

beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2024/01/11

Abstract

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

Contents

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

6	Implementation	8
6.1	Package declarations	8
6.2	Facility layer: definitions and naming	8
6.3	logging	10
6.4	Facility layer: Variables	10
6.4.1	Regex	15
6.4.2	Token lists	17
6.4.3	Strings	20
6.4.4	Sequences	21
6.4.5	Integers	23
6.4.6	Prop	23
6.5	Debug facilities	23
6.6	Debug messages	24
6.7	Variable facilities	24
6.8	Testing facilities	24
6.9	Local variables	24
6.10	Infinite loop management	25
6.11	Overlay specification	26
6.12	Basic functions	26
6.13	Functions with cache	28
6.13.1	Implicit value counter	30
6.13.2	Implicit index counter	32
6.13.3	Regular expressions	34
6.13.4	beamer.cls interface	36
6.13.5	Defining named slide ranges	36
6.13.6	Scanning named overlay specifications	46
6.13.7	Resolution	50
6.13.8	Evaluation bricks	56
6.13.9	Index counter	68
6.13.10	Value counter	70
6.13.11	Evaluation	74
6.13.12	Functions for the resolution	74
6.13.13	Reseting counters	95

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

```

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

2 Named overlay sets

2.1 Presentation

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

2.2 Named overlay reference

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

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

$\langle \text{frame id} \rangle! \langle \text{short name} \rangle$: 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 $\langle \text{spec} \rangle$ hereafter, $\langle \text{value} \rangle$, $\langle \text{first} \rangle$, $\langle \text{length} \rangle$ and $\langle \text{last} \rangle$ are algebraic expression (with algebraic operators $+$, $-$, \dots) 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 time¹.

2.3.2 List specifiers

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

$\langle \text{key} \rangle = \{ \langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle, \langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle, \dots, \langle \text{ref}_n \rangle = \langle \text{spec}_n \rangle \}$
is a convenient shortcut for
 $\langle \text{key} \rangle . \langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle,$
 $\langle \text{key} \rangle . \langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle,$
 $\dots,$
 $\langle \text{key} \rangle . \langle \text{ref}_n \rangle = \langle \text{spec}_n \rangle.$

The rules above can apply individually to each line.

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

2.3.3 .n specifiers

$\langle \text{key} \rangle . \text{n} = \langle \text{value} \rangle$ 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 $\langle ?(\text{A}.1) \rangle$ 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 a signed integer whereas $\langle \text{first} \rangle$, $\langle \text{last} \rangle$ and $\langle \text{length} \rangle$ stand for integers, or integer valued algebraic expressions.

3.1 Simple definitions

$\langle \text{key} \rangle = \langle \text{value} \rangle$ For an unlimited range

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

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

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

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

Notice that $\langle key \rangle.\text{previous}$ and $\langle key \rangle.0$ are synonyms.

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

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

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

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

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

```

1 \Beanoves {
2   A = 3:8, % or similarly A = 3::6, A = ::6:8 and A = :8::6
3 }
4 \begin{frame} {Frame \insertframenum} {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 $?(\text{A.next})$, $?(\text{A.last}+1)$, $?(\text{A.1}+\text{A.length})$ give the same result as soon as the slide range named ‘A’ has been properly defined with a starting value and a length.

3.2 Counters

Each named overlay set defined has a dedicated value counter which is some kind of variable that can be used and incremented. A standalone $\langle key \rangle$ *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.

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 space².

$++\langle key \rangle$, 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.

For each named overlay set defined, we also have an implicit index counter always starting at 1, its actual value is an integer denoted $\langle n \rangle$ in the sequel. 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.

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 $\langle spec \rangle$.

²The parser for algebraic expression is very rudimentary.

```

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

```

$\langle key \rangle . \langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_k \rangle = \langle range\ spec \rangle$ 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 references that would lead to an infinite T_EX loop. One final resolution occurs with the other rules above if possible otherwise 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 integer $\langle n' \rangle$ 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, for example, it becomes a shortcut to Y!A.1 because the *last frame id* is then Y.

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

`beanoves id` `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 from 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 parenthesis, 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 $?(...)$ 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).

$_@@=bnvs$

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

6.1 Package declarations

```

 $\_NeedsTeXFormat{LaTeX2e}[2020/01/01]$ 
 $\_ProvidesExplPackage$ 
  {beanoves}
  {2024/01/11}
  {1.0}
  {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 $\text{\LaTeX}3$. The `c` and `v` argument specifiers take a slightly different meaning when used in a function which name contains with `bnvs` or `BNVS`. Where $\text{\LaTeX}3$ would transform `l__bnvs_key_tl` into `\l__bnvs_key_tl`, `bnvs` will directly transform `key` into `\l__bnvs_key_tl`. The type of the local variable used depends on the context and may be `seq` or `int` for example. There are however a pair of exceptions mentionned below. For a better reading experience, ‘`key`’ will generally stand for `\l__bnvs_key_tl`, whereas ‘`path sequence`’ will generally stand for `\l__bnvs_path_seq`. Other similar shortcuts are used as well.

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

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

These are naming functions.

```
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` enough before usage by `<function>` or `<tokens>`. The first argument of `<function>` has type `N`. The next token after `<tokens>` will have type `N` too. `<cs name>` is a full cs name whereas `<cs core>` will be prepended with the appropriate prefix.

```
11 \cs_new:Npn \BNVS_use_raw:N #1 { #1 }
12 \cs_new:Npn \BNVS_use_raw:c #1 {
13   \exp_last_unbraced:No
14   \BNVS_use_raw:N { \cs:w #1 \cs_end: }
15 }
16 \cs_new:Npn \BNVS_use:c #1 {
17   \BNVS_use_raw:c { \BNVS:c { #1 } }
18 }
19 \cs_new:Npn \BNVS_use_raw:NN #1 #2 {
20   #1 #2
21 }
22 \cs_new:Npn \BNVS_use_raw:nN #1 #2 {
23   #1 #2
24 }
25 \cs_new:Npn \BNVS_use_raw:Nc #1 #2 {
26   \exp_last_unbraced:NNNo
27   \BNVS_use_raw:NN #1 { \cs:w #2 \cs_end: }
28 }
```

```

29 \cs_new:Npn \BNVS_use_raw:nc #1 #2 {
30   \exp_last_unbraced:Nno
31   \BNVS_use_raw:nN { #1 } { \cs:w #2 \cs_end: }
32 }
33 \cs_new:Npn \BNVS_use:Nc #1 #2 {
34   \BNVS_use_raw:Nc #1 { \BNVS:c { #2 } }
35 }
36 \cs_new:Npn \BNVS_use:nc #1 #2 {
37   \BNVS_use_raw:nc { #1 } { \BNVS:c { #2 } }
38 }
39 \cs_new:Npn \BNVS_log:n #1 { }
40 \cs_generate_variant:Nn \BNVS_log:n { x }
41 \cs_new:Npn \BNVS_DEBUG_on: {
42   \cs_set:Npn \BNVS_DEBUG_log:n { \BNVS_log:n }
43 }
44 \cs_new:Npn \BNVS_DEBUG_off: {
45   \cs_set:Npn \BNVS_DEBUG_log:n { \use_none:n }
46 }
47 \BNVS_DEBUG_off:

```

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

```

48 \cs_new:Npn \BNVS_new:cpn #1 {
49   \cs_new:cpn { \BNVS:c { #1 } }
50 }
51 \cs_new:Npn \BNVS_set:cpn #1 {
52   \cs_set:cpn { \BNVS:c { #1 } }
53 }
54 \cs_generate_variant:Nn \cs_generate_variant:Nn { c }
55 \cs_new:Npn \BNVS_generate_variant:cn #1 {
56   \cs_generate_variant:cn { \BNVS:c { #1 } }
57 }

```

6.3 logging

Utility message.

```

58 \msg_new:nnn { beanoves } { :n } { #1 }
59 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
60 \BNVS_new:cpn { warning:n } {
61   \msg_warning:nnn { beanoves } { :n }
62 }
63 \BNVS_generate_variant:cn { warning:n } { x }
64 \cs_new:Npn \BNVS_error:n {
65   \msg_error:nnn { beanoves } { :n }
66 }
67 \cs_new:Npn \BNVS_error:x {
68   \msg_error:nnx { beanoves } { :n }
69 }
70 \cs_new:Npn \BNVS_fatal:n {
71   \msg_fatal:nnn { beanoves } { :n }

```

```

72 }
73 \cs_new:Npn \BNVS_fatal:x {
74   \msg_fatal:nxx { beanoves } { :n }
75 }

```

6.4 Facility layer: Variables

`\BNVS_N_new:c` `\BNVS_N_new:n` {*<type>*}

`\BNVS_v_new:c`

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

```

76 \cs_new:Npn \BNVS_N_new:c #1 {
77   \cs_new:cpn { BNVS_#1:c } ##1 {
78     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
79   }
80   \cs_new:cpn { BNVS_#1_new:c } ##1 {
81     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
82   }
83   \cs_new:cpn { BNVS_#1_use:c } ##1 {
84     \use:c { \use:c { BNVS_#1:c } { ##1 } }
85   }
86   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
87     \BNVS_use_raw:Nc
88     ##1 { \use:c { BNVS_#1:c } { ##2 } }
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 }
95 \cs_new:Npn \BNVS_v_new:c #1 {
96   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
97     \BNVS_use_raw:nc
98     { \exp_args:Nv ##1 }
99     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
100  }
101  \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
102    \BNVS_use_raw:nc
103    { \exp_args:NnV \use:n { ##1 } }
104    { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
105  }
106 }
107 \BNVS_N_new:c { bool }
108 \BNVS_N_new:c { int }
109 \BNVS_v_new:c { int }
110 \BNVS_N_new:c { tl }
111 \BNVS_v_new:c { tl }
112 \BNVS_N_new:c { str }
113 \BNVS_v_new:c { str }
114 \BNVS_N_new:c { seq }
115 \BNVS_v_new:c { seq }
116 \cs_undefine:N \BNVS_N_new:c

```

\BNVS_use:Ncn \BNVS_use:Ncn {*function*} {(*core name*)} {(*type*)}

```

117 \cs_new:Npn \BNVS_use:Ncn #1 #2 #3 {
118   \BNVS_use_raw:c { BNVS_#3_use:Nc }   #1   { #2 }
119 }
120 \cs_new:Npn \BNVS_use:ncn #1 #2 #3 {
121   \BNVS_use_raw:c { BNVS_#3_use:nc } { #1 } { #2 }
122 }
123 \cs_new:Npn \BNVS_use:Nvn #1 #2 #3 {
124   \BNVS_use_raw:c { BNVS_#3_use:Nv }   #1   { #2 }
125 }
126 \cs_new:Npn \BNVS_use:nvn #1 #2 #3 {
127   \BNVS_use_raw:c { BNVS_#3_use:nv } { #1 } { #2 }
128 }
129 \cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
130   \BNVS_use:ncn {
131     \BNVS_use:Ncn   #1   { #2 } { #3 }
132   }
133 }
134 \cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
135   \BNVS_use:ncn {
136     \BNVS_use:ncn { #1 } { #2 } { #3 }
137   }
138 }
139 \cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
140   \BNVS_use:ncn {
141     \BNVS_use:Nvn   #1   { #2 } { #3 }
142   }
143 }
144 \cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
145   \BNVS_use:ncn {
146     \BNVS_use:nvn { #1 } { #2 } { #3 }
147   }
148 }
149 \cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
150   \BNVS_use:ncn {
151     \BNVS_use:Ncncn   #1   { #2 } { #3 } { #4 } { #5 }
152   }
153 }
154 \cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
155   \BNVS_use:ncn {
156     \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
157   }
158 }

```

\BNVS_new_c:cn \BNVS_new_c:nc {(*type*)} {(*core name*)}

```

159 \cs_new:Npn \BNVS_new_c:nc #1 #2 {
160   \BNVS_new_cpn { #1_#2:c } {
161     \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
162   }
163 }
164 \cs_new:Npn \BNVS_new_cn:nc #1 #2 {
165   \BNVS_new_cpn { #1_#2:cn } ##1 {

```

```

166     \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
167 }
168 }
169 \cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
170   \BNVS_new:cpn { #2:cnn } ##1 {
171     \BNVS_use:Ncn { #3 } { ##1 } { #1 }
172   }
173 }
174 \cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
175   \BNVS_use_raw:nc {
176     \BNVS_new_cnn:ncN { #1 } { #1_#2 }
177   } { #1_#2:Nnn }
178 }
179 \cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
180   \BNVS_new:cpn { #2:cnv } ##1 ##2 {
181     \BNVS_tl_use:nv {
182       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
183     }
184   }
185 }
186 \cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
187   \BNVS_use_raw:nc {
188     \BNVS_new_cnv:ncN { #1 } { #1_#2 }
189   } { #1_#2:Nnn }
190 }
191 \cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
192   \BNVS_new:cpn { #2:cnx } ##1 ##2 {
193     \exp_args:Nnx \use:n {
194       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
195     }
196   }
197 }
198 \cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
199   \BNVS_use_raw:nc {
200     \BNVS_new_cnx:ncN { #1 } { #1_#2 }
201   } { #1_#2:Nnn }
202 }
203 \cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
204   \BNVS_new:cpn { #2:cc } ##1 ##2 {
205     \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
206   }
207 }
208 \cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
209   \BNVS_use_raw:nc {
210     \BNVS_new_cc:ncNn { #1 } { #1_#2 }
211   } { #1_#2:NN }
212 }
213 \cs_new:Npn \BNVS_new_cc:nc #1 #2 {
214   \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
215 }
216 \cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
217   \BNVS_new:cpn { #2:cn } ##1 {
218     \BNVS_use:Ncn #3 { ##1 } { #1 }
219   }

```

```

220 }
221 \cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
222   \BNVS_use_raw:nc {
223     \BNVS_new_cn:ncNn { #1 } { #1_#2 }
224   } { #1_#2:Nn }
225 }
226 \cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
227   \BNVS_new_cpn { #2:cv } ##1 ##2 {
228     \BNVS_use:nvn {
229       \BNVS_use:Ncn #3 { ##1 } { #1 }
230     } { ##2 } { #4 }
231   }
232 }
233 \cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
234   \BNVS_use_raw:nc {
235     \BNVS_new_cv:ncNn { #1 } { #1_#2 }
236   } { #1_#2:Nn }
237 }
238 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
239   \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
240 }
241 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
242   \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
243 }
244 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
245   \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
246 }
247 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
248   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
249 }
250 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
251   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
252 }
253 \cs_new:Npn \BNVS_g_prop_use:Nc #1 #2 {
254   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { prop } }
255 }
256 \cs_new:Npn \BNVS_g_prop_use:nc #1 #2 {
257   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { prop } }
258 }
259 \cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
260   \BNVS_use:ncncncn { \exp_args:NVVV #1 }
261   { #2 } { t1 } { #3 } { t1 } { #4 } { t1 }
262 }

```

\BNVS_new_conditional:cpnn \BNVS_new_conditional:cpnn {<core name>} <parameter> {<conditions>} {<code>}

```

263 \cs_generate_variant:Nn \prg_new_conditional:Npnn { c }
264 \cs_new:Npn \BNVS_new_conditional:cpnn #1 {
265   \prg_new_conditional:cpnn { \BNVS:c { #1 } }
266 }
267 \cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
268 \cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
269   \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
270 }

```

```

271 \cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
272   \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
273     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
274       \prg_return_true:
275     } {
276       \prg_return_false:
277     }
278   }
279 }
280 \cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
281   \BNVS_use:nc {
282     \BNVS_new_conditional_vn:cNnn { #1 }
283   } { #1:nn TF } { #2 }
284 }
285 \cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
286   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
287     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
288       \prg_return_true:
289     } {
290       \prg_return_false:
291     }
292   }
293 }
294 \cs_new:Npn \BNVS_new_conditional_vc:cnn #1 {
295   \BNVS_use:nc {
296     \BNVS_new_conditional_vc:cNnn { #1 }
297   } { #1:ncTF }
298 }
299 \cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
300   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
301     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
302       \prg_return_true:
303     } {
304       \prg_return_false:
305     }
306   }
307 }
308 \cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
309   \BNVS_use:nc {
310     \BNVS_new_conditional_vc:cNn { #1 }
311   } { #1:ncTF }
312 }

```

6.4.1 Regex

```

313 \cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
314   \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
315 }

```

```

\__bnvs_match_once:NnTF \__bnvs_match_once:NnTF <regex variable> {<expression>}
\__bnvs_match_once:NvTF {<yes code>} {<no code>}
\__bnvs_match_once:nnTF \__bnvs_match_once:nnTF {<regex>} {<expression>}
\__bnvs_regex_split:cnTF {<yes code>} {<no code>}

```

```

\__bnvs_regex_split:cncTF <regex core> {<expression>} <seq core> {<yes code>} {<no
code>}
\__bnvs_regex_split:cnTF <regex core> {<expression>} {<yes code>} {<no code>}

```

These are shortcuts to

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

```

316 \BNVS_new_conditional:cpnn { match_once:Nn } #1 #2 { T, F, TF } {
317   \BNVS_use:ncn {
318     \regex_extract_once:NnNTF #1 { #2 }
319   } { match } { seq } {
320     \prg_return_true:
321   } {
322     \prg_return_false:
323   }
324 }
325 \BNVS_new_conditional:cpnn { match_once:Nv } #1 #2 { T, F, TF } {
326   \BNVS_seq_use:nc {
327     \BNVS_tl_use:nv {
328       \regex_extract_once:NnNTF #1
329     } { #2 }
330   } { match } {
331     \prg_return_true:
332   } {
333     \prg_return_false:
334   }
335 }
336 \BNVS_new_conditional:cpnn { match_once:nn } #1 #2 { T, F, TF } {
337   \BNVS_seq_use:nc {
338     \regex_extract_once:nnNTF { #1 } { #2 }
339   } { match } {
340     \prg_return_true:
341   } {
342     \prg_return_false:
343   }
344 }
345 \BNVS_new_conditional:cpnn { regex_split:cnc } #1 #2 #3 { T, F, TF } {
346   \BNVS_seq_use:nc {
347     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
348   } { #3 } {
349     \prg_return_true:
350   } {
351     \prg_return_false:
352   }
353 }
354 \BNVS_new_conditional:cpnn { regex_split:cn } #1 #2 { T, F, TF } {

```

```

355 \BNVS_seq_use:nc {
356   \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
357 } { split } {
358   \prg_return_true:
359 } {
360   \prg_return_false:
361 }
362 }

```

6.4.2 Token lists

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

These are shortcuts to

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

\BNVS_new_conditional_vnc:cn \BNVS_new_conditional_vnc:cn {<core>} {<conditions>}

<function> is the test function with signature ...:nncTF. <core>:nncTF is used for testing.

```

363 \cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
364   \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
365     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
366       \prg_return_true:
367     } {
368       \prg_return_false:
369     }
370   }
371 }
372 \cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
373   \BNVS_use:nc {

```

```

374     \BNVS_new_conditional_vnc:cNn { #1 }
375   } { #1:nncTF }
376 }

```

```

\BNVS_new_conditional_vnc:cn \BNVS_new_conditional_vnc:cn {<core>} {<conditions>}

```

Forwards to \BNVS_new_conditional_vnc:cNn with \<core>:nncTF as function argument. Used for testing.

```

377 \cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
378   \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
379     \BNVS_tl_use:nv {
380       \BNVS_tl_use:Nv #2 { ##1 }
381     } { ##2 } { ##3 } { ##4 } {
382       \prg_return_true:
383     } {
384       \prg_return_false:
385     }
386   }
387 }
388 \cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
389   \BNVS_use:nc {
390     \BNVS_new_conditional_vvnc:cNn { #1 }
391   } { #1:nnncTF }
392 }
393 \cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
394   \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
395     \BNVS_tl_use:nv {
396       \BNVS_tl_use:nv {
397         \BNVS_tl_use:Nv #2 { ##1 }
398       } { ##2 }
399     } { ##3 } { ##4 } {
400       \prg_return_true:
401     } {
402       \prg_return_false:
403     }
404   }
405 }
406 \cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
407   \BNVS_use:nc {
408     \BNVS_new_conditional_vvvc:cNn { #1 }
409   } { #1:nnncTF }
410 }
411 \cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
412   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
413     \BNVS_tl_use:nv {
414       \BNVS_tl_use:Nv #2 { ##1 }
415     } { ##2 } { ##3 } {
416       \prg_return_true:
417     } {
418       \prg_return_false:
419     }
420   }
421 }

```

```

422 \cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
423   \BNVS_use:nc {
424     \BNVS_new_conditional_vvc:cNn { #1 }
425   } { #1:nncTF }
426 }
427 \cs_new:Npn \BNVS_new_tl_c:c {
428   \BNVS_new_c:nc { tl }
429 }
430 \BNVS_new_tl_c:c { clear }
431 \BNVS_new_tl_c:c { use }
432 \BNVS_new_tl_c:c { count }
433
434 \BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
435   \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
436 }
437 \cs_new:Npn \BNVS_new_tl_cn:c {
438   \BNVS_new_cn:nc { tl }
439 }
440 \cs_new:Npn \BNVS_new_tl_cv:c #1 {
441   \BNVS_new_cv:ncn { tl } { #1 } { tl }
442 }
443 \BNVS_new_tl_cn:c { set }
444 \BNVS_new_tl_cv:c { set }
445 \BNVS_new:cpn { tl_set:cx } {
446   \exp_args:Nnx \__bnvs_tl_set:cn
447 }
448 \BNVS_new_tl_cn:c { put_right }
449 \BNVS_new_tl_cv:c { put_right }
450 % \BNVS_generate_variant:cn { tl_put_right:cn } { cx }
451 \BNVS_new:cpn { tl_put_right:cx } {
452   \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
453 }
454 \BNVS_new_tl_cn:c { put_left }
455 \BNVS_new_tl_cv:c { put_left }
456 % \BNVS_generate_variant:cn { tl_put_left:cn } { cx }
457 \BNVS_new:cpn { tl_put_left:cx } {
458   \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
459 }

```

<u>__bnvs_tl_if_empty:cTF</u>	<u>__bnvs_tl_if_empty:cTF</u>	{\core}	{\yes code}	{\no code}
<u>__bnvs_tl_if_blank:vTF</u>	<u>__bnvs_tl_if_blank:vTF</u>	{\core}	{\yes code}	{\no code}
<u>__bnvs_tl_if_eq:cnTF</u>	<u>__bnvs_tl_if_eq:cnTF</u>	{\core}	{\tl}	{\yes code} {\no code}

These are shortcuts to

- \tl_if_empty:cTF {l__bnvs_<core>_tl} {\yes code} {\no code}
- \tl_if_eq:cnTF {l__bnvs_<core>_tl}{\tl} {\yes code} {\no code}

```

460 \cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
461   \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
462     \BNVS_use:Ncn #3 { ##1 } { #1 } {
463       \prg_return_true:
464     } {

```

```

465     \prg_return_false:
466   }
467 }
468 }
469 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
470   \BNVS_use_raw:nc {
471     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
472   } { #1_#2:NTF }
473 }
474 \BNVS_new_conditional_c:ncn { t1 } { if_empty } { p, T, F, TF }
475 \BNVS_new_conditional:cpnn { t1_if_blank:v } #1 { T, F, TF } {
476   \BNVS_tl_use:Nv \t1_if_blank:nTF { #1 } {
477     \prg_return_true:
478   } {
479     \prg_return_false:
480   }
481 }
482 \cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
483   \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
484     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
485       \prg_return_true:
486     } {
487       \prg_return_false:
488     }
489   }
490 }
491 \cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
492   \BNVS_use_raw:nc {
493     \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
494   } { #1_#2:NnTF }
495 }
496 \BNVS_new_conditional_cn:ncn { t1 } { if_eq } { T, F, TF }
497 \cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
498   \BNVS_new_conditional:cpnn { #2:cv } ##1 ##2 { #4 } {
499     \BNVS_use:nvn {
500       \BNVS_use:Ncn #3 { ##1 } { #1 }
501     } { ##2 } { #1 } {
502       \prg_return_true:
503     } {
504       \prg_return_false:
505     }
506   }
507 }
508 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
509   \BNVS_use_raw:nc {
510     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
511   } { #1_#2:NnTF }
512 }
513 \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>} {<tl>} {<yes code>} {<no code>}

These are shortcuts to

- \str_if_eq:ccTF {l_bnvs_<core>_tl}{<yes code>} {<no code>}

```

514 \cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
515   \BNVS_new_conditional:cpnn { #2:vn } ##1 ##2 { #4 } {
516     \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
517       \prg_return_true:
518     } {
519       \prg_return_false:
520     }
521   }
522 }
523 \cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
524   \BNVS_use_raw:nc {
525     \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
526   } { #1_#2:nnTF }
527 }
528 \BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }
529 \cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
530   \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
531     \BNVS_use:nvn {
532       \BNVS_use:Nvn #3 { ##1 } { #1 }
533     } { ##2 } { #1 } {
534       \prg_return_true:
535     } {
536       \prg_return_false:
537     }
538   }
539 }
540 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
541   \BNVS_use_raw:nc {
542     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
543   } { #1_#2:nnTF }
544 }
545 \BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }

```

6.4.4 Sequences

<code>_bnvs_seq_count:c</code>	<code>_bnvs_seq_new:c {<core>}</code>
<code>_bnvs_seq_clear:c</code>	<code>_bnvs_seq_count:c {<core>}</code>
<code>_bnvs_seq_set_eq:cc</code>	<code>_bnvs_seq_clear:c {<core>}</code>
<code>_bnvs_seq_use:cn</code>	<code>_bnvs_seq_set_eq:cc {<core₁>} {<core₂>}</code>
<code>_bnvs_seq_item:cn</code>	<code>_bnvs_seq_use:cn {<core>} {<separator>}</code>
<code>_bnvs_seq_remove_all:cn</code>	<code>_bnvs_seq_item:cn {<core>} {<integer expression>}</code>
<code>_bnvs_seq_put_left:cv</code>	<code>_bnvs_seq_remove_all:cn {<core>} {<tl>}</code>
<code>_bnvs_seq_put_right:cn</code>	<code>_bnvs_seq_put_right:cn {<seq core>} {<tl>}</code>
<code>_bnvs_seq_put_right:cv</code>	<code>_bnvs_seq_put_right:cv {<seq core>} {<tl core>}</code>
<code>_bnvs_seq_set_split:cnn</code>	<code>_bnvs_seq_set_split:cnn {<seq core>} {<tl>} {<separator>}</code>
<code>_bnvs_seq_set_split:(cnv cnx)</code>	<code>_bnvs_seq_pop_left:cc {<core₁>} {<core₂>}</code>
<code>_bnvs_seq_pop_left:cc</code>	

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>}`

```

546 \BNVS_new_c:nc { seq } { count }
547 \BNVS_new_c:nc { seq } { clear }
548 \BNVS_new_cn:nc { seq } { use }
549 \BNVS_new_cn:nc { seq } { item }
550 \BNVS_new_cn:nc { seq } { remove_all }
551 \BNVS_new_cn:nc { seq } { map_inline }
552 \BNVS_new_cc:nc { seq } { set_eq }
553 \BNVS_new_cv:ncn { seq } { put_left } { tl }
554 \BNVS_new_cn:ncn { seq } { put_right } { tl }
555 \BNVS_new_cv:ncn { seq } { put_right } { tl }
556 \BNVS_new_cnn:nc { seq } { set_split }
557 \BNVS_new_cnv:nc { seq } { set_split }
558 \BNVS_new_cnx:nc { seq } { set_split }
559 \BNVS_new_cc:ncn { seq } { pop_left } { tl }
560 \BNVS_new_cc:ncn { seq } { pop_right } { tl }

```

<code>_bnvs_seq_if_empty:cTF</code>	<code>_bnvs_seq_if_empty:cTF {<seq core name>} {<yes code>} {<no code>}</code>
<code>_bnvs_seq_get_right:ccTF</code>	<code>_bnvs_seq_get_right:ccTF {<seq core name>} {<tl core name>} {<yes code>} {<no code>}</code>
<code>_bnvs_seq_pop_left:ccTF</code>	<code>code}</code>
<code>_bnvs_seq_pop_right:ccTF</code>	

```

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

```

6.4.5 Integers

```

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

```

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>}

```

579 \BNVS_new_c:nc { int } { new }
580 \BNVS_new_c:nc { int } { use }
581 \BNVS_new_c:nc { int } { zero }
582 \BNVS_new_c:nc { int } { incr }
583 \BNVS_new_c:nc { int } { decr }
584 \BNVS_new_cn:nc { int } { set }
585 \BNVS_new_cv:ncn { int } { set } { int }

```

6.4.6 Prop

```

\__bnvs_prop_get:NncTF

```

```

586 \BNVS_new_conditional:cpnn { prop_get:Nnc } #1 #2 #3 { T, F, TF } {
587   \BNVS_use:ncn {
588     \prop_get:NnNTF #1 { #2 }
589   } { #3 } { t1 } {
590     \prg_return_true:
591   } {
592     \prg_return_false:
593   }
594 }

```

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.

```

595 \tl_new:N \l__bnvs_id_last_tl
596 \tl_set:Nn \l__bnvs_id_last_tl { ?! }
597 \tl_new:N \l__bnvs_a_tl
598 \tl_new:N \l__bnvs_b_tl
599 \tl_new:N \l__bnvs_c_tl
600 \tl_new:N \l__bnvs_V_tl
601 \tl_new:N \l__bnvs_A_tl
602 \tl_new:N \l__bnvs_L_tl
603 \tl_new:N \l__bnvs_Z_tl
604 \tl_new:N \l__bnvs_ans_tl
605 \tl_new:N \l__bnvs_key_tl
606 \tl_new:N \l__bnvs_key_base_tl
607 \tl_new:N \l__bnvs_id_tl
608 \tl_new:N \l__bnvs_n_tl
609 \tl_new:N \l__bnvs_path_tl
610 \tl_new:N \l__bnvs_group_tl
611 \tl_new:N \l__bnvs_scan_tl
612 \tl_new:N \l__bnvs_query_tl
613 \tl_new:N \l__bnvs_token_tl
614 \tl_new:N \l__bnvs_root_tl
615 \tl_new:N \l__bnvs_n_incr_tl
616 \tl_new:N \l__bnvs_incr_tl

```

```

617 \tl_new:N \l__bnvs_post_tl
618 \tl_new:N \l__bnvs_suffix_tl
619 \int_new:N \g__bnvs_call_int
620 \int_new:N \l__bnvs_int
621 \seq_new:N \g__bnvs_def_seq
622 \seq_new:N \l__bnvs_a_seq
623 \seq_new:N \l__bnvs_b_seq
624 \seq_new:N \l__bnvs_ans_seq
625 \seq_new:N \l__bnvs_match_seq
626 \seq_new:N \l__bnvs_split_seq
627 \seq_new:N \l__bnvs_path_seq
628 \seq_new:N \l__bnvs_path_base_seq
629 \seq_new:N \l__bnvs_query_seq
630 \seq_new:N \l__bnvs_token_seq
631 \bool_new:N \l__bnvs_in_frame_bool
632 \bool_set_false:N \l__bnvs_in_frame_bool
633 \bool_new:N \l__bnvs_parse_bool

```

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

```

634 \bool_new:N \l__bnvs_provide_bool
635 \BNVS_new:cpn { provide_on: } {
636   \bool_set_true:N \l__bnvs_provide_bool
637 }
638 \BNVS_new:cpn { provide_off: } {
639   \bool_set_false:N \l__bnvs_provide_bool
640 }
641 \__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.

```

642 \BNVS_new_conditional:cpnn { if_provide: } { p, T, F, TF } {
643   \bool_if:NTF \l__bnvs_provide_bool {
644     \prg_return_true:
645   } {
646     \prg_return_false:
647   }
648 }

```

6.10 Infinite loop management

Unending recursivity is managed here.

`\g__bnvs_call_int` Some functions calls, as well as some loop bodies, decrement this counter. When this counter reaches 0, an error is raised or a computation is aborted.

(End of definition for `\g__bnvs_call_int`.)

```

649 \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.

```

650 \cs_set:Npn \__bnvs_call_greset: {
651   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
652 }

```

`__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.

```

653 \BNVS_new_conditional:cpnn { call: } { T, F, TF } {
654   \int_gdecr:N \g__bnvs_call_int
655   \int_compare:nNnTF \g__bnvs_call_int > 0 {
656     \prg_return_true:
657   } {
658     \prg_return_false:
659   }
660 }

```

6.11 Overlay specification

6.12 Basic functions

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

`<id>!<key>/V` for the value

`<id>!<key>/A` for the first index

`<id>!<key>/L` for the length when provided

`<id>!<key>/Z` for the last index when provided

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

```

661 \prop_new:N \g__bnvs_prop

```

(End of definition for `\g__bnvs_prop`.)

<code>__bnvs_gput:nnn</code>	<code>__bnvs_gput:nnn {<subkey>} {<key>} {<value>}</code>
<code>__bnvs_gput:nnv</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 `<key>/<subkey>`.

```

662 \BNVS_new:cpn { gput:nnn } #1 #2 {
663   \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
664 }

```

```

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

```

```

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

```

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

```

689 \BNVS_new_conditional:cpnn { if_in:nn } #1 #2 { p, T, F, TF } {
690   \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
691     \prg_return_true:
692   } {
693     \prg_return_false:
694   }
695 }
696 \BNVS_new_conditional:cpnn { if_in:n } #1 { p, T, F, TF } {
697   \bool_if:nTF {
698     \__bnvs_if_in_p:nn V { #1 }
699     || \__bnvs_if_in_p:nn A { #1 }
700     || \__bnvs_if_in_p:nn Z { #1 }
701   } {
702     \prg_return_true:
703   } {
704     \prg_return_false:
705   }
706 }

```

```

707 \BNVS_new_conditional:cpnn { if_in:v } #1 { p, T, F, TF } {
708   \BNVS_tl_use:Nv \__bnvs_if_in:nTF { #1 }
709   { \prg_return_true: } { \prg_return_false: }
710 }

```

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

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

```

711 \BNVS_new:cpn { gprovide:nnnT } #1 #2 #3 #4 {
712   \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
713     #4
714     \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
715   }
716 }

```

__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.

```

717 \BNVS_new_conditional:cpnn { get:nnc } #1 #2 #3 { T, F, TF } {
718   \BNVS_tl_use:nc {
719     \prop_get:NnNTF \g__bnvs_prop { #2 / #1 }
720   } { #3 } {
721     \prg_return_true:
722   } {
723     \prg_return_false:
724   }
725 }
726 \BNVS_new_conditional:cpnn { get:nvc } #1 #2 #3 { T, F, TF } {
727   \BNVS_tl_use:nv {
728     \__bnvs_get:nncTF { #1 }
729   } { #2 } { #3 } {
730     \prg_return_true:
731   } {
732     \prg_return_false:
733   }
734 }

```

6.13 Functions with cache

\g__bnvs_prop {<key>—<value>} property list to store the named overlay sets. Other keys are eventually used to cache results when some attributes are defined from other slide ranges.

<id>!<key>/V for the cached static value of the value

<id>!<key>/A for the cached static value of the first index

<id>!<key>/L for the cached static value of the length

<id>!<key>/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.

735 `\prop_new:N \g__bnvs_cache_prop`

(End of definition for `\g__bnvs_prop`.)

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

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

```

736 \BNVS_new:cpn { cache_gput:nnn } #1 #2 {
737   \prop_gput:Nnn \g__bnvs_cache_prop { #2 / #1 }
738 }
739 \cs_generate_variant:Nn \__bnvs_cache_gput:nnn { nV, nnV }
740 \BNVS_new:cpn { cache_gput:nvn } #1 {
741   \BNVS_tl_use:nc {
742     \__bnvs_cache_gput:nVn { #1 }
743   }
744 }
745 \BNVS_new:cpn { cache_gput:nnv } #1 #2 {
746   \BNVS_tl_use:nc {
747     \__bnvs_cache_gput:nnV { #1 } { #2 }
748   }
749 }
750 \BNVS_new:cpn { cache_item:nn } #1 #2 {
751   \prop_item:Nn \g__bnvs_cache_prop { #2 / #1 }
752 }
753 \BNVS_new:cpn { cache_gremove:nn } #1 #2 {
754   \prop_gremove:Nn \g__bnvs_cache_prop { #2 / #1 }
755 }
756 \BNVS_new:cpn { cache_gclear:n } #1 {
757   \clist_map_inline:nn { V, A, Z, L, P, N } {
758     \prop_gremove:Nn \g__bnvs_cache_prop { #1 / ##1 }
759   }
760 }
761 \BNVS_new:cpn { cache_gclear: } {
762   \prop_gclear:N \g__bnvs_cache_prop
763 }
```

<code>__bnvs_cache_if_in_p:nn</code>	<code>__bnvs_cache_if_in_p:n {<subkey>} {<key>}</code>
<code>__bnvs_cache_if_in:nnTF</code>	<code>__bnvs_cache_if_in:nTF {<subkey>} {<key>} {<yes code>} {<no code>}</code>

Convenient shortcuts to test for the existence of $\langle subkey \rangle / \langle key \rangle$, it makes the code more concise and readable.

```

764 \prg_new_conditional:Npnn \__bnvs_cache_if_in:nn #1 #2 { p, T, F, TF } {
765   \prop_if_in:NnTF \g__bnvs_cache_prop { #2 / #1 } {
766     \prg_return_true:
767   } {
768     \prg_return_false:
769   }
770 }

```

```

\__bnvs_cache_get:nncTF \__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 `\prop_get:NnNTF` is not expandable.

```

771 \BNVS_new_conditional:cpnn { cache_get:nnc } #1 #2 #3 { p, T, F, TF } {
772   \BNVS_tl_use:nc {
773     \prop_get:NnNTF \g__bnvs_cache_prop { #2 / #1 }
774   } { #3 } {
775     \prg_return_true:
776   } {
777     \prg_return_false:
778   }
779 }

```

6.13.1 Implicit value counter

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

`\g__bnvs_v_prop` *<key>*–*<value>* property list to store the contents or the named value counters. The keys are *<id>!**<key>*.

```

780 \prop_new:N \g__bnvs_v_prop

```

(End of definition for `\g__bnvs_v_prop`.)

```

\__bnvs_v_gput:nn \__bnvs_v_gput:nn {<key>} {<value>}
\__bnvs_v_gput:(nV|Vn) \__bnvs_v_item:n {<key>}
\__bnvs_v_item:n \__bnvs_v_gremove:n {<key>}
\__bnvs_v_gremove:n \__bnvs_v_gclear:
\__bnvs_v_gclear:

```

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

```

781 \BNVS_new:cpn { v_gput:nn } {
782   \prop_gput:Nnn \g__bnvs_v_prop
783 }

```

```

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

```

```

\__bnvs_v_if_in_p:n * \__bnvs_v_if_in_p:n {<key>}
\__bnvs_v_if_in:nTF * \__bnvs_v_if_in:nTF {<key>} {<yes code>} {<no code>}

```

Convenient shortcuts to test for the existence of the *<key>* value counter.

```

798 \BNVS_new_conditional:cpnn { v_if_in:n } #1 { p, T, F, TF } {
799   \prop_if_in:NnTF \g__bnvs_v_prop { #1 } {
800     \prg_return_true:
801   } {
802     \prg_return_false:
803   }
804 }

```

```

\__bnvs_v_get:ncTF \__bnvs_v_get:ncTF {<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 `\prop_get:NnNTF` is not expandable.

```

805 \BNVS_new_conditional:cpnn { v_get:nc } #1 #2 { T, F, TF } {
806   \BNVS_tl_use:nc {
807     \prop_get:NnNTF \g__bnvs_v_prop { #1 }
808   } { #2 } {
809     \prg_return_true:
810   } {
811     \prg_return_false:
812   }
813 }

```

<pre> __bnvs_v_greset:nnTF __bnvs_v_greset:vnTF __bnvs_greset_all:nn </pre>	<pre> __bnvs_v_greset:nnTF {<key>} {<initial value>} {<true code>} {<false code>} __bnvs_greset_all:nnTF {<key>} {<initial value>} {<true code>} {<false code>} </pre>
--	--

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.


```

814 \BNVS_new_conditional:cpnn { v_greset:nn } #1 #2 { T, F, TF } {
815   \__bnvs_v_if_in:nTF { #1 } {
816     \__bnvs_v_gremove:n { #1 }
817     \tl_if_empty:nF { #2 } {
818       \__bnvs_v_gput:nn { #1 } { #2 }
819     }
820     \prg_return_true:
821   } {
822     \prg_return_false:
823   }
824 }
825 \BNVS_new_conditional:cpnn { v_greset:vn } #1 #2 { T, F, TF } {
826   \BNVS_tl_use:Nv \__bnvs_v_greset:nnTF { #1 } { #2 }
827   { \prg_return_true: } { \prg_return_false: }
828 }
829 \BNVS_new_conditional:cpnn { greset_all:nn } #1 #2 { T, F, TF } {
830   \__bnvs_if_in:nTF { #1 } {
831     \BNVS_begin:
832     \clist_map_inline:nn { V, A, Z, L } {
833       \__bnvs_get:nncT { ##1 } { #1 } { a } {
834         \__bnvs_quark_if_nil:cT { a } {
835           \__bnvs_cache_get:nncTF { ##1 } { #1 } { a } {
836             \__bnvs_gput:nnv { ##1 } { #1 } { a }
837           } {
838             \__bnvs_gput:nnn { ##1 } { #1 } { 1 }
839           }
840         }
841       }
842     }
843     \BNVS_end:
844     \__bnvs_cache_gclear:n { #1 }
845     \__bnvs_v_greset:nnT { #1 } { #2 } {}
846     \prg_return_true:
847   } {
848     \prg_return_false:
849   }
850 }
851 \BNVS_new_conditional:cpnn { greset_all:vn } #1 #2 { T, F, TF } {
852   \BNVS_tl_use:Nv \__bnvs_greset_all:nnTF { #1 } { #2 }
853   { \prg_return_true: } { \prg_return_false: }
854 }

```

<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.

```

855 \BNVS_new:cpn { gclear_all: } {
856   \__bnvs_gclear:
857   \__bnvs_cache_gclear:
858   \__bnvs_n_gclear:
859   \__bnvs_v_gclear:
860 }

```

```

861 \BNVS_new:cpn { gclear_all:n } #1 {
862   \__bnvs_gclear:n { #1 }
863   \__bnvs_cache_gclear:n { #1 }
864   \__bnvs_n_gremove:n { #1 }
865   \__bnvs_v_gremove:n { #1 }
866 }

```

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 $\langle id \rangle!$ $\langle name \rangle$.

```

867 \prop_new:N \g__bnvs_n_prop

```

(End of definition for `\g__bnvs_n_prop`.)

<code>__bnvs_n_gput:nn</code>	<code>__bnvs_n_gput:nn {$\langle key \rangle$} {$\langle value \rangle$}</code>
<code>__bnvs_n_gput:(nv vn)</code>	<code>__bnvs_n_item:n {$\langle key \rangle$}</code>
<code>__bnvs_n_gprovide:nn</code>	<code>__bnvs_n_gremove:n {$\langle key \rangle$}</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.

```

868 \BNVS_new:cpn { n_gput:nn } {
869   \prop_gput:Nnn \g__bnvs_n_prop
870 }
871 \cs_generate_variant:Nn \__bnvs_n_gput:nn { nV }
872 \BNVS_new:cpn { n_gput:nv } #1 {
873   \BNVS_tl_use:nc {
874     \__bnvs_n_gput:nV { #1 }
875   }
876 }
877 \BNVS_new:cpn { n_gprovide:nn } #1 #2 {
878   \prop_if_in:NnF \g__bnvs_n_prop { #1 } {
879     \prop_gput:Nnn \g__bnvs_n_prop { #1 } { #2 }
880   }
881 }
882 \BNVS_new:cpn { n_item:n } #1 {
883   \prop_item:Nn \g__bnvs_n_prop { #1 }
884 }
885 \BNVS_new:cpn { n_gremove:n } {
886   \prop_gremove:Nn \g__bnvs_n_prop
887 }
888 \BNVS_generate_variant:cn { n_gremove:n } { V }
889 \BNVS_new:cpn { n_gremove:v } {
890   \BNVS_tl_use:nc {
891     \__bnvs_n_gremove:V
892   }
893 }
894 \BNVS_new:cpn { n_gclear: } {
895   \prop_gclear:N \g__bnvs_n_prop

```

```

896 }
897 \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.

```

898 \prg_new_conditional:Npnn \__bnvs_n_if_in:n #1 { p, T, F, TF } {
899   \prop_if_in:NnTF \g__bnvs_n_prop { #1 } {
900     \prg_return_true:
901   } {
902     \prg_return_false:
903   }
904 }

```

```

\__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.

```

905 \prg_new_conditional:Npnn \__bnvs_n_get:nc #1 #2 { T, F, TF } {
906   \__bnvs_prop_get:NncTF \g__bnvs_n_prop { #1 } { #2 } {
907     \prg_return_true:
908   } {
909     \prg_return_false:
910   }
911 }

```

6.13.3 Regular expressions

$\backslash c_bnvs_name_regex$ The short name of an overlay set consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```

912 \regex_const:Nn \c__bnvs_name_regex {
913   [[:alpha:]]_ [[:alnum:]]_*
914 }

```

(End of definition for $\backslash c_bnvs_name_regex$.)

$\backslash c_bnvs_id_regex$ The frame identifier consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```

915 \regex_const:Nn \c__bnvs_id_regex {
916   (?: \ur{c__bnvs_name_regex} | [?] )? !
917 }

```

(End of definition for $\backslash c_bnvs_id_regex$.)

$\backslash c_bnvs_path_regex$ A sequence of $\langle positive\ integer \rangle$ or $\langle short\ name \rangle$ items representing a path.

```

918 \regex_const:Nn \c__bnvs_path_regex {
919   (?: \. \ur{c__bnvs_name_regex} | \. [-+]? \d+ )*
920 }

```

(End of definition for `\c__bnvs_path_regex`.)

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

(End of definition for `\c__bnvs_A_key_Z_regex`.)

```
921 \regex_const:Nn \c__bnvs_A_key_Z_regex {  
  
    1: The range name including the slide  $\langle id \rangle$  and question mark if any  
    2: slide  $\langle id \rangle$  including the question mark  
  
922 \A ( ( \ur{c__bnvs_id_regex} ? ) \ur{c__bnvs_name_regex} )  
  
    3: the path, if any.  
  
923 ( \ur{c__bnvs_path_regex} ) \Z  
924 }
```

`\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 of definition for `\c__bnvs_TEST_A_key_n_Z_regex`.)

```
925 \regex_const:Nn \c__bnvs_TEST_A_key_n_Z_regex {  
  
    1: The full match  
  
    2: The overlay set name including the slide  $\langle id \rangle$  and question mark if any, the dotted  
       path but excluding the trailing .n  
  
    3: slide  $\langle id \rangle$  including the question mark  
  
926 \A ( ( \ur{c__bnvs_id_regex} ? )  
927       \ur{c__bnvs_name_regex}  
928       (?: \. \ur{c__bnvs_name_regex} | \. [-+]? \d+ )*? )  
  
    4: the last .n component if any.  
  
929 ( \. n )? \Z  
930 }
```

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

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

(End of definition for `\c__bnvs_colons_regex`.)

`\c__bnvs_split_regex` Used to parse slide list overlay specifications in queries. Next are the 9 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

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

We start with ‘++’ instrussions³.

934 \+\+

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

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

- 3: optionally followed by a dotted path

936 (\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

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

- 6: optionally followed by a dotted path

938 (\ur{c__bnvs_path_regex})

We continue with other expressions

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

939 (?: \.(\+)\+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.

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

- 9: the post increment

941 | (\+)\+

942)?

943) \s*

944 }

(End of definition for `\c__bnvs_split_regex`.)

6.13.4 beamer.cls interface

Work in progress.

945 \RequirePackage{keyval}

946 \define@key{beamerframe}{beanoves-id}[] {

947 \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }

948 }

949 \AddToHook{env/beamer@frameslide/before}{

950 __bnvs_n_gclear:

951 __bnvs_v_gclear:

952 \bool_set_true:N \l__bnvs_in_frame_bool

953 }

954 \AddToHook{env/beamer@frameslide/after}{

955 \bool_set_false:N \l__bnvs_in_frame_bool

956 }

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

6.13.5 Defining named slide ranges

```

957 \_bnvs_range_set:cccnTF \_bnvs_range_set:cccnTF {<core first>} {<core end>} {<core length>} {<tl>} {<yes
958 code>} {<no code>}

```

Parse $\langle tl \rangle$ as a range according to `\c__bnvs_colons_regex` and set the variables accordingly. $\langle tl \rangle$ is expected to only contain colons and integers.

```

957 \BNVS_new_conditional:cpnn { split_pop_left:c } #1 { T, F, TF } {
958   \_bnvs_seq_pop_left:ccTF { split } { #1 } {
959     \prg_return_true:
960   } {
961     \prg_return_false:
962   }
963 }
964 \exp_args_generate:n { VVV }
965 \BNVS_new_conditional:cpnn { range_set:cccn } #1 #2 #3 #4 { T, F, TF } {
966   \BNVS_begin:
967     \_bnvs_tl_clear:c { a }
968     \_bnvs_tl_clear:c { b }
969     \_bnvs_tl_clear:c { c }
970     \_bnvs_regex_split:cnTF { colons } { #4 } {
971       \_bnvs_seq_pop_left:ccT { split } { a } {

```

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

```

972   \_bnvs_seq_pop_left:ccT { split } { b } {
973     \_bnvs_tl_if_empty:cTF { b } {

```

This is a one colon range.

```

974     \_bnvs_split_pop_left:cTF { b } {

```

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

```

975     \_bnvs_seq_pop_left:ccT { split } { c } {
976       \_bnvs_tl_if_empty:cTF { c } {

```

A :: was expected:

```

977       \BNVS_error:n { Invalid-range-expression(1):~#4 }
978     } {
979       \int_compare:nNtT { \_bnvs_tl_count:c { c } } > { 1 } {
980         \BNVS_error:n { Invalid-range-expression(2):~#4 }
981       }
982       \_bnvs_split_pop_left:cTF { c } {

```

$\backslash l_bnvs_c_tl$ may contain the $\langle length \rangle$.

```

983       \_bnvs_seq_if_empty:cF { split } {
984         \BNVS_error:n { Invalid-range-expression(3):~#4 }
985       }
986     } {
987       \BNVS_error:n { Internal-error }
988     }
989   }
990 }
991 } {
992 }
993 } {

```

This is a two colon range component.

```

994         \int_compare:nNtT { \__bnvs_tl_count:c { b } } > { 1 } {
995             \BNVS_error:n { Invalid~range~expression(4):~#4 }
996         }
997         \__bnvs_seq_pop_left:ccT { split } { c } {

```

c contains the $\langle length \rangle$.

```

998             \__bnvs_split_pop_left:cTF { b } {
999                 \__bnvs_tl_if_empty:cTF { b } {
1000                     \__bnvs_seq_pop_left:cc { split } { b }

```

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

```

1001                 \__bnvs_seq_if_empty:cF { split } {
1002                     \BNVS_error:n { Invalid~range~expression(5):~#4 }
1003                 }
1004             } {
1005                 \BNVS_error:n { Invalid~range~expression(6):~#4 }
1006             }
1007         } {
1008             \__bnvs_tl_clear:c { b }
1009         }
1010     }
1011 }
1012 }
1013 }

```

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.

```

1014     \cs_set:Npn \BNVS_next: { }
1015     \__bnvs_tl_if_empty:cT { a } {
1016         \__bnvs_tl_if_empty:cT { b } {
1017             \__bnvs_tl_if_empty:cT { c } {
1018                 \cs_set:Npn \BNVS_next: {
1019                     \BNVS_error:n { Invalid~range~expression(7):~#3 }
1020                 }
1021             }
1022         }
1023     }
1024     \BNVS_next:
1025     \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1026         \BNVS_end:
1027         \__bnvs_tl_set:cn { #1 } { ##1 }
1028         \__bnvs_tl_set:cn { #2 } { ##2 }
1029         \__bnvs_tl_set:cn { #3 } { ##3 }
1030     }
1031     \BNVS_exp_args:Nvvv \BNVS:nnn { a } { b } { c }
1032     \prg_return_true:
1033 } {
1034     \BNVS_end:
1035     \prg_return_false:
1036 }
1037 }

```

```

__bnvs_range:nnnn  __bnvs_range:nnnn {<key>} {<start>} {<end>} {<length>}
__bnvs_range:nvvv

```

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

```

1038 \BNVS_new:cpn { range:nnnn } #1 {
1039   __bnvs_if_provide:TF {
1040     __bnvs_if_in:nnTF A { #1 } {
1041       \use_none:nnn
1042     } {
1043       __bnvs_if_in:nnTF Z { #1 } {
1044         \use_none:nnn
1045       } {
1046         __bnvs_if_in:nnTF L { #1 } {
1047           \use_none:nnn
1048         } {
1049           __bnvs_do_range:nnnn { #1 }
1050         }
1051       }
1052     }
1053   } {
1054     __bnvs_do_range:nnnn { #1 }
1055   }
1056 }
1057 \BNVS_new:cpn { range:nvvv } #1 #2 #3 #4 {
1058   \BNVS_tl_use:nv {
1059     \BNVS_tl_use:nv {
1060       \BNVS_tl_use:nv {
1061         \BNVS_use:c { range:nnnn } { #1 }
1062       } { #2 }
1063     } { #3 }
1064   } { #4 }
1065 }

```

```

__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.

```

1066 \BNVS_new:cpn { parse_record:n } #1 {
1067   __bnvs_if_provide:TF {
1068     __bnvs_gprovide:nnnT V { #1 } { 1 } {
1069       __bnvs_gclear:n { #1 }
1070     }
1071   } {
1072     __bnvs_gclear:n { #1 }
1073     __bnvs_gput:nnn V { #1 } { 1 }
1074   }

```



```

1075 }
1076 \cs_generate_variant:Nn \__bnvs_parse_record:n { V }
1077 \BNVS_new:cpn { parse_record:v } {
1078   \BNVS_tl_use:nc {
1079     \__bnvs_parse_record:V
1080   }
1081 }
1082 \BNVS_new:cpn { parse_record:nn } #1 #2 {
1083   \__bnvs_range_set:cccnTF { a } { b } { c } { #2 } {
1084     \__bnvs_range:nvvv { #1 } { a } { b } { c }
1085   } {
1086     \__bnvs_if_provide:TF {
1087       \__bnvs_gprovide:nnnT V { #1 } { #2 } {
1088         \__bnvs_gclear_all:n { #1 }
1089       }
1090     } {
1091       \__bnvs_gclear_all:n { #1 }
1092       \__bnvs_gput:nnn V { #1 } { #2 }
1093     }
1094   }
1095 }
1096 \cs_generate_variant:Nn \__bnvs_parse_record:nn { x, V }
1097 \BNVS_new:cpn { parse_record:vn } {
1098   \BNVS_tl_use:nc {
1099     \__bnvs_parse_record:Vn
1100   }
1101 }
1102 \BNVS_new:cpn { n_parse_record:n } #1 {
1103   \bool_if:NTF \l__bnvs_n_provide_bool {
1104     \__bnvs_n_gprovide:nn
1105   } {
1106     \__bnvs_n_gput:nn
1107   }
1108   { #1 } { 1 }
1109 }
1110 \cs_generate_variant:Nn \__bnvs_n_parse_record:n { V }
1111 \BNVS_new:cpn { n_parse_record:v } {
1112   \BNVS_tl_use:nc {
1113     \__bnvs_n_parse_record:V
1114   }
1115 }
1116 \BNVS_new:cpn { n_parse_record:nn } #1 #2 {
1117   \__bnvs_range_set:cccnTF { a } { b } { c } { #2 } {
1118     \BNVS_error:n { Unexpected~range:~#2 }
1119   } {
1120     \__bnvs_if_provide:TF {
1121       \__bnvs_n_gprovide:nn { #1 } { #2 }
1122     } {
1123       \__bnvs_n_gput:nn { #1 } { #2 }
1124     }
1125   }
1126 }
1127 \cs_generate_variant:Nn \__bnvs_n_parse_record:nn { x, V }
1128 \BNVS_new:cpn { n_parse_record:vn } {

```

```

1129 \BNVS_tl_use:Nc \__bnvs_n_parse_record:Vn
1130 }

```

```

\__bnvs_name_id_n_get:nTF \__bnvs_name_id_n_set:nTF {<key>} {< yes code>} {< no code>}
\__bnvs_name_id_n_get:vTF

```

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.

```

1131 \BNVS_new:cpn { name_id_n_end_export: } {
1132   \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1133     \BNVS_end:
1134     \__bnvs_tl_set:cn { key } { ##1 }
1135     \__bnvs_tl_set:cn { id } { ##2 }
1136     \__bnvs_tl_set:cn { n } { ##3 }
1137   }
1138   \__bnvs_tl_if_empty:cTF { id } {
1139     \BNVS_exp_args:Nvvv
1140     \BNVS:nnn { key } { id_last } { n }
1141     \__bnvs_tl_put_left:cv { key } { id_last }
1142   } {
1143     \BNVS_exp_args:Nvvv
1144     \BNVS:nnn { key } { id } { n }
1145     \__bnvs_tl_set:cv { id_last } { id }
1146   }
1147 }
1148 \BNVS_new_conditional:cpnn { name_id_n_get:n } #1 { T, F, TF } {
1149   \BNVS_begin:
1150   \__bnvs_match_once:NnTF \c__bnvs_TEST_A_key_n_Z_regex { #1 } {
1151     \__bnvs_match_pop_left:cTF { key } {
1152       \__bnvs_match_pop_left:cTF { key } {
1153         \__bnvs_match_pop_left:cTF { id } {
1154           \__bnvs_match_pop_left:cTF { n } {
1155             \__bnvs_name_id_n_end_export:
1156             \prg_return_true:
1157           } {
1158             \BNVS_end:
1159             \BNVS_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/n }
1160             \prg_return_false:
1161           }
1162         } {
1163           \BNVS_end:
1164           \BNVS_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/id }
1165           \prg_return_false:
1166         }
1167       } {
1168         \BNVS_end:
1169         \BNVS_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/name }
1170         \prg_return_false:
1171       }
1172     } {
1173       \BNVS_end:
1174       \BNVS_error:n { LOGICALLY_UNREACHABLE_A_key_n_Z/n }

```

```

1175     \prg_return_false:
1176   }
1177 } {
1178   \BNVS_end:
1179   \prg_return_false:
1180 }
1181 }
1182 \BNVS_new_conditional:cpnn { name_id_n_get:v } #1 { T, F, TF } {
1183   \BNVS_tl_use:nv { \BNVS_use:c { name_id_n_get:nTF } } { #1 } {
1184     \prg_return_true:
1185   } {
1186     \prg_return_false:
1187   }
1188 }

```

```

\__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 of definition for \l__bnvs_match_seq.)

```

1189 \BNVS_new:cpn { parse:n } #1 {
1190   \peek_remove_spaces:n {
1191     \peek_catcode:NTF \c_group_begin_token {
1192       \__bnvs_tl_if_empty:CTF { root } {
1193         \BNVS_error:n { Unexpected-list-at-top-level. }
1194       }
1195       \BNVS_begin:
1196       \__bnvs_int_incr:c { }
1197       \__bnvs_tl_set:cx { root } { \__bnvs_int_use:c { } . }
1198       \cs_set:Npn \bnvs:nw #####1 #####2 \s_stop {
1199         \regex_match:nnT { \S* } { #####2 } {
1200           \BNVS_error:n { Unexpected-#####2 }
1201         }
1202         \keyval_parse:nnn {
1203           \__bnvs_parse:n
1204         } {
1205           \__bnvs_parse:nn
1206         } { #####1 }
1207         \BNVS_end:
1208       }
1209       \bnvs:nw
1210     } {
1211       \__bnvs_tl_if_empty:CTF { root } {
1212         \__bnvs_name_id_n_get:nTF { #1 } {
1213           \__bnvs_tl_if_empty:CTF { n } {
1214             \__bnvs_parse_record:v
1215           } {
1216             \__bnvs_n_parse_record:v
1217           }

```

```

1218         { key }
1219     } {
1220         \BNVS_error:n { Unexpected~key:~#1 }
1221     }
1222 } {
1223     \__bnvs_int_incr:c { }
1224     \__bnvs_tl_if_empty:cTF { n } {
1225         \__bnvs_parse_record:xn
1226     } {
1227         \__bnvs_n_parse_record:xn
1228     } {
1229         \__bnvs_tl_use:c { root } . \__bnvs_int_use:c { }
1230     } { #1 }
1231 }
1232 \use_none_delimit_by_s_stop:w
1233 }
1234 }
1235 #1 \s_stop
1236 }
1237 \BNVS_new:cpn { do_range:nnnn } #1 #2 #3 #4 {
1238     \__bnvs_gclear_all:n { #1 }
1239     \tl_if_empty:nTF { #4 } {
1240         \tl_if_empty:nTF { #2 } {
1241             \tl_if_empty:nTF { #3 } {
1242                 \BNVS_error:n { Not~a~range::~~#1 }
1243             } {
1244                 \__bnvs_gput:nnn Z { #1 } { #3 }
1245                 \__bnvs_gput:nnn V { #1 } { \q_nil }
1246             }
1247         } {
1248             \__bnvs_gput:nnn A { #1 } { #2 }
1249             \__bnvs_gput:nnn V { #1 } { \q_nil }
1250             \tl_if_empty:nF { #3 } {
1251                 \__bnvs_gput:nnn Z { #1 } { #3 }
1252                 \__bnvs_gput:nnn L { #1 } { \q_nil }
1253             }
1254         }
1255     } {
1256         \tl_if_empty:nTF { #2 } {
1257             \__bnvs_gput:nnn L { #1 } { #4 }
1258             \tl_if_empty:nF { #3 } {
1259                 \__bnvs_gput:nnn Z { #1 } { #3 }
1260                 \__bnvs_gput:nnn A { #1 } { \q_nil }
1261                 \__bnvs_gput:nnn V { #1 } { \q_nil }
1262             }
1263         } {
1264             \__bnvs_gput:nnn A { #1 } { #2 }
1265             \__bnvs_gput:nnn L { #1 } { #4 }
1266             \__bnvs_gput:nnn Z { #1 } { \q_nil }
1267             \__bnvs_gput:nnn V { #1 } { \q_nil }
1268         }
1269     }
1270 }
1271 \cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {

```

```

1272 \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
1273 { #4 }
1274 }
1275 \cs_new:Npn \BNVS_end_tl_set:cv #1 #2 {
1276 \BNVS_tl_use:nv {
1277 \BNVS_end: \__bnvs_tl_set:cn { #1 }
1278 } { #2 }
1279 }
1280 \BNVS_new:cpn { parse:nn } #1 #2 {
1281 \BNVS_begin:
1282 \__bnvs_tl_set:cn { a } { #1 }
1283 \__bnvs_tl_put_left:cv { a } { root }
1284 \__bnvs_name_id_n_get:vTF { a } {
1285 \regex_match:nnTF { \S } { #2 } {
1286 \peek_remove_spaces:n {
1287 \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.

```

1288 \__bnvs_tl_if_empty:cF { n } {
1289 \__bnvs_warning:n { Ignoring-unexpected-suffix~.n:~#1 }
1290 }
1291 \BNVS_begin:
1292 \__bnvs_tl_set:cv { root } { key }
1293 \int_set:Nn \l__bnvs_int { 0 }
1294 \cs_set:Npn \BNVS:nn ##1 ##2 \s_stop {
1295 \regex_match:nnT { \S } { ##2 } {
1296 \BNVS_error:n { Unexpected~value~#2 }
1297 }
1298 \keyval_parse:nnn {
1299 \__bnvs_parse:n
1300 } {
1301 \__bnvs_parse:nn
1302 } { ##1 }
1303 \BNVS_end:
1304 }
1305 \BNVS:nn
1306 } {
1307 \__bnvs_tl_if_empty:cTF { n } {
1308 \__bnvs_parse_record:vn
1309 } {
1310 \__bnvs_n_parse_record:vn
1311 }
1312 { key } { #2 }
1313 \use_none_delimit_by_s_stop:w
1314 }
1315 }
1316 #2 \s_stop
1317 } {

```

Empty value given: remove the reference.

```

1318 \__bnvs_tl_if_empty:cTF { n } {
1319 \__bnvs_gclear:v
1320 } {

```

```

1321         \__bnvs_n_gremove:v
1322     }
1323     { key }
1324 }
1325 } {
1326     \BNVS_error:n { Invalid-key:~#2 }
1327 }

```

We export \l__bnvs_id_last_tl:

```

1328     \BNVS_end_tl_set:cv { id_last } { id_last }
1329 }

```

```

1330 \BNVS_new:cpn { parse_prepare:N } #1 {
1331     \tl_set:Nx #1 #1
1332     \bool_set_false:N \l__bnvs_parse_bool
1333     \bool_do_until:Nn \l__bnvs_parse_bool {
1334         \tl_if_in:NnTF #1 {%---[
1335     ]} {
1336         \regex_replace_all:nnNF { \[ ([^\]]%---)
1337     ]*%---[(
1338     ) \] } { { { \1 } } } #1 {
1339         \bool_set_true:N \l__bnvs_parse_bool
1340     }
1341     } {
1342         \bool_set_true:N \l__bnvs_parse_bool
1343     }
1344 }
1345 \tl_if_in:NnTF #1 {%---[
1346 ]} {
1347     \BNVS_error:n { Unbalanced~%---[
1348 ]}
1349 } {
1350     \tl_if_in:NnT #1 { [%---]
1351     } {
1352         \BNVS_error:n { Unbalanced~[ %---]
1353     }
1354     }
1355 }
1356 }

```

\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.

```

1357 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1358     \BNVS_tl_use:nv {
1359         \BNVS_end:
1360         \__bnvs_tl_put_right:cn { #1 }
1361     } { #2 }
1362 }
1363 \cs_new:Npn \BNVS_end_v_gput:nc #1 #2 {
1364     \BNVS_tl_use:nv {
1365         \BNVS_end:

```

```

1366     \__bnvs_v_gput:nn { #1 }
1367   } { #2 }
1368 }
1369 \NewDocumentCommand \Beanoves { sm } {
1370   \tl_if_empty:NTF \@currentenv {

```

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

```

1371     \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
1372   } {
1373     \tl_if_eq:NnT \@currentenv { document } {

```

At the top level, clear everything.

```

1374     \__bnvs_gclear:
1375   }
1376   \BNVS_begin:
1377   \__bnvs_tl_clear:c { root }
1378   \int_zero:N \l__bnvs_int
1379   \__bnvs_tl_set:cn { a } { #2 }
1380   \tl_if_eq:NnT \@currentenv { document } {

```

At the top level, use the global definitions.

```

1381     \seq_if_empty:NF \g__bnvs_def_seq {
1382       \__bnvs_tl_put_left:cx { a } {
1383         \seq_use:Nn \g__bnvs_def_seq , ,
1384       }
1385     }
1386   }
1387   \__bnvs_parse_prepare:N \l__bnvs_a_tl
1388   \IfBooleanTF {#1} {
1389     \__bnvs_provide_on:
1390   } {
1391     \__bnvs_provide_off:
1392   }
1393   \BNVS_tl_use:nv {
1394     \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn }
1395   } { a }
1396   \BNVS_end_tl_set:cv { id_last } { id_last }
1397   \ignorespaces
1398 }
1399 }

```

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

```

1400 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}

```

6.13.6 Scanning named overlay specifications

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

<code>_bnvs@frame</code>	<code>_bnvs@frame {<overlay specification>}</code>
<code>_bnvs@masterdecode</code>	<code>_bnvs@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 of definition for `\l__bnvs_ans_tl`.)

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

```

1401 \makeatletter
1402 \cs_set:Npn \_bnvs@frame < #1 > {
1403   \BNVS_begin:
1404   \_bnvs_tl_clear:c { ans }
1405   \_bnvs_scan:nNc { #1 } \_bnvs_eval:nc { ans }
1406   \BNVS_tl_use:nv {
1407     \BNVS_end:
1408     \_bnvs_saved_beamer@frame <
1409   } { ans } >
1410 }
1411 \cs_set:Npn \_bnvs@masterdecode #1 {
1412   \BNVS_begin:
1413   \_bnvs_tl_clear:c { ans }
1414   \_bnvs_scan:nNc { #1 } \_bnvs_eval:nc { ans }
1415   \BNVS_tl_use:nv {
1416     \BNVS_end:
1417     \_bnvs_beamer@masterdecode
1418   } { ans }
1419 }
1420 \cs_new:Npn \BeanovesPrepare {
1421   \cs_if_exist_use:NTF \beamer@frame {
1422     \cs_set_eq:NN \_bnvs_saved_beamer@frame \beamer@frame
1423   } {
1424     \BNVS_error:n {Missing~package~beamer}
1425   }
1426   \cs_if_exist_use:NTF \beamer@masterdecode {
1427     \cs_set_eq:NN \_bnvs_saved_beamer@masterdecode \beamer@masterdecode
1428   } {
1429     \BNVS_error:n {Missing~package~beamer}
1430   }
1431 }
1432
1433 \cs_new:Npn \BeanovesOff {
1434   \cs_set_eq:NN \beamer@frame \_bnvs_saved_beamer@frame
1435   \cs_set_eq:NN \beamer@masterdecode \_bnvs_saved_beamer@masterdecode
1436 }
1437 \cs_new:Npn \BeanovesOn {
1438   \cs_set_eq:NN \beamer@frame \_bnvs@frame
1439   \cs_set_eq:NN \beamer@masterdecode \_bnvs@masterdecode
1440 }
1441 \makeatother
1442 \AddToHook{begindocument/before}{
1443   \BeanovesOn
1444 }
```

__bnvs_scan:nNc __bnvs_scan:nNc {*<named overlay expression>*} *<eval>* *<tl core>*
 Scan the *<named overlay expression>* argument and feed the *<tl variable>* replacing *?(...)* instructions by their static counterpart with help from the *<eval>* function, which is __bnvs_eval:nN. A group is created to use local variables:

\l__bnvs_ans_tl The token list that will be appended to *<tl variable>* on return.
(End of definition for \l__bnvs_ans_tl.)

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

\l__bnvs_query_tl Storage for the overlay query expression to be evaluated.
(End of definition for \l__bnvs_query_tl.)

\l__bnvs_token_seq The *<overlay expression>* is split into the sequence of its tokens.
(End of definition for \l__bnvs_token_seq.)

\l__bnvs_token_tl Storage for just one token.
(End of definition for \l__bnvs_token_tl.)
 Next are helpers.

__bnvs_scan_question:T __bnvs_scan_question:T {*<code>*}
 At top level state, scan the tokens of the *<named overlay expression>* looking for a ‘?’ character. If a ‘?(...)’ is found, then the *<code>* is executed.

```

1445 \BNVS_new:cpn { scan_question:T } #1 {
1446   \__bnvs_seq_pop_left:ccT { token } { token } {
1447     \__bnvs_tl_if_eq:cnTF { token } { ? } {
1448       \__bnvs_scan_require_open:
1449         #1
1450       } {
1451         \__bnvs_tl_put_right:cv { ans } { token }
1452       }
1453     }
1454     \__bnvs_scan_question:T { #1 }
1455   }
1456 }
```

__bnvs_scan_require_open: __bnvs_scan_require_open:

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

```

1457 \BNVS_new:cpn { scan_require_open: } {
```

Get next token.

```

1458 \__bnvs_seq_pop_left:ccTF { token } { token } {
1459   \tl_if_eq:NnTF \l__bnvs_token_tl { ( %)
1460   } {

```

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

```

1461   \__bnvs_int_set:cn { } { 1 }

```

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

```

1462   \__bnvs_tl_clear:c { query }
1463   \__bnvs_scan_require_close:
1464   } {

```

Ignore this token and loop.

```

1465   \__bnvs_scan_require_open:
1466   }
1467   } {

```

End reached but no opening parenthesis found, raise.

```

1468   \BNVS_fatal:x {Missing~'('%---)
1469   ~after~a~? }
1470   }
1471   }

```

```

\__bnvs_scan_require_close: \__bnvs_scan_require_close:

```

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

```

1472 \BNVS_new:cpn { scan_require_close: } {

```

Get next token.

```

1473 \__bnvs_seq_pop_left:ccTF { token } { token } {
1474   \__bnvs_tl_if_eq:cnTF { token } { ( %---)
1475   } {

```

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

```

1476   \__bnvs_int_incr:c { }
1477   \__bnvs_tl_put_right:cv { query } { token }
1478   \__bnvs_scan_require_close:
1479   } {

```

This is not a ‘(’.

```

1480   \__bnvs_tl_if_eq:cnTF { token } { %(---
1481   )
1482   } {

```

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

```

1483   \__bnvs_int_decr:c {}
1484   \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 ?.

```
1485         } {
```

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

```
1486         \__bnvs_tl_put_right:cv { query } { token }
1487         \__bnvs_scan_require_close:
1488     }
1489 } {
```

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

```
1490         \__bnvs_tl_put_right:cv { query } { token }
1491         \__bnvs_scan_require_close:
1492     }
1493 }
1494 } {
```

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 ‘)’.

```
1495     \BNVS_error:x { Missing~%(---
1496     `)' }
1497     \__bnvs_tl_put_right:cx { query } {
1498     \prg_replicate:nn { \l__bnvs_int } {%(---
1499     )}
1500 }
1501 }
1502 }
```

```
1503 \BNVS_new:cpn { scan:nNc } #1 #2 #3 {
1504     \BNVS_begin:
1505     \BNVS_set:cpn { fatal:x } ##1 {
1506         \msg_fatal:nnx { beanoves } { :n }
1507         { \tl_to_str:n { #1 } :~##1}
1508     }
1509     \BNVS_set:cpn { error:x } ##1 {
1510         \msg_error:nnx { beanoves } { :n }
1511         { \tl_to_str:n { #1 } :~##1}
1512     }
1513     \__bnvs_tl_set:cn { scan } { #1 }
1514     \__bnvs_tl_clear:c { ans }
1515     \__bnvs_seq_clear:c { token }
```

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

```
1516     \regex_split:nnN { } { #1 } \l__bnvs_token_seq
```

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

```
1517     \__bnvs_scan_question:T {
1518         \BNVS_tl_use:Nv #2 { query } { ans }
1519     }
1520     \BNVS_tl_use:nv {
1521         \BNVS_end:
1522         \__bnvs_tl_put_right:cn { #3 }
1523     } { ans }
1524 }
```

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.

```

1525 \exp_args_generate:n { VVx }
1526 \quark_new:N \q__bnvs
1527 \BNVS_new:cpn { end_kip_export_seq:nnccc } #1 #2 #3 #4 #5 #6 {
1528   \BNVS_end:
1529   \tl_if_empty:nTF { #2 } {
1530     \__bnvs_tl_set:cn { #4 } { #1 }
1531     \__bnvs_tl_put_left:cv { #4 } { #5 }
1532   } {
1533     \__bnvs_tl_set:cn { #4 } { #1 }
1534     \__bnvs_tl_set:cn { #5 } { #2 }
1535   }
1536   \__bnvs_seq_set_split:cn { #6 } { \q__bnvs } { #3 }
1537   \__bnvs_seq_remove_all:cn { #6 } { }
1538 }
1539 \BNVS_new:cpn { end_kip_export:ccc } {
1540   \exp_args:Nnnx \BNVS_tl_use:nv {
1541     \BNVS_tl_use:Nv \__bnvs_end_kip_export_seq:nnccc { key }
1542   } { id } {
1543     \__bnvs_seq_use:cn { path } { \q__bnvs }
1544   }
1545 }
1546 \BNVS_new_conditional:cpnn { match_pop_kip: } { T, F, TF } {
1547   \__bnvs_match_pop_left:cTF { key } {
1548     \__bnvs_match_pop_left:cTF { key } {
1549       \__bnvs_match_pop_left:cTF { id } {
1550         \__bnvs_match_pop_left:cTF { path } {
1551           \__bnvs_seq_set_split:cnv { path } { . } { path }
1552           \__bnvs_seq_remove_all:cn { path } { }
1553           \prg_return_true:
1554         } {
1555           \prg_return_false:
1556         }
1557       } {
1558         \prg_return_false:
1559       }
1560     } {
1561       \prg_return_false:
1562     }
1563   } {
1564     \prg_return_false:
1565   }

```

```

1566 }
1567 \BNVS_new_conditional:cpnn { kip:ccc } #1 #2 #3 { T, F, TF } {
1568   \BNVS_begin:
1569   \__bnvs_match_once:NvTF \c__bnvs_A_key_Z_regex { #1 } {

```

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

```

1570   \__bnvs_match_pop_kip:TF {
1571     \__bnvs_end_kip_export:ccc { #1 } { #2 } { #3 }
1572     \prg_return_true:
1573   } {
1574     \BNVS_end:
1575     \prg_return_false:
1576   }
1577 } {
1578   \BNVS_end:
1579   \prg_return_false:
1580 }
1581 }

```

```

\__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}

```

```

1582 \BNVS_new:cpn { kip_x_path_resolve:TFF } #1 #2 {
1583   \__bnvs_kip_x_path_resolve:TF {
1584     \__bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
1585   }
1586 }

```

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.

```

1587 \BNVS_new:cpn { end_kip_export: } {
1588   \exp_args:Nnnx
1589   \BNVS_tl_use:nv {
1590     \BNVS_tl_use:Nv \__bnvs_end_kip_export_seq:nnccc { key }
1591   } { id } {
1592     \__bnvs_seq_use:cn { path } { \q__bnvs }
1593   } { key } { id } { path }
1594 }
1595 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
1596   \__bnvs_seq_if_empty:cF { #2 } {
1597     \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
1598       \__bnvs_seq_use:cn { #1 } { \q__bnvs }
1599       \exp_not:n { \q__bnvs }
1600       \__bnvs_seq_use:cn { #2 } { \q__bnvs }
1601     }
1602     \__bnvs_seq_remove_all:cn { #1 } { }
1603   }
1604 }
1605 \BNVS_new:cpn { kip_x_path_resolve:nFF } #1 #2 #3 {
1606   \__bnvs_get:nvcTF #1 { a } { b } {
1607     \__bnvs_kip:cccTF { b } { id } { path } {
1608       \__bnvs_tl_set_eq:cc { key } { b }
1609       \__bnvs_seq_merge:cc { path } { b }
1610       \__bnvs_seq_clear:c { b }
1611       \__bnvs_seq_set_eq:cc { a } { path }
1612       \__bnvs_kip_x_path_resolve_loop_or_end_return:
1613     } {
1614       \__bnvs_seq_if_empty:cTF { b } {
1615         \__bnvs_tl_set_eq:cc { key } { b }
1616         \__bnvs_seq_clear:c { path }
1617         \__bnvs_seq_clear:c { a }
1618         \__bnvs_kip_x_path_resolve_loop_or_end_return:
1619       } {
1620         #2
1621       }
1622     }
1623   } {
1624     #3
1625   }
1626 }

```

```

1627 \BNVS_new:cpn { kip_x_path_resolve_VAL_loop_or_end_return:F } #1 {
1628   \__bnvs_kip_x_path_resolve:nFF V { #1 } {
1629     \__bnvs_kip_x_path_resolve:nFF A { #1 } {
1630       \__bnvs_kip_x_path_resolve:nFF L { #1 } { #1 }
1631     }
1632   }
1633 }
1634 \BNVS_new:cpn { kip_x_path_resolve_end_return_true: } {
1635   \__bnvs_seq_pop_left:ccTF { path } { a } {
1636     \__bnvs_seq_if_empty:cTF { path } {
1637       \__bnvs_tl_clear:c { b }
1638       \__bnvs_index_can:vTF { key } {
1639         \__bnvs_index_append:vcTF { key } { a } { b } {
1640           \__bnvs_tl_set:cv { key } { b }
1641         } {
1642           \__bnvs_tl_set:cv { key } { a }
1643         }
1644       } {
1645         \__bnvs_tl_set:cv { key } { a }
1646       }
1647     } {
1648       \BNVS_error:x { Path~too~long~.\BNVS_tl_use:c { a }
1649         .\__bnvs_seq_use:cn { path } . }
1650     }
1651   } {
1652     \__bnvs_value_resolve:vcT { key } { key } {}
1653   }
1654   \__bnvs_end_kip_export:
1655   \prg_return_true:
1656 }
1657 \BNVS_new_conditional:cpnn { kip_x_path_resolve: } { T, F, TF } {
1658   \BNVS_begin:
1659   \__bnvs_seq_set_eq:cc { a } { path }
1660   \__bnvs_seq_clear:c { b }
1661   \__bnvs_kip_x_path_resolve_loop_or_end_return:
1662 }
1663 \BNVS_new:cpn { kip_x_path_resolve_loop_or_end_return: } {
1664   \__bnvs_call:TF {
1665     \__bnvs_tl_set_eq:cc { a } { key }
1666     \__bnvs_seq_if_empty:cTF { a } {
1667       \__bnvs_kip_x_path_resolve_VAL_loop_or_end_return:F {
1668         \__bnvs_kip_x_path_resolve_end_return_true:
1669       }
1670     } {
1671       \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { a } . }
1672       \__bnvs_kip_x_path_resolve_VAL_loop_or_end_return:F {
1673         \__bnvs_seq_pop_right:ccT { a } { c } {
1674           \__bnvs_seq_put_left:cv { b } { c }
1675         }
1676       }
1677     }
1678   }
1679 } {
1680   \BNVS_end:

```

```

1681     \prg_return_false:
1682   }
1683 }

1684 \BNVS_new:cpn { kip_n_path_resolve_or_end_return:nF } #1 #2 {
1685   \__bnvs_get:nvcTF { #1 } { a } { b } {
1686     \__bnvs_kip:cccTF { b } { id } { path } {
1687       \__bnvs_tl_set_eq:cc { key } { b }
1688       \__bnvs_seq_merge:cc { path } { b }
1689       \__bnvs_seq_set_eq:cc { a } { path }
1690       \__bnvs_seq_clear:c { b }
1691       \__bnvs_kip_n_path_resolve_loop_or_end_return:
1692     } {
1693       \__bnvs_seq_pop_right:ccTF { a } { c } {
1694         \__bnvs_seq_put_left:cv { b } { c }
1695         \__bnvs_kip_n_path_resolve_loop_or_end_return:
1696       } {
1697         \__bnvs_kip_n_path_resolve_end_return_true:
1698       }
1699     }
1700   } {
1701     #2
1702   }
1703 }

1704 \BNVS_new:cpn { kip_n_path_resolve_VAL_loop_or_end_return: } {
1705   \__bnvs_kip_n_path_resolve_or_end_return:nF V {
1706     \__bnvs_kip_n_path_resolve_or_end_return:nF A {
1707       \__bnvs_kip_n_path_resolve_or_end_return:nF L {
1708         \__bnvs_seq_pop_right:ccTF { a } { c } {
1709           \__bnvs_seq_put_left:cv { b } { c }
1710           \__bnvs_kip_n_path_resolve_loop_or_end_return:
1711         } {
1712           \__bnvs_kip_n_path_resolve_end_return_true:
1713         }
1714       }
1715     }
1716   }
1717 }

1718 \BNVS_new:cpn { kip_n_path_resolve_end_return_false: } {
1719   \BNVS_end:
1720   \prg_return_false:
1721 }

1722 \BNVS_new:cpn { kip_n_path_resolve_end_return_true: } {
1723   \__bnvs_end_kip_export:
1724   \prg_return_true:
1725 }

```

__bnvs_kip_n_path_resolve_loop_or_end_return:

Loop to resolve the path.

```

1726 \BNVS_new:cpn { kip_n_path_resolve_loop_or_end_return: } {
1727   \__bnvs_call:TF {

```



```

1728     \__bnvs_tl_set_eq:cc { a } { key }
1729     \__bnvs_seq_if_empty:cTF { a } {
1730         \__bnvs_seq_if_empty:cTF { b } {
1731             \__bnvs_kip_n_path_resolve_end_return_true:
1732         } {
1733             \__bnvs_kip_n_path_resolve_VAL_loop_or_end_return:
1734         }
1735     } {
1736         \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { a } . }
1737         \__bnvs_kip_n_path_resolve_VAL_loop_or_end_return:
1738     }
1739 } {
1740     \BNVS_end:
1741     \prg_return_false:
1742 }
1743 }

```

`__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.

```

1744 \BNVS_new_conditional:cpnn { kip_n_path_resolve: } { T, F, TF } {
1745     \BNVS_begin:
1746     \__bnvs_seq_set_eq:cc { a } { path }
1747     \__bnvs_seq_clear:c { b }
1748     \__bnvs_kip_n_path_resolve_loop_or_end_return:
1749 }

```

6.13.8 Evaluation bricks

We start by helpers.

<code>__bnvs_round_ans:n</code>	<code>__bnvs_round:c <tl core name></code>
<code>__bnvs_round:c</code>	<code>__bnvs_round_ans:</code>
<code>__bnvs_round_ans:</code>	<code>__bnvs_round_ans:n {<expression>}</code>

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.

```

1750 \BNVS_new:cpn { round_ans:n } #1 {
1751     \tl_if_empty:nTF { #1 } {
1752         \__bnvs_tl_put_right:cn { ans } { 0 }

```

```

1753 } {
1754   \__bnvs_tl_put_right:cx { ans } { \fp_eval:n { round(#1) } }
1755 }
1756 }
1757 \BNVS_new:cpn { round:N } #1 {
1758   \tl_if_empty:NTF #1 {
1759     \tl_set:Nn #1 { 0 }
1760   } {
1761     \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
1762   }
1763 }
1764 \BNVS_new:cpn { round:c } {
1765   \BNVS_tl_use:Nc \__bnvs_round:N
1766 }

```

```

\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.

```

1767 \cs_new:Npn \BNVS_end_return_false: {
1768   \BNVS_end:
1769   \prg_return_false:
1770 }
1771 \cs_new:Npn \BNVS_end_return_false:x #1 {
1772   \BNVS_error:x { #1 }
1773   \BNVS_end_return_false:
1774 }

```

```

\__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.

```

1775 \BNVS_new:cpn { value_resolve_return:nnnT } #1 #2 #3 #4 {
1776   \__bnvs_tl_if_empty:cTF { #3 } {
1777     \prg_return_false:
1778   } {
1779     \__bnvs_cache_gput:nnv V { #2 } { #3 }
1780     #4
1781     \prg_return_true:
1782   }
1783 }

```

```

1784 \BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
1785   \BNVS_tl_use:Nc \quark_if_nil:NTF { #1 } {
1786     \prg_return_true:
1787   } {
1788     \prg_return_false:
1789   }
1790 }
1791 \BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
1792   \BNVS_tl_use:Nc \quark_if_no_value:NTF { #1 } {
1793     \prg_return_true:
1794   } {
1795     \prg_return_false:
1796   }
1797 }
1798 \BNVS_new_conditional:cpnn { value_resolve:nc } #1 #2 { T, F, TF } {
1799   \__bnvs_cache_get:nncTF V { #1 } { #2 } {
1800     \prg_return_true:
1801   } {
1802     \__bnvs_get:nncTF V { #1 } { #2 } {
1803       \__bnvs_quark_if_nil:cTF { #2 } {

```

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

```

1804   \__bnvs_gput:nnn V { #1 } { \q_no_value }
1805   \__bnvs_first_resolve:ncTF { #1 } { #2 } {
1806     \__bnvs_value_resolve_return:nnnT A { #1 } { #2 } {
1807       \__bnvs_gput:nnn V { #1 } { \q_nil }
1808     }
1809   } {
1810     \__bnvs_last_resolve:ncTF { #1 } { #2 } {
1811       \__bnvs_value_resolve_return:nnnT Z { #1 } { #2 } {
1812         \__bnvs_gput:nnn V { #1 } { \q_nil }
1813       }
1814     } {
1815       \__bnvs_gput:nnn V { #1 } { \q_nil }
1816       \prg_return_false:
1817     }
1818   }
1819 } {
1820   \__bnvs_quark_if_no_value:cTF { #2 } {
1821     \BNVS_fatal:n {Circular~definition:~#1}
1822   } {

```

Possible recursive call.

```

1823   \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1824     \__bnvs_value_resolve_return:nnnT V { #1 } { #2 } {
1825       \__bnvs_gput:nnn V { #1 } { \q_nil }
1826     }
1827   } {
1828     \__bnvs_gput:nnn V { #1 } { \q_nil }
1829     \prg_return_false:
1830   }
1831 }
1832 }
1833 } {

```

```

1834     \prg_return_false:
1835   }
1836 }
1837 }
1838 \BNVS_new_conditional:cpnn { value_resolve:vc } #1 #2 { T, F, TF } {
1839   \BNVS_tl_use:Nv \_bnvs_value_resolve:ncTF { #1 } { #2 } {
1840     \prg_return_true:
1841   } {
1842     \prg_return_false:
1843   }
1844 }
1845 \BNVS_new:cpn { end_put_right:vc } #1 #2 {
1846   \BNVS_tl_use:nv {
1847     \BNVS_end:
1848     \_bnvs_tl_put_right:cn { #2 }
1849   } { #1 }
1850 }
1851 \BNVS_new_conditional:cpnn { value_append:nc } #1 #2 { T, F, TF } {
1852   \BNVS_begin:
1853   \_bnvs_value_resolve:ncTF { #1 } { #2 } {
1854     \BNVS_end_tl_put_right:cv { #2 } { #2 }
1855     \prg_return_true:
1856   } {
1857     \BNVS_end:
1858     \prg_return_true:
1859   }
1860 }
1861 \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.

```

1862 \BNVS_new_conditional:cpnn { first_resolve:nc } #1 #2 { T, F, TF } {
1863   \_bnvs_cache_get:nncTF A { #1 } { #2 } {
1864     \prg_return_true:
1865   } {
1866     \_bnvs_get:nncTF A { #1 } { #2 } {
1867       \_bnvs_quark_if_nil:cTF { #2 } {
1868         \_bnvs_gput:nnn A { #1 } { \q_no_value }

```

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

```

1869   \_bnvs_last_resolve:ncTF { #1 } { #2 } {
1870     \_bnvs_tl_put_right:cn { #2 } { - }
1871   \_bnvs_length_append:ncTF { #1 } { #2 } {
1872     \_bnvs_tl_put_right:cn { #2 } { + 1 }
1873     \_bnvs_round:c { #2 }
1874     \_bnvs_tl_if_empty:cTF { #2 } {

```

```

1875         \_bnvs_gput:nnn A { #1 } { \q_nil }
1876     \prg_return_false:
1877 } {
1878     \_bnvs_gput:nnn A { #1 } { \q_nil }
1879     \_bnvs_cache_gput:nnv A { #1 } { #2 }
1880     \prg_return_true:
1881 }
1882 } {
1883     \BNVS_error:n {
1884 Unavailable~length~for~#1~(\token_to_str:N\_bnvs_first_resolve:ncTF/2) }
1885     \_bnvs_gput:nnn A { #1 } { \q_nil }
1886     \prg_return_false:
1887 }
1888 } {
1889     \BNVS_error:n {
1890 Unavailable~last~for~#1~(\token_to_str:N\_bnvs_first_resolve:ncTF/1) }
1891     \_bnvs_gput:nnn A { #1 } { \q_nil }
1892     \prg_return_false:
1893 }
1894 } {
1895     \_bnvs_quark_if_no_value:cTF { a } {
1896         \BNVS_fatal:n {Circular~definition:~#1}
1897     } {
1898         \_bnvs_if_resolve:vcTF { #2 } { #2 } {
1899             \_bnvs_cache_gput:nnv A { #1 } { #2 }
1900             \prg_return_true:
1901         } {
1902             \prg_return_false:
1903         }
1904     }
1905 }
1906 } {
1907     \prg_return_false:
1908 }
1909 }
1910 }
1911 \BNVS_new_conditional_vc:cn { first_resolve } { T, F, TF }
1912 \BNVS_new_conditional_cpnn { first_append:nc } #1 #2 { T, F, TF } {
1913     \BNVS_begin:
1914     \_bnvs_first_resolve:ncTF { #1 } { #2 } {
1915         \BNVS_end_tl_put_right:cv { #2 } { #2 }
1916         \prg_return_true:
1917     } {
1918         \prg_return_false:
1919     }
1920 }

```

```

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

```

1921 \BNVS_new_conditional:cpnn { last_resolve:nc } #1 #2 { T, F, TF } {
1922   \__bnvs_cache_get:nncTF Z { #1 } { #2 } {
1923     \prg_return_true:
1924   } {
1925     \__bnvs_get:nncTF Z { #1 } { #2 } {
1926       \__bnvs_quark_if_nil:cTF { #2 } {
1927         \__bnvs_gput:nnn Z { #1 } { \q_no_value }

```

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

```

1928     \__bnvs_first_resolve:ncTF { #1 } { #2 } {
1929       \__bnvs_tl_put_right:cn { #2 } { + }
1930       \__bnvs_length_append:ncTF { #1 } { #2 } {
1931         \__bnvs_tl_put_right:cn { #2 } { - 1 }
1932         \__bnvs_round:c { #2 }
1933         \__bnvs_cache_gput:nnv Z { #1 } { #2 }
1934         \__bnvs_gput:nnn Z { #1 } { \q_nil }
1935         \prg_return_true:
1936       } {
1937         \BNVS_error:x {
1938   Unavailable~length~for~#1~(\token_to_str:N \__bnvs_last_resolve:ncTF/1) }
1939         \__bnvs_gput:nnn Z { #1 } { \q_nil }
1940         \prg_return_false:
1941       }
1942     } {
1943       \BNVS_error:x {
1944   Unavailable~first~for~#1~(\token_to_str:N \__bnvs_last_resolve:ncTF/1) }
1945       \__bnvs_gput:nnn Z { #1 } { \q_nil }
1946       \prg_return_false:
1947     }
1948   } {
1949     \__bnvs_quark_if_no_value:cTF { #2 } {
1950       \BNVS_fatal:n {Circular~definition:~#1}
1951     } {
1952       \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1953         \__bnvs_cache_gput:nnv Z { #1 } { #2 }
1954         \prg_return_true:
1955       } {
1956         \prg_return_false:
1957       }
1958     }
1959   }
1960 } {
1961   \prg_return_false:
1962 }
1963 }
1964 }
1965 \BNVS_new_conditional_vc:cn { last_resolve } { T, F, TF }
1966 \prg_new_conditional:Npnn \__bnvs_last_append:nc #1 #2 { T, F, TF } {
1967   \BNVS_begin:
1968   \__bnvs_last_resolve:ncTF { #1 } { #2 } {
1969     \BNVS_end_tl_put_right:cv { #2 } { #2 }
1970     \prg_return_true:
1971   } {

```

```

1972     \BNVS_end:
1973     \prg_return_false:
1974   }
1975 }
1976 \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 *<key>* slide range into *<tl variable>*, or append the length of the *<key>* slide range to *<tl variable>*. Execute *<yes code>* when there is a *<length>*, *<no code>* otherwise.

```

1977 \BNVS_new_conditional:cpnn { length_resolve:nc } #1 #2 { T, F, TF } {
1978   \__bnvs_cache_get:nncTF L { #1 } { #2 } {
1979     \prg_return_true:
1980   } {
1981     \__bnvs_get:nncTF L { #1 } { #2 } {
1982       \__bnvs_quark_if_nil:cTF { #2 } {
1983         \__bnvs_gput:nnn L { #1 } { \q_no_value }

```

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

```

1984   \__bnvs_last_resolve:ncTF { #1 } { #2 } {
1985     \__bnvs_tl_put_right:cn { #2 } { - }
1986     \__bnvs_first_append:ncTF { #1 } { #2 } {
1987       \__bnvs_tl_put_right:cn { #2 } { + 1 }
1988       \__bnvs_round:c { #2 }
1989       \__bnvs_gput:nnn L { #1 } { \q_nil }
1990       \__bnvs_cache_gput:nnv L { #1 } { #2 }
1991       \prg_return_true:
1992     } {
1993       \BNVS_error:n {
1994         Unavailable~first~for~#1~(\__bnvs_length_resolve:ncTF/2) }
1995       \return_false:
1996     }
1997   } {
1998     \BNVS_error:n {
1999       Unavailable~last~for~#1~(\__bnvs_length_resolve:ncTF/1) }
2000     \return_false:
2001   }
2002 } {
2003   \__bnvs_quark_if_no_value:cTF { #2 } {
2004     \BNVS_fatal:n {Circular~definition:~#1}
2005   } {
2006     \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2007       \__bnvs_cache_gput:nnv L { #1 } { #2 }
2008       \prg_return_true:
2009     } {
2010       \prg_return_false:
2011     }
2012   }
2013 }
2014 } {

```

```

2015     \prg_return_false:
2016   }
2017 }
2018 }
2019 \BNVS_new_conditional_vc:cn { length_resolve } { T, F, TF }
2020 \BNVS_new_conditional:cpnn { length_append:nc } #1 #2 { T, F, TF } {
2021   \BNVS_begin:
2022   \__bnvs_length_resolve:ncTF { #1 } { #2 } {
2023     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2024     \prg_return_true:
2025   } {
2026     \prg_return_false:
2027   }
2028 }
2029 \BNVS_new_conditional_vc:cn { length_append } { T, F, TF }

```

```

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

```

2030 \BNVS_new_conditional:cpnn { range_append:nc } #1 #2 { T, F, TF } {
2031   \BNVS_begin:
2032   \__bnvs_first_resolve:ncTF { #1 } { a } {
2033     \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
2034       \__bnvs_tl_set:cn { a } { 0 }
2035     }
2036     \__bnvs_last_resolve:ncTF { #1 } { b } {

```

Limited from above and below.

```

2037     \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2038       \__bnvs_tl_set:cn { b } { 0 }
2039     }
2040     \__bnvs_tl_put_right:cn { a } { - }
2041     \__bnvs_tl_put_right:cv { a } { b }
2042     \BNVS_end_tl_put_right:cv { #2 } { a }
2043     \prg_return_true:
2044   } {

```

Limited from below.

```

2045     \BNVS_end_tl_put_right:cv { #2 } { a }
2046     \__bnvs_tl_put_right:cn { #2 } { - }
2047     \prg_return_true:
2048   }
2049 } {
2050   \__bnvs_last_resolve:ncTF { #1 } { b } {

```

Limited from above.

```

2051     \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2052       \__bnvs_tl_set:cn { b } { 0 }
2053     }
2054     \__bnvs_tl_put_left:cn { b } { - }
2055     \BNVS_end_tl_put_right:cv { #2 } { b }

```



```

2056     \prg_return_true:
2057   } {
2058     \__bnvs_value_resolve:ncTF { #1 } { b } {
2059       \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2060         \__bnvs_tl_set:cn { b } { 0 }
2061       }

```

Unlimited range.

```

2062     \BNVS_end_tl_put_right:cv { #2 } { b }
2063     \__bnvs_tl_put_right:cn { #2 } { - }
2064     \prg_return_true:
2065   } {
2066     \BNVS_end:
2067     \prg_return_false:
2068   }
2069 }
2070 }
2071 }
2072 \BNVS_new_conditional_vc:cn { range_append } { T, F, TF }
2073 \BNVS_new_conditional:cpnn { range_resolve:nc } #1 #2 { T, F, TF } {
2074   \__bnvs_tl_clear:c { #2 }
2075   \__bnvs_range_append:ncTF { #1 } { #2 } {
2076     \prg_return_true:
2077   } {
2078     \prg_return_false:
2079   }
2080 }
2081 \BNVS_new_conditional_vc:cn { range_resolve } { T, F, TF }

```

```

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

```

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

```

2082 \BNVS_new_conditional:cpnn { previous_resolve:nc } #1 #2 { T, F, TF } {
2083   \__bnvs_cache_get:nncTF P { #1 } { #2 } {
2084     \prg_return_true:
2085   } {
2086     \__bnvs_first_resolve:ncTF { #1 } { #2 } {
2087       \__bnvs_tl_put_right:cn { #2 } { -1 }
2088       \__bnvs_round:c { #2 }
2089       \__bnvs_cache_gput:nnv P { #1 } { #2 }
2090       \prg_return_true:
2091     } {
2092       \prg_return_false:
2093     }
2094   }
2095 }
2096 \BNVS_new_conditional_vc:cn { previous_resolve } { T, F, TF }
2097 \BNVS_new_conditional:cpnn { previous_append:nc } #1 #2 { T, F, TF } {

```

```

2098 \BNVS_begin:
2099 \__bnvs_previous_resolve:ncTF { #1 } { #2 } {
2100   \BNVS_end_t1_put_right:cv { #2 } { #2 }
2101   \prg_return_true:
2102 } {
2103   \BNVS_end:
2104   \prg_return_false:
2105 }
2106 }
2107 \BNVS_new_conditional_vc:cn { previous_append } { T, F, TF }

```

```

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

```

2108 \BNVS_new_conditional:cpnn { next_resolve:nc } #1 #2 { T, F, TF } {
2109   \__bnvs_cache_get:nncTF N { #1 } { #2 } {
2110     \prg_return_true:
2111   } {
2112     \__bnvs_last_resolve:ncTF { #1 } { #2 } {
2113       \__bnvs_t1_put_right:cn { #2 } { +1 }
2114       \__bnvs_round:c { #2 }
2115       \__bnvs_cache_gput:nv N { #1 } { #2 }
2116       \prg_return_true:
2117     } {
2118       \prg_return_false:
2119     }
2120   }
2121 }
2122 \BNVS_new_conditional_vc:cn { next_resolve } { T, F, TF }
2123 \BNVS_new_conditional:cpnn { next_append:nc } #1 #2 { T, F, TF } {
2124   \BNVS_begin:
2125   \__bnvs_next_resolve:ncTF { #1 } { #2 } {
2126     \BNVS_end_t1_put_right:cv { #2 } { #2 }
2127     \prg_return_true:
2128   } {
2129     \BNVS_end:
2130     \prg_return_true:
2131   }
2132 }
2133 \BNVS_new_conditional_vc:cn { next_append } { T, F, TF }

```

```

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

```

2134 \BNVS_new_conditional:cpnn { v_resolve:nc } #1 #2 { T, F, TF } {

```

```

2135 \__bnvs_v_get:ncTF { #1 } { #2 } {
2136   \__bnvs_quark_if_no_value:cTF { #2 } {
2137     \BNVS_fatal:n {Circular~definition:~#1}
2138     \prg_return_false:
2139   } {
2140     \prg_return_true:
2141   }
2142 } {
2143   \__bnvs_v_gput:nn { #1 } { \q_no_value }
2144   \__bnvs_value_resolve:ncTF { #1 } { #2 } {
2145     \__bnvs_v_gput:nv { #1 } { #2 }
2146     \prg_return_true:
2147   } {
2148     \__bnvs_first_resolve:ncTF { #1 } { #2 } {
2149       \__bnvs_v_gput:nv { #1 } { #2 }
2150       \prg_return_true:
2151     } {
2152       \__bnvs_last_resolve:ncTF { #1 } { #2 } {
2153         \__bnvs_v_gput:nv { #1 } { #2 }
2154         \prg_return_true:
2155       } {
2156         \__bnvs_v_gremove:n { #1 }
2157         \prg_return_false:
2158       }
2159     }
2160   }
2161 }
2162 }
2163 \BNVS_new_conditional_vc:cn { v_resolve } { T, F, TF }
2164 \BNVS_new_conditional_cpnn { v_append:nc } #1 #2 { T, F, TF } {
2165   \BNVS_begin:
2166   \__bnvs_v_resolve:ncTF { #1 } { #2 } {
2167     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2168     \prg_return_true:
2169   } {
2170     \BNVS_end:
2171     \prg_return_false:
2172   }
2173 }
2174 \BNVS_new_conditional_vc:cn { v_append } { T, F, TF }

```

```

2175 \__bnvs_index_can:nTF      \__bnvs_index_can:nTF {<key>} {<yes code>} {<no code>}
2176 \__bnvs_index_can:vTF      \__bnvs_index_resolve:nncTF {<key>} {<integer>} {<tl core name>} {<yes code>}
2177 \__bnvs_index_resolve:nncTF {<no code>}
2178 \__bnvs_index_resolve:vvcTF \__bnvs_index_append:nncTF {<key>} {<integer>} {<tl core name>} {<yes code>}
2179 \__bnvs_index_append:nncTF {<no code>}
2180 \__bnvs_index_append:vvcTF

```

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.

```

2175 \BNVS_new_conditional:cpnn { index_can:n } #1 { p, T, F, TF } {
2176   \bool_if:nTF {
2177     \__bnvs_if_in_p:nn V { #1 }
2178     || \__bnvs_if_in_p:nn A { #1 }
2179     || \__bnvs_if_in_p:nn Z { #1 }
2180   } {
2181     \prg_return_true:
2182   } {
2183     \prg_return_false:
2184   }
2185 }
2186 \BNVS_new_conditional:cpnn { index_can:v } #1 { p, T, F, TF } {
2187   \BNVS_tl_use:Nv \__bnvs_index_can:nTF { #1 } {
2188     \prg_return_true:
2189   } {
2190     \prg_return_false:
2191   }
2192 }
2193 \BNVS_new_conditional:cpnn { index_resolve:nnc } #1 #2 #3 { T, F, TF } {
2194   \exp_args:Nx \__bnvs_value_resolve:ncTF { #1.#2 } { #3 } {
2195     \prg_return_true:
2196   } {
2197     \__bnvs_first_resolve:ncTF { #1 } { #3 } {
2198       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2199       \__bnvs_round:c { #3 }
2200     \prg_return_true:

```

Limited overlay set.

```

2201   } {
2202     \__bnvs_last_resolve:ncTF { #1 } { #3 } {
2203       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2204       \__bnvs_round:c { #3 }
2205     \prg_return_true:
2206   } {
2207     \__bnvs_value_resolve:ncTF { #1 } { #3 } {
2208       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2209       \__bnvs_round:c { #3 }
2210     \prg_return_true:
2211   } {

```

```

2212         \prg_return_false:
2213     }
2214 }
2215 }
2216 }
2217 }
2218 \BNVS_new_conditional:cpnn { index_resolve:nvc } #1 #2 #3 { T, F, TF } {
2219     \BNVS_tl_use:nv {
2220         \__bnvs_index_resolve:nncTF { #1 }
2221     } { #2 } { #3 } {
2222         \prg_return_true:
2223     } {
2224         \prg_return_false:
2225     }
2226 }
2227 \BNVS_new_conditional:cpnn { index_resolve:vvc } #1 #2 #3 { T, F, TF } {
2228     \BNVS_tl_use:nv {
2229         \BNVS_tl_use:Nv \__bnvs_index_resolve:nncTF { #1 }
2230     } { #2 } { #3 } {
2231         \prg_return_true:
2232     } {
2233         \prg_return_false:
2234     }
2235 }
2236 \BNVS_new_conditional:cpnn { index_append:nnc } #1 #2 #3 { T, F, TF } {
2237     \BNVS_begin:
2238     \__bnvs_index_resolve:nncTF { #1 } { #2 } { #3 } {
2239         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2240         \prg_return_true:
2241     } {
2242         \BNVS_end:
2243         \prg_return_false:
2244     }
2245 }
2246 \BNVS_new_conditional:cpnn { index_append:vvc } #1 #2 #3 { T, F, TF } {
2247     \BNVS_tl_use:nv {
2248         \BNVS_tl_use:Nv \__bnvs_index_append:nncTF { #1 }
2249     } { #2 } { #3 } {
2250         \prg_return_true:
2251     } {
2252         \prg_return_false:
2253     }
2254 }

```

6.13.9 Index counter

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

__bnvs_n_append:ncTF

__bnvs_n_append:VcTF

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

```

2255 \BNVS_new_conditional:cpnn { n_resolve:nc } #1 #2 { T, F, TF } {

```

```

2256 \__bnvs_n_get:ncF { #1 } { #2 } {
2257   \__bnvs_tl_set:cn { #2 } { 1 }
2258   \__bnvs_n_gput:nn { #1 } { 1 }
2259 }
2260 \prg_return_true:
2261 }
2262 \BNVS_new_conditional:cpnn { n_append:nc } #1 #2 { T, F, TF } {
2263   \BNVS_begin:
2264     \__bnvs_n_resolve:ncTF { #1 } { #2 } {
2265       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2266       \prg_return_true:
2267     } {
2268       \BNVS_end:
2269       \prg_return_false:
2270     }
2271 }
2272 \BNVS_new_conditional_vc:cn { n_append } { T, F, TF }

```

```

\__bnvs_n_index_resolve:ncTF \__bnvs_n_index_resolve:ncTF {<key>} <tl variable> {<yes code>} {<no
\__bnvs_n_index_append:ncTF   code>}
\__bnvs_n_index_resolve:nncTF \__bnvs_n_index_append:ncTF {<key>} <tl variable> {<yes code>} {<no code>}
\__bnvs_n_index_append:nncTF \__bnvs_n_index_resolve:nncTF {<key>} {<base key>} <tl variable> {<yes
code>} {<no code>}
\__bnvs_n_index_append:nncTF {<key>} {<base key>} <tl variable> {<yes
code>} {<no code>}

```

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

```

2273 \BNVS_new_conditional:cpnn { n_index_resolve:nc } #1 #2 { T, F, TF } {
2274   \__bnvs_n_resolve:ncTF { #1 } { #2 } {
2275     \__bnvs_index_resolve:nvcTF { #1 } { #2 } { #2 } {
2276       \prg_return_true:
2277     } {
2278       \prg_return_false:
2279     }
2280   } {
2281     \prg_return_false:
2282   }
2283 }
2284 \BNVS_new_conditional:cpnn { n_index_resolve:nnc } #1 #2 #3 { T, F, TF } {
2285   \__bnvs_n_resolve:ncTF { #1 } { #3 } {
2286     \__bnvs_tl_put_left:cn { #3 } { #2. }
2287     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2288       \prg_return_true:
2289     } {
2290       \prg_return_false:
2291     }
2292   } {

```

```

2293     \prg_return_false:
2294 }
2295 }
2296 \BNVS_new_conditional:cpnn { n_index_append:nc } #1 #2 { T, F, TF } {
2297     \BNVS_begin:
2298     \__bnvs_n_index_resolve:ncTF { #1 } { #2 } {
2299         \BNVS_end_tl_put_right:cv { #2 } { #2 }
2300         \prg_return_true:
2301     } {
2302         \BNVS_end:
2303         \prg_return_false:
2304     }
2305 }
2306 \BNVS_new_conditional:cpnn { n_index_append:nnc } #1 #2 #3 { T, F, TF } {
2307     \BNVS_begin:
2308     \__bnvs_n_index_resolve:nncTF { #1 } { #2 } { #3 } {
2309         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2310         \prg_return_true:
2311     } {
2312         \BNVS_end:
2313         \prg_return_false:
2314     }
2315 }
2316 \BNVS_new_conditional_vc:cn { n_index_append } { T, F, TF }
2317 \BNVS_new_conditional_vvc:cn { n_index_append } { T, F, TF }

```

6.13.10 Value counter

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

```

2318 \BNVS_new_conditional:cpnn { v_incr_resolve:nnc } #1 #2 #3 { T, F, TF } {
2319     \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2320         \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2321             \__bnvs_v_resolve:ncTF { #1 } { #3 } {
2322                 \prg_return_true:
2323             } {
2324                 \prg_return_false:
2325             }
2326         } {
2327             \__bnvs_tl_put_right:cn { #3 } { + }
2328             \__bnvs_v_append:ncTF { #1 } { #3 } {
2329                 \__bnvs_round:c { #3 }
2330                 \__bnvs_v_gput:nv { #1 } { #3 }
2331                 \prg_return_true:
2332             } {

```

```

2333         \prg_return_false:
2334     }
2335 }
2336 } {
2337     \prg_return_false:
2338 }
2339 }
2340 \BNVS_new_conditional_vnc:cn { v_incr_resolve } { T, F, TF }
2341 \BNVS_new_conditional_cpnn { v_incr_append:nnc } #1 #2 #3 { T, F, TF } {
2342     \BNVS_begin:
2343     \__bnvs_v_incr_resolve:nncTF { #1 } { #2 } { #3 } {
2344         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2345         \prg_return_true:
2346     } {
2347         \prg_return_false:
2348     }
2349 }
2350 \BNVS_new_conditional_vnc:cn { v_incr_append } { T, F, TF }
2351 \BNVS_new_conditional_vvc:cn { v_incr_append } { T, F, TF }
2352 \BNVS_new_conditional_cpnn { v_post_resolve:nnc } #1 #2 #3 { T, F, TF } {
2353     \__bnvs_v_resolve:ncTF { #1 } { #3 } {
2354         \BNVS_begin:
2355         \__bnvs_if_resolve:ncTF { #2 } { a } {
2356             \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
2357                 \BNVS_end:
2358                 \prg_return_true:
2359             } {
2360                 \__bnvs_tl_put_right:cn { a } { + }
2361                 \__bnvs_tl_put_right:cv { a } { #3 }
2362                 \__bnvs_round:c { a }
2363                 \BNVS_end_v_gput:nc { #1 } { a }
2364                 \prg_return_true:
2365             }
2366         } {
2367             \BNVS_end:
2368             \prg_return_false:
2369         }
2370     } {
2371         \prg_return_false:
2372     }
2373 }
2374 \BNVS_new_conditional_vvc:cn { v_post_resolve } { T, F, TF }
2375 \BNVS_new_conditional_cpnn { v_post_append:nnc } #1 #2 #3 { T, F, TF } {
2376     \BNVS_begin:
2377     \__bnvs_v_post_resolve:nncTF { #1 } { #2 } { #3 } {
2378         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2379         \prg_return_true:
2380     } {
2381         \prg_return_true:
2382     }
2383 }
2384 \BNVS_new_conditional_vnc:cn { v_post_append } { T, F, TF }
2385 \BNVS_new_conditional_vvc:cn { v_post_append } { T, F, TF }

```

<code>__bnvs_n_incr_resolve:nnncTF</code>	<code>__bnvs_n_incr_resolve:nnctf {<key>} {<base key>} {<offset>} <tl core name> {<yes code>} {<no code>}</code>
<code>__bnvs_n_incr_resolve:vncTF</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:vncTF</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>`.

```

2386 \BNVS_new_conditional:cpnn { n_incr_resolve:nnnc } #1 #2 #3 #4 { T, F, TF } {
2387   \__bnvs_if_resolve:ncTF { #3 } { #4 } {
2388     \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
2389       \__bnvs_n_resolve:ncTF { #1 } { #4 } {
2390         \__bnvs_index_resolve:nvcTF { #1 } { #4 } { #4 } {
2391           \prg_return_true:
2392         } {
2393           \prg_return_false:
2394         }
2395       } {
2396         \prg_return_false:
2397       }
2398     } {
2399       \__bnvs_tl_put_right:cn { #4 } { + }
2400       \__bnvs_n_append:ncTF { #1 } { #4 } {
2401         \__bnvs_round:c { #4 }
2402         \__bnvs_n_gput:nv { #1 } { #4 }
2403         \__bnvs_index_resolve:nvcTF { #2 } { #4 } { #4 } {
2404           \prg_return_true:
2405         } {
2406           \prg_return_false:
2407         }
2408       } {
2409         \prg_return_false:
2410       }
2411     }
2412   } {
2413     \prg_return_false:
2414   }
2415 }
2416 \BNVS_new_conditional:cpnn { n_incr_resolve:nnnc } #1 #2 #3 { T, F, TF } {
2417   \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2418     \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2419       \__bnvs_n_resolve:ncTF { #1 } { #3 } {
2420         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2421           \prg_return_true:
2422         } {
2423           \prg_return_false:
2424         }
2425       } {

```

```

2426     \prg_return_false:
2427 }
2428 } {
2429     \__bnvs_tl_put_right:cn { #3 } { + }
2430     \__bnvs_n_append:ncTF { #1 } { #3 } {
2431         \__bnvs_round:c { #3 }
2432         \__bnvs_n_gput:nv { #1 } { #3 }
2433         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2434             \prg_return_true:
2435         } {
2436             \prg_return_false:
2437         }
2438     } {
2439         \prg_return_false:
2440     }
2441 }
2442 } {
2443     \prg_return_false:
2444 }
2445 }
2446 \BNVS_new_conditional_vnc:cn { n_incr_resolve } { T, F, TF }
2447 \BNVS_new_conditional_vvc:cn { n_incr_resolve } { T, F, TF }
2448 \BNVS_new_conditional_cpnn { n_incr_append:nnnc } #1 #2 #3 #4 { T, F, TF } {
2449     \BNVS_begin:
2450     \__bnvs_n_incr_resolve:nnncTF { #1 } { #2 } { #3 } { #4 }{
2451         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2452         \prg_return_true:
2453     } {
2454         \BNVS_end:
2455         \prg_return_false:
2456     }
2457 }
2458 \BNVS_new_conditional_vvnc:cn { n_incr_append } { T, F, TF }
2459 \BNVS_new_conditional_vvvc:cn { n_incr_append } { T, F, TF }
2460 \BNVS_new_conditional_cpnn { n_incr_append:nnc } #1 #2 #3 { T, F, TF } {
2461     \BNVS_begin:
2462     \__bnvs_n_incr_resolve:nncTF { #1 } { #2 } { #3 } {
2463         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2464         \prg_return_true:
2465     } {
2466         \BNVS_end:
2467         \prg_return_false:
2468     }
2469 }
2470 \BNVS_new_conditional_vnc:cn { n_incr_append } { T, F, TF }
2471 \BNVS_new_conditional_vvc:cn { n_incr_append } { T, F, TF }

```

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

```

2472 \BNVS_new_conditional:cpnn { n_post_resolve:nnc } #1 #2 #3 { T, F, TF } {
2473   \__bnvs_n_resolve:ncTF { #1 } { #3 } {
2474     \BNVS_begin:
2475     \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2476       \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2477         \BNVS_end:
2478         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2479           \prg_return_true:
2480         } {
2481           \prg_return_false:
2482         }
2483       } {
2484         \__bnvs_tl_put_right:cn { #3 } { + }
2485         \__bnvs_n_append:ncTF { #1 } { #3 } {
2486           \__bnvs_round:c { #3 }
2487           \__bnvs_n_gput:nv { #1 } { #3 }
2488         \BNVS_end:
2489         \__bnvs_index_resolve:nvcTF { #1 } { #3 } { #3 } {
2490           \prg_return_true:
2491         } {
2492           \prg_return_false:
2493         }
2494       } {
2495         \BNVS_end:
2496         \prg_return_false:
2497       }
2498     }
2499   } {
2500     \BNVS_end:
2501     \prg_return_false:
2502   }
2503 } {
2504   \prg_return_false:
2505 }
2506 }
2507 \BNVS_new_conditional:cpnn { n_post_append:nnc } #1 #2 #3 { T, F, TF } {
2508   \BNVS_begin:
2509   \__bnvs_n_post_resolve:nncTF { #1 } { #2 } { #3 } {
2510     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2511     \prg_return_true:
2512   } {
2513     \BNVS_end:
2514     \prg_return_false:
2515   }
2516 }
2517 \BNVS_new_conditional_vnc:cn { n_post_append } { T, F, TF }
2518 \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.

```

2519 \BNVS_set:cpn { round_ans: } {
2520   \_bnvs_round:c { ans }
2521 }
```

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⟩}`

```

2522 \BNVS_new:cpn { path_resolve_n:TFF } #1 #2 {
2523   \_bnvs_kip_n_path_resolve:TF {
2524     \_bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
2525   }
2526 }
```

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

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

```

2527 \BNVS_new:cpn { if_resolve_end_return_false:n } #1 {
2528   \BNVS_end:
2529   \prg_return_false:
2530 }
2531 \BNVS_new:cpn { path_resolve_n:T } #1 {
2532   \_bnvs_path_resolve_n:TFF {
2533     #1
2534   } {
2535     \_bnvs_if_resolve_end_return_false:n {
2536       Too~many~dotted~components
2537     }
2538   } {
2539     \_bnvs_if_resolve_end_return_false:n {
2540       Unknown~dotted~path
2541     }
2542   }
2543 }
```

```

2544 \BNVS_set:cpn { resolve_x:T } #1 {
2545   \__bnvs_kip_x_path_resolve:TFF {
2546     #1
2547   } {
2548     \__bnvs_if_resolve_end_return_false:n {
2549       Too-many-dotted-components
2550     }
2551   } {
2552     \__bnvs_if_resolve_end_return_false:n { Unknown-dotted-path }
2553   }
2554 }

```

__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.

```

2555 \BNVS_new_conditional:cpnn { path_pop_right_n:c } #1 { T, F, TF } {
2556   \__bnvs_seq_pop_right:ccTF { path } { #1 }
2557   { \prg_return_true: } { \prg_return_false: }
2558 }

```

__bnvs_if_resolve_pop_kip:TTF	__bnvs_if_resolve_pop_kip:TTF {<blank code>} {<black code>}
__bnvs_if_resolve_pop_complete_white:T	{<end code>}
__bnvs_if_resolve_pop_complete_black:T	__bnvs_if_resolve_pop_complete_white:T {<blank code>}
	__bnvs_if_resolve_pop_complete_black:T {<black code>}

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

For __bnvs_if_resolve_pop_complete_white:T: pops the three heading items of the split sequence into the three variables n_incr, incr, post. Then execute <blank code>.

For __bnvs_if_resolve_pop_complete_black:T: pops the six heading items of the split sequence then execute <blank code>.

```

2559 \BNVS_new:cpn { if_resolve_pop_kip_complete: } {
2560   \__bnvs_tl_if_blank:vT { id } {
2561     \__bnvs_tl_put_left:cv { key } { id_last }
2562     \__bnvs_tl_set:cv { id } { id_last }
2563   }
2564   \__bnvs_tl_if_blank:vTF { path } {
2565     \__bnvs_seq_clear:c { path }
2566   } {
2567     \__bnvs_seq_set_split:cnv { path } { . } { path }
2568     \__bnvs_seq_remove_all:cn { path } { }
2569   }
2570   \__bnvs_tl_set_eq:cc { key_base } { key }
2571   \__bnvs_seq_set_eq:cc { path_base } { path }
2572 }
2573 \BNVS_new:cpn { if_resolve_pop_kip:TTF } #1 #2 #3 {

```

```

2574 \__bnvs_split_pop_left:cTF { key } {
2575 \__bnvs_split_pop_left:cTF { id } {
2576 \__bnvs_split_pop_left:cTF { path } {
2577 \__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.

```

2578 #1
2579 } {
2580 \__bnvs_if_resolve_pop_kip_complete:
2581 #2
2582 }
2583 } {
2584 \__bnvs_end_unreachable_return_false:n { if_resolve_pop_kip:TTF/2 }
2585 }
2586 } {
2587 \__bnvs_end_unreachable_return_false:n { if_resolve_pop_kip:TTF/1 }
2588 }
2589 } { #3 }
2590 }

```

```

\__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.

```

2591 \BNVS_new:cpn { if_resolve_pop_complete_white:T } #1 {
2592 \__bnvs_split_pop_left:cTF { n_incr } {
2593 \__bnvs_split_pop_left:cTF { incr } {
2594 \__bnvs_split_pop_left:cTF { post } {
2595 #1
2596 } {
2597 \__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/3 }
2598 }
2599 } {
2600 \__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/2 }
2601 }
2602 } {
2603 \__bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_white:T/1 }
2604 }
2605 }
2606 \BNVS_new:cpn { if_resolve_pop_complete_black:T } #1 {
2607 \__bnvs_split_pop_left:cTF { a } {
2608 \__bnvs_split_pop_left:cTF { a } {

```

```

2609     \_bnvs_split_pop_left:cTF { a } {
2610         \_bnvs_split_pop_left:cTF { a } {
2611             \_bnvs_split_pop_left:cTF { a } {
2612                 \_bnvs_split_pop_left:cTF { a } {
2613                     #1
2614                 } {
2615     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/6 }
2616         }
2617     } {
2618     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/5 }
2619         }
2620     } {
2621     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/4 }
2622         }
2623     } {
2624     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/3 }
2625         }
2626     } {
2627     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/2 }
2628         }
2629     } {
2630     \_bnvs_end_unreachable_return_false:n { if_resolve_pop_complete_black:T/1 }
2631         }
2632 }

```

<code>_bnvs_if_resolve:ncTF</code> <code>_bnvs_if_resolve:vcTF</code> <code>_bnvs_if_append:ncTF</code> <code>_bnvs_if_append:(vc xc)TF</code>	<code>_bnvs_if_append:ncTF {<expression>} <tl variable> {<yes code>} {<no code>}</code> 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:
---	--

`\l__bnvs_ans_tl` To feed *<tl variable>* with.

(End of definition for `\l__bnvs_ans_tl`.)

`\l__bnvs_split_seq` The sequence of caught query groups and non queries.

(End of definition for `\l__bnvs_split_seq`.)

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

(End of definition for `\l__bnvs_split_int`.)

2633 `\BNVS_int_new:c { split }`

`\l__bnvs_key_tl` Storage for `split` sequence items that represent names.

(End of definition for `\l__bnvs_key_tl`.)

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

(End of definition for `\l__bnvs_path_tl`.)

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

```

2634 \BNVS_new:cpn { kip_x_path_resolve_or_end_return_false:nt } #1 #2 {
2635   \_bnvs_kip_x_path_resolve:TFF {
2636     #2
2637   } {
2638     \BNVS_end_return_false:x { Too-many-dotted-components:~#1 }
2639   } {
2640     \BNVS_end_return_false:x { Unknown-dotted-path:~#1 }
2641   }
2642 }
2643 \BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { T, F, TF } {
2644   \BNVS_begin:
2645     \_bnvs_if_resolve:ncTF { #1 } { #2 } {
2646       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2647     }
2648     \BNVS_DEBUG_log_if_append_ncTF:nn { ... } { ...TRUE }
2649   }
2650   \prg_return_true:
2651 } {
2652   \BNVS_end:
2653 }
2654 \BNVS_DEBUG_log_if_append_ncTF:nn { ... } { ...FALSE }
2655 }
2656 \prg_return_false:
2657 }
```



```

2658 }
2659 \BNVS_new:cpn { end_unreachable_return_false:n } #1 {
2660   \BNVS_error:x { UNREACHABLE/#1 }
2661   \BNVS_end:
2662   \prg_return_false:
2663 }
2664 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { T, F, TF } {
2665   \__bnvs_call:TF {
2666     \BNVS_begin:
2667     \BNVS_set:cpn { if_resolve_warning:n } ##1 {
2668       \__bnvs_warning:n { #1:~##1 }
2669       \BNVS_set:cpn { if_resolve_warning:n } {
2670         \use_none:n
2671       }
2672     }

```

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

```

2673   \__bnvs_regex_split:cnTF { split } { #1 } {

```

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

```

2674     \BNVS_set:cpn { if_resolve_end_return_true: } {

```

Normal and unique end of the loop.

```

2675       \__bnvs_if_resolve_round_ans:
2676       \BNVS_tl_use:nv {
2677         \BNVS_end:
2678         \__bnvs_tl_set:cn { #2 }
2679       } { ans }
2680       \prg_return_true:
2681     }
2682     \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round_ans: }
2683     \__bnvs_tl_clear:c { ans }
2684     \__bnvs_if_resolve_loop_or_end_return:
2685   } {
2686     \__bnvs_tl_clear:c { ans }
2687     \__bnvs_round_ans:n { #1 }
2688     \BNVS_end_tl_set:cv { #2 } { ans }
2689     \prg_return_true:
2690   }
2691 } {
2692   \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
2693   \prg_return_false:
2694 }
2695 }
2696 \BNVS_new_conditional:cpnn { if_append:vc } #1 #2 { T, F, TF } {
2697   \BNVS_tl_use:Nv \__bnvs_if_append:ncTF { #1 } { #2 } {
2698     \prg_return_true:
2699   } {
2700     \prg_return_false:
2701   }
2702 }
2703 \BNVS_new_conditional:cpnn { if_resolve:vc } #1 #2 { T, F, TF } {
2704   \BNVS_tl_use:Nv \__bnvs_if_resolve:ncTF { #1 } { #2 } {
2705     \prg_return_true:

```

```

2706 } {
2707   \prg_return_false:
2708 }
2709 }

```

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.

```

2710 \BNVS_new:cpn { if_resolve_loop_or_end_return: } {
2711   \__bnvs_split_pop_left:cTF { a } {
2712     \__bnvs_tl_put_right:cv { ans } { a }
2713     \__bnvs_if_resolve_pop_kip:TTF {
2714       \__bnvs_if_resolve_pop_kip:TTF {
2715         \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/3 }
2716       } {
2717         \__bnvs_if_resolve_pop_complete_white:T {
2718           \__bnvs_tl_if_blank:vTF { n_incr } {
2719             \__bnvs_tl_if_blank:vTF { incr } {
2720               \__bnvs_tl_if_blank:vTF { post } {
2721                 \__bnvs_if_resolve_value_loop_or_end_return_true:F {

```

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

```

2722           \__bnvs_seq_pop_right:ccTF { path } { a } {
2723             \BNVS_tl_use:Nv \str_case:nnF { a } {
2724               { n      } { \BNVS_use:c { if_resolve_loop_or_end_return[.n]: } }
2725               { length } { \BNVS_use:c { if_resolve_loop_or_end_return[.length]: } }
2726               { last   } { \BNVS_use:c { if_resolve_loop_or_end_return[.last]: } }
2727               { range  } { \BNVS_use:c { if_resolve_loop_or_end_return[.range]: } }
2728               { previous } { \BNVS_use:c { if_resolve_loop_or_end_return[.previous]: } }
2729               { next    } { \BNVS_use:c { if_resolve_loop_or_end_return[.next]: } }
2730               { reset   } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset]: } }
2731               { reset_all } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset_all]: } }
2732             } {
2733               \BNVS_use:c { if_resolve_loop_or_end_return[...<integer>]: }
2734             }
2735           } {
2736             \BNVS_use:c { if_resolve_loop_or_end_return[...]: }
2737           }
2738         }
2739       } {
2740         \BNVS_use:c { if_resolve_loop_or_end_return[...++]: }
2741       }
2742     } {
2743       \__bnvs_path_suffix:nTF { n } {
2744         \BNVS_use:c { if_resolve_loop_or_end_return[...n+=...]: }
2745       } {
2746         \BNVS_use:c { if_resolve_loop_or_end_return[...+=...]: }
2747       }
2748     }
2749   } {

```

```

2750 \BNVS_use:c { if_resolve_loop_or_end_return[...++n]: }
2751     }
2752     }
2753     } {
2754 % split sequence empty
2755 \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/2 }
2756     }
2757     } {
2758         \__bnvs_if_resolve_pop_complete_black:T {
2759             \__bnvs_path_suffix:nTF { n } {
2760 \BNVS_use:c { if_resolve_loop_or_end_return[+...n]: }
2761         } {
2762 \BNVS_use:c { if_resolve_loop_or_end_return[+...]: }
2763         }
2764     }
2765     } {
2766         \__bnvs_if_resolve_end_return_true:
2767     }
2768     } {
2769 \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/1 }
2770     }
2771 }
2772 \BNVS_set:cpn { if_resolve_value_loop_or_end_return_true:F } #1 {
2773     \__bnvs_tl_set:cx { a } {
2774         \BNVS_tl_use:c { key } \BNVS_tl_use:c { path }
2775     }
2776     \__bnvs_v_resolve:vcTF { a } { a } {
2777         \__bnvs_tl_put_right:cv { ans } { a }
2778         \__bnvs_if_resolve_loop_or_end_return:
2779     } {
2780         \__bnvs_value_resolve:vcTF { a } { a } {
2781             \__bnvs_tl_put_right:cv { ans } { a }
2782             \__bnvs_if_resolve_loop_or_end_return:
2783         } {
2784             #1
2785         }
2786     }
2787 }
2788 \BNVS_new:cpn { end_return_error:n } #1 {
2789     \BNVS_error:n { #1 }
2790     \BNVS_end:
2791     \prg_return_false:
2792 }
2793 \BNVS_new:cpn { if_resolve_loop_or_end_return[.n]: } {

```

- Case ...n.

```

2794 \__bnvs_path_resolve_n:T {
2795     \__bnvs_base_resolve_n:
2796     \__bnvs_n_index_append:vcTF { key } { key_base } { ans } {
2797         \__bnvs_if_resolve_loop_or_end_return:
2798     } {
2799         \__bnvs_end_return_error:n {
2800             Undefined-dotted-path

```

```

2801     }
2802   }
2803 }
2804 }

2805 \BNVS_new_conditional:cpnn { path_suffix:n } #1 { T, F, TF } {
2806   \__bnvs_seq_get_right:ccTF { path } { a } {
2807     \__bnvs_tl_if_eq:cnTF { a } { #1 } {
2808       \__bnvs_seq_pop_right:ccT { path } { a } { }
2809       \prg_return_true:
2810     } {
2811       \prg_return_false:
2812     }
2813   } {
2814     \prg_return_false:
2815   }
2816 }
2817 \BNVS_new:cpn { if_resolve_loop_or_end_return[.length]: } {

```

- Case ...length.

```

2818   \__bnvs_path_resolve_n:T {
2819     \__bnvs_length_append:vcTF { key } { ans } {
2820   % \begin{BNVS/gobble}
2821   <!*final>
2822   \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../length }
2823   </!final>
2824   % \end{BNVS/gobble}
2825     \__bnvs_if_resolve_loop_or_end_return:
2826   } {
2827     \__bnvs_if_resolve_end_return_false:n { NO~length }
2828   }
2829   }
2830 }
2831 \BNVS_new:cpn { if_resolve_loop_or_end_return[.last]: } {

```

- Case ...last.

```

2832   \__bnvs_path_resolve_n:T {
2833     \__bnvs_last_append:vcTF { key } { ans } {
2834   % \begin{BNVS/gobble}
2835   <!*final>
2836   \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../last }
2837   </!final>
2838   % \end{BNVS/gobble}
2839     \__bnvs_if_resolve_loop_or_end_return:
2840   } {
2841     \BNVS_end_return_false:x { NO~last }
2842   }
2843   }
2844 }
2845 \BNVS_new:cpn { if_resolve_loop_or_end_return[.range]: } {

```

- Case ...range.

```

2846 \__bnvs_path_resolve_n:T {
2847   \__bnvs_range_append:vcTF { key } { ans } {
2848     \BNVS_set:cpn { if_resolve_round_ans: } { \prg_do_nothing: }
2849 % \begin{BNVS/gobble}
2850 <!*final>
2851 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../range }
2852 </!final>
2853 % \end{BNVS/gobble}
2854   \__bnvs_if_resolve_loop_or_end_return:
2855   } {
2856     \__bnvs_if_resolve_end_return_false:n { NO~range }
2857   }
2858 }
2859 }
2860 \BNVS_new:cpn { if_resolve_loop_or_end_return[.previous]: } {

```

- Case ...previous.

```

2861 \__bnvs_path_resolve_n:T {
2862   \__bnvs_previous_append:vcTF { key } { ans } {
2863 % \begin{BNVS/gobble}
2864 <!*final>
2865 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../previous }
2866 </!final>
2867 % \end{BNVS/gobble}
2868   \__bnvs_if_resolve_loop_or_end_return:
2869   } {
2870     \__bnvs_if_resolve_end_return_false:n { NO~previous }
2871   }
2872 }
2873 }
2874 \BNVS_new:cpn { if_resolve_loop_or_end_return[.next]: } {

```

- Case ...next.

```

2875 \__bnvs_path_resolve_n:T {
2876   \__bnvs_next_append:vcTF { key } { ans } {
2877 % \begin{BNVS/gobble}
2878 <!*final>
2879 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../next }
2880 </!final>
2881 % \end{BNVS/gobble}
2882   \__bnvs_if_resolve_loop_or_end_return:
2883   } {
2884     \__bnvs_if_resolve_end_return_false:n { NO~next }
2885   }
2886 }
2887 }
2888 \BNVS_new:cpn { if_resolve_loop_or_end_return[.reset]: } {

```

- Case ...reset.

```

2889 \__bnvs_path_resolve_n:T {
2890   \__bnvs_v_greset:vnT { key } { } { }
2891   \__bnvs_value_append:vcTF { key } { ans } {

```

```

2892 % \begin{BNVS/gobble}
2893 <*\final>
2894 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../reset }
2895 </\final>
2896 % \end{BNVS/gobble}
2897     \__bnvs_if_resolve_loop_or_end_return:
2898     } {
2899     \__bnvs_if_resolve_end_return_false:n { NO~reset }
2900     }
2901 }
2902 }
2903 \BNVS_new:cpn { if_resolve_loop_or_end_return[.reset_all]: } {

    • Case ...reset_all.

2904     \__bnvs_path_resolve_n:T {
2905         \__bnvs_greset_all:vnT { key } { } { }
2906         \__bnvs_value_append:vcTF { key } { ans } {
2907 % \begin{BNVS/gobble}
2908 <*\final>
2909 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../reset }
2910 </\final>
2911 % \end{BNVS/gobble}
2912     \__bnvs_if_resolve_loop_or_end_return:
2913     } {
2914     \__bnvs_if_resolve_end_return_false:n { NO~reset }
2915     }
2916 }
2917 }
2918 \BNVS_set:cpn { if_resolve_loop_or_end_return[...<integer>]: } {

    • Case ...<integer>.

2919     \__bnvs_path_resolve_n:T {
2920         \__bnvs_index_append:vcTF { key } { a } { ans } {
2921 % \begin{BNVS/gobble}
2922 <*\final>
2923 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../<integer> }
2924 </\final>
2925 % \end{BNVS/gobble}
2926     \__bnvs_if_resolve_loop_or_end_return:
2927     } {
2928     \__bnvs_if_resolve_end_return_false:n { NO~integer }
2929     }
2930 }
2931 }
2932 \BNVS_set:cpn { if_resolve_loop_or_end_return[...]: } {

    • Case ....

2933     \__bnvs_path_resolve_n:T {
2934         \__bnvs_value_append:vcTF { key } { ans } {
2935 % \begin{BNVS/gobble}
2936 <*\final>
2937 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../... }
2938 </\final>

```

```

2939 % \end{BNVS/gobble}
2940     \__bnvs_if_resolve_loop_or_end_return:
2941     } {
2942         \__bnvs_if_resolve_end_return_false:n { NO~value }
2943     }
2944 }
2945 }
2946 \BNVS_set:cpn { if_resolve_loop_or_end_return[...++]: } {

    • Case ...++.

2947     \__bnvs_path_suffix:nTF { reset } {
2948         \__bnvs_path_resolve_n:T {
2949             \__bnvs_v_greset:vnT { key } { } { }
2950             \__bnvs_v_post_append:vncTF { key } { 1 } { ans } {
2951                 \__bnvs_if_resolve_loop_or_end_return:
2952             } {
2953                 \__bnvs_if_resolve_end_return_false:n { NO~post }
2954             }
2955         }
2956     } {
2957         \__bnvs_path_suffix:nTF { reset_all } {
2958             \__bnvs_path_resolve_n:T {
2959                 \__bnvs_greset_all:vnT { key } { } { }
2960                 \__bnvs_v_post_append:vncTF { key } { 1 } { ans } {
2961                     \__bnvs_if_resolve_loop_or_end_return:
2962                 } {
2963                     \__bnvs_if_resolve_end_return_false:n { NO~post }
2964                 }
2965             }
2966         } {
2967             \__bnvs_path_resolve_n:T {
2968                 \__bnvs_v_post_append:vncTF { key } { 1 } { ans } {
2969                     \__bnvs_if_resolve_loop_or_end_return:
2970                 } {
2971                     \__bnvs_if_resolve_end_return_false:n { NO~post }
2972                 }
2973             }
2974         }
2975     }
2976 }
2977 \BNVS_set:cpn { if_resolve_loop_or_end_return[...n+=...]: } {

    • Case ....n+=integer.

2978     \__bnvs_path_resolve_n:T {
2979         \__bnvs_base_resolve_n:
2980         \__bnvs_n_incr_append:vvvcTF { key } { key_base } { incr } { ans } {
2981 % \begin{BNVS/gobble}
2982 <{*!final}
2983 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../...n+=... }
2984 </!final}
2985 % \end{BNVS/gobble}
2986     \__bnvs_if_resolve_loop_or_end_return:
2987     } {

```

```

2988     \__bnvs_if_resolve_end_return_false:n {
2989         NO~n~incrementation
2990     }
2991 }
2992 }
2993 }
2994 \BNVS_set:cpn { if_resolve_loop_or_end_return[...+=...]: } {
    • Case A+=<integer>.

2995     \__bnvs_path_resolve_n:T {
2996         \__bnvs_v_incr_append:vcTF { key } { incr } { ans } {
2997 % \begin{BNVS/gobble}
2998 <!*final>
2999 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../...+=... }
3000 </!final>
3001 % \end{BNVS/gobble}
3002         \__bnvs_if_resolve_loop_or_end_return:
3003     } {
3004         \__bnvs_if_resolve_end_return_false:n {
3005             NO~incremented~value
3006         }
3007     }
3008 }
3009 }
3010 \BNVS_new:cpn { base_resolve_n: } {
3011     \__bnvs_seq_if_empty:cF { path_base } {
3012         \__bnvs_seq_pop_right:cc { path_base } { a }
3013         \__bnvs_seq_if_empty:cF { path_base } {
3014             \__bnvs_tl_put_right:cx { key_base } {
3015                 . \__bnvs_seq_use:cn { path_base } { . }
3016             }
3017         }
3018     }
3019 }
3020 \BNVS_new:cpn { base_resolve: } {
3021     \__bnvs_seq_if_empty:cF { path_base } {
3022         \__bnvs_tl_put_right:cx { key_base } {
3023             . \__bnvs_seq_use:cn { path_base } { . }
3024         }
3025     }
3026 }
3027 \BNVS_new:cpn { if_resolve_loop_or_end_return[...++n]: } {
    • Case ...++n.

3028     \__bnvs_path_resolve_n:T {
3029         \__bnvs_base_resolve:
3030         \__bnvs_n_incr_append:vnctF { key } { key_base } { 1 } { ans } {
3031             \__bnvs_if_resolve_loop_or_end_return:
3032         } {
3033             \__bnvs_if_resolve_end_return_false:n { NO~...++n }
3034         }
3035     }
3036 }
3037 \BNVS_set:cpn { if_resolve_loop_or_end_return[++...n]: } {

```


- Case ++...n.

```

3038 \__bnvs_path_resolve_n:T {
3039   \__bnvs_base_resolve_n:
3040   \__bnvs_n_incr_append:vvncTF { key } { key_base } { 1 } { ans } {
3041     \__bnvs_if_resolve_loop_or_end_return:
3042   } {
3043     \__bnvs_if_resolve_end_return_false:n { NO~++...n }
3044   }
3045 }
3046 }
3047 \BNVS_new:cpn { if_resolve_loop_or_end_return[++...]: } {

```

- Case ++....

```

3048 \__bnvs_path_suffix:nTF { reset } {
3049   \__bnvs_path_resolve_n:T {
3050     \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
3051 % \begin{BNVS/gobble}
3052 <!*final>
3053 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../++...reset }
3054 </!final>
3055 % \end{BNVS/gobble}
3056 % \begin{macrocode}
3057   \__bnvs_v_greset:vnT { key } { } { }
3058   \__bnvs_if_resolve_loop_or_end_return:
3059 } {
3060   \__bnvs_v_greset:vnT { key } { } { }
3061   \__bnvs_if_resolve_end_return_false:n { No~increment }
3062 }
3063 }
3064 } {
3065   \__bnvs_path_suffix:nTF { reset_all } {
3066     \__bnvs_path_resolve_n:T {
3067       \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
3068 % \begin{BNVS/gobble}
3069 <!*final>
3070 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../++...reset_all }
3071 </!final>
3072 % \end{BNVS/gobble}
3073 % \begin{macrocode}
3074     \__bnvs_greset_all:vnT { key } { } { }
3075     \__bnvs_if_resolve_loop_or_end_return:
3076 } {
3077     \__bnvs_greset_all:vnT { key } { } { }
3078     \__bnvs_if_resolve_end_return_false:n { No~increment }
3079 }
3080 }
3081 } {
3082   \__bnvs_path_resolve_n:T {
3083     \__bnvs_v_incr_append:vncTF { key } { 1 } { ans } {
3084 % \begin{BNVS/gobble}
3085 <!*final>
3086 \BNVS_DEBUG_log_if_resolve_ncTF:nn { ... } { .../++... }
3087 </!final>

```

```

3088 % \end{BNVS/gobble}
3089 % \begin{macrocode}
3090     \_bnvs_if_resolve_loop_or_end_return:
3091     } {
3092     \_bnvs_if_resolve_end_return_false:n { No~increment }
3093     }
3094   }
3095 }
3096 }
3097 }

```

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

Evaluates the single `⟨overlay query⟩`, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of `\l__bnvs_ans_tl`. Ranges are supported with the colon syntax. This is executed within a local `TeX` group managed by the caller. Below are local variables and constants.

`\l__bnvs_V_tl` Storage for a single value out of a range.
(End of definition for `\l__bnvs_V_tl`.)

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

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

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

`\c__bnvs_A_cln_Z_regex` Used to parse 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 of definition for `\c__bnvs_A_cln_Z_regex`.)

```

3098 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
3099   \A \s* (?:
      • 2: V
3100       ( [^:]+? )
      • 3, 4, 5: A : Z? or A :: L?
3101       | ( [^:]+? ) \s* : (?: ( \s* [^:]*? ) | : ( \s* [^:]*? ) )
      • 6, 7: ::(L:Z)?
3102       | :: \s* (?: ( [^:]+? ) \s* : \s* ( [^:]+? ) )?
      • 8, 9: :(Z::L)?
3103       | : \s* (?: ( [^:]+? ) \s* :: \s* ( [^:]*? ) )?
3104     )
3105     \s* \Z
3106   }

3107 \BNVS_set:cpn { query_eval_end_return_true: } {
3108   \group_end:
3109   \prg_return_true:
3110 }
3111 \BNVS_new:cpn { query_eval_end_return_false: } {
3112   \BNVS_end:
3113   \prg_return_false:

```

```

3114 }
3115 \BNVS_new:cpn { query_eval_end_return_false:n } #1 {
3116   \BNVS_end:
3117   \prg_return_false:
3118 }
3119 \BNVS_new:cpn { query_eval_error_end_return_false:n } #1 {
3120   \BNVS_error:x { #1 }
3121   \__bnvs_query_eval_end_return_false:
3122 }
3123 \BNVS_new:cpn { query_eval_unreachable: } {
3124   \__bnvs_query_eval_error_end_return_false:n { UNREACHABLE }
3125 }
3126 \BNVS_new:cpn { if_blank:cTF } #1 {
3127   \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3128 }
3129 \BNVS_new_conditional:cpnn { match_pop_left:c } #1 { T, F, TF } {
3130   \BNVS_tl_use:nc {
3131     \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
3132   } { #1 } {
3133     \prg_return_true:
3134   } {
3135     \prg_return_false:
3136   }
3137 }

```

__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.

```

3138 \BNVS_new_conditional:cpnn { query_eval_match_branch: } { T, F, TF } {
3139   \__bnvs_match_pop_left:cT V {
3140     \__bnvs_match_pop_left:cT V {
3141       \__bnvs_if_blank:cTF V {
3142         \__bnvs_match_pop_left:cT A {
3143           \__bnvs_match_pop_left:cT Z {
3144             \__bnvs_match_pop_left:cT L {
3145               \__bnvs_if_blank:cTF A {
3146                 \__bnvs_match_pop_left:cT L {
3147                   \__bnvs_match_pop_left:cT Z {
3148                     \__bnvs_if_blank:cTF Z {
3149                       \__bnvs_if_blank:cTF L {
3150                         \__bnvs_match_pop_left:cT Z {
3151                           \__bnvs_match_pop_left:cT L {
3152                             \__bnvs_if_blank:cTF L {
3153                               \__bnvs_if_blank:cTF Z {
3154                                 \BNVS_use:c { query_eval_return[:]: }
3155                               } {
3156                                 \BNVS_use:c { query_eval_return[:Z]: }
3157                               }
3158                             } {

```

```

3159         \_bnvs_if_blank:cTF Z {
3160 \_bnvs_query_eval_error_end_return_false:n { Missing~first~or~last }
3161         } {
3162         \BNVS_use:c { query_eval_return[:Z::L]: }
3163         }
3164     }
3165 }
3166 }
3167 } {
3168 \_bnvs_query_eval_error_end_return_false:n { Missing~first~or~last }
3169     }
3170 } {
3171     \_bnvs_if_blank:cTF L {
3172     \_bnvs_query_eval_unreachable:
3173     } {
3174     \BNVS_use:c { query_eval_return[:Z::L]: }
3175     }
3176 }
3177 }
3178 }
3179 } {
3180     \_bnvs_if_blank:cTF Z {
3181     \_bnvs_if_blank:cTF L {
3182     \BNVS_use:c { query_eval_return[A:]: }
3183     } {
3184     \BNVS_use:c { query_eval_return[A::L]: }
3185     }
3186 } {
3187     \_bnvs_if_blank:cTF L {
3188     \BNVS_use:c { query_eval_return[A:Z]: }
3189     } {
3190     \_bnvs_query_eval_error_end_return_false:n {
3191     Only~two~of~first,~last~or~length
3192     }
3193     }
3194 }
3195 }
3196 }
3197 }
3198 }
3199 } {
3200     \BNVS_use:c { query_eval_return[V]: }
3201     }
3202 }
3203 }
3204 }
3205 \BNVS_new:cpn { query_eval_return[V]: } {

```

Single value

```

3206 \_bnvs_if_resolve:vcTF { V } { ans } {
3207     \prg_return_true:
3208     } {
3209     \prg_return_false:
3210     }

```

```

3211 }
3212 \BNVS_new:cpn { query_eval_return[A:Z]: } {
3213   <first>:<last> range
3214   \__bnvs_if_resolve:vcTF { A } { ans } {
3215     \__bnvs_tl_put_right:cn { ans } { - }
3216     \__bnvs_if_append:vcTF { Z } { ans } {
3217       \prg_return_true:
3218     } {
3219       \prg_return_false:
3220     } {
3221       \prg_return_false:
3222     }
3223   }
3224   \BNVS_new:cpn { query_eval_return[A::L]: } {
3225     <first>::<length> range
3226     \__bnvs_if_resolve:vcTF { A } { A } {
3227       \__bnvs_if_resolve:vcTF { L } { ans } {
3228         \__bnvs_tl_put_right:cn { ans } { + }
3229         \__bnvs_tl_put_right:cv { ans } { A }
3230         \__bnvs_tl_put_right:cn { ans } { -1 }
3231         \__bnvs_round_ans:
3232         \__bnvs_tl_put_left:cn { ans } { - }
3233         \__bnvs_tl_put_left:cv { ans } { A }
3234         \prg_return_true:
3235       } {
3236         \prg_return_false:
3237       } {
3238         \prg_return_false:
3239       }
3240     }
3241     \BNVS_new:cpn { query_eval_return[A:]: } {
3242       <first>: and <first>:: range
3243       \__bnvs_if_resolve:vcTF { A } { ans } {
3244         \__bnvs_tl_put_right:cn { ans } { - }
3245         \prg_return_true:
3246       } {
3247         \prg_return_false:
3248       }
3249     }
3250     \BNVS_new:cpn { query_eval_return[:Z::L]: } {
3251       :Z::L or ::L:Z range
3252       \__bnvs_if_resolve:vcTF { Z } { Z } {
3253         \__bnvs_if_resolve:vcTF { L } { ans } {
3254           \__bnvs_tl_put_left:cn { ans } { 1- }
3255           \__bnvs_tl_put_right:cn { ans } { + }
3256           \__bnvs_tl_put_right:cv { ans } { Z }
3257           \__bnvs_round_ans:
3258           \__bnvs_tl_put_right:cn { ans } { - }

```

```

3257     \_bnvs_tl_put_right:cv { ans } { Z }
3258     \prg_return_true:
3259   } {
3260     \prg_return_false:
3261   }
3262 } {
3263   \prg_return_false:
3264 }
3265 }
3266 \BNVS_new:cpn { query_eval_return[:]: } {
  : or :: range
3267   \_bnvs_tl_set:cn { ans } { - }
3268   \prg_return_true:
3269 }
3270 \BNVS_new:cpn { query_eval_return[Z]: } {
  ::(last) range
3271   \_bnvs_tl_set:cn { ans } { - }
3272   \_bnvs_if_append:vcTF { Z } { ans } {
3273     \prg_return_true:
3274   } {
3275     \prg_return_false:
3276   }
3277 }
3278 \BNVS_new_conditional:cpnn { query_eval:nc } #1 #2 { T, F, TF } {
3279   \_bnvs_call_greset:
3280   \_bnvs_match_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
3281     \BNVS_begin:
3282     \_bnvs_query_eval_match_branch:TF {
3283       \BNVS_end_tl_set:cv { #2 } { ans }
3284       \prg_return_true:
3285     } {
3286       \BNVS_end:
3287       \prg_return_false:
3288     }
3289   } {
Error
3290   \BNVS_error:n { Syntax~error:~#1 }
3291   \prg_return_false:
3292 }
3293 }

```

```

\__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 of definition for `\l__bnvs_query_seq`.)

```

\l__bnvs_ans_seq Storage of the evaluated result.

```

(End of definition for `\l__bnvs_ans_seq`.)

```

\c__bnvs_comma_regex Used to parse slide range overlay specifications.

```

```

3294 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End of definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

3295 \BNVS_new:cpn { eval:nc } #1 #2 {

```

```

3296   \BNVS_begin:

```

Local variables declaration

```

3297   \__bnvs_seq_clear:c { query }

```

```

3298   \__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`.

```

3299   \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.

```

3300   \__bnvs_seq_map_inline:cn { query } {
3301     \__bnvs_tl_clear:c { ans }
3302     \__bnvs_query_eval:ncTF { ##1 } { ans } {
3303       \__bnvs_seq_put_right:cv { ans } { ans }
3304     } {
3305       \seq_map_break:n {
3306         \BNVS_error:n { Circular/Undefined-dependency-in-#1}
3307       }
3308     }
3309   }

```

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

```

3310   \exp_args:NNnx
3311   \BNVS_end:
3312   \__bnvs_tl_put_right:cn { #2 } { \__bnvs_seq_use:cn { ans } , }
3313 }

```

`\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.

```

3314 \NewDocumentCommand \BeanovesEval { O{} m } {
3315   \BNVS_begin:
3316   \keys_define:nn { BeanovesEval } {
3317     in:N .tl_set:N = \l__bnvs_eval_in_tl,
3318     in:N .initial:n = { },
3319     see .bool_set:N = \l__bnvs_eval_see_bool,
3320     see .default:n = true,
3321     see .initial:n = false,
3322   }
3323   \keys_set:nn { BeanovesEval } { #1 }
3324   \__bnvs_tl_clear:c { ans }
3325   \__bnvs_eval:nc { #2 } { ans }
3326   \__bnvs_tl_if_empty:cTF { eval_in } {
3327     \bool_if:nTF { \l__bnvs_eval_see_bool } {
3328       \BNVS_tl_use:Nv \BNVS_end: { ans }
3329     } {
3330       \BNVS_end:
3331     }
3332   } {
3333     \bool_if:nTF { \l__bnvs_eval_see_bool } {
3334       \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3335         \BNVS_end:
3336         \tl_set:Nn ##1 { ##2 }
3337         ##2
3338       }
3339       \BNVS_tl_use:nv {
3340         \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_eval_in_tl
3341       } { ans }
3342     } {
3343       \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3344         \BNVS_end:
3345         \tl_set:Nn ##1 { ##2 }
3346       }
3347       \BNVS_tl_use:nv {
3348         \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_eval_in_tl
3349       } { ans }
3350     }
3351   }
3352 }
```

6.13.13 Reseting counters

`\BeanovesReset` `\BeanovesReset` [*<first value>*] {*<key>*}

`\BeanovesReset*` `\BeanovesReset*` [*<first value>*] {*<key>*}

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

```

3353 \NewDocumentCommand \BeanovesReset { s O{} m } {
3354   \__bnvs_name_id_n_get:nTF { #3 } {
3355     \BNVS_tl_use:nv {
3356       \IfBooleanTF { #1 } {
3357         \__bnvs_greset_all:nnF
3358       } {
3359         \__bnvs_v_greset:nnF
3360       }
3361     } { key } { #2 } {
3362       % \__bnvs_warning:n { Unknown~name:~#3 }
3363     }
3364   } {
3365     \__bnvs_warning:n { Bad~name:~#3 }
3366   }
3367   \ignorespaces
3368 }

3369 \ExplSyntaxOff

<*internal> </internal>

```