <!*final>
2979             \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../...+=... }
2980             </!final>
2981             % \end{bnvs.gobble}
2982             \__bnvs_if_resolve_loop_or_end_return:
2983             } {
2984                 \__bnvs_if_resolve_end_return_false:n {
2985                     NO~incremented~value
2986                 }
2987             }
2988         }
2989     }
2990 \BNVS_new:cpn { base_resolve_n: } {
2991     \__bnvs_seq_if_empty:cF { path_base } {
2992         \__bnvs_seq_pop_right:cc { path_base } { a }
2993         \__bnvs_seq_if_empty:cF { path_base } {
2994             \__bnvs_tl_put_right:cx { key_base } {
2995                 . \__bnvs_seq_use:cn { path_base } { . }
2996             }
2997         }
2998     }
2999 }
3000 \BNVS_new:cpn { base_resolve: } {
3001     \__bnvs_seq_if_empty:cF { path_base } {
3002         \__bnvs_tl_put_right:cx { key_base } {
3003             . \__bnvs_seq_use:cn { path_base } { . }
3004         }
3005     }
3006 }
3007 \BNVS_new:cpn { if_resolve_loop_or_end_return[...++n]: } {
    • Case ...++n.

3008     \__bnvs_path_resolve_n:T {
3009         \__bnvs_base_resolve:
3010         \__bnvs_n_incr_append:vnCTF { key } { key_base } { 1 } { ans } {
3011             \__bnvs_if_resolve_loop_or_end_return:
3012             } {
3013                 \__bnvs_if_resolve_end_return_false:n { NO~...++n }
3014             }
3015         }
3016     }
3017 \BNVS_set:cpn { if_resolve_loop_or_end_return[++...n]: } {

```

- Case ++...n.

```

3018 \__bnvs_path_resolve_n:T {
3019   \__bnvs_base_resolve_n:
3020   \__bnvs_n_incr_append:vvncTF { key } { key_base } { 1 } { ans } {
3021     \__bnvs_if_resolve_loop_or_end_return:
3022   } {
3023     \__bnvs_if_resolve_end_return_false:n { NO~++...n }
3024   }
3025 }
3026 }
3027 \BNVS_new:cpn { if_resolve_loop_or_end_return[++...]: } {

```

- Case ++....

```

3028 \__bnvs_path_suffix:nTF { reset } {
3029   \__bnvs_path_resolve_n:T {
3030     \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
3031 % \begin{bnvs.gobble}
3032 <!*final>
3033 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../++...reset }
3034 </!final>
3035 % \end{bnvs.gobble}
3036 % \begin{macrocode}
3037   \__bnvs_v_greset:vnT { key } { } { }
3038   \__bnvs_if_resolve_loop_or_end_return:
3039 } {
3040   \__bnvs_v_greset:vnT { key } { } { }
3041   \__bnvs_if_resolve_end_return_false:n { No~increment }
3042 }
3043 }
3044 } {
3045   \__bnvs_path_suffix:nTF { reset_all } {
3046     \__bnvs_path_resolve_n:T {
3047       \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
3048 % \begin{bnvs.gobble}
3049 <!*final>
3050 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../++...reset_all }
3051 </!final>
3052 % \end{bnvs.gobble}
3053 % \begin{macrocode}
3054   \__bnvs_greset_all:vnT { key } { } { }
3055   \__bnvs_if_resolve_loop_or_end_return:
3056 } {
3057   \__bnvs_greset_all:vnT { key } { } { }
3058   \__bnvs_if_resolve_end_return_false:n { No~increment }
3059 }
3060 }
3061 } {
3062   \__bnvs_path_resolve_n:T {
3063     \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
3064 % \begin{bnvs.gobble}
3065 <!*final>
3066 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../++... }
3067 </!final>

```



```

3068 % \end{bnvs.gobble}
3069 % \begin{macrocode}
3070     \_bnvs_if_resolve_loop_or_end_return:
3071     } {
3072     \_bnvs_if_resolve_end_return_false:n { No~increment }
3073     }
3074   }
3075 }
3076 }
3077 }

```

`__bnvs_query_eval:ncTF` `__bnvs_query_eval:ncTF {⟨overlay query⟩} {⟨tl core⟩} {⟨yes code⟩} {⟨no code⟩}`

Evaluates the single $\langle overlay\ query \rangle$, 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 definition for \l__bnvs_V_tl.)

`\l__bnvs_TEST_A_tl` Storage for the first component of a range.
(End definition for \l__bnvs_TEST_A_tl.)

`\l__bnvs_Z_tl` Storage for the last component of a range.
(End definition for \l__bnvs_Z_tl.)

`\l__bnvs_L_tl` Storage for the length component of a range.
(End definition for \l__bnvs_L_tl.)

`\c__bnvs_A_cln_Z_regex` Used to parse slide range overlay specifications. A, A:Z, A::L on one side, :Z, :Z::L and ::L:Z on the other sides. Next are the capture groups.
(End definition for \c__bnvs_A_cln_Z_regex.)

```

3078 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
3079   \A \s* (?:
      • 2: V
3080       ( [^:]* )
      • 3, 4, 5: A : Z? or A :: L?
3081       | ( [^:]* ) \s* : (?: ( \s* [^:]* ) | : ( \s* [^:]* ) )
      • 6, 7: ::(L:Z)?
3082       | :: \s* (?: ( [^:]* ) \s* : \s* ( [^:]* ) )?
      • 8, 9: :(Z::L)?
3083       | : \s* (?: ( [^:]* ) \s* :: \s* ( [^:]* ) )?
3084   )
3085   \s* \Z
3086 }

3087 \BNVS_set:cpn { query_eval_end_return_true: } {
3088   \group_end:
3089   \prg_return_true:
3090 }
3091 \BNVS_new:cpn { query_eval_end_return_false: } {
3092   \BNVS_end:
3093   \prg_return_false:

```

```

3094 }
3095 \BNVS_new:cpn { query_eval_end_return_false:n } #1 {
3096   \BNVS_end:
3097   \prg_return_false:
3098 }
3099 \BNVS_new:cpn { query_eval_error_end_return_false:n } #1 {
3100   \__bnvs_error:x { #1 }
3101   \__bnvs_query_eval_end_return_false:
3102 }
3103 \BNVS_new:cpn { query_eval_unreachable: } {
3104   \__bnvs_query_eval_error_end_return_false:n { UNREACHABLE }
3105 }
3106 \BNVS_new:cpn { if_blank:cTF } #1 {
3107   \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3108 }
3109 \BNVS_new_conditional:cpnn { match_pop_left:c } #1 { T, F, TF } {
3110   \BNVS_tl_use:nc {
3111     \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
3112   } { #1 } {
3113     \prg_return_true:
3114   } {
3115     \prg_return_false:
3116   }
3117 }

```

__bnvs_query_eval_match_branch:TF __bnvs_query_eval_match_branch:TF {<true code>} {<false code>}

Puts the proper items of \l__bnvs_match_seq in \l__bnvs_V_tl, \l__bnvs_TEST_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 ...ans_tl variable and they end with either \prg_return_true: or \prg_return_false:. This is not inlined for readability.

```

3118 \BNVS_new_conditional:cpnn { query_eval_match_branch: } { T, F, TF } {
3119   \__bnvs_match_pop_left:cT V {
3120     \__bnvs_match_pop_left:cT V {
3121       \__bnvs_if_blank:cTF V {
3122         \__bnvs_match_pop_left:cT A {
3123           \__bnvs_match_pop_left:cT Z {
3124             \__bnvs_match_pop_left:cT L {
3125               \__bnvs_if_blank:cTF A {
3126                 \__bnvs_match_pop_left:cT L {
3127                   \__bnvs_match_pop_left:cT Z {
3128                     \__bnvs_if_blank:cTF Z {
3129                       \__bnvs_if_blank:cTF L {
3130                         \__bnvs_match_pop_left:cT Z {
3131                           \__bnvs_match_pop_left:cT L {
3132                             \__bnvs_if_blank:cTF L {
3133                               \__bnvs_if_blank:cTF Z {
3134                                 \BNVS_use:c { query_eval_return[:]: }
3135                               } {
3136                                 \BNVS_use:c { query_eval_return[:Z]: }
3137                               }
3138                             } {

```

```

3139         \_bnvs_if_blank:cTF Z {
3140 \_bnvs_query_eval_error_end_return_false:n { Missing~first~or~last }
3141     } {
3142         \BNVS_use:c { query_eval_return[:Z::L]: }
3143     }
3144 }
3145 }
3146 }
3147 } {
3148 \_bnvs_query_eval_error_end_return_false:n { Missing~first~or~last }
3149 }
3150 } {
3151     \_bnvs_if_blank:cTF L {
3152         \_bnvs_query_eval_unreachable:
3153     } {
3154         \BNVS_use:c { query_eval_return[:Z::L]: }
3155     }
3156 }
3157 }
3158 }
3159 } {
3160     \_bnvs_if_blank:cTF Z {
3161         \_bnvs_if_blank:cTF L {
3162             \BNVS_use:c { query_eval_return[A:]: }
3163         } {
3164             \BNVS_use:c { query_eval_return[A::L]: }
3165         }
3166     } {
3167         \_bnvs_if_blank:cTF L {
3168             \BNVS_use:c { query_eval_return[A:Z]: }
3169         } {
3170             \_bnvs_query_eval_error_end_return_false:n {
3171                 Only~two~of~first,~last~or~length
3172             }
3173         }
3174     }
3175 }
3176 }
3177 }
3178 }
3179 } {
3180     \BNVS_use:c { query_eval_return[V]: }
3181 }
3182 }
3183 }
3184 }
3185 \BNVS_new:cpn { query_eval_return[V]: } {

```

Single value

```

3186 \_bnvs_if_resolve:vcTF { V } { ans } {
3187     \prg_return_true:
3188 } {
3189     \prg_return_false:
3190 }

```

```

3191 }
3192 \BNVS_new:cpn { query_eval_return[A:Z]: } {
  ☛ <first>:<last> range
3193   \__bnvs_if_resolve:vcTF { A } { ans } {
3194     \__bnvs_tl_put_right:cn { ans } { - }
3195     \__bnvs_if_append:vcTF { Z } { ans } {
3196       \prg_return_true:
3197     } {
3198       \prg_return_false:
3199     }
3200   } {
3201     \prg_return_false:
3202   }
3203 }
3204 \BNVS_new:cpn { query_eval_return[A::L]: } {
  ☛ <first>::<length> range
3205   \__bnvs_if_resolve:vcTF { A } { A } {
3206     \__bnvs_if_resolve:vcTF { L } { ans } {
3207       \__bnvs_tl_put_right:cn { ans } { + }
3208       \__bnvs_tl_put_right:cv { ans } { A }
3209       \__bnvs_tl_put_right:cn { ans } { -1 }
3210       \__bnvs_round_ans:
3211       \__bnvs_tl_put_left:cn { ans } { - }
3212       \__bnvs_tl_put_left:cv { ans } { A }
3213       \prg_return_true:
3214     } {
3215       \prg_return_false:
3216     }
3217   } {
3218     \prg_return_false:
3219   }
3220 }
3221 \BNVS_new:cpn { query_eval_return[A:]: } {
  ☛ <first>: and <first>:: range
3222   \__bnvs_if_resolve:vcTF { A } { ans } {
3223     \__bnvs_tl_put_right:cn { ans } { - }
3224     \prg_return_true:
3225   } {
3226     \prg_return_false:
3227   }
3228 }
3229 \BNVS_new:cpn { query_eval_return[:Z::L]: } {
  ☛ :Z::L or ::L:Z range
3230   \__bnvs_if_resolve:vcTF { Z } { Z } {
3231     \__bnvs_if_resolve:vcTF { L } { ans } {
3232       \__bnvs_tl_put_left:cn { ans } { 1- }
3233       \__bnvs_tl_put_right:cn { ans } { + }
3234       \__bnvs_tl_put_right:cv { ans } { Z }
3235       \__bnvs_round_ans:
3236       \__bnvs_tl_put_right:cn { ans } { - }

```

```

3237     \_bnvs_tl_put_right:cv { ans } { Z }
3238     \prg_return_true:
3239   } {
3240     \prg_return_false:
3241   }
3242 } {
3243   \prg_return_false:
3244 }
3245 }
3246 \BNVS_new:cpn { query_eval_return[:]: } {
  : or :: range
3247   \_bnvs_tl_set:cn { ans } { - }
3248   \prg_return_true:
3249 }
3250 \BNVS_new:cpn { query_eval_return[Z]: } {
  ::(last) range
3251   \_bnvs_tl_set:cn { ans } { - }
3252   \_bnvs_if_append:vcTF { Z } { ans } {
3253     \prg_return_true:
3254   } {
3255     \prg_return_false:
3256   }
3257 }
3258 \BNVS_new_conditional:cpnn { query_eval:nc } #1 #2 { T, F, TF } {
3259   \_bnvs_call_greset:
3260   \_bnvs_match_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
3261     \BNVS_begin:
3262     \_bnvs_query_eval_match_branch:TF {
3263       \BNVS_end_tl_set:cv { #2 } { ans }
3264       \prg_return_true:
3265     } {
3266       \BNVS_end:
3267       \prg_return_false:
3268     }
3269   } {
Error
3270   \_bnvs_error:n { Syntax~error:~#1 }
3271   \prg_return_false:
3272 }
3273 }

```

```

__bnvs_eval:nc  __bnvs_eval:nN {<overlay query list>} <tl variable>

```

This is called by the *named overlay specifications* scanner. Evaluates the comma separated list of *<overlay query>*'s, replacing all the named overlay specifications and integer expressions by their static counterparts by calling `__bnvs_query_eval:nc`, then append the result to the right of the *<tl variable>*. This is executed within a local group. Below are local variables and constants used throughout the body of this function.

```

\l__bnvs_query_seq  Storage for a sequence of <query>'s obtained by splitting a comma separated list.
                    (End definition for \l__bnvs_query_seq.)

```

```

\l__bnvs_ans_seq    Storage of the evaluated result.
                    (End definition for \l__bnvs_ans_seq.)

```

```

\c__bnvs_comma_regex  Used to parse slide range overlay specifications.
3274 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }
                    (End definition for \c__bnvs_comma_regex.)
No other variable is used.

```

```

3275 \BNVS_new:cpn { eval:nc } #1 #2 {
3276   \BNVS_begin:

```

Local variables declaration

```

3277   __bnvs_seq_clear:c { query }
3278   __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`.

```

3279   \regex_split:NnN \c__bnvs_comma_regex { #1 } \l__bnvs_query_seq

```

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

```

3280   __bnvs_seq_map_inline:cn { query } {
3281     __bnvs_tl_clear:c { ans }
3282     __bnvs_query_eval:ncTF { ##1 } { ans } {
3283       __bnvs_seq_put_right:cv { ans } { ans }
3284     } {
3285       \seq_map_break:n {
3286         __bnvs_error:n { Circular/Undefined~dependency~in~#1}
3287       }
3288     }
3289   }

```

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

```

3290   \exp_args:NNnx
3291   \BNVS_end:
3292   __bnvs_tl_put_right:cn { #2 } { __bnvs_seq_use:cn { ans } , }
3293 }

```

\BeanovesEval

\BeanovesEval [*<tl variable>*] {*<overlay queries>*}

<overlay queries> is the argument of ?(...) instructions. This is a comma separated list of single *<overlay query>*'s.

This function evaluates the *<overlay queries>* and store the result in the *<tl variable>* when provided or leave the result in the input stream. Forwards to `__bnvs_eval:nN` within a group. `\...ans_tl` is used locally to store the result.

```
3294 \NewDocumentCommand \BeanovesEval { 0{ } m } {
3295   \BNVS_begin:
3296   \keys_define:nn { BeanovesEval } {
3297     in:N .tl_set:N = \l__bnvs_BeanovesEval_tl,
3298     in:N .initial:n = { },
3299     see .bool_set:N = \l__bnvs_BeanovesEval_bool,
3300     see .default:n = true,
3301     see .initial:n = false,
3302   }
3303   \keys_set:nn { BeanovesEval } { #1 }
3304   \__bnvs_tl_clear:c { ans }
3305   \__bnvs_eval:nc { #2 } { ans }
3306   \__bnvs_tl_if_empty:cTF { BeanovesEval } {
3307     \bool_if:nTF { \l__bnvs_BeanovesEval_bool } {
3308       \BNVS_tl_use:Nv \BNVS_end: { ans }
3309     } {
3310       \BNVS_end:
3311     }
3312   } {
3313     \bool_if:nTF { \l__bnvs_BeanovesEval_bool } {
3314       \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3315         \BNVS_end:
3316         \tl_set:Nn ##1 { ##2 }
3317         ##2
3318       }
3319       \BNVS_tl_use:nv {
3320         \exp_last_unbraced:Nv \BNVS_end:Nn \l__bnvs_BeanovesEval_tl
3321       } { ans }
3322     } {
3323       \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3324         \BNVS_end:
3325         \tl_set:Nn ##1 { ##2 }
3326       }
3327       \BNVS_tl_use:nv {
3328         \exp_last_unbraced:Nv \BNVS_end:Nn \l__bnvs_BeanovesEval_tl
3329       } { ans }
3330     }
3331   }
3332 }
```

6.13.13 Resetting counters

\BeanovesReset
\BeanovesReset*

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

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


```

3333 \NewDocumentCommand \BeanovesReset { s O{} m } {
3334   \__bnvs_name_id_n_get:nTF { #3 } {
3335     \BNVS_tl_use:nv {
3336       \IfBooleanTF { #1 } {
3337         \__bnvs_greset_all:nnF
3338       } {
3339         \__bnvs_v_greset:nnF
3340       }
3341     } { key } { #2 } {
3342       % \__bnvs_warning:n { Unknown~name:~#3 }
3343     }
3344   } {
3345     \__bnvs_warning:n { Bad~name:~#3 }
3346   }
3347   \ignorespaces
3348 }

3349 \makeatother
3350 \ExplSyntaxOff

<*internal> </internal>

```