

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](#). This is a solution to this [latex.org forum query](#).

Contents

1	Installation	1
1.1	Package manager	1
1.2	Manual installation	1
1.3	Usage	1
2	Minimal example	2
3	Named overlay sets	2
3.1	Presentation	2
3.2	Named overlay reference	3
3.3	Defining named overlay sets	3
3.3.1	Basic specifiers	3
3.3.2	List specifiers	4
4	Resolution of <code>?(...)</code> query expressions	4
4.1	Number and range overlay queries	5
4.2	Counters	6
4.3	Dotted paths	7
4.4	The beamer counters	8
4.5	Multiple queries	8
4.6	Frame id	8
4.7	Resolution command	9
5	Support	9

6	Implementation	9
6.1	Package declarations	9
6.2	Facility layer: definitions and naming	9
6.3	logging	12
6.4	Facility layer: Variables	13
6.4.1	Regex	19
6.4.2	Token lists	21
6.4.3	Strings	25
6.4.4	Sequences	26
6.4.5	Integers	27
6.5	Debug facilities	28
6.6	Debug messages	28
6.7	Testing facilities	28
6.8	Local variables	28
6.9	Infinite loop management	29
6.10	Overlay specification	30
6.10.1	Registration	30
6.10.2	Basic functions	34
6.10.3	Functions with cache	39
6.11	Implicit value counter	40
6.12	Implicit index counter	43
6.13	Regular expressions	43
6.14	beamer.cls interface	47
6.15	Defining named slide ranges	47
6.16	Scanning named overlay specifications	61
6.17	Resolution	66
6.18	Evaluation bricks	68
6.19	Index counter	83
6.20	Value counter	84
6.21	Functions for the resolution	88
6.22	Resetting counters and values	109

1 Installation

1.1 Package manager

When not already available, `beanoves` package may be installed using a TEX distribution's package manager, either from the graphical user interface, or with the relevant command (`tlmgr` for $\text{T}_{\text{E}}\text{X}$ Live and `mpm` for $\text{M}_{\text{I}}\text{K}_{\text{T}}\text{E}_{\text{X}}$). This should install files `beanoves.sty` and its debug version `beanoves-debug.sty` as well as `beanoves-doc.pdf` documentation.

1.2 Manual installation

The `beanoves` source files are available from the [source repository](#). They can also be fetched from the [CTAN repository](#).

1.3 Usage

The `beanoves` package is imported by putting `\RequirePackage{beanoves}` in the preamble of a \LaTeX document that uses the `beamer` class. Should the package cause problems, its features can be temporarily deactivated with simple commands `\BeanovesOff` and `\BeanovesOn`.

2 Minimal example

The \LaTeX document below is a contrived example to show how the `beamer` overlay specifications have been extended. More demonstration files are available from the [beanoves source repository](#).

```
1 \documentclass{beamer}
2 \RequirePackage{beanoves}
3 \begin{document}
4 \Beanoves {
5   A = 1:4,
6   B = A.last::3,
7   C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \insertframenum}
11 {\Large Slide \insertslidenumber}
12 - \visible<?(A.1)> {Only on slide 1}\\
13 - \visible<?(B.range)> {Only on slides 4 to 6}\\
14 - \visible<?(C.1)> {Only on slide 7}\\
15 - \visible<?(A.2)> {Only on slide 2}\\
16 - \visible<?(B.2:B.last)> {Only on slides 5 to 6}\\
17 - \visible<?(C.2)> {Only on slide 8}\\
18 - \visible<?(A.next)-> {From slide 5}\\
19 - \visible<?(B.3:B.last)> {Only on slide 6}\\
20 - \visible<?(C.3)> {Only on slide 9}\\
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 4. 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 5. On line 6, we declare a second overlay set named ‘B’, starting after the 3 slides of ‘A’ namely 4. Its length is 3 meaning that its last slide number is 6, thus each `?(B.last)` is replaced by 6. The next slide number after slide range ‘B’ is 7 which is also the start of the third slide range due to line 7.

3 Named overlay sets

3.1 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.

3.2 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 referencing a well defined static integer or range to be used in `beamer` overlay specifications. They can take one of the next forms.

$\langle \text{short name} \rangle$: 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 *dotted names* like `A.1` and `B.C.2`,

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

The *short names* and *frame ids* are alphanumerical case sensitive identifiers, with possible underscores but with no space nor leading digit. Unicode symbols above `U+00A0` are allowed if the underlying `TEX` engine supports it. Only the *frame id* is allowed to be empty, in which case it may apply to any frame.

The *dotted path* is a string $\langle c_1 \rangle . \langle c_2 \rangle \dots \langle c_j \rangle$. Each component $\langle c_i \rangle$ denotes a $\langle \text{short name} \rangle$ or a decimal integer. The *dotted path* can be empty for which j is 0.

Identifiers consisting only of lowercase letters may have special meaning as detailed below. This includes components $\langle c \rangle$ s, unless explicitly documented like for “`n`”.

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

3.3 Defining named overlay sets

In order to define *named overlay sets*, we can either execute the next `\Beanoves` command before a `beamer` frame environment, or use the new `beanoves` option of this environment.

<code>\Beanoves</code>	<code>\Beanoves{\langle ref_1 \rangle [= \langle spec_1 \rangle], \langle ref_2 \rangle [= \langle spec_2 \rangle], \dots, [\langle ref_j \rangle = \langle spec_j \rangle]}</code>
<code>\Beanoves*</code>	

<code>beanoves</code>	<code>beanoves = {\langle ref_1 \rangle [= \langle spec_1 \rangle], \langle ref_2 \rangle [= \langle spec_2 \rangle], \dots, \langle ref_j \rangle [= \langle spec_j \rangle]}</code>
-----------------------	---

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

Notice that $\langle ref_i \rangle = 1$ can be shortened to $\langle ref_i \rangle$. The `\Beanoves` arguments take precedence over both the `\Beanoves*` arguments and the `beanoves` options. This allows to provide an overlay name only when not already defined, which is helpful when the very same frame source is included multiple times in different contexts.

3.3.1 Basic specifiers

In the possible values for $\langle spec \rangle$ hereafter, $\langle value \rangle$, $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$ are numerical expression (with algebraic operators $+$, $-$, ...) possibly involving any *named overlay reference* defined above.

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

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

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

$: \langle last \rangle$, a shortcut for $1 : \langle last \rangle$.

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside \LaTeX environments, each new call cumulates with the previous one. Notice that the argument of this function can contain macros: they will be exhaustively expanded at resolution time¹.

3.3.2 List specifiers

Also possible values are *list specifiers* which are comma separated lists of $\langle path \rangle = \langle spec \rangle$ definitions. They come in three different flavours.

The definition

$\langle a \rangle = \{ \langle path_1 \rangle = \langle spec_1 \rangle, \langle path_2 \rangle = \langle spec_2 \rangle, \dots, \langle path_j \rangle = \langle spec_j \rangle \}$

removes previous $\langle a \rangle$ index definitions, and executes

$\langle a \rangle . \langle path_1 \rangle = \langle spec_1 \rangle$,

$\langle a \rangle . \langle path_2 \rangle = \langle spec_2 \rangle$,

\dots ,

$\langle a \rangle . \langle path_j \rangle = \langle spec_j \rangle$.

The rules above can apply individually to each line. The $\langle a \rangle$ counter defined below is left unmodified. If $= \langle spec_j \rangle$ is omitted, it defaults to $= 1$.

The definition

$\langle a \rangle = [\langle path_1 \rangle = \langle spec_1 \rangle, \langle path_2 \rangle = \langle spec_2 \rangle, \dots, \langle path_j \rangle = \langle spec_j \rangle]$

is similar to the previous list, except that what can be omitted is $\langle path_j \rangle =$, which defaults to $\langle i \rangle =$ where $\langle i \rangle$ is the smallest positive integer such that $\langle a \rangle . \langle i \rangle$ is not already defined.

Finally, the definition

$\langle a \rangle = \{ \{ \langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_j \rangle \} \}$

corresponds to beamer range specifier.

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

4 Resolution of $?(\dots)$ query expressions

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

The $\langle \textit{queries} \rangle$ argument is a comma separated list of individual $\langle \textit{query} \rangle$ ’s processed from left to right as explained below. Notice that nesting a $?(\dots)$ query expressions inside another query expression is not supported.

The named overlay sets defined above are queried for integer numerical values that will be passed to `beamer`. Turning an *overlay query* into the static expression it represents, as when above $?(\text{A.1})$ was replaced by 1, is denoted by *overlay query resolution* or simply *resolution*. The process starts by replacing any *query reference* by its value as explained below until obtaining numerical expressions that are evaluated and finally rounded to the nearest integer to feed `beamer` with either a range or a number. When the *query reference* is a previously declared $\langle \textit{a} \rangle$, like X after $\text{X}=1$, it is simply replaced by the corresponding declared $\langle \textit{value} \rangle$, here 1. Otherwise, we use *implicit overlay queries* and their *resolution rules* depending on the definition of the named overlay set. Hereafter $\langle \textit{i} \rangle$ denotes a signed integer whereas $\langle \textit{value} \rangle$, $\langle \textit{first} \rangle$, $\langle \textit{last} \rangle$ and $\langle \textit{length} \rangle$ stand for raw integers or more general numerical expressions. We assume that $\langle \textit{first} \rangle \leq \langle \textit{last} \rangle$ and $\langle \textit{length} \rangle \geq 0$.

Resolution occurs only when required and the result is cached for performance reason.

4.1 Number and range overlay queries

$\langle \textit{a} \rangle = \langle \textit{value} \rangle$ For an unlimited range

overlay query	resolution
$\langle \textit{a} \rangle.1$	$\langle \textit{value} \rangle$
$\langle \textit{a} \rangle.2$	$\langle \textit{value} \rangle + 1$
$\langle \textit{a} \rangle.\langle \textit{i} \rangle$	$\langle \textit{value} \rangle + \langle \textit{i} \rangle - 1$

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

overlay query	resolution
$\langle \textit{a} \rangle.1$	$\langle \textit{first} \rangle$
$\langle \textit{a} \rangle.2$	$\langle \textit{first} \rangle + 1$
$\langle \textit{a} \rangle.\langle \textit{i} \rangle$	$\langle \textit{first} \rangle + \langle \textit{i} \rangle - 1$
$\langle \textit{a} \rangle.\textit{previous}$	$\langle \textit{first} \rangle - 1$
$\langle \textit{a} \rangle.\textit{first}$	$\langle \textit{first} \rangle$

Notice that $\langle \textit{a} \rangle.\textit{previous}$ and $\langle \textit{a} \rangle.0$ are most of the time synonyms.

$\langle \textit{a} \rangle = : \langle \textit{last} \rangle$ For a range limited from above:

overlay query	resolution
$\langle \textit{a} \rangle.1$	$\langle \textit{last} \rangle$
$\langle \textit{a} \rangle.0$	$\langle \textit{last} \rangle - 1$
$\langle \textit{a} \rangle.\langle \textit{i} \rangle$	$\langle \textit{last} \rangle + \langle \textit{i} \rangle - 1$
$\langle \textit{a} \rangle.\textit{last}$	$\langle \textit{last} \rangle$
$\langle \textit{a} \rangle.\textit{next}$	$\langle \textit{last} \rangle + 1$

$\langle a \rangle = \langle first \rangle : \langle last \rangle$ 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:

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

Notice that the resolution of $\langle a \rangle .range$ 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 `\BeanovesResolve` 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 \insertframenumber} {Slide \insertslidenumber}
5 \ttfamily
6 \BeanovesResolve[show] (A.1)      == 3,
7 \BeanovesResolve[show] (A.-1)    == 1,
8 \BeanovesResolve[show] (A.previous) == 2,
9 \BeanovesResolve[show] (A.first)  == 3,
10 \BeanovesResolve[show] (A.last)   == 8,
11 \BeanovesResolve[show] (A.next)   == 9,
12 \BeanovesResolve[show] (A.length) == 6,
13 \BeanovesResolve[show] (A.range)  == 3-8,
14 \end{frame}

```

For example both $?(A.next)$, $?(A.last+1)$, $?(A.1+A.length)$ give the same result as soon as the slide range named 'A' has been properly defined with a starting value and a length, and not overridden by some rule below.

4.2 Counters

Each named overlay set defined has a dedicated value counter which is some kind of integer variable that can be used and incremented. A standalone $\langle name \rangle$ *overlay query* is resolved into the position of this value counter. For each frame, this variable is initialized to the first available resolution amongst $\langle value \rangle$, $\langle name \rangle .first$, $\langle name \rangle .1$ or $\langle name \rangle .last$. If none is available, the counter is initialized to 1.

Additionally, resolution rules are provided for dedicated *overlay queries*:

$\langle name \rangle = \langle integer\ expression \rangle$, resolve $\langle integer\ expression \rangle$ into $\langle integer \rangle$, set the value counter to $\langle integer \rangle$ and use the new position. Here $\langle integer\ expression \rangle$ is the longest character sequence with no space².

$\langle name \rangle += \langle integer\ expression \rangle$, resolve $\langle integer\ expression \rangle$ into $\langle integer \rangle$, advance the value counter by $\langle integer \rangle$ and use the new position.

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

$\langle name \rangle ++$, use the actual position and advance the value counter for $\langle name \rangle$ 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 $\langle name \rangle.n$ *named index reference* is resolved into $\langle name \rangle.\langle n \rangle$, which in turn is resolved according to the preceding rules.

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

$\langle name \rangle.n = \langle integer\ expression \rangle$, resolve $\langle integer\ expression \rangle$ into $\langle integer \rangle$, set the implicit index counter associate to $\langle name \rangle$ to $\langle integer \rangle$ and use the resolution of $\langle name \rangle.n$.

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

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

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

$\langle name \rangle.n ++$, use the resolution of $\langle name \rangle.n$ and increment the implicit index counter associate to $\langle name \rangle$ 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 $\langle name \rangle.n$ counter, and whichever $\langle name \rangle$ first or last value is defined for the $\langle name \rangle$ counter. Sometimes, resetting the counter manually is necessary, for example when managing tikz overlay material.

\BeanovesReset [*options*] { $\langle ref_1 \rangle [= \langle spec_1 \rangle]$, $\langle ref_2 \rangle [= \langle spec_2 \rangle]$, ..., $\langle ref_j \rangle [= \langle spec_j \rangle]$ }

This command is very similar to **\Beanoves**, except that a standalone $\langle ref_i \rangle$ resets the counter to its default value and that it is meant to be used inside a **frame** environment. When the **all** option is provided, some internals that were cached for performance reasons are cleared.

²The parser for algebraic expression is very rudimentary.

4.3 Dotted paths

In previous overlay queries, $\langle name \rangle$ can be formally replaced by $\langle name \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_j \rangle$. If the whole does not correspond to a definition or an assignment, the longest included qualified dotted name $\langle name \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_k \rangle$ where $0 \leq k \leq j$ is first replaced by its definition $\langle name' \rangle.\langle c'_1 \rangle \dots \langle c'_l \rangle$ if any and then the modified overlay query is resolved with preceding rules as well as this one. For example, with `\Beanoves{A.B=D, D.C=E}`, `A.B.C` is resolved like `E`.

4.4 The beamer counters

While inside a `frame` environment, it is possible to save the current value of the `beamerpauses` counter that controls whether elements should appear on the current slide. For that, we can execute one of `\Beanoves{\langle a \rangle=pauses}` or in a query `?(\langle a \rangle=pauses) \dots`. Then later on, we can use `?(\dots \langle a \rangle \dots)` to refer to this saved value in the same frame³. Next frame source is an example of usage.

```

1 \begin{frame}
2 \visible<+>{A}\\
3 \visible<+>{B\Beanoves{afterB=pauses}}\\
4 \visible<+>{C}\\
5 \visible<?(afterB)>{other C}\\
6 \visible<?(afterB.previous)>{other B}\\
7 \end{frame}

```

“A” first appears on slide 1, “B” on slide 2 and “C” on slide 3. On line 2, `afterB` takes the value of the `beamerpauses` counter once updated, *id est* 3. “B” and “other B” as well as “C” and “other C” appear at the same time. If the `beamerpauses` counter is not suitable, we can execute instead one of `\Beanoves{\langle a \rangle=slideinframe}` or in a query `?(\dots \langle a \rangle=slideinframe) \dots`. It uses the numerical value of `\insertslideinframe`.

4.5 Multiple queries

It is possible to replace the comma separated list `?(\langle query_1 \rangle), \dots ?(\langle query_j \rangle)` with the shorter `?(\langle query_1 \rangle, \dots \langle query_j \rangle)`.

4.6 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 is empty by default. 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,`

³See [stackexchange](#) for an alternative that needs at least two passes.

We can use the same $\langle frame\ id \rangle$ for different frames to share named overlay sets. When a query contains an undefined *qualified dotted name* with an explicit $\langle frame\ id \rangle$, the resolution uses instead the *qualified dotted name* with an empty $\langle frame\ id \rangle$ instead, if possible. For example, if $X!A$ is not defined, $!A$ is used instead.

4.7 Resolution command

```
\BeanovesResolve [ $\langle setup \rangle$ ] [ $\langle overlay\ queries \rangle$ ]
```

This function resolves the $\langle overlay\ queries \rangle$, which are like the argument of $?(\dots)$ instructions: a comma separated list of single $\langle overlay\ query \rangle$'s. The optional $\langle setup \rangle$ is a key-value:

`show` the result is left into the input stream

`in:N= $\langle command \rangle$` the result is stored into $\langle command \rangle$.

5 Support

See the [source repository](#). One can report issues there.

6 Implementation

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

```
1 \MY=bnvs
```

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

6.1 Package declarations

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

6.2 Facility layer: definitions and naming

In order to make the code shorter and easier to read during development, we add a layer over L^AT_EX3. The `c` and `v` argument specifiers take a slightly different meaning when used in a function which name contains with `bnvs` or `BNVS`. Where L^AT_EX3 would transform `l__bnvs_ref_tl` into `\l__bnvs_ref_tl`, `bnvs` will directly transform `ref` into `\l__bnvs_ref_tl`. The type of the local variable used depends on the context and may be `seq` or `int` for example. There are however a pair of exceptions mentioned below. For a better reading experience, ‘`ref`’ will generally stand for `\l__bnvs_ref_tl`, whereas ‘`path sequence`’ will generally stand for `\l__bnvs_path_seq`. Other similar shortcuts are used as well.

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

<code>\BNVS:c</code>	<code>\BNVS:c {⟨cs core name⟩}</code>
<code>\BNVS_l:cn</code>	<code>\BNVS_l:cn {⟨local variable core name⟩} {⟨ type ⟩}</code>
<code>\BNVS_g:cn</code>	<code>\BNVS_g:cn {⟨global variable core name⟩} {⟨ type ⟩}</code>

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 }

```

<code>\BNVS_use_raw:c</code>	<code>\BNVS_use_raw:c {⟨cs name⟩}</code>
<code>\BNVS_use_raw:Nc</code>	<code>\BNVS_use_raw:Nc ⟨function⟩ {⟨cs name⟩}</code>
<code>\BNVS_use_raw:nc</code>	<code>\BNVS_use_raw:nc {⟨tokens⟩} {⟨cs name⟩}</code>
<code>\BNVS_use:c</code>	<code>\BNVS_use:c {⟨cs core⟩}</code>
<code>\BNVS_use:Nc</code>	<code>\BNVS_use:Nc ⟨function⟩ {⟨cs core⟩}</code>
<code>\BNVS_use:nc</code>	<code>\BNVS_use:nc {⟨tokens⟩} {⟨cs core⟩}</code>

`\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:NNo
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_tl_use:nvv #1 #2 {
40   \BNVS_tl_use:nv { \BNVS_tl_use:nv { #1 } { #2 } }
41 }
42 \cs_new:Npn \BNVS_tl_use:nvvv #1 #2 {
43   \BNVS_tl_use:nvv { \BNVS_tl_use:nv { #1 } { #2 } }
44 }
45 \cs_new:Npn \BNVS_log:n #1 { }
46 \cs_generate_variant:Nn \BNVS_log:n { x }

```

\BNVS_DEBUG_on:n	\BNVS_DEBUG_on:n {<type>}
\BNVS_DEBUG_off:n	\BNVS_DEBUG_off:n {<type>}
\BNVS_DEBUG_push:n	\BNVS_DEBUG_push:n {<types>}
\BNVS_DEBUG_pop:	\BNVS_DEBUG_pop:

These functions are only available in debug mode. Manage debug messaging for one given *<type>* or *<types>*. The implementation is not publicly exposed.

```

47 \cs_new:Npn \BNVS_DEBUG:c #1 {
48   BNVS_DEBUG~#1~
49 }
50 \cs_new:Npn \BNVS_DEBUG_on:n #1 {
51   \tl_if_empty:nT { #1 } {
52     \typein { Empty~argument~not~allowed }
53   }
54   \cs_set:cpn { \BNVS_DEBUG:c { #1 } log:n } { \BNVS_log:n }
55   \cs_generate_variant:cn { \BNVS_DEBUG:c { #1 } log:n } { x }
56 }
57 \cs_new:Npn \BNVS_DEBUG_off:n #1 {
58   \cs_set:cpn { \BNVS_DEBUG:c { #1 } log:n } { \use_none:n }
59 }
60 \seq_new:N \l_BNVS_DEBUG_push_n_seq
61 \cs_new:Npn \BNVS_DEBUG_push:n #1 {
62   \tl_if_empty:nT { #1 } {
63     \typein { Empty~argument~not~allowed }
64   }
65   \tl_map_inline:nn { #1 } {
66     \BNVS_DEBUG_on:n { ##1 }
67   }
68   \seq_put_left:Nn \l_BNVS_DEBUG_push_n_seq {
69     \tl_map_inline:nn { #1 } {
70       \BNVS_DEBUG_off:n { ##1 }
71     }
72   }
73 }
74 \tl_new:N \l_BNVS_DEBUG_push_n_tl
75 \cs_new:Npn \BNVS_DEBUG_pop: {
76   \seq_pop_left:NNTF \l_BNVS_DEBUG_push_n_seq \l_BNVS_DEBUG_push_n_tl {
77     \l_BNVS_DEBUG_push_n_tl
78   } {
79     \BNVS_error:n { Unbalanced~\BNVS_DEBUG_pop: }

```

```

80 }
81 }
82 \AddToHookNext { env/BNVS.test/begin } {
83   \BNVS_DEBUG_push:n {CDBGpfarsRomqi}
84   \BNVS_DEBUG_pop:
85 }
86 \cs_new:Npn \BNVS_DEBUG_log:nn #1 {
87   \cs_if_exist_use:cF { \BNVS_DEBUG:c { #1 } log:n } {
88     \BNVS_warning:n { Undeclared~DEBUG~type:~#1 }
89     \cs_new:cpn { \BNVS_DEBUG:c { #1 } log:n } { \use_none:n }
90     \use_none:n
91   }
92 }
93 \cs_new:Npn \BNVS_DEBUG_on: {
94   \BNVS_DEBUG_on:n {C}
95 }
96 \cs_new:Npn \BNVS_DEBUG_off: {
97   \BNVS_DEBUG_off:n {C}
98 }

```

`\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`.

```

99 \cs_new:Npn \BNVS_new:cpn #1 {
100   \cs_new:cpn { \BNVS:c { #1 } }
101 }
102 \cs_new:Npn \BNVS_set:cpn #1 {
103   \cs_set:cpn { \BNVS:c { #1 } }
104 }
105 \cs_generate_variant:Nn \cs_generate_variant:Nn { c }
106 \cs_new:Npn \BNVS_generate_variant:cn #1 {
107   \cs_generate_variant:cn { \BNVS:c { #1 } }
108 }

```

6.3 logging

Utility messaging.

```

109 \msg_new:nnn { beanoves } { :n } { #1 }
110 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
111 \cs_new:Npn \BNVS_warning:n {
112   \msg_warning:nnn { beanoves } { :n }
113 }
114 \cs_new:Npn \BNVS_warning:x {
115   \msg_warning:nnx { beanoves } { :n }
116 }

```

```

117 \cs_new:Npn \BNVS_error:n {
118   \msg_error:nnn { beanoves } { :n }
119 }
120 \cs_new:Npn \BNVS_error:x {
121   \msg_error:nnx { beanoves } { :n }
122 }
123 \cs_new:Npn \BNVS_fatal:n {
124   \msg_fatal:nnn { beanoves } { :n }
125 }
126 \cs_new:Npn \BNVS_fatal:x {
127   \msg_fatal:nnx { beanoves } { :n }
128 }

```

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`...

```

129 \cs_new:Npn \BNVS_N_new:c #1 {
130   \cs_new:cpn { BNVS_#1:c } ##1 {
131     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
132   }
133   \cs_new:cpn { BNVS_#1_new:c } ##1 {
134     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
135   }
136   \cs_new:cpn { BNVS_#1_use:c } ##1 {
137     \use:c { \cs:w BNVS_#1:c \cs_end: { ##1 } }
138   }
139   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
140     \BNVS_use_raw:Nc
141     ##1 { \cs:w BNVS_#1:c \cs_end: { ##2 } }
142   }
143   \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
144     \BNVS_use_raw:nc
145     { ##1 } { \cs:w BNVS_#1:c \cs_end: { ##2 } }
146   }
147 }
148 \cs_new:Npn \BNVS_v_new:c #1 {
149   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
150     \BNVS_use_raw:nc
151     { \exp_args:Nv ##1 }
152     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
153   }
154   \cs_new:cpn { BNVS_#1_use:cv } ##1 ##2 {
155     \BNVS_use_raw:nc
156     { \exp_args:NnV \BNVS_use:c { ##1 } }
157     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
158   }
159   \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {

```

```

160     \BNVS_use_raw:nc
161     { \exp_args:NnV \use:n { ##1 } }
162     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
163 }
164 }

165 \BNVS_N_new:c { bool }
166 \BNVS_N_new:c { int }
167 \BNVS_v_new:c { int }
168 \BNVS_N_new:c { tl }
169 \BNVS_v_new:c { tl }
170 \cs_new:Npn \BNVS_tl_use:Nvv #1 {
171     \BNVS_exp_args:Nvv #1
172 }
173 \BNVS_N_new:c { str }
174 \BNVS_v_new:c { str }
175 \BNVS_N_new:c { seq }
176 \BNVS_v_new:c { seq }
177 \cs_undefine:N \BNVS_N_new:c

```

\BNVS_use:Ncn \BNVS_use:Ncn *<function>* {*<core name>*} {*<type>*}

```

178 \cs_new:Npn \BNVS_use:Ncn #1 #2 #3 {
179     \BNVS_use_raw:c { BNVS_#3_use:Nc } #1 { #2 }
180 }

181 \cs_new:Npn \BNVS_use:ncn #1 #2 #3 {
182     \BNVS_use_raw:c { BNVS_#3_use:nc } { #1 } { #2 }
183 }

184 \cs_new:Npn \BNVS_use:Nvn #1 #2 #3 {
185     \BNVS_use_raw:c { BNVS_#3_use:Nv } #1 { #2 }
186 }

187 \cs_new:Npn \BNVS_use:nvn #1 #2 #3 {
188     \BNVS_use_raw:c { BNVS_#3_use:nv } { #1 } { #2 }
189 }

190 \cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
191     \BNVS_use:ncn {
192         \BNVS_use:Ncn #1 { #2 } { #3 }
193     }
194 }

195 \cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
196     \BNVS_use:ncn {
197         \BNVS_use:ncn { #1 } { #2 } { #3 }
198     }
199 }

200 \cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
201     \BNVS_use:ncn {
202         \BNVS_use:Nvn #1 { #2 } { #3 }
203     }
204 }

```

```

205 \cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
206   \BNVS_use:ncn {
207     \BNVS_use:nvn { #1 } { #2 } { #3 }
208   }
209 }

210 \cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
211   \BNVS_use:ncn {
212     \BNVS_use:Ncncn #1 { #2 } { #3 } { #4 } { #5 }
213   }
214 }

215 \cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
216   \BNVS_use:ncn {
217     \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
218   }
219 }

```

\BNVS_new_c:cn \BNVS_new_c:nc {<type>} {<core name>}

```

220 \cs_new:Npn \BNVS_new_c:nc #1 #2 {
221   \BNVS_new:cpn { #1_#2:c } {
222     \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
223   }
224 }

225 \cs_new:Npn \BNVS_new_cn:nc #1 #2 {
226   \BNVS_new:cpn { #1_#2:cn } ##1 {
227     \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
228   }
229 }

230 \cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
231   \BNVS_new:cpn { #2:cnn } ##1 {
232     \BNVS_use:Ncn { #3 } { ##1 } { #1 }
233   }
234 }

235 \cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
236   \BNVS_use_raw:nc {
237     \BNVS_new_cnn:ncN { #1 } { #1_#2 }
238   } { #1_#2:Nnn }
239 }

240 \cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
241   \BNVS_new:cpn { #2:cnv } ##1 ##2 {
242     \BNVS_tl_use:nv {
243       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
244     }
245   }
246 }

247 \cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
248   \BNVS_use_raw:nc {
249     \BNVS_new_cnv:ncN { #1 } { #1_#2 }
250   } { #1_#2:Nnn }
251 }

```



```

252 \cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
253   \BNVS_new_cpn { #2:cnx } ##1 ##2 {
254     \exp_args:Nnx \use:n {
255       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
256     }
257   }
258 }

259 \cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
260   \BNVS_use_raw:nc {
261     \BNVS_new_cnx:ncN { #1 } { #1_#2 }
262   } { #1_#2:Nnn }
263 }

264 \cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
265   \BNVS_new_cpn { #2:cc } ##1 ##2 {
266     \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
267   }
268 }

269 \cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
270   \BNVS_use_raw:nc {
271     \BNVS_new_cc:ncNn { #1 } { #1_#2 }
272   } { #1_#2:NN }
273 }

274 \cs_new:Npn \BNVS_new_cc:nc #1 #2 {
275   \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
276 }

277 \cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
278   \BNVS_new_cpn { #2:cn } ##1 {
279     \BNVS_use:Ncn #3 { ##1 } { #1 }
280   }
281 }

282 \cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
283   \BNVS_use_raw:nc {
284     \BNVS_new_cn:ncNn { #1 } { #1_#2 }
285   } { #1_#2:Nn }
286 }

287 \cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
288   \BNVS_new_cpn { #2:cv } ##1 ##2 {
289     \BNVS_use:nvn {
290       \BNVS_use:Ncn #3 { ##1 } { #1 }
291     } { ##2 } { #4 }
292   }
293 }

294 \cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
295   \BNVS_use_raw:nc {
296     \BNVS_new_cv:ncNn { #1 } { #1_#2 }
297   } { #1_#2:Nn }
298 }

```

```

299 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
300   \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
301 }
302 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
303   \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
304 }
305 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
306   \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
307 }
308 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
309   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
310 }
311 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
312   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
313 }
314 \cs_new:Npn \BNVS_exp_args:Nvv #1 #2 #3 {
315   \BNVS_use:ncncn { \exp_args:NVV #1 }
316   { #2 } { t1 } { #3 } { t1 }
317 }
318 \cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
319   \BNVS_use:ncncncn { \exp_args:NVVV #1 }
320   { #2 } { t1 } { #3 } { t1 } { #4 } { t1 }
321 }
322 \cs_new:Npn \BNVS_exp_args:Nvvvv #1 #2 #3 #4 #5 {
323   \BNVS_tl_use:nc {
324     \exp_args:NnV \use:n {
325       \BNVS_exp_args:Nvvv #1 { #2 } { #3 } { #4 }
326     }
327   } { #5 }
328 }

```

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

```

329 \cs_generate_variant:Nn \prg_new_conditional:Npnn { c }
330 \cs_new:Npn \BNVS_new_conditional:cpnn #1 {
331   \prg_new_conditional:cpnn { \BNVS:c { #1 } }
332 }
333 \cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
334 \cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
335   \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
336 }
337 \cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
338   \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
339     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
340       \prg_return_true:
341     } {
342       \prg_return_false:
343     }
344   }
345 }

```

```

346 \cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
347   \BNVS_use:nc {
348     \BNVS_new_conditional_vn:cNnn { #1 }
349   } { #1:nn TF } { #2 }
350 }

351 \cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
352   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
353     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
354       \prg_return_true:
355     } {
356       \prg_return_false:
357     }
358   }
359 }

360 \cs_new:Npn \BNVS_new_conditional_vc:cnn #1 {
361   \BNVS_use:nc {
362     \BNVS_new_conditional_vc:cNnn { #1 }
363   } { #1:ncTF }
364 }

365 \cs_new:Npn \BNVS_new_conditional_vvc:cNnnn #1 #2 #3 #4 #5 {
366   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #5 } {
367     \BNVS_use:nvn {
368       \BNVS_use:Nvn #2 { ##1 } { #3 }
369     } { ##2 } { #4 } { ##3 } {
370       \prg_return_true:
371     } {
372       \prg_return_false:
373     }
374   }
375 }

376 \cs_new:Npn \BNVS_new_conditional_vvc:cnnn #1 {
377   \BNVS_use:nc {
378     \BNVS_new_conditional_vvc:cNnnn { #1 }
379   } { #1:nncTF }
380 }

381 \cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
382   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
383     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
384       \prg_return_true:
385     } {
386       \prg_return_false:
387     }
388   }
389 }

390 \cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
391   \BNVS_use:nc {
392     \BNVS_new_conditional_vc:cNn { #1 }
393   } { #1:ncTF }
394 }

```

```

395 \cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
396   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
397     \BNVS_tl_use:nv {
398       \BNVS_tl_use:Nv #2 { ##1 }
399     } { ##2 } { ##3 } {
400       \prg_return_true:
401     } {
402       \prg_return_false:
403     }
404   }
405 }

406 \cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
407   \BNVS_use:nc {
408     \BNVS_new_conditional_vvc:cNn { #1 }
409   } { #1:nncTF }
410 }

```

6.4.1 Regex

```

411 \cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
412   \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
413 }

```

```

\__bnvs_match_if_once:NnTF \__bnvs_match_if_once:NnTF <regex variable> {<expression>}
\__bnvs_match_if_once:NvTF {<yes code>} {<no code>}
\__bnvs_match_if_once:nnTF \__bnvs_match_if_once:nnTF {<regex>} {<expression>}
\__bnvs_if_regex_split:cnTF {<yes code>} {<no code>}
    \__bnvs_if_regex_split:cncTF {<regex core>} {<expression>} <seq core> {<yes
    code>} {<no code>}
    \__bnvs_if_regex_split:cnTF {<regex core>} {<expression>} {<yes code>} {<no
    code>}

```

These are shortcuts to

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

```

414 \BNVS_new_conditional:cpnn { if_extract_once:Ncn } #1 #2 #3 { T, F, TF } {
415   \BNVS_use:ncn {
416     \regex_extract_once:NnNTF #1 { #3 }
417   } { #2 } { seq } {
418     \prg_return_true:
419   } {
420     \prg_return_false:
421   }
422 }

```

```

423 \BNVS_new_conditional:cpnn { match_if_once:Nn } #1 #2 { T, F, TF } {
424   \BNVS_use:ncn {
425     \regex_extract_once:NnNTF #1 { #2 }
426   } { match } { seq } {
427     \prg_return_true:
428   } {
429     \prg_return_false:
430   }
431 }

432 \BNVS_new_conditional:cpnn { if_extract_once:Ncv } #1 #2 #3 { T, F, TF } {
433   \BNVS_seq_use:nc {
434     \BNVS_tl_use:nv {
435       \regex_extract_once:NnNTF #1
436     } { #3 }
437   } { #2 } {
438     \prg_return_true:
439   } {
440     \prg_return_false:
441   }
442 }

443 \BNVS_new_conditional:cpnn { match_if_once:Nv } #1 #2 { T, F, TF } {
444   \BNVS_seq_use:nc {
445     \BNVS_tl_use:nv {
446       \regex_extract_once:NnNTF #1
447     } { #2 }
448   } { match } {
449     \prg_return_true:
450   } {
451     \prg_return_false:
452   }
453 }

454 \BNVS_new_conditional:cpnn { match_if_once:nn } #1 #2 { T, F, TF } {
455   \BNVS_seq_use:nc {
456     \regex_extract_once:nnNTF { #1 } { #2 }
457   } { match } {
458     \prg_return_true:
459   } {
460     \prg_return_false:
461   }
462 }

463 \BNVS_new_conditional:cpnn { if_regex_split:cnc } #1 #2 #3 { T, F, TF } {
464   \BNVS_seq_use:nc {
465     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
466   } { #3 } {
467     \prg_return_true:
468   } {
469     \prg_return_false:
470   }
471 }

```

```

472 \BNVS_new_conditional:cpnn { if_regex_split:cn } #1 #2 { T, F, TF } {
473   \BNVS_seq_use:nc {
474     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
475   } { split } {
476     \prg_return_true:
477   } {
478     \prg_return_false:
479   }
480 }

```

6.4.2 Token lists

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

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.

```

481 \cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
482   \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
483     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
484       \prg_return_true:
485     } {
486       \prg_return_false:
487     }
488   }
489 }

```

```

490 \cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
491   \BNVS_use:nc {
492     \BNVS_new_conditional_vnc:cNn { #1 }
493   } { #1:nncTF }
494 }

```

\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.

```

495 \cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
496   \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
497     \BNVS_tl_use:nv {
498       \BNVS_tl_use:Nv #2 { ##1 }
499     } { ##2 } { ##3 } { ##4 } {
500       \prg_return_true:
501     } {
502       \prg_return_false:
503     }
504   }
505 }

506 \cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
507   \BNVS_use:nc {
508     \BNVS_new_conditional_vvnc:cNn { #1 }
509   } { #1:nnncTF }
510 }

511 \cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
512   \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
513     \BNVS_tl_use:nvv {
514       \BNVS_tl_use:Nv #2 { ##1 }
515     } { ##2 } { ##3 } { ##4 } {
516       \prg_return_true:
517     } {
518       \prg_return_false:
519     }
520   }
521 }

522 \cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
523   \BNVS_use:nc {
524     \BNVS_new_conditional_vvvc:cNn { #1 }
525   } { #1:nnncTF }
526 }

527 \cs_new:Npn \BNVS_new_tl_c:c {
528   \BNVS_new_c:nc { tl }
529 }
530 \BNVS_new_tl_c:c { clear }
531 \BNVS_new_tl_c:c { use }
532 \BNVS_new_tl_c:c { count }

```

```

533 \BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
534   \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
535 }

536 \cs_new:Npn \BNVS_new_tl_cn:c {
537   \BNVS_new_cn:nc { tl }
538 }

539 \cs_new:Npn \BNVS_new_tl_cv:c #1 {
540   \BNVS_new_cv:ncn { tl } { #1 } { tl }
541 }
542 \BNVS_new_tl_cn:c { set }
543 \BNVS_new_tl_cv:c { set }

544 \BNVS_new:cpn { tl_set:cx } {
545   \exp_args:Nnx \__bnvs_tl_set:cn
546 }
547 \BNVS_new_tl_cn:c { put_right }
548 \BNVS_new_tl_cv:c { put_right }
549 % \BNVS_generate_variant:cn { tl_put_right:cn } { cx }

550 \BNVS_new:cpn { tl_put_right:cx } {
551   \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
552 }
553 \BNVS_new_tl_cn:c { put_left }
554 \BNVS_new_tl_cv:c { put_left }
555 % \BNVS_generate_variant:cn { tl_put_left:cn } { cx }

556 \BNVS_new:cpn { tl_put_left:cx } {
557   \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
558 }

```

```

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

```

These are shortcuts to

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

```

559 \cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
560   \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
561     \BNVS_use:Ncn #3 { ##1 } { #1 } {
562       \prg_return_true:
563     } {
564       \prg_return_false:
565     }
566   }
567 }

```



```

568 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
569   \BNVS_use_raw:nc {
570     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
571   } { #1_#2:NnTF }
572 }
573 \BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }
574 \BNVS_new_conditional_cpnn { tl_if_blank:v } #1 { T, F, TF } {
575   \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
576     \prg_return_true:
577   } {
578     \prg_return_false:
579   }
580 }

581 \cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
582   \BNVS_new_conditional_cpnn { #2:cn } ##1 ##2 { #4 } {
583     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
584       \prg_return_true:
585     } {
586       \prg_return_false:
587     }
588   }
589 }

590 \cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
591   \BNVS_use_raw:nc {
592     \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
593   } { #1_#2:NnTF }
594 }
595 \BNVS_new_conditional_cn:ncn { tl } { if_eq } { T, F, TF }

596 \cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
597   \BNVS_new_conditional_cpnn { #2:cv } ##1 ##2 { #4 } {
598     \BNVS_use:nvn {
599       \BNVS_use:Ncn #3 { ##1 } { #1 }
600     } { ##2 } { #1 } {
601       \prg_return_true:
602     } {
603       \prg_return_false:
604     }
605   }
606 }

607 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
608   \BNVS_use_raw:nc {
609     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
610   } { #1_#2:NnTF }
611 }
612 \BNVS_new_conditional_cv:ncn { tl } { 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>}

```

613 \cs_new:Npn \BNVS_new_conditional_vv:cNn #1 #2 #3 {
614   \BNVS_new_conditional:cpnn { #1:vv } ##1 ##2 { #3 } {
615     \BNVS_tl_use:nv {
616       \BNVS_tl_use:Nv #2 { ##1 }
617     } { ##2 } {
618       \prg_return_true:
619     } {
620       \prg_return_false:
621     }
622   }
623 }

624 \cs_new:Npn \BNVS_new_conditional_vv:cn #1 {
625   \BNVS_use:nc {
626     \BNVS_new_conditional_vvnc:cNn { #1 }
627   } { #1:nnTF }
628 }

629 \cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
630   \BNVS_new_conditional:cpnn { #2:vn } ##1 ##2 { #4 } {
631     \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
632       \prg_return_true:
633     } {
634       \prg_return_false:
635     }
636   }
637 }

638 \cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
639   \BNVS_use_raw:nc {
640     \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
641   } { #1_#2:nnTF }
642 }
643 \BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }

644 \cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
645   \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
646     \BNVS_use:nvn {
647       \BNVS_use:Nvn #3 { ##1 } { #1 }
648     } { ##2 } { #1 } {
649       \prg_return_true:
650     } {
651       \prg_return_false:
652     }
653   }
654 }
```

```

655 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
656   \BNVS_use_raw:nc {
657     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
658   } { #1_#2:nnTF }
659 }
660 \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_gset_eq:cc</code>	<code>__bnvs_seq_set_eq:cc {<core₁>} {<core₂>}</code>
<code>__bnvs_seq_use:cn</code>	<code>__bnvs_seq_use:cn {<core>} {<separator>}</code>
<code>__bnvs_seq_item:cn</code>	<code>__bnvs_seq_item:cn {<core>} {<integer expression>}</code>
<code>__bnvs_seq_remove_all:cn</code>	<code>__bnvs_seq_remove_all:cn {<core>} {<tl>}</code>
<code>__bnvs_seq_put_left:cv</code>	<code>__bnvs_seq_put_right:cn {<seq core>} {<tl>}</code>
<code>__bnvs_seq_put_right:cn</code>	<code>__bnvs_seq_put_right:cv {<seq core>} {<tl core>}</code>
<code>__bnvs_seq_put_right:cv</code>	<code>__bnvs_seq_set_split:cnn {<seq core>} {<tl>} {<separator>}</code>
<code>__bnvs_seq_set_split:cnn</code>	<code>__bnvs_seq_pop_left:cc {<core₁>} {<core₂>}</code>
<code>__bnvs_seq_set_split:(cnv cnx)</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>}`

```

661 \BNVS_new_c:nc { seq } { count }
662 \BNVS_new_c:nc { seq } { clear }
663 \BNVS_new_cn:nc { seq } { use }
664 \BNVS_new_cn:nc { seq } { item }
665 \BNVS_new_cn:nc { seq } { remove_all }
666 \BNVS_new_cn:nc { seq } { map_inline }
667 \BNVS_new_cc:nc { seq } { set_eq }
668 \BNVS_new_cc:nc { seq } { gset_eq }
669 \BNVS_new_cv:ncn { seq } { put_left } { tl }
670 \BNVS_new_cn:ncn { seq } { put_right } { tl }
671 \BNVS_new_cv:ncn { seq } { put_right } { tl }
672 \BNVS_new_cnn:nc { seq } { set_split }

```

```

673 \BNVS_new_cnv:nc { seq } { set_split }
674 \BNVS_new_cnx:nc { seq } { set_split }
675 \BNVS_new_cc:ncn { seq } { pop_left } { tl }
676 \BNVS_new_cc:ncn { seq } { pop_right } { tl }

```

```

\__bnvs_seq_if_empty:cTF \__bnvs_seq_if_empty:cTF {<seq core>} {<yes code>} {<no code>}
\__bnvs_seq_get_right:ccTF \__bnvs_seq_get_right:ccTF {<seq core>} {<tl core>} {<yes code>} {<no code>}
\__bnvs_seq_pop_left:ccTF
\__bnvs_seq_pop_right:ccTF

```

```

677 \cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
678   \BNVS_new_conditional_cpnn { #1_#2:cc } ##1 ##2 { #4 } {
679     \BNVS_use:ncncn {
680       \BNVS_use_raw:c { #1_#2:NNTF }
681     } { ##1 } { #1 } { ##2 } { #3 } {
682       \prg_return_true:
683     } {
684       \prg_return_false:
685     }
686   }
687 }
688 \BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
689 \BNVS_new_conditional_cc:ncnn
690   { seq } { get_right } { tl } { T, F, TF }
691 \BNVS_new_conditional_cc:ncnn
692   { seq } { pop_left } { tl } { T, F, TF }
693 \BNVS_new_conditional_cc:ncnn
694   { 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_zero:c \__bnvs_int_incr:c {<core>}
\__bnvs_int_inc:c \__bnvs_int_decr:c {<core>}
\__bnvs_int_decr:c \__bnvs_int_set:cn {<core>} {<value>}
\__bnvs_int_set:cn
\__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>

```

695 \BNVS_new_c:nc { int } { new }
696 \BNVS_new_c:nc { int } { use }
697 \BNVS_new_c:nc { int } { zero }
698 \BNVS_new_c:nc { int } { incr }

```

```

699 \BNVS_new_c:nc { int } { decr }
700 \BNVS_new_cn:nc { int } { set }
701 \BNVS_new_cv:ncn { int } { set } { int }

```

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 Testing facilities

6.8 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.

```

702 \tl_new:N \l__bnvs_id_last_tl
703 \tl_new:N \l__bnvs_id_tl
704 \tl_new:N \l__bnvs_kri_tl
705 \tl_new:N \l__bnvs_short_tl
706 \tl_new:N \l__bnvs_path_tl
707 \tl_new:N \l__bnvs_n_tl
708 \tl_new:N \l__bnvs_ref_tl
709 \tl_new:N \l__bnvs_tag_tl
710 \tl_new:N \l__bnvs_a_tl
711 \tl_new:N \l__bnvs_b_tl
712 \tl_new:N \l__bnvs_c_tl
713 \tl_new:N \l__bnvs_V_tl
714 \tl_new:N \l__bnvs_A_tl
715 \tl_new:N \l__bnvs_L_tl
716 \tl_new:N \l__bnvs_Z_tl
717 \tl_new:N \l__bnvs_ans_tl
718 \tl_new:N \l__bnvs_QD_name_tl
719 \tl_new:N \l__bnvs_base_tl
720 \tl_new:N \l__bnvs_group_tl
721 \tl_new:N \l__bnvs_scan_tl
722 \tl_new:N \l__bnvs_query_tl
723 \tl_new:N \l__bnvs_token_tl
724 \tl_new:N \l__bnvs_root_tl
725 \tl_new:N \l__bnvs_n_incr_tl
726 \tl_new:N \l__bnvs_incr_tl
727 \tl_new:N \l__bnvs_plus_tl
728 \tl_new:N \l__bnvs_rhs_tl
729 \tl_new:N \l__bnvs_post_tl
730 \tl_new:N \l__bnvs_suffix_tl
731 \tl_new:N \l__bnvs_index_tl
732 \int_new:N \g__bnvs_call_int

```

```

733 \int_new:N \l__bnvs_int
734 \int_new:N \l__bnvs_i_int
735 \seq_new:N \g__bnvs_def_seq
736 \seq_new:N \l__bnvs_a_seq
737 \seq_new:N \l__bnvs_b_seq
738 \seq_new:N \l__bnvs_ans_seq
739 \seq_new:N \l__bnvs_match_seq
740 \seq_new:N \l__bnvs_split_seq
741 \seq_new:N \l__bnvs_path_seq
742 \seq_new:N \l__bnvs_path_head_seq
743 \seq_new:N \l__bnvs_path_tail_seq
744 \seq_new:N \l__bnvs_query_seq
745 \seq_new:N \l__bnvs_token_seq
746 \bool_new:N \l__bnvs_in_frame_bool
747 \bool_set_false:N \l__bnvs_in_frame_bool
748 \bool_new:N \l__bnvs_parse_bool
749 \bool_set_false:N \l__bnvs_parse_bool
750 \bool_new:N \l__bnvs_deep_bool
751 \bool_set_false:N \l__bnvs_deep_bool

752 \cs_new:Npn \BNVS_error_ans:x {
753   \__bnvs_tl_put_right:cn { ans } { 0 }
754   \BNVS_error:x
755 }

```

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

```

756 \bool_new:N \l__bnvs_provide_bool

757 \BNVS_new:cpn { set_true:c } #1 {
758   \exp_args:Nc \bool_set_true:N { l__bnvs_#1_bool }
759 }

760 \BNVS_new:cpn { set_false:c } #1 {
761   \exp_args:Nc \bool_set_false:N { l__bnvs_#1_bool }
762 }

763 \BNVS_new:cpn { provide_on: } {
764   \__bnvs_set_true:c { provide }
765 }

766 \BNVS_new:cpn { provide_off: } {
767   \__bnvs_set_false:c { provide }
768 }
769 \__bnvs_provide_off:

```

6.9 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.)

```

770 \int_const:Nn \c__bnvs_max_call_int { 8192 }

```

__bnvs_greset_call: __bnvs_greset_call:

Reset globally the call stack counter to its maximum value.

```

771 \BNVS_new:cpn { greset_call: } {
772   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
773 }

```

__bnvs_if_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.

```

774 \BNVS_new_conditional:cpnn { if_call: } { T, F, TF } {
775   \int_gdecr:N \g__bnvs_call_int
776   \int_compare:nNnTF \g__bnvs_call_int > 0 {
777     \prg_return_true:
778   } {
779     \prg_return_false:
780   }
781 }

```

6.10 Overlay specification

6.10.1 Registration

We keep track of the `<id>` `<a>` combinations and provide looping mechanisms.

<code>__bnvs_name:nnn</code>	<code>__bnvs_name:nnn {<subkey>} {<id>} {<a>}</code>
<code>__bnvs_name:nn</code>	<code>__bnvs_name:nn {<id>} {<a>}</code>
<code>__bnvs_id_seq:n</code>	<code>__bnvs_id_seq:nn {<id>}</code>

Create a unique name from the arguments.

```

782 \BNVS_new:cpn { name:nnn } #1 #2 #3 { __bnvs_#2!#3/#1: }
783 \BNVS_new:cpn { name:nn } #1 #2 { __bnvs_#1!#2: }
784 \BNVS_new:cpn { id_seq:n } #1 { g__bnvs_#1!_seq }

```

`\g__bnvs_I_seq` List of registered identifiers.

(End of definition for `\g__bnvs_I_seq`.)

```

785 \seq_new:N \g__bnvs_I_seq

```

<code>__bnvs_register:nn</code>	<code>__bnvs_register:nn {<id>} {<a>}</code>
<code>__bnvs_unregister:nn</code>	<code>__bnvs_unregister:nn {<id>} {<a>}</code>
<code>__bnvs_unregister:n</code>	<code>__bnvs_unregister:n {<id>}</code>
<code>__bnvs_unregister:</code>	<code>__bnvs_unregister:</code>

Register and unregister according to the arguments.

```

786 \seq_new:N \l__bnvs_register_NNnn_seq
787 \BNVS_new:cpn { register:NNnn } #1 #2 #3 #4 {
788   \cs_if_exist:NF #1 {

```

```

789 \cs_gset:Npn #1 { }
790 \seq_if_exist:NTF #2 {
791   \__bnvs_seq_clear:c { register_NNnn }
792   \cs_set:Npn \BNVS_register_NNnn: {
793     \__bnvs_seq_put_right:cn { register_NNnn } { #4 }
794     \cs_set:Npn \BNVS_register_NNnn: { }
795   }
796   \cs_set:Npn \BNVS_register_NNnn:w ##1 ##2 {
797     \str_compare:nNnTF { ##2 } < { #4 } {
798       \__bnvs_seq_put_right:cn { register_NNnn } { ##2 }
799     } {
800       \BNVS_register_NNnn:
801       \__bnvs_seq_put_right:cn { register_NNnn } { ##2 }
802       \cs_set:Npn \BNVS_register_NNnn:w ####1 ####2 {
803         \__bnvs_seq_put_right:cn { register_NNnn } { ####2 }
804       }
805     }
806   }
807   \__bnvs_foreach_T:nNTF { #3 } \BNVS_register_NNnn:w {
808     \BNVS_register_NNnn:
809     \seq_gset_eq:NN #2 \l__bnvs_register_NNnn_seq
810   } {
811     \BNVS_error:n { Unreachable/register:NNnn-id-#3 }
812   }
813 } {
814   \seq_new:N #2
815   \seq_gput_right:Nn #2 { #4 }
816   \__bnvs_seq_clear:c { register_NNnn }
817   \cs_set:Npn \BNVS_register_NNnn: {
818     \__bnvs_seq_put_right:cn { register_NNnn } { #3 }
819     \cs_set:Npn \BNVS_register_NNnn: {}
820   }
821   \cs_set:Npn \BNVS_register_NNnn:w ##1 {
822     \str_compare:nNnTF { ##1 } < { #3 } {
823       \__bnvs_seq_put_right:cn { register_NNnn } { ##1 }
824     } {
825       \BNVS_register_NNnn:
826       \__bnvs_seq_put_right:cn { register_NNnn } { ##1 }
827       \cs_set:Npn \BNVS_register_NNnn:w ####1 {
828         \__bnvs_seq_put_right:cn { register_NNnn } { ####1 }
829       }
830     }
831   }
832   \__bnvs_foreach_I:N \BNVS_register_NNnn:w
833   \BNVS_register_NNnn:
834   \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_register_NNnn_seq
835 }
836 }
837 }
838 \BNVS_new:cpn { register:nn } #1 #2 {
839   \exp_args:Ncc \__bnvs_register:NNnn
840   { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
841   { #1 } { #2 }
842 }

```

```

__bnvs_unregister:NNnn  \__bnvs_unregister:NNnn <cs> <seq> {<id>} {<a>}

```

Unregistering a $\langle id \rangle$ $\langle a \rangle$ combination is not straightforward. $\langle cs \rangle$ and $\langle seq \rangle$ are respectively the command and the sequence uniquely associated to this combination.

```

843 \seq_new:N \l__bnvs_unregister_NNnn_seq
844 \BNVS_new:cpn { unregister:NNnn } #1 #2 #3 #4 {
845   \cs_if_exist:NT #1 {
846     \cs_undefine:N #1
847     \__bnvs_seq_clear:c { unregister_NNnn }
848     \cs_set:Npn \BNVS_unregister_NNnn:n ##1 { ##1 }
849     \cs_set:Npn \BNVS_unregister_NNnn:w ##1 ##2 {
850       \str_compare:nNnTF { ##2 } < { #4 } {
851         \__bnvs_seq_put_right:cn { unregister_NNnn } { ##2 }
852         \cs_set:Npn \BNVS_unregister_NNnn:n #####1 { }
853       } {
854         \cs_set:Npn \BNVS_unregister_NNnn:w #####1 #####2 {
855           \__bnvs_seq_put_right:cn { unregister_NNnn } { #####2 }
856           \cs_set:Npn \BNVS_unregister_NNnn:n #####1 { }
857         }
858       }
859     }
860     \__bnvs_foreach_T:nNTF { #3 } \BNVS_unregister_NNnn:w {
861       \seq_gset_eq:NN #2 \l__bnvs_unregister_NNnn_seq
862     } {
863       \BNVS_error:n { Unreachable / unregister:NNnn~#3!#4 }
864     }
865     \BNVS_unregister_NNnn:n {
866       \__bnvs_seq_clear:c { unregister_NNnn }
867       \cs_set:Npn \BNVS_unregister_NNnn:w ##1 {
868         \str_compare:nNnTF { ##1 } < { #3 } {
869           \__bnvs_seq_put_right:cn { unregister_NNnn } { ##1 }
870         } {
871           \cs_set:Npn \BNVS_unregister_NNnn:n #####1 {
872             \__bnvs_seq_put_right:cn { unregister_NNnn } { #####1 }
873           }
874         }
875       }
876       \__bnvs_foreach_I:N \BNVS_unregister_NNnn:w
877       \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_unregister_NNnn_seq
878       \cs_undefine:N #2
879     }
880   }
881 }

882 \BNVS_new:cpn { unregister:nn } #1 #2 {
883   \exp_args:Ncc \__bnvs_unregister:NNnn
884   { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
885   { #1 } { #2 }
886 }

```

```

__bnvs_foreach_I:N  \__bnvs_foreach_I:N <function:n>
__bnvs_foreach_I:n  \__bnvs_foreach_I:n {<code>}

```

Execute the $\langle function:n \rangle$ or the $\langle code \rangle$ for each declared identifier.

```

887 \BNVS_new:cpn { foreach_I:N } {
888   \seq_map_function:NN \g__bnvs_I_seq
889 }
890 \BNVS_new:cpn { foreach_I:n } {
891   \seq_map_inline:Nn \g__bnvs_I_seq
892 }

```

```

\__bnvs_foreach_T:nNTF \__bnvs_foreach_T:nNTF {<id>} {<function:nn>} {<yes code>} {<no code>}
\__bnvs_foreach_T:nnTF \__bnvs_foreach_T:nnTF {<id>} {<code>} {<yes code>} {<no code>}

```

If $\langle id \rangle$ is a declared identifier, execute $\langle function:nn \rangle$ or $\langle code \rangle$ for each combination of $\langle id \rangle$ and its associate $\langle a \rangle$ s.

```

893 \BNVS_new_conditional:cpnn { foreach_T:nN } #1 #2 { T, F, TF } {
894   \seq_if_exist:CTF { g__bnvs_#1!_seq } {
895     \seq_map_inline:cn { g__bnvs_#1!_seq } { #2 { #1 } { ##1 } }
896     \prg_return_true:
897   } { \prg_return_false: }
898 }
899 \BNVS_new_conditional:cpnn { foreach_T:nn } #1 #2 { T, F, TF } {
900   \seq_if_exist:CTF { g__bnvs_#1!_seq } {
901     \cs_set:Npn \BNVS_foreach_T_nn:nn ##1 ##2 { #2 }
902     \seq_map_inline:cn { g__bnvs_#1!_seq }
903     { \BNVS_foreach_T_nn:nn { #1 } { ##1 } }
904     \prg_return_true:
905   } { \prg_return_false: }
906 }

```

```

\__bnvs_foreach_IT:N \__bnvs_foreach_IT:N {<function:nn>}
\__bnvs_foreach_IT:n \__bnvs_foreach_IT:n {<code>}

```

Execute the $\langle function:nn \rangle$ or the $\langle code \rangle$ for each combination of $\langle id \rangle$ and $\langle a \rangle$.

```

907 \BNVS_new:cpn { foreach_IT:N } #1 {
908   \__bnvs_foreach_I:n {
909     \__bnvs_foreach_T:nNT { ##1 } #1 { }
910   }
911 }
912 \BNVS_new:cpn { foreach_IT:n } #1 {
913   \cs_set:Npn \BNVS_foreach_IT_n:nn ##1 ##2 { #1 }
914   \__bnvs_foreach_I:n {
915     \__bnvs_foreach_T:nNT { ##1 } \BNVS_foreach_IT_n:nn { }
916   }
917 }

```

```

\__bnvs_foreach_I:N \__bnvs_foreach_key:N {<function:n>}
\__bnvs_foreach_I:n \__bnvs_foreach_key:n {<code>}
\__bnvs_foreach_key_main:N {<function:n>}
\__bnvs_foreach_key_main:n {<code>}
\__bnvs_foreach_key_sub:N {<function:n>}
\__bnvs_foreach_key_sub:n {<code>}
\__bnvs_foreach_key_cache:N {<function:n>}
\__bnvs_foreach_key_cache:n {<code>}

```

Execute the $\langle function:n \rangle$ or the $\langle code \rangle$ for each concerned key.

```

918 \BNVS_new:cpn { foreach_key_main:N } {
919   \tl_map_function:nN { VWAZL }
920 }
921 \BNVS_new:cpn { foreach_key_main:n } {
922   \tl_map_inline:nn { VWAZL }
923 }
924 \BNVS_new:cpn { foreach_key_sub:N } {
925   \tl_map_function:nN { PNvn }
926 }
927 \BNVS_new:cpn { foreach_key_sub:n } {
928   \tl_map_inline:nn { PNvn }
929 }
930 \BNVS_new:cpn { foreach_key:n } #1 {
931   \__bnvs_foreach_key_main:n { #1 }
932   \__bnvs_foreach_key_sub:n { #1 }
933 }
934 \BNVS_new:cpn { foreach_key:N } #1 {
935   \__bnvs_foreach_key_main:N #1
936   \__bnvs_foreach_key_sub:N #1
937 }
938 \BNVS_new:cpn { foreach_key_cache:N } {
939   \tl_map_function:nN { {V*}{A*}{Z*}{L*}{P*}{N*} }
940 }
941 \BNVS_new:cpn { foreach_key_cache:n } {
942   \tl_map_inline:nn { {V*}{A*}{Z*}{L*}{P*}{N*} }
943 }

```

6.10.2 Basic functions

```

\__bnvs_gset:nnnn \__bnvs_gset:nnnn {<key>} {<id>} {<a>} {<spec>}
\__bnvs_gset:(nnnv|nvvn|nvvv)

```

Convenient shortcuts to manage the storage, it makes the code more concise and readable.

```

944 \BNVS_new:cpn { gset:nnnn } #1 #2 #3 {
945   \regex_match:nnTF { ^[a-z_]+$ } { #3 } {
946     \use_none:n
947   } {
948     \__bnvs_register:nn { #2 } { #3 }
949     \cs_gset:cpn { \__bnvs_name:nnn { #1 } { #2 } { #3 } }
950   }
951 }
952 \BNVS_new:cpn { gset:nvvn } #1 {
953   \BNVS_tl_use:nv { \__bnvs_gset:nnnn { #1 } }
954 }

```

```

955 \BNVS_new:cpn { gset:nnnv } #1 #2 #3 {
956   \BNVS_tl_use:nv {
957     \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
958   }
959 }

960 \BNVS_new:cpn { gset:nvvv } #1 {
961   \BNVS_tl_use:nvvv { \__bnvs_gset:nnnn { #1 } }
962 }

```

__bnvs_gunset:nnn	__bnvs_gunset:nnn {<key>} {<id>} {<tag>}
__bnvs_gunset:nn	__bnvs_gunset:nn {<id>} {<tag>}
__bnvs_gunset:n	__bnvs_gunset:n {<id>}
__bnvs_gunset:	__bnvs_gunset:

Removes the specifications for the $\langle key \rangle$, $\langle id \rangle$, $\langle tag \rangle$ combination. In the variant, all possible $\langle key \rangle$ s and $\langle tag \rangle$ s are used.

```

963 \BNVS_new:cpn { gunset:nnn } #1 #2 #3 {
964   \cs_undefine:c { \__bnvs_name:nnn { #1 } { #2 } { #3 } }
965 }

966 \BNVS_new:cpn { gunset:nvv } #1 {
967   \BNVS_tl_use:nvv { \__bnvs_gunset:nnn { #1 } }
968 }

969 \BNVS_new:cpn { gunset:nn } #1 #2 {
970   \tl_map_inline:nn {
971     \__bnvs_foreach_key_main:n
972     \__bnvs_foreach_key_sub:n
973     \__bnvs_foreach_key_cache:n
974   } {
975     ##1 {
976       \__bnvs_gunset:nnn { #####1 } { #1 } { #2 }
977     }
978   }
979 }

980 \BNVS_new:cpn { gunset_deep:nn } #1 #2 {
981   \__bnvs_foreach_IT:n {
982     \tl_if_eq:nnT { #1 } { ##1 } {
983       \tl_if_in:nnT { .. ##2 } { .. #2 . } {
984         \__bnvs_gunset:nn { #1 } { ##2 }
985       }
986     }
987   }
988 }

989 \BNVS_new:cpn { gunset:vv } {
990   \BNVS_tl_use:Nvv \__bnvs_gunset:nn
991 }

992 \BNVS_new:cpn { gunset_deep:vv } {
993   \BNVS_tl_use:Nvv \__bnvs_gunset_deep:nn
994 }

```

```

995 \seq_new:N \l__bnvs_gunset_n_seq
996 \BNVS_new:cpn { gunshot:n } #1 {
997   \__bnvs_seq_clear:c { gunshot_n }
998   \__bnvs_foreach_I:n {
999     \tl_if_eq:nnTF { ##1 } { #1 } {
1000       \__bnvs_foreach_T:nn { #1 } {
1001         \__bnvs_gunset:nn { #1 } { ####1 }
1002       }
1003     } {
1004       \__bnvs_seq_put_right:cn { gunshot_n } { ##1 }
1005     }
1006   }
1007   \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_gunset_n_seq
1008 }

1009 \BNVS_new:cpn { gunshot: } {
1010   \__bnvs_foreach_IT:N \__bnvs_gunset:nn
1011 }

```

<u>__bnvs_is_gset:nnnTF</u>	__bnvs_is_gset:nnnTF {<key>} {<id>} {<tag>} {<yes code>} {<no code>}
<u>__bnvs_is_gset:nnTF</u>	__bnvs_is_gset:nnTF {<id>} {<tag>} {<yes code>} {<no code>}
<u>__bnvs_if_spec:nnnTF</u>	__bnvs_if_spec:nnnTF {<key>} {<id>} {<tag>} {<yes code>} {<no code>}
<u>__bnvs_if_spec:nnTF</u>	__bnvs_if_spec:nnTF {<id>} {<tag>} {<yes code>} {<no code>}

Convenient shortcuts to test for the existence of a *<spec>* for that *<key>* metaid, metatag combination. The version with no *<key>* is the or combination for keys V, A and Z.

The *_spec:...* variant is similar except that it uses *<key>* metaid, metaref or *<key>* empty metaid, metatag combinations.

```

1012 \BNVS_new_conditional:cpnn { is_gset:nnn } #1 #2 #3 { T, F, TF } {
1013   \cs_if_exist:cTF { \__bnvs_name:nnn { #1 } { #2 } { #3 } } {
1014     \prg_return_true:
1015   } {
1016     \prg_return_false:
1017   }
1018 }

1019 \BNVS_new_conditional:cpnn { is_gset:nvv } #1 #2 #3 { T, F, TF } {
1020   \BNVS_tl_use:nvv {
1021     \__bnvs_is_gset:nnnTF { #1 }
1022   } { #2 } { #3 } {
1023     \prg_return_true:
1024   } {
1025     \prg_return_false:
1026   }
1027 }

1028 \BNVS_new_conditional:cpnn { is_gset:nn } #1 #2 { T, F, TF } {
1029   \__bnvs_is_gset:nnnTF V { #1 } { #2 } {
1030     \prg_return_true:
1031   } {
1032     \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
1033       \prg_return_true:
1034     } {
1035       \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {

```

```

1036         \prg_return_true:
1037     } {
1038         \prg_return_false:
1039     }
1040 }
1041 }
1042 }

1043 \BNVS_new_conditional:cpnn { if_spec:nnn } #1 #2 #3 { T, F, TF } {
1044     \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
1045         \prg_return_true:
1046     } {
1047         \tl_if_empty:nTF { #2 } {
1048             \prg_return_false:
1049         } {
1050             \__bnvs_is_gset:nnnTF { #1 } { } { #3 } {
1051                 \prg_return_true:
1052             } {
1053                 \prg_return_false:
1054             }
1055         }
1056     }
1057 }

1058 \BNVS_new_conditional:cpnn { if_spec:nn } #1 #2 { T, F, TF } {
1059     \__bnvs_is_gset:nnTF { #1 } { #2 } {
1060         \prg_return_true:
1061     } {
1062         \tl_if_empty:nTF { #1 } {
1063             \prg_return_false:
1064         } {
1065             \__bnvs_is_gset:nnTF { } { #2 } {
1066                 \prg_return_true:
1067             } {
1068                 \prg_return_false:
1069             }
1070         }
1071     }
1072 }

```

<u>__bnvs_if_get:nnncTF</u> <u>__bnvs_spec:nnncTF</u>	__bnvs_if_get:nnncTF {<key>} {<id>} {<a>} {<ans>} {<yes code>} {<no code>} __bnvs_spec:nnncTF {<key>} {<id>} {<a>} {<ans>} {<yes code>} {<no code>}
--	--

The `__bnvs_if_get:nnnc...` variant puts what was stored for `<key>`, `<id>` and `<a>` into the `<ans>` variable, if any, then executes the `<yes code>`. Otherwise executes the `<no code>` without changing the contents of the `<ans>` `tl` variable.

The `__bnvs_spec:nnnc...` is similar except that it uses what was stored for `<key>`, `<id>` and `<a>` or `<key>`, an empty `<id>` and `<a>`.

```

1073 \BNVS_new_conditional:cpnn { if_get:nnnc } #1 #2 #3 #4 { T, F, TF } {
1074     \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {

```

```

1075     \exp_args:Nnc \use:n { \exp_args:Nno \cs_set:cpn { \BNVS_1:cn { #4 } { t1 } } } { \_bnv
1076     \prg_return_true:
1077   } {
1078     \prg_return_false:
1079   }
1080 }

1081 \BNVS_new_conditional:cpnn { if_get:nvvc } #1 #2 #3 #4 { T, F, TF } {
1082   \BNVS_tl_use:nvv {
1083     \_bnvs_if_get:nnncTF { #1 }
1084   } { #2 } { #3 } { #4 } {
1085     \prg_return_true:
1086   } {
1087     \prg_return_false:
1088   }
1089 }

1090 \BNVS_new_conditional:cpnn { if_spec:nnnc } #1 #2 #3 #4 { T, F, TF } {
1091   \_bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #4 } {
1092     \prg_return_true:
1093   } {
1094     \tl_if_empty:nTF { #2 } {
1095       \prg_return_false:
1096     } {
1097       \_bnvs_if_get:nnncTF { #1 } { } { #3 } { #4 } {
1098         \prg_return_true:
1099       } {
1100         \prg_return_false:
1101       }
1102     }
1103   }
1104 }

```

$_bnvs_gprovide:TnnnnF$ $_bnvs_gprovide:TnvvnF$	$_bnvs_gprovide:TnnnnF \{ \langle yes \ code \rangle \} \{ \langle key \rangle \} \{ \langle id \rangle \} \{ \langle tag \rangle \} \{ \langle value \rangle \} \{ \langle no \ code \rangle \}$
--	---

Execute $\{ \langle no \ code \rangle \}$ exclusively when not in provide mode. Does nothing when something was set for the $\langle key \rangle$, $\langle id \rangle$! $\langle tag \rangle$ combination. Execute $\langle yes \ code \rangle$ before providing.

```

1105 \BNVS_new:cpn { gprovide:TnnnnF } #1 #2 #3 #4 #5 {
1106   \_bnvs_if:cTF { provide } {
1107     \_bnvs_is_gset:nnnF { #2 } { #3 } { #4 } {
1108       #1
1109     } \_bnvs_gset:nnnn { #2 } { #3 } { #4 } { #5 }
1110   }
1111 }
1112 }

1113 \BNVS_new:cpn { gprovide:TnvvnF } #1 #2 {
1114   \BNVS_tl_use:nvv { \_bnvs_gprovide:TnnnnF { #1 } { #2 } }
1115 }

```

6.10.3 Functions with cache

<code>_bnvs_gset_cache:nnnn</code>	<code>_bnvs_gset_cache:nnnn {<key>} {<id>} {<a>} {<value>}</code>
<code>_bnvs_gset_cache:(nnnv nvvn)</code>	

Wrapper over the functions above for $\langle key \rangle^*$ instead of $\langle key \rangle$.

```

1116 \BNVS_new:cpn { gset_cache:nnnn } #1 {
1117   \_bnvs_gset:nnnn { #1 * }
1118 }

1119 \BNVS_new:cpn { gset_cache:nvvn } #1 #2 {
1120   \BNVS_tl_use:nv {
1121     \BNVS_tl_use:nv {
1122       \_bnvs_gset_cache:nnnn { #1 }
1123     } { #2 }
1124   }
1125 }

1126 \BNVS_new:cpn { gset_cache:nnnv } #1 #2 #3 {
1127   \BNVS_tl_use:nv {
1128     \_bnvs_gset_cache:nnnn { #1 } { #2 } { #3 }
1129   }
1130 }
```

<code>_bnvs_if_get_cache:nnncTF</code>	<code>_bnvs_if_get_cache:nnncTF {<key>} {<id>} {<a>} {<ans>}</code>
	<code>{<yes code>} {<false code>}</code>

Wrapper over the functions above for $\langle key \rangle^*$ instead of $\langle key \rangle$.

```

1131 \BNVS_new_conditional:cpnn { if_get_cache:nnnc } #1 #2 #3 #4 { T, F, TF } {
1132   \_bnvs_if_get:nnncTF { #1 * } { #2 } { #3 } { #4 } {
1133     \prg_return_true:
1134   } {
1135     \prg_return_false:
1136   }
1137 }
```

<code>_bnvs_gunset_cache:nnn</code>	<code>_bnvs_gunset_cache:nnn {<key>} {<id>} {<a>}</code>
<code>_bnvs_gunset_cache:nvv</code>	<code>_bnvs_gunset_cache:nn {<id>} {<a>}</code>
<code>_bnvs_gunset_cache:nn</code>	<code>_bnvs_gunset_cache:n {<id>}</code>
<code>_bnvs_gunset_cache:n</code>	<code>_bnvs_gunset_cache:</code>

Wrapper over the functions above for $\langle key \rangle^*$ instead of $\langle key \rangle$.

```

1138 \BNVS_new:cpn { gunset_cache:nnn } #1 {
1139   \_bnvs_gunset:nnn { #1 * }
1140 }

1141 \BNVS_new:cpn { gunset_cache:nvv } #1 {
1142   \_bnvs_gunset:nvv { #1 * }
1143 }
```



```

1144 \BNVS_new:cpn { gunset_cache:nn } #1 #2 {
1145   \__bnvs_foreach_key_cache:n {
1146     \__bnvs_gunset:nnn { ##1 } { #1 } { #2 }
1147   }
1148 }

1149 \BNVS_new:cpn { gunset_cache:n } #1 {
1150   \__bnvs_foreach_IT:n {
1151     \tl_if_eq:nnT { #1 } { ##1 } {
1152       \__bnvs_gunset_cache:nn { ##1 } { ##2 }
1153     }
1154   }
1155 }

1156 \BNVS_new:cpn { gunset_cache: } {
1157   \__bnvs_foreach_IT:n {
1158     \__bnvs_gunset_cache:nn { ##1 } { ##2 }
1159   }
1160 }

```

6.11 Implicit value counter

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

`\g__bnvs_v_prop` $\langle key \rangle$ – $\langle value \rangle$ property list to store the contents or the named value counters. The keys are qualified dotted names $\langle frame id \rangle / \langle name \rangle . \langle c_1 \rangle \dots \langle c_j \rangle$ denoted as $\langle a \rangle$.

```

1161 \prop_new:N \g__bnvs_v_prop
(End of definition for \g__bnvs_v_prop.)

```

<code>__bnvs_v_gunset:nn</code>	<code>__bnvs_v_gunset:n {<id>} {<a>}</code>
<code>__bnvs_v_gunset:n</code>	<code>__bnvs_v_gunset:n {<id>}</code>
<code>__bnvs_v_gunset:</code>	<code>__bnvs_v_gunset:</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.

```

1162 \BNVS_new:cpn { v_gunset: } {
1163   \__bnvs_foreach_IT:n {
1164     \__bnvs_gunset:nnn v { ##1 } { ##2 }
1165     \__bnvs_gunset_cache:nnn v { ##1 } { ##2 }
1166   }
1167 }

```

<code>__bnvs_n_if_greset:nnnTF</code>	<code>__bnvs_n_if_greset:nnnTF {<id>} {<tag>} {<initial value>} {<yes</code>
<code>__bnvs_n_if_greset:(nnv vvn vvv)TF</code>	<code>code)} {<no code>}</code>
<code>__bnvs_if_greset_all:nnnTF</code>	<code>__bnvs_if_greset_all:nnnTF {<id>} {<tag>} {<initial value>} {<yes</code>
<code>__bnvs_if_greset_all:vvnTF</code>	<code>code)} {<no code>}</code>

If the $\langle id \rangle ! \langle tag \rangle$ combination is known, reset the value counter or the `n` counter to the given $\langle initial value \rangle$ and execute $\langle yes code \rangle$ otherwise $\langle no code \rangle$ is executed. The `..._all` variant also cleans the cached values and all the subvalues.

```

1168 \BNVS_new_conditional:cpnn { if_greset:nnnn } #1 #2 #3 #4 { T, F, TF } {
1169   \_bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
1170     \_bnvs_gunset_deep:nn { #2 } { #3 }
1171     \tl_if_empty:nTF { #4 } {
1172       \_bnvs_gunset:nnn { #1 } { #2 } { #3 }
1173     } {
1174       \_bnvs_gset:nnnn { #1 } { #2 } { #3 } { #4 }
1175     }
1176     \prg_return_true:
1177   } {
1178     \prg_return_false:
1179   }
1180 }

1181 \BNVS_new_conditional:cpnn { if_greset:nnnv } #1 #2 #3 #4 { T, F, TF } {
1182   \BNVS_tl_use:nv {
1183     \_bnvs_if_greset:nnTF { #1 } { #2 } { #3 }
1184   } { #4 } { \prg_return_true: } { \prg_return_false: }
1185 }

1186 \BNVS_new_conditional:cpnn { if_greset:nnvn } #1 #2 #3 #4 { T, F, TF } {
1187   \BNVS_tl_use:nv {
1188     \_bnvs_if_greset:nnnnTF { #1 } { #2 }
1189   } { #3 } { #4 }
1190   { \prg_return_true: } { \prg_return_false: }
1191 }

1192 \BNVS_new_conditional:cpnn { if_greset:nnvv } #1 #2 #3 #4 { T, F, TF } {
1193   \BNVS_tl_use:nvv {
1194     \_bnvs_if_greset:nnnnTF { #1 }
1195   } { #2 } { #3 } { #4 }
1196   { \prg_return_true: } { \prg_return_false: }
1197 }

1198 \BNVS_new_conditional:cpnn { n_if_greset:nnn } #1 #2 #3 { T, F, TF } {
1199   \_bnvs_is_gset:nnnTF n { #1 } { #2 } {
1200     \_bnvs_gunset:nnn n { #1 } { #2 }
1201     \tl_if_empty:nF { #3 } {
1202       \_bnvs_gset:nnnn n { #1 } { #2 } { #3 }
1203     }
1204     \prg_return_true:
1205   } {
1206     \prg_return_false:
1207   }
1208 }

1209 \BNVS_new_conditional:cpnn { n_if_greset:nnv } #1 #2 #3 { T, F, TF } {
1210   \BNVS_tl_use:nv { \_bnvs_n_if_greset:nnnTF { #1 } { #2 } } { #3 }
1211   { \prg_return_true: } { \prg_return_false: }
1212 }

1213 \BNVS_new_conditional:cpnn { n_if_greset:vvv } #1 #2 #3 { T, F, TF } {
1214   \BNVS_tl_use:nv {
1215     \BNVS_tl_use:Nv \_bnvs_n_if_greset:nnnTF { #1 }
1216   } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
1217 }

```

```

1218 \BNVS_new_conditional:cpnn { n_if_greset:vvv } #1 #2 #3 { T, F, TF } {
1219     \BNVS_tl_use:nvv {
1220         \BNVS_tl_use:Nv \__bnvs_n_if_greset:nnnTF { #1 }
1221     } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
1222 }

1223 \BNVS_new_conditional:cpnn { v_if_greset:nnn } #1 #2 #3 { T, F, TF } {
1224     \__bnvs_is_gset:nnnTF v { #1 } { #2 } {
1225         \__bnvs_gunset:nnn v { #1 } { #2 }
1226         \tl_if_empty:nF { #3 } {
1227             \__bnvs_gset:nnnn v { #1 } { #2 } { #3 }
1228         }
1229         \prg_return_true:
1230     } {
1231         \prg_return_false:
1232     }
1233 }

1234 \BNVS_new_conditional:cpnn { v_if_greset:nnv } #1 #2 #3 { T, F, TF } {
1235     \BNVS_tl_use:nv { \__bnvs_v_if_greset:nnnTF { #1 } { #2 } } { #3 }
1236     { \prg_return_true: } { \prg_return_false: }
1237 }

1238 \BNVS_new_conditional:cpnn { v_if_greset:vvv } #1 #2 #3 { T, F, TF } {
1239     \BNVS_tl_use:nv {
1240         \BNVS_tl_use:Nv \__bnvs_v_if_greset:nnnTF { #1 }
1241     } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
1242 }

1243 \BNVS_new_conditional:cpnn { v_if_greset:vvv } #1 #2 #3 { T, F, TF } {
1244     \BNVS_tl_use:nvv {
1245         \BNVS_tl_use:Nv \__bnvs_v_if_greset:nnnTF { #1 }
1246     } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
1247 }

1248 \BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
1249     \BNVS_tl_use:nc { \exp_args:No \quark_if_nil:nTF } { #1 } {
1250         \prg_return_true:
1251     } {
1252         \prg_return_false:
1253     }
1254 }

1255 \BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
1256     \BNVS_tl_use:nc { \exp_args:No \quark_if_no_value:nTF } { #1 } {
1257         \prg_return_true:
1258     } {
1259         \prg_return_false:
1260     }
1261 }

1262 \BNVS_new_conditional:cpnn { if_greset_all:nnn } #1 #2 #3 { T, F, TF } {
1263     \__bnvs_is_gset:nnTF { #1 } { #2 } {
1264         \BNVS_begin:

```

```

1265 \__bnvs_foreach_key_main:n {
1266   \__bnvs_if_get:nnncT { ##1 } { #1 } { #2 } { a } {
1267     \__bnvs_quark_if_nil:cT { a } {
1268       \__bnvs_if_get_cache:nnncTF { ##1 } { #1 } { #2 } { a } {
1269         \__bnvs_gset:nnnv { ##1 } { #1 } { #2 } { a }
1270       } {
1271         \__bnvs_gset:nnnn { ##1 } { #1 } { #2 } { 1 }
1272       }
1273     }
1274   }
1275 }
1276 \BNVS_end:
1277 \__bnvs_gunset_cache:nn { #1 } { #2 }
1278 \__bnvs_foreach_key_sub:n {
1279   \__bnvs_gunset:nnn { ##1 } { #1 } { #2 }
1280 }
1281 \prg_return_true:
1282 } {
1283 \prg_return_false:
1284 }
1285 }

1286 \BNVS_new_conditional:cpnn { if_greset_all:vvnn } #1 #2 #3 { T, F, TF } {
1287   \BNVS_tl_use:nv {
1288     \BNVS_tl_use:Nv \__bnvs_if_greset_all:nnnTF { #1 }
1289   } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
1290 }

```

6.12 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.

__bnvs_n_gunset: __bnvs_n_gunset:

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

```

1291 \BNVS_new:cpn { n_gunset: } {
1292   \__bnvs_foreach_IT:n {
1293     \__bnvs_gunset:nnn n { ##1 } { ##2 }
1294     \__bnvs_gunset_cache:nnn n { ##1 } { ##2 }
1295   }
1296 }

```

6.13 Regular expressions

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

```

1297 \regex_const:Nn \c__bnvs_short_regex {
1298   [[[:alpha:]]_][[:alnum:]]_*
1299 }

```

(End of definition for `\c__bnvs_short_regex`.)

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

```
1300 \regex_const:Nn \c__bnvs_path_regex {
1301   (? : \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ ) *
1302 }
```

(End of definition for `\c__bnvs_path_regex`.)

`\c__bnvs_A_index_Z_regex`

(End of definition for `\c__bnvs_A_index_Z_regex`.)

```
1303 \regex_const:Nn \c__bnvs_A_index_Z_regex { \A [-+]? \d+ \Z }
```

`\c__bnvs_A_reserved_Z_regex`

(End of definition for `\c__bnvs_A_reserved_Z_regex`.)

```
1304 \regex_const:Nn \c__bnvs_A_reserved_Z_regex {
1305   \A * [a-z] [_a-z0-9] * \Z
1306 }
```

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

(End of definition for `\c__bnvs_A_ref_Z_regex`.)

```
1307 \regex_const:Nn \c__bnvs_A_ref_Z_regex {

1: the frame id

1308   \A (?: ( \ur{c__bnvs_short_regex} )? ! )?

2: The short name.

1309   ( \ur{c__bnvs_short_regex} )

3: the path, if any.

1310   ( \ur{c__bnvs_path_regex} ) \Z
1311 }
```

`\c__bnvs_A_ISPn_Z_regex` Matches the whole string. Catch the ending `.n`.

(End of definition for `\c__bnvs_A_ISPn_Z_regex`.)

```
1312 \regex_const:Nn \c__bnvs_A_ISPn_Z_regex {

1: The full match,

2: the frame id

1313   \A (?: ( \ur{c__bnvs_short_regex} )? (!) )?

3: The short name
```

1314 (\ur{c__bnvs_short_regex})

4: The dotted path excluding the trailing .n.

1315 ((?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+) *?)

5: the last .n component if any.

1316 (\. n)? \Z
1317 }

\c__bnvs_A_SPn_Z_regex Matches the whole string. Catch the ending .n.

(End of definition for \c__bnvs_A_SPn_Z_regex.)

1318 \regex_const:Nn \c__bnvs_A_SPn_Z_regex {

1: The full match,

2: the frame $\langle id \rangle$

1319 \A (\ur{c__bnvs_short_regex} | [-+]? \d+)

3: The dotted path excluding the trailing .n.

1320 ((?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+) *?)

4: the last .n component if any.

1321 (\. n)? \Z
1322 }

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

1323 \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 12 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.

1324 \regex_const:Nn \c__bnvs_split_regex {
1325 \s* (? :

We start with ‘++’ instrussions⁴.

1 incrementation prefix

1326 \+\+

1.1: optional identifier: optional $\langle frame id \rangle$ followed by !

1327 (?: (\ur{c__bnvs_short_regex})? (!))?

1.2: $\langle short\ name \rangle$

1328 (\ur{c__bnvs_short_regex})

1.3: optionally followed by a dotted path with a heading dot

1329 (\ur{c__bnvs_path_regex})

2: without incement prefix

2.1: optional $\langle frame\ id \rangle$

1330 | (?: (\ur{c__bnvs_short_regex})? (!))?

2.2: $\langle short\ name \rangle$

1331 (\ur{c__bnvs_short_regex})

2.3: optionally followed by a dotted path

1332 (\ur{c__bnvs_path_regex})

We continue with other expressions

2.4: the $\langle ++n \rangle$ attribute

1333 (?: \.(+)\+n

2.5: the ‘+’ in ‘+=’ versus standalone ‘=’.

2.6: 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.

1334 | \s* (\+?)= \s* (\S+)

2.7: the post increment

1335 | (\+)\+

1336)?

1337) \s*

1338 }

”

(End of definition for `\c__bnvs_split_regex`.)

⁴At the same time an instruction and an expression... this is a synonym of exprection

6.14 beamer.cls interface

Work in progress.

```

1339 \RequirePackage{keyval}

1340 \define@key{beamerframe}{beanoves~id}[]{}
1341 \tl_set:Nx \l__bnvs_id_last_tl { #1 }
1342 }

1343 \AddToHook{env/beamer@frameslide/before}{
1344   \__bnvs_greset_call:
1345   \__bnvs_n_gunset:
1346   \__bnvs_v_gunset:
1347   \__bnvs_set_true:c { in_frame }
1348 }

1349 \AddToHook{env/beamer@frameslide/after}{
1350   \__bnvs_set_false:c { in_frame }
1351 }

```

6.15 Defining named slide ranges

```

\__bnvs_range_if_set:cccnTF \__bnvs_range_if_set:cccnTF {<core first>} {<core end>} {<core length>}
{<tl>} {<yes 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.

```

1352 \BNVS_new_conditional:cpnn { split_if_pop_left:c } #1 { T, F, TF } {
1353   \__bnvs_seq_pop_left:ccTF { split } { #1 } {
1354     \prg_return_true:
1355   } {
1356     \prg_return_false:
1357   }
1358 }

1359 \BNVS_new:cpn { split_if_pop_left:cTn } #1 #2 #3 {
1360   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS_split_F:n { #3 } }
1361 }

1362 \BNVS_new:cpn { split_if_pop_left_or:cT } #1 #2 {
1363   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS_split_F:n { #1 } }
1364 }

1365 \exp_args_generate:n { VVV }

1366 \BNVS_new_conditional:cpnn { range_if_set:cccn } #1 #2 #3 #4 { T, F, TF } {
1367   \BNVS_begin:
1368   \__bnvs_tl_clear:c { a }
1369   \__bnvs_tl_clear:c { b }
1370   \__bnvs_tl_clear:c { c }
1371   \__bnvs_if_regex_split:cnTF { colons } { #4 } {
1372     \__bnvs_seq_pop_left:ccT { split } { a } {

```

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

```

1373     \__bnvs_seq_pop_left:ccT { split } { b } {
1374     \__bnvs_tl_if_empty:cTF { b } {

```


This is a one colon range.

```
1375         \__bnvs_split_if_pop_left:cTF { b } {
```

b may contain the *⟨end⟩*.

```
1376         \__bnvs_seq_pop_left:ccT { split } { c } {
1377         \__bnvs_tl_if_empty:cTF { c } {
```

A :: was expected:

```
1378         \BNVS_error:n { Invalid-range-expression(1):~#4 }
1379     } {
1380     \int_compare:nNnT { \__bnvs_tl_count:c { c } } > { 1 } {
1381     \BNVS_error:n { Invalid-range-expression(2):~#4 }
1382     }
1383     \__bnvs_split_if_pop_left:cTF { c } {
```

\l__bnvs_c_tl may contain the *⟨length⟩*.

```
1384         \__bnvs_seq_if_empty:cF { split } {
1385         \BNVS_error:n { Invalid-range-expression(3):~#4 }
1386     }
1387     } {
1388     \BNVS_error:n { Internal-error }
1389     }
1390     }
1391     }
1392     } {
1393     }
1394     } {
```

This is a two colon range component.

```
1395     \int_compare:nNnT { \__bnvs_tl_count:c { b } } > { 1 } {
1396     \BNVS_error:n { Invalid-range-expression(4):~#4 }
1397     }
1398     \__bnvs_seq_pop_left:ccT { split } { c } {
```

c contains the *⟨length⟩*.

```
1399         \__bnvs_split_if_pop_left:cTF { b } {
1400         \__bnvs_tl_if_empty:cTF { b } {
1401         \__bnvs_seq_pop_left:cc { split } { b }
```

b may contain the *⟨end⟩*.

```
1402         \__bnvs_seq_if_empty:cF { split } {
1403         \BNVS_error:n { Invalid-range-expression(5):~#4 }
1404     }
1405     } {
1406     \BNVS_error:n { Invalid-range-expression(6):~#4 }
1407     }
1408     } {
1409     \__bnvs_tl_clear:c { b }
1410     }
1411     }
1412     }
1413     }
1414     }
```

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

```

1415 \cs_set:Npn \BNVS_range_if_set_cccnTF:w { }
1416 \__bnvs_tl_if_empty:cT { a } {
1417   \__bnvs_tl_if_empty:cT { b } {
1418     \__bnvs_tl_if_empty:cT { c } {
1419       \cs_set:Npn \BNVS_range_if_set_cccnTF:w {
1420         \BNVS_error:n { Invalid~range~expression(7):~#3 }
1421       }
1422     }
1423   }
1424 }
1425 \BNVS_range_if_set_cccnTF:w
1426 \cs_set:Npn \BNVS_range_if_set_cccnTF:w ##1 ##2 ##3 {
1427   \BNVS_end:
1428   \__bnvs_tl_set:cn { #1 } { ##1 }
1429   \__bnvs_tl_set:cn { #2 } { ##2 }
1430   \__bnvs_tl_set:cn { #3 } { ##3 }
1431 }
1432 \BNVS_exp_args:Nvvv \BNVS_range_if_set_cccnTF:w { a } { b } { c }
1433 \prg_return_true:
1434 } {
1435   \BNVS_end:
1436   \prg_return_false:
1437 }
1438 }

```

__bnvs_parse_IT: __bnvs_parse_IT:

Auxiliary function for __bnvs_parse:n and __bnvs_parse:nn below. The id, short and path variables are expected to be set. These are bottlenecks.

```

1439 \BNVS_new:cpn { parse_IT: } {
1440   \__bnvs_gprovide:TnvvvF {
1441     \__bnvs_gunset_cache:vv { id } { tag }
1442   } V { id } { tag } { 1 } {
1443     \__bnvs_if:cTF { reset } {
1444       \__bnvs_if:cT { reset_all } {
1445         \__bnvs_if_greset_all:vvvT { id } { tag } {} {}
1446       }
1447       \__bnvs_if:cF { only } {
1448         \__bnvs_foreach_IT:n {
1449           \__bnvs_tl_if_eq:cnT { id } { ##1 } {
1450             \exp_args:Ne \tl_if_in:nnT {
1451               .. \__bnvs_tl_use:c { tag } . } { .. ##2
1452             } {
1453               \__bnvs_gunset_cache:nn { ##1 } { ##2 }
1454             }
1455           }
1456         }
1457       }
1458     } \__bnvs_gunset:nvv v { id } { tag }
1459   } {
1460     \__bnvs_gunset:vv { id } { tag }
1461     \__bnvs_gset:nvvv V { id } { tag } { 1 }

```

```

1462     }
1463   }
1464 }

```

```

\__bnvs_parse_IT:n \__bnvs_parse_IT:n {<value>}
\__bnvs_n_parse_IT:n \__bnvs_n_parse_IT:n {<value>}

```

Auxiliary function for __bnvs_parse:n and __bnvs_parse:nn below. If *<value>* does not correspond to a range, the V key is used. The *_n* variant concerns the index counter. These are bottlenecks.

```

\__bnvs_range:nnnnn \__bnvs_range:nnnnn {<id>} {<a>} {<start>} {<end>} {<length>}
\__bnvs_range:nnvvv

```

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

```

1465 \BNVS_new:cpn { range:nnnnn } #1 #2 {
1466   \__bnvs_if:cTF { provide } {
1467     \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
1468       \use_none:nnn
1469     } {
1470       \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {
1471         \use_none:nnn
1472       } {
1473         \__bnvs_is_gset:nnnTF L { #1 } { #2 } {
1474           \use_none:nnn
1475         } {
1476           \__bnvs_do_range:nnnnn { #1 } { #2 }
1477         }
1478       }
1479     } {
1480     } {
1481       \__bnvs_do_range:nnnnn { #1 } { #2 }
1482     }
1483   }

1484 \BNVS_new:cpn { range:nnvvv } #1 #2 {
1485   \BNVS_tl_use:nnvv {
1486     \__bnvs_range:nnnnn { #1 } { #2 }
1487   }
1488 }

1489 \BNVS_new:cpn { do_range:nnnnn } #1 #2 #3 #4 #5 {
1490   \__bnvs_gunset:nn { #1 } { #2 }
1491   \tl_if_empty:nTF { #5 } {
1492     \tl_if_empty:nTF { #3 } {
1493       \tl_if_empty:nTF { #4 } {
1494         \BNVS_error:n { Not~a~range::~~#1!#2 }
1495       } {
1496         \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
1497         \__bnvs_gset:nnnn A { #1 } { #2 } { 1 }
1498         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1499       }
1500     } {
1501       \__bnvs_gset:nnnn A { #1 } { #2 } { #3 }
1502       \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }

```

```

1503     \tl_if_empty:nF { #4 } {
1504         \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
1505         \__bnvs_gset:nnnn L { #1 } { #2 } { \q_nil }
1506     }
1507 }
1508 } {
1509     \tl_if_empty:nTF { #3 } {
1510         \__bnvs_gset:nnnn L { #1 } { #2 } { #5 }
1511         \tl_if_empty:nF { #4 } {
1512             \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
1513             \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
1514             \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1515         }
1516     } {
1517         \__bnvs_gset:nnnn A { #1 } { #2 } { #3 }
1518         \__bnvs_gset:nnnn L { #1 } { #2 } { #5 }
1519         \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
1520         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1521     }
1522 }
1523 }

1524 \BNVS_new:cpn { range_IT:vvv } {
1525     \BNVS_tl_use:nvvv {
1526         \BNVS_tl_use:nv {
1527             \BNVS_tl_use:Nv \__bnvs_range:nnnnn { id }
1528         } { tag }
1529     }
1530 }

1531 \BNVS_new:cpn { parse_IT:n } #1 {
1532     \__bnvs_range_if_set:cccnTF { a } { b } { c } { #1 } {
1533         \__bnvs_range_IT:vvv { a } { b } { c }
1534     } {
1535         \__bnvs_gprovide:TnvvvF {
1536             \__bnvs_gunset:vv { id } { tag }
1537         } V { id } { tag } { #1 } {
1538             \__bnvs_if:cTF { reset } {
1539                 \__bnvs_if:cT { reset_all } {
1540                     \__bnvs_if_greset_all:vvvT { id } { tag } { #1 } { }
1541                 }
1542                 \__bnvs_if_greset:nvvvT v { id } { tag } { #1 } { }
1543             } {
1544                 \__bnvs_gunset:vv { id } { tag }
1545                 \__bnvs_gset:nvvv V { id } { tag } { #1 }
1546             }
1547         }
1548     }
1549 }

1550 \BNVS_new:cpn { n_parse_IT:n } #1 {
1551     \__bnvs_range_if_set:cccnTF { a } { b } { c } { #1 } {
1552         \BNVS_error:n { Unexpected~range:~#1 }
1553     } {
1554         \__bnvs_gprovide:TnvvvF {} n { id } { tag } { #1 } {

```

```

1555     \_bnvs_gset:nvvn      n { id } { tag } { #1 }
1556   }
1557 }
1558 }

```

```

\_bnvs_if_ref:nTF      \_bnvs_if_ref:nTF {<name>} {<yes code>} {<no code>}
\_bnvs_if_ref:nnTF     \_bnvs_if_ref:nnTF {<root>} {<relative>} {<yes code>} {<no code>}
\_bnvs_if_ref:vnTF     \_bnvs_if_ref_relative:nnTF {<root>} {<relative>} {<yes code>} {<no code>}
\_bnvs_if_ref_relative:nnTF

```

If *<name>* is a reference, put the frame id it defines into *id* the short name into *short*, the dotted path into *path*, without an eventual trailing *.n*, an eventual trailing *.n* into *n*, then execute *<yes code>*. The *n* *tl* variable is empty except when *<a>* ends with *.n*. Otherwise execute *<no code>*.

The second version calls the first one with *<name>* equals *<relative>* prepended with *<root>*.

The third version accepts integers as *<relative>* argument. It assumes that *<id>*, *<short>* and *<path>* are already set. The *<path>* and *<tag>* are updated accordingly

```

1559 \BNVS_new_conditional:cpnn { if_ref:n } #1 { T, F, TF } {
1560   \BNVS_begin:
1561     \_bnvs_match_if_once:NnTF \c\_bnvs_A_ISPn_Z_regex { #1 } {
1562       \_bnvs_if_match_pop_left:cTF { n } {
1563         \_bnvs_if_match_pop_left:cTF { id } {
1564           \_bnvs_if_match_pop_left:cTF { kri } {
1565             \_bnvs_if_match_pop_left:cTF { short } {
1566               \_bnvs_if_match_pop_left:cTF { path } {
1567                 \_bnvs_if_match_pop_left:cTF { n } {
1568                   \cs_set:Npn \BNVS_aux_if_ref_nTF:nnnn ##1 ##2 ##3 ##4 {
1569                     \BNVS_end:
1570                     \_bnvs_tl_set:cn { id } { ##1 }
1571                     \_bnvs_tl_set:cn { short } { ##2 }
1572                     \_bnvs_tl_set:cn { path } { ##3 }
1573                     \_bnvs_tl_set:cn { n } { ##4 }
1574                   }
1575                   \_bnvs_tl_if_empty:cTF { kri } {
1576                     \BNVS_exp_args:Nvvvv
1577                     \BNVS_aux_if_ref_nTF:nnnn
1578                     { id_last }
1579                   } {
1580                     \BNVS_exp_args:Nvvvv
1581                     \BNVS_aux_if_ref_nTF:nnnn
1582                     { id }
1583                   } { short } { path } { n }
1584                   \_bnvs_tl_if_empty:cTF n {
1585                     \_bnvs_set_false:c
1586                   } {
1587                     \_bnvs_set_true:c
1588                   } n
1589                   \_bnvs_tl_set:cv { tag } { path }
1590                   \_bnvs_tl_put_left:cv { tag } { short }
1591                   \_bnvs_tl_set:cv { id_last } { id }

```

```

1592         \prg_return_true:
1593     } {
1594         \BNVS_end_unreachable_return_false:n { A_ISPn_Z/n }
1595     }
1596 } {
1597     \BNVS_end_unreachable_return_false:n { A_ISPn_Z/path }
1598 }
1599 } {
1600     \BNVS_end_unreachable_return_false:n { A_ISPn_Z/short }
1601 }
1602 } {
1603     \BNVS_end_unreachable_return_false:n { A_ISPn_Z/kri }
1604 }
1605 } {
1606     \BNVS_end_unreachable_return_false:n { A_ISPn_Z/id }
1607 }
1608 } {
1609     \BNVS_end_unreachable_return_false:n { A_ISPn_Z/full_match }
1610 }
1611 } {
1612     \BNVS_end:
1613     \prg_return_false:
1614 }
1615 }

1616 \BNVS_new_conditional:cpnn { if_ref_relative:nn } #1 #2 { T, F, TF } {
1617     \BNVS_begin:
1618     \__bnvs_match_if_once:NnTF \c__bnvs_A_SPn_Z_regex { #2 } {
1619         \__bnvs_if_match_pop_left:cTF { n } {
1620             \__bnvs_if_match_pop_left:cTF { short } {
1621                 \__bnvs_if_match_pop_left:cTF { path } {
1622                     \__bnvs_if_match_pop_left:cTF { n } {
1623                         \cs_set:Npn \BNVS_aux_if_ref_nTF:nnn ##1 ##2 ##3 {
1624                             \BNVS_end:
1625                             \__bnvs_tl_put_right:cn { path } { . ##1 ##2 }
1626                             \__bnvs_tl_set:cn { n } { ##3 }
1627                         }
1628                         \BNVS_exp_args:Nvvv
1629                         \BNVS_aux_if_ref_nTF:nnn { short } { path } { n }
1630                         \__bnvs_tl_if_empty:cTF n {
1631                             \__bnvs_set_false:c
1632                         } {
1633                             \__bnvs_set_true:c
1634                         } n
1635                         \__bnvs_tl_set:cv { tag } { path }
1636                         \__bnvs_tl_put_left:cv { tag } { short }
1637                         \prg_return_true:
1638                     } {
1639                         \BNVS_end_unreachable_return_false:n { A_SPn_Z/n }
1640                     }
1641                 } {
1642                     \BNVS_end_unreachable_return_false:n { A_SPn_Z/path }
1643                 }
1644             } {
1645                 \BNVS_end_unreachable_return_false:n { A_SPn_Z/short }

```

```

1646     }
1647   } {
1648     \BNVS_end_unreachable_return_false:n { A_SPn_Z/full_match }
1649   }
1650 } {
1651   \BNVS_end:
1652   \prg_return_false:
1653 }
1654 }

1655 \BNVS_new_conditional:cpnn { if_ref:nn } #1 #2 { T, F, TF } {
1656   \tl_if_empty:nTF { #1 } {
1657     \__bnvs_if_ref:nTF { #2 } {
1658       \prg_return_true:
1659     } {
1660       \prg_return_false:
1661     }
1662   } {
1663     \__bnvs_if_ref_relative:nnTF { #1 } { #2 } {
1664       \prg_return_true:
1665     } {
1666       \prg_return_false:
1667     }
1668   }
1669 }

1670 \BNVS_new_conditional:cpnn { if_ref:vn } #1 #2 { T, F, TF } {
1671   \BNVS_tl_use:Nv \__bnvs_if_ref:nnTF { #1 } { #2 } {
1672     \prg_return_true:
1673   } {
1674     \prg_return_false:
1675   }
1676 }

```

__bnvs_keyval_parse:Nn	__bnvs_keyval_parse:Nn <i><function></i> { <i><definition></i> }
__bnvs_keyval_named:n	__bnvs_keyval_defined:n { <i><definition></i> }
__bnvs_keyval_defined:n	__bnvs_keyval_named:n { <i><name></i> }

Wrapper over \keyval_parse:nnn.

```

1677 \BNVS_new:cpn { keyval_parse:Nn } #1 {
1678   \keyval_parse:nnn { #1 } { \__bnvs_parse:nn }
1679 }

1680 \BNVS_new:cpn { keyval_named:n } {
1681   \keyval_parse:nnn { \__bnvs_parse_named:n } { \__bnvs_parse:nn }
1682 }

1683 \BNVS_new:cpn { keyval_defined:n } {
1684   \keyval_parse:nnn { \__bnvs_parse_defined:n } { \__bnvs_parse:nn }
1685 }

```

<code>__bnvs_parse:nn</code>	<code>__bnvs_parse_named:n</code>	<code>__bnvs_parse:nn {<name>} {<definition>}</code>
<code>__bnvs_parse_defined:n</code>	<code>__bnvs_parse_named:n {<name>}</code>	
	<code>__bnvs_parse_defined:n {<definition>}</code>	

Auxiliary functions called within a group by `\keyval:nnn`. `<name>` is the overlay set name, including eventually a dotted path or a frame identifier, `<definition>` is the corresponding definition. `__bnvs_parse_named:n` is `__bnvs_parse:nn` with the definition 1. It is used when parsing a `{{} }` `__bnvs_parse_defined:n` is `__bnvs_parse:nn` with the first available index as `n`. `__bnvs_parse:n` is just `__bnvs_parse_defined:n` when in list mode and `__bnvs_parse_named:n` otherwise.

`\l__bnvs_match_seq` Local storage for the match result.

(End of definition for `\l__bnvs_match_seq`.)

```

1686 \exp_args_generate:n { nne }
1687 \exp_args_generate:n { nnne }
1688 \BNVS_new:cpn { parse_named:n } #1 {
1689   \__bnvs_set_true:c { deep }
1690   \__bnvs_parse:nn { #1 } { 1 }
1691 }

1692 \BNVS_new:cpn { parse_defined:n } #1 {
1693   \__bnvs_tl_if_empty:cTF { root } {
1694     \BNVS_error:n { Unexpected~list~at~top-level. }
1695   } {
1696     \BNVS_begin:
1697     \cs_set:Npn \BNVS_aux_parse_defined_n: {
1698       \__bnvs_int_incr:c { i }
1699       \__bnvs_tl_set_eq:cc { a } { root }
1700       \__bnvs_tl_put_right:cn { a } { . }
1701       \BNVS_int_use:nc { \exp_args:NnV \__bnvs_tl_put_right:cn { a } } { i }
1702       \__bnvs_is_gset:nvT V { id } { a } { \BNVS_aux_parse_defined_n: }
1703     }
1704     \BNVS_aux_parse_defined_n:
1705     \BNVS_int_use:nv { \__bnvs_parse:nn } { i } { #1 }
1706     \BNVS_end:
1707   }
1708 }

1709 \cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
1710   \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
1711   { #4 }
1712 }
1713 \cs_new:Npn \BNVS_end_tl_set:cv #1 {
1714   \BNVS_tl_use:nv {
1715     \BNVS_end: \__bnvs_tl_set:cn { #1 }
1716   }
1717 }
```

Helper for `\keyval_parse:nnn` used in `\Beanoves` command. We have three requirements:

- raw beamer lists $X=A$ or $X=\{A\}$,
- key-value lists $X=\{\{A,B\}\}$,

- integer-value lists $X=[A,B]$.

```

1718 \regex_const:Nn \c__bnvs_one_suffix_regex { \A(.*)\.(?:1|first)\Z }
1719 \BNVS_new:cpn { parse:nn } #1 #2 {
1720   \BNVS_begin:

```

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

```

1721   \__bnvs_if_ref:vnTF { root } { #1 } {
1722     \regex_match:nnTF { \S } { #2 } {

```

This is not a $X=$.

```

1723     \peek_meaning:NNTF \BNVS_square_brackets:w {

```

This is a $X=[\dots]$ list, for an indexed list of range specification.

```

1724     \BNVS_begin:

```

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

```

1725     \cs_set:Npn \BNVS_square_brackets:w ##1 \s_stop {
1726       \__bnvs_tl_put_right:cn { root } { #1 }
1727       \__bnvs_keyval_defined:n { ##1 }
1728       \BNVS_end:
1729     }
1730   } {

```

Not a $X=[\dots]$.

```

1731     \peek_catcode:NNTF \c_group_begin_token {
1732       \__bnvs_if:cT n {
1733         \BNVS_warning:n { Ignoring~unexpected~suffix~.n::~#1 }
1734       }
1735       \cs_set:Npn \BNVS_aux_parse_nn:w ##1 ##2 \s_stop {
1736         \regex_match:nnT { \S } { ##2 } {
1737           \BNVS_warning:n { Ignoring~##2 }
1738         }
1739         \BNVS_use:c { X={{...}}:n } { ##1 }
1740       }
1741       \BNVS_aux_parse_nn:w
1742     } {

```

This is not a raw list, not a $X={{\dots}}$.

```

1743       \__bnvs_if:cTF n {
1744         \cs_set:Npn \BNVS_aux_parse_nn:n ##1 {
1745           \__bnvs_n_parse_IT:n { ##1 }
1746           \cs_set:Npn \BNVS_aux_parse_nn:n #####1 {
1747             \BNVS_warning:n { Ignored::~##1 }
1748           }
1749         }
1750         \cs_set:Npn \BNVS_aux_parse_nn:nn ##1 ##2 {
1751           \BNVS_warning:n { Ignored::~##1=##2 }
1752         }
1753         \keyval_parse:nnn
1754         { \BNVS_aux_parse_nn:n } { \BNVS_aux_parse_nn:nn } { #2 }
1755       } {
1756         \BNVS_use:c { X=...:n } { #2 }
1757       }

```

This is not a compound definition. Next character is not a group begin token.

```

1758         \use_none_delimit_by_s_stop:w
1759     }
1760 }
1761 #2 \s_stop
1762 } {

```

Empty value given: completely remove the reference.

```

1763     \__bnvs_if:cTF n {
1764         \__bnvs_gunset:nvv      n { id } { tag }
1765         \__bnvs_gunset_cache:nvv n
1766     } {
1767         \__bnvs_if:cT { deep } {
1768             \__bnvs_gunset_deep:vv { id } { tag }
1769         }
1770         \__bnvs_gunset:vv
1771     } { id } { tag }
1772 }
1773 } {
1774     \BNVS_error:n { Invalid-name:~#2 }
1775 }

```

We export \l__bnvs_id_last_tl:

```

1776     \__bnvs_match_if_once:NvT \c__bnvs_one_suffix_regex { tag } {
1777         \__bnvs_if_match_pop_left:cTF { a } {
1778             \__bnvs_if_match_pop_left:cTF { a } {
1779                 \cs_set:Npn \BNVS_aux_parse_nn: {
1780                     \__bnvs_gset:nvvn V { id } { a } { #2 }
1781                 }
1782                 \__bnvs_if_get:nvvcT V { id } { a } { b } {
1783                     \__bnvs_quark_if_nil:cF { b } {
1784                         \cs_set:Npn \BNVS_aux_parse_nn: { }
1785                     }
1786                 }
1787                 \BNVS_aux_parse_nn:
1788             } {
1789                 \BNVS_error:n { Unreachable~2 }
1790             }
1791         } {
1792             \BNVS_error:n { Unreachable~1 }
1793         }
1794     }

```

We export \l__bnvs_id_last_tl:

```

1795     \BNVS_end_tl_set:cv { id_last } { id_last }
1796 }

1797 \BNVS_new:cpn { X={{...}}:n } #1 {
1798     \__bnvs_gunset_deep:vv { id } { tag }

```

S

A $X={{\dots}}$ list, for a $\langle name \rangle$ – $\langle definition \rangle$ dictionary.

```

1799     \BNVS_begin:

```

Remove the elements that contain a =.

```

1800 \__bnvs_tl_put_right:cv { root } { tag }
1801 \__bnvs_tl_put_right:cn { root } { . }
1802 \__bnvs_keyval_named:n { #1 }
1803 \BNVS_end:
1804 }

1805 \BNVS_new:cpn { X=...:n } #1 {
1806   \BNVS_begin:
1807     \__bnvs_tl_clear:c { a }
1808     \__bnvs_seq_clear:c { a }
1809     \cs_set:Npn \BNVS_aux_parse_nn:n ##1 {
1810       \__bnvs_tl_set:cn { a } { ##1 }
1811       \cs_set:Npn \BNVS_aux_parse_nn:n #####1 {
1812         \__bnvs_seq_put_right:cn { a } { #####1 }
1813       }
1814     }
1815     \cs_set:Npn \BNVS_aux_parse_nn:nn ##1 ##2 {
1816       \BNVS_warning:n { Ignored:~##1=##2 }
1817     }
1818     \keyval_parse:nnn
1819     { \BNVS_aux_parse_nn:n } { \BNVS_aux_parse_nn:nn } { #1 }
1820     \__bnvs_tl_if_empty:cTF { a } {

```

Clean everything, whether in provide mode or not, including the .n counter.

```

1821 \__bnvs_gunset_deep:vv { id } { tag }
1822 \__bnvs_gunset:vv { id } { tag }
1823 } {

```

The first definition.

```

1824 \BNVS_tl_use:Nv \__bnvs_parse_IT:n { a }
1825 \__bnvs_seq_if_empty:cTF { a } {
1826   \__bnvs_gunset:nvv W { id } { tag }
1827   \__bnvs_gunset_cache:nvv W { id } { tag }
1828 } {
1829   \exp_args:Nnx \use:n {
1830     \__bnvs_gset:nvvn W { id } { tag }
1831   } { \seq_use:Nn \l__bnvs_a_seq { \q__bnvs } }
1832 }
1833 }
1834 \BNVS_end:
1835 }

```

```

\__bnvs_parse_IT[=...]:n \__bnvs_parse_n_IT[=...]:n \__bnvs_parse_IT[=...]:n {(definitions)}
\__bnvs_parse_n_IT[=...]:n {(definitions)}

```

Used by __bnvs_parse:nn . $\langle id \rangle$ and $\langle tag \rangle$ are set. Store the associate values. $\langle definitions \rangle$ is a comma separated list of $\langle definition \rangle$'s, either for ranges or values. The first $\langle definition \rangle$ is for the VWAZL keys, the other ones are for the W key.

```

1836 \BNVS_new:cpn { parse_IT[=...]:n } #1 {
1837   \BNVS_begin:

```

```

1838 \__bnvs_tl_clear:c { a }
1839 \__bnvs_seq_clear:c { a }
1840 \cs_set:Npn \BNVS_aux_parse_nn:n ##1 {
1841   \__bnvs_tl_set:cn { a } { ##1 }
1842   \cs_set:Npn \BNVS_aux_parse_nn:n ##1 {
1843     \__bnvs_seq_put_right:cn { a } { ##1 }
1844   }
1845 }
1846 \cs_set:Npn \BNVS_aux_parse_nn:nn ##1 ##2 {
1847   \BNVS_warning:n { Ignored:~##1=##2 }
1848 }
1849 \keyval_parse:nnn
1850 { \BNVS_aux_parse_nn:n } { \BNVS_aux_parse_nn:nn } { #1 }
1851 \__bnvs_tl_if_empty:cTF { a } {

```

Clean everything, whether in provide mode or not, including the .n counter.

```

1852   \__bnvs_gunset:vv { id } { tag }
1853 } {

```

The first definition.

```

1854   \BNVS_tl_use:Nv \__bnvs_parse_IT:n { a }
1855   \__bnvs_seq_if_empty:cTF { a } {
1856     \__bnvs_gunset:nvv W { id } { tag }
1857     \__bnvs_gunset_cache:nvv W { id } { tag }
1858   } {
1859     \__bnvs_seq_use:cn {
1860       \__bnvs_set:nvv W { id } { tag }
1861     } { a } { \q__bnvs }
1862   }
1863 }
1864 \BNVS_end:
1865 }

1866 \BNVS_new:cpn { parse_IT[.n=...]:n } #1 {
1867   \BNVS_begin:
1868   \__bnvs_tl_clear:c { a }
1869   \cs_set:Npn \BNVS_aux_parse_nn:n ##1 {
1870     \__bnvs_tl_set:cn { a } { ##1 }
1871     \cs_set:Npn \BNVS_aux_parse_nn:n ##1 {
1872       \BNVS_warning:n { Ignored:~##1 }
1873     }
1874   }
1875   \cs_set:Npn \BNVS_aux_parse_nn:nn ##1 ##2 {
1876     \BNVS_warning:n { Ignored:~##1=##2 }
1877   }
1878   \keyval_parse:nnn
1879   { \BNVS_aux_parse_nn:n } { \BNVS_aux_parse_nn:nn } { #1 }
1880   \__bnvs_tl_if_empty:cTF { a } {

```

Clean everything, whether in provide mode or not, including the .n counter.

```

1881   \__bnvs_gunset:nvv n { id } { tag }
1882   \__bnvs_gunset_cache:nvv n { id } { tag }
1883 } {

```

The first definition.

```

1884 \BNVS_tl_use:Nv \__bnvs_n_parse_IT:n { a }
1885 }
1886 \BNVS_end:
1887 }

1888 \BNVS_new:cpn { parse_prepare:N } #1 {
1889 \tl_set:Nx #1 #1
1890 \__bnvs_set_false:c { parse }
1891 \bool_do_until:Nn \l__bnvs_parse_bool {
1892 \tl_if_in:NnTF #1 {%---[
1893 ]} {
1894 \regex_replace_all:nnNF { \[ ([^\]%---)
1895 ]*%---[(
1896 ) \] } { \c{BNVS_square_brackets:w} { \1 } } #1 {
1897 \__bnvs_set_true:c { parse }
1898 }
1899 } {
1900 \__bnvs_set_true:c { parse }
1901 }
1902 }
1903 \tl_if_in:NnTF #1 {%---[
1904 ]} {
1905 \BNVS_error:n { Unbalanced~%---[
1906 ]}
1907 } {
1908 \tl_if_in:NnT #1 { [%---]
1909 } {
1910 \BNVS_error:n { Unbalanced~[ %---]
1911 }
1912 }
1913 }
1914 }

```

\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.

```

1915 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1916 \BNVS_tl_use:nv {
1917 \BNVS_end:
1918 \__bnvs_tl_put_right:cn { #1 }
1919 } { #2 }
1920 }
1921 \cs_new:Npn \BNVS_end_gset:nnnv #1 #2 #3 {
1922 \BNVS_tl_use:nv {
1923 \BNVS_end:
1924 \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
1925 }
1926 }
1927 \NewDocumentCommand \Beanoves { sm } {
1928 \__bnvs_set_false:c { reset }
1929 \__bnvs_set_false:c { reset_all }
1930 \__bnvs_set_false:c { only }

```

```
1931 \tl_if_empty:NTF \@currentenvir {
```

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

```
1932 \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
1933 } {
1934 \tl_if_eq:NnT \@currentenvir { document } {
```

At the top level, clear everything.

```
1935 \__bnvs_gunset:
1936 }
1937 \BNVS_begin:
1938 \__bnvs_tl_clear:c { root }
1939 \__bnvs_int_zero:c { i }
1940 \__bnvs_tl_set:cn { a } { #2 }
1941 \tl_if_eq:NnT \@currentenvir { document } {
```

At the top level, use the global definitions.

```
1942 \seq_if_empty:NF \g__bnvs_def_seq {
1943 \__bnvs_tl_put_left:cx { a } {
1944 \seq_use:Nn \g__bnvs_def_seq , ,
1945 }
1946 }
1947 }
1948 \__bnvs_parse_prepare:N \l__bnvs_a_tl
1949 \IfBooleanTF {#1} {
1950 \__bnvs_provide_on:
1951 } {
1952 \__bnvs_provide_off:
1953 }
1954 \BNVS_tl_use:Nv \__bnvs_keyval_named:n { a }
1955 \BNVS_end_tl_set:cv { id_last } { id_last }
1956 \ignorespaces
1957 }
1958 }
```

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

```
1959 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}
```

6.16 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.

```

1960 \makeatletter
1961 \cs_set:Npn \__bnvs@frame < #1 > {
1962   \BNVS_begin:
1963   \__bnvs_tl_clear:c { ans }
1964   \__bnvs_scan:nNc { #1 } \__bnvs_if_resolve:ncTF { ans }
1965   \BNVS_set:cpn { :n } ##1 { \BNVS_end: \BNVS_saved@frame < ##1 > }
1966   \BNVS_tl_use:cv { :n } { ans }
1967 }

1968 \cs_set:Npn \__bnvs@masterdecode #1 {
1969   \BNVS_begin:
1970   \__bnvs_tl_clear:c { ans }
1971   \__bnvs_scan:nNc { #1 } \__bnvs_if_resolve_queries:ncTF { ans }
1972   \BNVS_tl_use:nv {
1973     \BNVS_end:
1974     \BNVS_saved@masterdecode
1975   } { ans }
1976 }

1977 \cs_new:Npn \BeanovesOff {
1978   \cs_set_eq:NN \beamer@frame \BNVS_saved@frame
1979   \cs_set_eq:NN \beamer@masterdecode \BNVS_saved@masterdecode
1980 }

1981 \cs_new:Npn \BeanovesOn {
1982   \cs_set_eq:NN \beamer@frame \__bnvs@frame
1983   \cs_set_eq:NN \beamer@masterdecode \__bnvs@masterdecode
1984 }

1985 \AddToHook{begindocument/before}{
1986   \cs_if_exist:NTF \beamer@frame {
1987     \cs_set_eq:NN \BNVS_saved@frame \beamer@frame
1988     \cs_set_eq:NN \BNVS_saved@masterdecode \beamer@masterdecode
1989   } {
1990     \cs_set:Npn \BNVS_saved@frame < #1 > {
1991       \BNVS_error:n {Missing~package~beamer}
1992     }
1993     \cs_set:Npn \BNVS_saved@masterdecode < #1 > {
1994       \BNVS_error:n {Missing~package~beamer}
1995     }
1996   }
1997   \BeanovesOn
1998 }
1999 \makeatother

```

_bnvs_scan:nNc _bnvs_scan:nNc {<overlay query>} <resolve> {<ans>}

Scan the <overlay query> argument and feed the <ans> tl variable replacing ?(...) instructions by their static counterpart with help from the <resolve> function, which is _bnvs_if_resolve:nCTF. 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.)

_bnvs_scan:nNcTF _bnvs_scan:nNcTF {<overlay query>} <resolve> {<ans>} {<yes code>} {<no code>}

Next are helpers.

_bnvs_scan_for_query_then_end_return: _bnvs_scan_for_query_then_end_return:

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.

```

2000 \BNVS_new:cpn { scan_for_query_then_end_return: } {
2001   \_bnvs_seq_pop_left:ccTF { token } { token } {
2002     \_bnvs_tl_if_eq:cnTF { token } { ? } {
2003       \_bnvs_scan_require_open_end_return:
2004     } {
2005       \_bnvs_tl_put_right:cv { ans } { token }
2006       \_bnvs_scan_for_query_then_end_return:
2007     }
2008   } {
2009     \_bnvs_scan_end_return_true:
2010   }
2011 }
```

_bnvs_scan_require_open_end_return: _bnvs_scan_require_open_end_return:

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

```

2012 \BNVS_new:cpn { scan_require_open_end_return: } {
```


Get next token.

```

2013  \__bnvs_seq_pop_left:ccTF { token } { token } {
2014    \str_if_eq:VnTF \l__bnvs_token_t1 { ( % )
2015  } {

```

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

```

2016    \__bnvs_int_set:cn { } { 1 }

```

Record the forthcomming content in the \l__bnvs_query_t1 variable, up to the next balancing ‘)’.

```

2017    \__bnvs_t1_clear:c { query }
2018    \__bnvs_scan_require_close_and_return:
2019  } {

```

Ignore this token and loop.

```

2020    \__bnvs_scan_require_open_end_return:
2021  }
2022 } {

```

Get next token.

End reached but no opening parenthesis found, raise. As this is a standalone raising ?, this is not a fatal error.

```

2023    \BNVS_error:x {Missing~'('%---)
2024    ~after~a~? }
2025    \__bnvs_scan_end_return_true:
2026  }
2027 }

```

```

\__bnvs_scan_require_close_and_return: \__bnvs_scan_require_close_and_return:

```

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

```

2028 \BNVS_new:cpn { scan_require_close_and_return: } {

```

Get next token.

```

2029  \__bnvs_seq_pop_left:ccTF { token } { token } {
2030    \str_case:VnF \l__bnvs_token_t1 {
2031      { ( %---)
2032    } {

```

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

```

2033    \__bnvs_int_incr:c { }
2034    \__bnvs_t1_put_right:cv { query } { token }
2035    \__bnvs_scan_require_close_and_return:
2036  }
2037  { %(---
2038    )
2039  } {

```

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

```

2040    \__bnvs_int_decr:c { }
2041    \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 `'?'`.

```
2042     \__bnvs_scan_handle_query_then_end_return:
2043     } {
```

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

```
2044         \__bnvs_tl_put_right:cv { query } { token }
2045         \__bnvs_scan_require_close_and_return:
2046     }
2047     }
2048     } {
```

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

```
2049         \__bnvs_tl_put_right:cv { query } { token }
2050         \__bnvs_scan_require_close_and_return:
2051     }
2052     } {
```

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 `)`.

```
2053     \BNVS_error:x { Missing~%(---
2054     `)' }
2055     \__bnvs_tl_put_right:cx { query } {
2056         \prg_replicate:nn { \l__bnvs_int } {%(---
2057         )}
2058     }
2059     \__bnvs_scan_end_return_true:
2060 }
2061 }
```

```
2062 \BNVS_new_conditional:cpnn { scan:nNc } #1 #2 #3 { T, F, TF } {
2063     \BNVS_begin:
2064     \BNVS_set:cpn { error:x } ##1 {
2065         \msg_error:nnx { beanoves } { :n }
2066         { \tl_to_str:n { #1 }:-##1}
2067     }
2068     \__bnvs_tl_set:cn { scan } { #1 }
2069     \__bnvs_tl_clear:c { ans }
2070     \__bnvs_seq_clear:c { token }
```

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

```
2071     \regex_split:nnN { } { #1 } \l__bnvs_token_seq
```

Run the top level loop to scan for a `'?'` character: Error recovery is missing.

```
2072     \BNVS_set:cpn { scan_handle_query_then_end_return: } {
2073         \BNVS_tl_use:Nv #2 { query } { ans } {
2074             \__bnvs_scan_for_query_then_end_return:
2075         } {
2076             \BNVS_end_tl_put_right:cv { #3 } { ans }
```

Stop on the first error.

```

2077     \prg_return_false:
2078   }
2079 }
2080 \BNVS_set:cpn { scan_end_return_true: } {
2081   \BNVS_end_tl_put_right:cv { #3 } { ans }
2082   \prg_return_true:
2083 }
2084 \BNVS_set:cpn { scan_end_return_false: } {
2085   \BNVS_end_tl_put_right:cv { #3 } { ans }
2086   \prg_return_false:
2087 }
2088 \__bnvs_scan_for_query_then_end_return:
2089 }
2090 \BNVS_new:cpn { scan:nNc } #1 #2 #3 {
2091   \BNVS_use:c { scan:nNcTF } { #1 } #2 { #3 } {} {}
2092 }

```

6.17 Resolution

Given a name, a frame id and a dotted 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 neither.

```

\__bnvs_if_TIP:cccTF \__bnvs_if_TIP:cccTF {<name>} {<id>} {<path>} {<yes code>} {<no code>}

```

Auxiliary function. On input, the *<name>* tl variable contains a set name whereas the *<id>* tl variable contains a frame id. If *<name>* tl variable contents is a recorded set, on return, *<a>* tl variable contains the resolved name, *<id>* tl variable contains the used frame id, *<path>* seq variable is prepended with new dotted path components, *<n>* tl variable is empty on return iff there is a trailing .n, *<yes code>* is executed, otherwise variables are left untouched and *<no code>* is executed.

```

2093 \BNVS_new_conditional:cpnn { if_TIP:ccc } #1 #2 #3 { T, F, TF } {
2094   \BNVS_begin:
2095   \__bnvs_match_if_once:NvTF \c__bnvs_A_ref_Z_regex { #1 } {

```

This is a correct *<a>*, update the path sequence accordingly.

```

2096   \__bnvs_if_match_pop_TIP:cccTF { #1 } { #2 } { #3 } {
2097     \__bnvs_export_TIP:cccN { #1 } { #2 } { #3 }
2098     \BNVS_end:
2099     \prg_return_true:
2100   } {
2101     \BNVS_end:
2102     \prg_return_false:
2103   }
2104 } {
2105   \BNVS_end:
2106   \prg_return_false:
2107 }
2108 }
2109 \quark_new:N \q__bnvs

```

```

2110 \tl_new:N \l__bnvs_export_TIP_cccN_tl
2111 \BNVS_new:cpn { export_TIP:cccN } #1 #2 #3 #4 {
2112   \cs_set:Npn \BNVS_export_TIP_cccN:w ##1 ##2 ##3 {
2113     #4
2114     \__bnvs_tl_set:cn { #1 } { ##1 }
2115     \__bnvs_tl_set:cn { #2 } { ##2 }
2116     \__bnvs_tl_set:cn { export_TIP_cccN } { ##3 }
2117   }
2118   \__bnvs_tl_set:cx { export_TIP_cccN }
2119   { \__bnvs_seq_use:cn { #1 } { \q__bnvs } }
2120   \BNVS_tl_use:nvv {
2121     \BNVS_tl_use:Nv \BNVS_export_TIP_cccN:w { #1 }
2122   } { #2 } { export_TIP_cccN }
2123   \BNVS_tl_use:nv {
2124     \__bnvs_seq_set_split:cnn { #3 } { \q__bnvs }
2125   } { export_TIP_cccN }
2126   \__bnvs_seq_remove_all:cn { #3 } { }
2127 }

2128 \tl_new:N \l__bnvs_if_match_export_ISPn_cccc_tl
2129 \BNVS_new:cpn { if_match_export_ISPn:ccccN } #1 #2 #3 #4 #5 {
2130   \cs_set:Npn \BNVS_if_match_export_ISPn_ccccN:w ##1 ##2 ##3 ##4 {
2131     #5
2132     \__bnvs_tl_set:cn { #1 } { ##1 }
2133     \__bnvs_tl_set:cn { #2 } { ##2 }
2134     \__bnvs_tl_set:cn { #3 } { ##3 }
2135     \__bnvs_tl_set:cn { #4 } { ##4 }
2136   }
2137   \__bnvs_tl_set:cx { if_match_export_ISPn_cccc }
2138   { \__bnvs_seq_use:cn { #1 } { \q__bnvs } }
2139   \BNVS_tl_use:nvvv {
2140     \BNVS_tl_use:Nv \BNVS_if_match_export_ISPn_ccccN:w { #1 }
2141   } { #2 } { if_match_export_ISPn_cccc } { #4 }
2142   \BNVS_tl_use:nv {
2143     \__bnvs_seq_set_split:cnn { #3 } { \q__bnvs }
2144   } { if_match_export_ISPn_cccc }
2145   \__bnvs_seq_remove_all:cn { #3 } { }
2146 }

2147 \BNVS_new_conditional:cpnn { if_match_pop_ISPn:cccc } #1 #2 #3 #4 { TF } {
2148   \BNVS_begin:
2149   \__bnvs_if_match_pop_left:cTF { #1 } {
2150     \__bnvs_if_match_pop_left:cTF { #1 } {
2151       \__bnvs_if_match_pop_left:cTF { #2 } {
2152         \__bnvs_if_match_pop_left:cTF { #3 } {
2153           \__bnvs_seq_set_split:cnv { #3 } { . } { #3 }
2154           \__bnvs_seq_remove_all:cn { #3 } { }
2155           \__bnvs_if_match_pop_left:cTF { #4 } {
2156             \__bnvs_if_match_export_ISPn:ccccN { #1 } { #2 } { #3 } { #4 }
2157             \BNVS_end:
2158             \prg_return_true:
2159           } {
2160             \BNVS_end_return_false:
2161           }

```

```

2162     } {
2163         \BNVS_end_return_false:
2164     }
2165 } {
2166     \BNVS_end_return_false:
2167 }
2168 } {
2169     \BNVS_end_return_false:
2170 }
2171 } {
2172     \BNVS_end_return_false:
2173 }
2174 }

```

Local variables:

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

```

2175 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
2176     \__bnvs_seq_if_empty:cF { #2 } {
2177         \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
2178             \__bnvs_seq_use:cn { #1 } { \q__bnvs }
2179             \exp_not:n { \q__bnvs }
2180             \__bnvs_seq_use:cn { #2 } { \q__bnvs }
2181         }
2182         \__bnvs_seq_remove_all:cn { #1 } { }
2183     }
2184 }

```

6.18 Evaluation bricks

We start by helpers.

__bnvs_round:N	__bnvs_round:N <i><tl variable></i>
__bnvs_round:c	__bnvs_round:c <i><{tl core name}></i>

Replaces the variable content with its rounded floating point evaluation.

```

2185 \BNVS_new:cpn { round:N } #1 {
2186     \tl_if_empty:NTF #1 {
2187         \tl_set:Nn #1 { 0 }
2188     } {
2189         \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
2190     }
2191 }

2192 \BNVS_new:cpn { round:c } {
2193     \BNVS_tl_use:Nc \__bnvs_round:N
2194 }

```

```

__bnvs_if_assign_value:nnnTF      __bnvs_if_assign_value:nnnTF {⟨id⟩} {⟨a⟩} ⟨value⟩ {⟨yes code⟩} {⟨no
__bnvs_if_assign_value:(nnv|vvv)TF code⟩}

```

```

2195 \BNVS_new_conditional:cpnn { if_assign_value:nnn } #1 #2 #3 { T, F, TF } {
2196   \BNVS_begin:
2197   __bnvs_if_resolve:ncTF { #3 } { a } {
2198     __bnvs_gunset:nn { #1 } { #2 }
2199     \tl_map_inline:nn { V { V * } v } {
2200       __bnvs_gset:nnnv { ##1 } { #1 } { #2 } { a }
2201     }
2202     \BNVS_end:
2203     \prg_return_true:
2204   } {
2205     \BNVS_end:
2206     \prg_return_false:
2207   }
2208 }

2209 \BNVS_new_conditional:cpnn { if_assign_value:nnv } #1 #2 #3 { T, F, TF } {
2210   \BNVS_tl_use:nv {
2211     __bnvs_if_assign_value:nnnTF { #1 } { #2 }
2212   } { #3 } {
2213     \prg_return_true:
2214   } {
2215     \prg_return_false:
2216   }
2217 }

2218 \BNVS_new_conditional:cpnn { if_assign_value:vvv } #1 #2 #3 { T, F, TF } {
2219   \BNVS_tl_use:nvv {
2220     \BNVS_tl_use:Nv \bnvs_if_assign_value:nnnTF { #1 }
2221   } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
2222 }

```

```

__bnvs_if_resolve_V:nncTF      __bnvs_if_resolve_V:nncTF {⟨id⟩} {⟨a⟩} ⟨ans⟩ {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_resolve_V:nvcTF      __bnvs_if_append_V:nncTF {⟨id⟩} {⟨a⟩} ⟨ans⟩ {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_append_V:nncTF
__bnvs_if_append_V:(nxc|nvc)TF

```

Resolve the content of the $\langle id \rangle$, $\langle a \rangle$ value counter into the $\langle ans \rangle$ `tl` variable or append this value to the right of this variable. Execute $\langle yes\ code \rangle$ when there is a $\langle value \rangle$, $\langle no\ code \rangle$ otherwise. Inside the $\langle no\ code \rangle$ branch, the content of the $\langle ans \rangle$ `tl` variable is undefined. Implementation detail: in $\langle ans \rangle$ we return the first in the cache for subkey V and in the general prop for subkey V (once resolved). 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 local group is created while appending but not while resolving.

```

2223 \BNVS_new:cpn { if_resolve_V_return:nnncT } #1 #2 #3 #4 #5 {
2224   __bnvs_tl_if_empty:cTF { #4 } {
2225     \prg_return_false:
2226   } {
2227     __bnvs_gset_cache:nnnv V { #2 } { #3 } { #4 }
2228     #5

```

```

2229     \prg_return_true:
2230   }
2231 }

2232 \makeatletter
2233 \BNVS_new_conditional:cpnn { if_resolve_V:nnc } #1 #2 #3 { T, F, TF } {
2234   \__bnvs_if_get_cache:nnncTF V { #1 } { #2 } { #3 } {
2235     \prg_return_true:
2236   } {
2237     \__bnvs_if_get:nnncTF V { #1 } { #2 } { #3 } {
2238       \__bnvs_quark_if_nil:cTF { #3 } {

```

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

```

2239     \__bnvs_gset:nnnn V { #1 } { #2 } { \q_no_value }
2240     \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2241       \__bnvs_if_resolve_V_return:nnncT A { #1 } { #2 } { #3 } {
2242         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2243       }
2244     } {
2245       \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2246         \__bnvs_if_resolve_V_return:nnncT Z { #1 } { #2 } { #3 } {
2247           \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2248         }
2249       } {
2250         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2251         \prg_return_false:
2252       }
2253     }
2254   } {

```

Possible recursive call.

```

2255     \__bnvs_quark_if_no_value:cTF { #3 } {
2256       \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2257       \__bnvs_gset:nnnn V { #1 } { #2 } { 1 }
2258       \__bnvs_tl_set:cn { #3 } { 1 }
2259       \prg_return_true:
2260     } {
2261       \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2262         \__bnvs_if_resolve_V_return:nnncT V { #1 } { #2 } { #3 } {
2263           \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2264         }
2265       } {
2266         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2267         \prg_return_false:
2268       }
2269     }
2270   }
2271 } {
2272   \tl_if_eq:nnTF { #2 } { pauses } {
2273     \cs_if_exist:NTF \c@beamerpauses {
2274       \exp_args:Nnx \__bnvs_tl_set:cn { #3 } { \the\c@beamerpauses }
2275       \__bnvs_gunset:nn { #1 } { #2 }
2276       \prg_return_true:
2277     } {

```

```

2278         \prg_return_false:
2279     }
2280 } {
2281     \tl_if_eq:nnTF { #2 } { slideinframe } {
2282         \cs_if_exist:NTF \beamer@slideinframe {
2283             \exp_args:Nnx \__bnvs_tl_set:cn { #3 } { \beamer@slideinframe }
2284             \__bnvs_gunset:nn { #1 } { #2 }
2285             \prg_return_true:
2286         } {
2287             \prg_return_false:
2288         }
2289     } {
2290         \prg_return_false:
2291     }
2292 }
2293 }
2294 }
2295 }
2296 \makeatother
2297 \BNVS_new_conditional_vvc:cn { if_resolve_V } { T, F, TF }
2298 \BNVS_new:cpn { end_put_right:vc } #1 #2 {
2299     \BNVS_tl_use:nv {
2300         \BNVS_end:
2301         \__bnvs_tl_put_right:cn { #2 }
2302     } { #1 }
2303 }
2304 \BNVS_new_conditional:cpnn { if_append_V:nnc } #1 #2 #3 { T, F, TF } {
2305     \BNVS_begin:
2306     \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
2307         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2308         \prg_return_true:
2309     } {
2310         \BNVS_end:
2311         \prg_return_false:
2312     }
2313 }
2314 \BNVS_new_conditional_vvc:cn { if_append_V } { T, F, TF }

```

$\underline{\underline{\backslash_bnvs_if_resolve_A:nncTF}}$	$\backslash_bnvs_if_resolve_A:nncTF \{ \langle id \rangle \} \{ \langle a \rangle \} \{ \langle ans \rangle \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$
$\underline{\underline{\backslash_bnvs_if_append_A:nncTF}}$	$\backslash_bnvs_if_append_A:nncTF \{ \langle id \rangle \} \{ \langle a \rangle \} \{ \langle ans \rangle \} \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$

Resolve the first index of the $\langle a \rangle$ slide range into the $\langle ans \rangle$ tl variable or append the first index of the $\langle a \rangle$ slide range to the $\langle ans \rangle$ tl variable. If no resolution occurs the content of the $\langle ans \rangle$ tl variable 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.

```

2315 \BNVS_new_conditional:cpnn { if_resolve_A:nnc } #1 #2 #3 { T, F, TF } {
2316     \__bnvs_if_get_cache:nncTF A { #1 } { #2 } { #3 } {
2317         \prg_return_true:
2318     } {
2319         \__bnvs_if_get:nncTF A { #1 } { #2 } { #3 } {
2320             \__bnvs_quark_if_nil:cTF { #3 } {
2321                 \__bnvs_gset:nncn A { #1 } { #2 } { \q_no_value }

```


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

```

2322     \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2323         \__bnvs_tl_put_right:cn { #3 } { - }
2324     \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 } {
2325         \__bnvs_tl_put_right:cn { #3 } { + 1 }
2326         \__bnvs_round:c { #3 }
2327         \__bnvs_tl_if_empty:cTF { #3 } {
2328             \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2329             \prg_return_false:
2330         } {
2331             \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2332             \__bnvs_gset_cache:nnnv A { #1 } { #2 } { #3 }
2333             \prg_return_true:
2334         }
2335     } {
2336         \BNVS_error:n {
2337 Unavailable~length~for~#1~(\token_to_str:N\__bnvs_if_resolve_A:nncTF/2) }
2338         \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2339         \prg_return_false:
2340     }
2341 } {
2342     \BNVS_error:n {
2343 Unavailable~last~for~#1~(\token_to_str:N\__bnvs_if_resolve_A:nncTF/1) }
2344     \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2345     \prg_return_false:
2346 }
2347 } {
2348     \__bnvs_quark_if_no_value:cTF { a } {
2349         \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2350         \__bnvs_gset:nnnn A { #1 } { #2 } { 1 }
2351         \__bnvs_tl_set:cn { #3 } { 1 }
2352         \prg_return_true:
2353     } {
2354         \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2355             \__bnvs_gset:nnnv A { #1 } { #2 } { #3 }
2356             \prg_return_true:
2357         } {
2358             \prg_return_false:
2359         }
2360     }
2361 }
2362 } {
2363     \prg_return_false:
2364 }
2365 }
2366 }

2367 \BNVS_new_conditional:cpnn { if_append_A:nnc } #1 #2 #3 { T, F, TF } {
2368     \BNVS_begin:
2369     \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2370         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2371         \prg_return_true:
2372     } {

```

```

2373     \BNVS_end:
2374     \prg_return_false:
2375 }
2376 }

```

```

\__bnvs_if_resolve_Z:nncTF \__bnvs_if_resolve_Z:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append_Z:nncTF \__bnvs_if_append_Z:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no code>}

```

Resolve the last index of the $\langle id \rangle!$ $\langle a \rangle$ range into or to the right of the $\langle ans \rangle$ t1 variable, when possible. Execute $\langle yes\ code \rangle$ when a last index was given, $\langle no\ code \rangle$ otherwise.

```

2377 \BNVS_new_conditional:cpnn { if_resolve_Z:nnc } #1 #2 #3 { T, F, TF } {
2378   \__bnvs_if_get_cache:nnncTF Z { #1 } { #2 } { #3 } {
2379     \prg_return_true:
2380   } {
2381     \__bnvs_if_get:nnncTF Z { #1 } { #2 } { #3 } {
2382       \__bnvs_quark_if_nil:cTF { #3 } {
2383         \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_no_value }

```

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

```

2384   \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2385     \__bnvs_tl_put_right:cn { #3 } { + }
2386     \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 } {
2387       \__bnvs_tl_put_right:cn { #3 } { - 1 }
2388       \__bnvs_round:c { #3 }
2389       \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
2390       \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
2391       \prg_return_true:
2392     } {
2393       \BNVS_error:x {
2394         Unavailable~last~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
2395       \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
2396       \prg_return_false:
2397     }
2398   } {
2399     \BNVS_error:x {
2400       Unavailable~first~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
2401     \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
2402     \prg_return_false:
2403   }
2404 } {
2405   \__bnvs_quark_if_no_value:cTF { #3 } {
2406     \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2407     \__bnvs_tl_set:cn { #3 } { 1 }
2408     \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
2409     \prg_return_true:
2410   } {
2411     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2412       \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
2413       \prg_return_true:
2414     } {
2415       \prg_return_false:
2416     }
2417   }

```

```

2418     }
2419   } {
2420     \prg_return_false:
2421   }
2422 }
2423 }
2424 \BNVS_new_conditional_vvc:cn { if_resolve_Z } { T, F, TF }
2425 \BNVS_new_conditional_cpnn { if_append_Z:nnc } #1 #2 #3 { T, F, TF } {
2426   \BNVS_begin:
2427     \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2428       \BNVS_end_tl_put_right:cv { #3 } { #3 }
2429       \prg_return_true:
2430     } {
2431       \BNVS_end:
2432       \prg_return_false:
2433     }
2434   }
2435   \BNVS_new_conditional_vvc:cn { if_append_Z } { T, F, TF }

```

```

\__bnvs_if_resolve_L:nncTF \__bnvs_if_resolve_L:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append_L:nncTF \__bnvs_if_append_L:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no code>}

```

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

```

2436 \BNVS_new_conditional_cpnn { if_resolve_L:nnc } #1 #2 #3 { T, F, TF } {
2437   \__bnvs_if_get_cache:nnncTF L { #1 } { #2 } { #3 } {
2438     \prg_return_true:
2439   } {
2440     \__bnvs_if_get:nnncTF L { #1 } { #2 } { #3 } {
2441       \__bnvs_quark_if_nil:cTF { #3 } {
2442         \__bnvs_gset:nnnn L { #1 } { #2 } { \q_no_value }

```

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

```

2443   \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2444     \__bnvs_tl_put_right:cn { #3 } { - }
2445     \__bnvs_if_append_A:nncTF { #1 } { #2 } { #3 } {
2446       \__bnvs_tl_put_right:cn { #3 } { + 1 }
2447       \__bnvs_round:c { #3 }
2448       \__bnvs_gset:nnnn L { #1 } { #2 } { \q_nil }
2449       \__bnvs_gset_cache:nnnv L { #1 } { #2 } { #3 }
2450       \prg_return_true:
2451     } {
2452       \BNVS_error:n {
2453         Unavailable~first~for~#1~(\__bnvs_if_resolve_L:nncTF/2) }
2454       \prg_return_false:
2455     }
2456   } {
2457     \BNVS_error:n {
2458       Unavailable~last~for~#1~(\__bnvs_if_resolve_L:nncTF/1) }
2459     \prg_return_false:
2460   }
2461 } {

```

```

2462     \_bnvs_quark_if_no_value:cTF { #3 } {
2463         \BNVS_error:n {Circular-definition:~#1!#2 (Error-recovery-1)}
2464         \_bnvs_gset_cache:nnnn L { #1 } { #2 } { 1 }
2465         \_bnvs_tl_set:cn { #3 } { 1 }
2466         \prg_return_true:
2467     } {
2468         \_bnvs_if_resolve:vcTF { #3 } { #3 } {
2469             \_bnvs_gset_cache:nnnv L { #1 } { #2 } { #3 }
2470             \prg_return_true:
2471         } {
2472             \prg_return_false:
2473         }
2474     }
2475 } {
2476     \prg_return_false:
2477 }
2478 }
2479 }
2480 }
2481 \BNVS_new_conditional_vvc:cn { if_resolve_L } { T, F, TF }
2482 \BNVS_new_conditional:cpnn { if_append_L:nnc } #1 #2 #3 { T, F, TF } {
2483     \BNVS_begin:
2484     \_bnvs_if_resolve_L:nncTF { #1 } { #2 } { #3 } {
2485         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2486         \prg_return_true:
2487     } {
2488         \BNVS_end:
2489         \prg_return_false:
2490     }
2491 }
2492 \BNVS_new_conditional_vvc:cn { if_append_L } { T, F, TF }

```

```

\_bnvs_if_resolve_previous:nncTF \_bnvs_if_append_previous:ncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
\_bnvs_if_append_previous:nncTF code>}

```

Resolve the index after the $\langle id \rangle!$ $\langle key \rangle$ slide range into the $\langle ans \rangle$ t1 variable, or append this index to that variable. Execute $\langle yes \ code \rangle$ when there is a $\langle next \rangle$ index, $\langle no \ code \rangle$ otherwise. In the latter case, the $\langle ans \rangle$ t1 is undefined on resolution only.

```

\_bnvs_if_resolve_first:nncTF \_bnvs_if_resolve_first:nncTF {<id>} {<tag>} <ans> {<yes code>} {<no
\_bnvs_if_resolve_first:vvcTF code>}
\_bnvs_if_append_first:nncTF \_bnvs_if_append_first:nncTF {<id>} {<tag>} <ans> {<yes code>} {<no
\_bnvs_if_append_first:vvcTF code>}

```

Resolve the first index starting the $\langle id \rangle!$ $\langle tag \rangle$ slide range into the $\langle ans \rangle$ t1 variable, or append this index to that variable. Execute $\langle yes \ code \rangle$ when there is a $\langle first \rangle$ index, $\langle no \ code \rangle$ otherwise. In the latter case, on resolution only, the content of the $\langle ans \rangle$ t1 variable is undefined.

```

2493 \BNVS_new_conditional:cpnn { if_resolve_first:nnc } #1 #2 #3 { T, F, TF } {

```

```

2494 \__bnvs_if_resolve_V:nncTF { #1 } { #2.first } { #3 }
2495 { \prg_return_true: }
2496 { \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 }
2497   { \prg_return_true: }
2498   { \__bnvs_if_resolve_v:nncTF { #1 } { #2.1 } { #3 }
2499     { \prg_return_true: } { \prg_return_false: }
2500   }
2501 }
2502 }
2503 \BNVS_new_conditional_vvc:cn { if_resolve_first } { T, F, TF }

2504 \BNVS_new_conditional:cpnn { if_append_first:nnc } #1 #2 #3 { T, F, TF } {
2505   \__bnvs_if_append_index:nncTF { #1 } { #2 } { 1 } { #3 } { \prg_return_true: } {
2506     \__bnvs_if_append_A:nncTF { #1 } { #2 } { #3 }
2507     { \prg_return_true: } { \prg_return_false: }
2508   }
2509 }
2510 \BNVS_new_conditional_vvc:cn { if_append_first } { T, F, TF }

```

```

\__bnvs_if_resolve_last:nncTF \__bnvs_if_resolve_last:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
\__bnvs_if_resolve_last:vvcTF code>}
\__bnvs_if_append_last:nncTF \__bnvs_if_append_last:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
\__bnvs_if_append_last:vvcTF code>}

```

Resolve the last index of the $\langle id \rangle! \langle a \rangle$ slide range into the $\langle ans \rangle$ t1 variable, or append this index to that variable. Execute $\langle yes \ code \rangle$ when there is a $\langle last \rangle$ index, $\langle no \ code \rangle$ otherwise. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only.

```

2511 \BNVS_new_conditional:cpnn { if_resolve_last:nnc } #1 #2 #3 { T, F, TF } {
2512   \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 }
2513   { \prg_return_true: } { \prg_return_false: }
2514 }
2515 \BNVS_new_conditional_vvc:cn { if_resolve_last } { T, F, TF }

2516 \BNVS_new_conditional:cpnn { if_append_last:nnc } #1 #2 #3 { T, F, TF } {
2517   \__bnvs_if_append_Z:nncTF { #1 } { #2 } { #3 }
2518   { \prg_return_true: } { \prg_return_false: }
2519 }

2520 \BNVS_new_conditional_vvc:cn { if_append_last } { T, F, TF }

```

```

\__bnvs_if_resolve_length:nncTF \__bnvs_if_resolve_length:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
\__bnvs_if_append_length:nncTF code>}
\__bnvs_if_append_length:vvcTF \__bnvs_if_append_length:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
\__bnvs_if_append_length:vvcTF code>}

```

Resolve the length of the $\langle id \rangle! \langle a \rangle$ slide range into the $\langle ans \rangle$ t1 variable, or append this number to that variable. Execute $\langle yes \ code \rangle$ when there is a $\langle last \rangle$ index, $\langle no \ code \rangle$ otherwise. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only.

```

2521 \BNVS_new_conditional:cpnn { if_resolve_length:nnc } #1 #2 #3 { T, F, TF } {

```

```

2522 \__bnvs_if_resolve_L:nncTF { #1 } { #2 } { #3 }
2523 { \prg_return_true: } { \prg_return_false: }
2524 }
2525 \BNVS_new_conditional_vvc:cn { if_resolve_length } { T, F, TF }
2526 \BNVS_new_conditional:cpnn { if_append_length:nnc } #1 #2 #3 { T, F, TF } {
2527 \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 }
2528 { \prg_return_true: } { \prg_return_false: }
2529 }
2530 \BNVS_new_conditional_vvc:cn { if_append_length } { T, F, TF }

```

```

\__bnvs_if_resolve_range:nncTF \__bnvs_if_resolve_range:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
\__bnvs_if_append_range:nncTF code>}
\__bnvs_if_append_range:nncTF {<id>} {<a>} {<ans>} {<yes code>} {<no
code>}

```

Resolve the range of the $\langle id \rangle!$ $\langle key \rangle$ slide range into the $\langle ans \rangle$ t1 variable or append this range to that variable. Execute $\langle yes\ code \rangle$ when there is a $\langle range \rangle$, $\langle no\ code \rangle$ otherwise, in that latter case the content the $\langle ans \rangle$ t1 variable is undefined on resolution only.

```

2531 \BNVS_new_conditional:cpnn { if_append_range:nnc } #1 #2 #3 { T, F, TF } {
2532 \BNVS_begin:
2533 \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { a } {
2534 \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
2535 \__bnvs_tl_set:cn { a } { 0 }
2536 }
2537 \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { b } {

```

Limited from above and below.

```

2538 \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2539 \__bnvs_tl_set:cn { b } { 0 }
2540 }
2541 \__bnvs_tl_put_right:cn { a } { - }
2542 \__bnvs_tl_put_right:cv { a } { b }
2543 \BNVS_end_tl_put_right:cv { #3 } { a }
2544 \prg_return_true:
2545 } {

```

Limited from below.

```

2546 \BNVS_end_tl_put_right:cv { #3 } { a }
2547 \__bnvs_tl_put_right:cn { #3 } { - }
2548 \prg_return_true:
2549 }
2550 } {
2551 \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { b } {

```

Limited from above.

```

2552 \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2553 \__bnvs_tl_set:cn { b } { 0 }
2554 }
2555 \__bnvs_tl_put_left:cn { b } { - }
2556 \BNVS_end_tl_put_right:cv { #3 } { b }

```

```

2557     \prg_return_true:
2558   } {
2559     \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { b } {
2560       \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2561         \__bnvs_tl_set:cn { b } { 0 }
2562       }

```

Unlimited range.

```

2563       \BNVS_end_tl_put_right:cv { #3 } { b }
2564       \__bnvs_tl_put_right:cn { #3 } { - }
2565       \prg_return_true:
2566     } {
2567       \BNVS_end:
2568       \prg_return_false:
2569     }
2570   }
2571 }
2572 }
2573 \BNVS_new_conditional_vvc:cn { if_append_range } { T, F, TF }

2574 \BNVS_new_conditional:cpnn { if_resolve_range:nnc } #1 #2 #3 { T, F, TF } {
2575   \__bnvs_tl_clear:c { #3 }
2576   \__bnvs_if_append_range:ncTF { #1 } { #2 } { #3 } {
2577     \prg_return_true:
2578   } {
2579     \prg_return_false:
2580   }
2581 }
2582 \BNVS_new_conditional_vvc:cn { if_resolve_range } { T, F, TF }

```

```

\__bnvs_if_resolve_previous:nncTF \__bnvs_if_resolve_previous:nncTF {<id>} {<tag>} {<ans>} {<yes code>}
\__bnvs_if_append_previous:nncTF {<no code>}
\__bnvs_if_append_previous:nncTF {<id>} {<tag>} {<ans>} {<yes code>}
{<no code>}

```

Resolve the index after the *<key>* slide range into the *<ans>* tl variable, or append this index to that 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.

```

2583 \BNVS_new_conditional:cpnn { if_resolve_previous:nnc } #1 #2 #3 { T, F, TF } {
2584   \__bnvs_if_get_cache:nnncTF P { #1 } { #2 } { #3 } {
2585     \prg_return_true:
2586   } {
2587     \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2588       \__bnvs_tl_put_right:cn { #3 } { -1 }
2589       \__bnvs_round:c { #3 }
2590       \__bnvs_gset_cache:nnnv P { #1 } { #2 } { #3 }
2591       \prg_return_true:
2592     } {
2593       \prg_return_false:
2594     }
2595   }
2596 }
2597 \BNVS_new_conditional_vvc:cn { if_resolve_previous } { T, F, TF }

```

```

2598 \BNVS_new_conditional:cpnn { if_append_previous:nnc } #1 #2 #3 { T, F, TF } {
2599   \BNVS_begin:
2600   \__bnvs_if_resolve_previous:nncTF { #1 } { #2 } { #3 } {
2601     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2602     \prg_return_true:
2603   } {
2604     \BNVS_end:
2605     \prg_return_false:
2606   }
2607 }
2608 \BNVS_new_conditional_vvc:cn { if_append_previous } { T, F, TF }

```

```

\__bnvs_if_resolve_next:nncTF \__bnvs_if_resolve_next:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no
\__bnvs_if_append_next:nncTF code>}
\__bnvs_if_append_next:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no
code>}

```

Resolve the index after the *<id>*! slide range into the *<ans>* t1 variable, or append this index to that variable. Execute *<yes code>* when there is a *<next>* index, *<no code>* otherwise. In the latter case, the content of the *<t1 variable>* is undefined, on resolution only.

```

2609 \BNVS_new_conditional:cpnn { if_resolve_next:nnc } #1 #2 #3 { T, F, TF } {
2610   \__bnvs_if_get_cache:nnncTF N { #1 } { #2 } { #3 } {
2611     \prg_return_true:
2612   } {
2613     \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2614       \__bnvs_t1_put_right:cn { #3 } { +1 }
2615       \__bnvs_round:c { #3 }
2616       \__bnvs_gset_cache:nnnv N { #1 } { #2 } { #3 }
2617       \prg_return_true:
2618     } {
2619       \prg_return_false:
2620     }
2621   }
2622 }
2623 \BNVS_new_conditional_vvc:cn { if_resolve_next } { T, F, TF }

2624 \BNVS_new_conditional:cpnn { if_append_next:nnc } #1 #2 #3 { T, F, TF } {
2625   \BNVS_begin:
2626   \__bnvs_if_resolve_next:nncTF { #1 } { #2 } { #3 } {
2627     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2628     \prg_return_true:
2629   } {
2630     \BNVS_end:
2631     \prg_return_true:
2632   }
2633 }
2634 \BNVS_new_conditional_vvc:cn { if_append_next } { T, F, TF }

```

<code>__bnvs_if_resolve_v:nncTF</code> <code>__bnvs_if_resolve_v:vvcTF</code> <code>__bnvs_if_append_v:nnc</code>	<code>__bnvs_if_resolve_v:nncTF {<id>} {<a>} <ans> {<yes code>} {<no code>}</code> <code>__bnvs_if_append_v:vvcTF</code> <code>__bnvs_if_append_v:nncTF {<id>} {<a>} <ans> {<yes code>} {<no code>}</code>
--	---

Resolve the value of the $\langle id \rangle!$ $\langle a \rangle$ overlay set into the $\langle ans \rangle$ t1 variable or append this value to the right of this variable. Execute $\langle yes\ code \rangle$ when there is a $\langle value \rangle$, $\langle no\ code \rangle$ otherwise. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only. Calls `__bnvs_if_resolve_V:nncTF`.

```

2635 \BNVS_new_conditional:cpnn { if_resolve_v:nnc } #1 #2 #3 { T, F, TF } {
2636   \__bnvs_if_get:nnncTF v { #1 } { #2 } { #3 } {
2637     \__bnvs_quark_if_no_value:cTF { #3 } {
2638       \BNVS_error:n {Circular-definition:~#1!#2 (Error~recovery~1)}
2639       \__bnvs_gset:nnnn V { #1 } { #2 } { 1 }
2640       \__bnvs_t1_set:cn { #3 } { 1 }
2641       \prg_return_true:
2642     } {
2643       \prg_return_true:
2644     }
2645   } {
2646     \__bnvs_gset:nnnn v { #1 } { #2 } { \q_no_value }
2647     \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
2648       \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
2649       \prg_return_true:
2650     } {
2651       \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2652         \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
2653         \prg_return_true:
2654       } {
2655         \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2656           \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
2657           \prg_return_true:
2658         } {
2659           \__bnvs_gunset:nnn v { #1 } { #2 }
2660           \prg_return_false:
2661         }
2662       }
2663     }
2664   }
2665 }
2666 \BNVS_new_conditional_vvc:cn { if_resolve_v } { T, F, TF }

2667 \BNVS_new_conditional:cpnn { if_append_v:nnc } #1 #2 #3 { T, F, TF } {
2668   \BNVS_begin:
2669   \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #3 } {
2670     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2671     \prg_return_true:
2672   } {
2673     \BNVS_end:
2674     \prg_return_false:
2675   }
2676 }
2677 \BNVS_new_conditional_vvc:cn { if_append_v } { T, F, TF }

```

```

__bnvs_index_can:nnTF      __bnvs_index_can:nnTF {<id>} {<a>} {<yes code>} {<no code>}
__bnvs_index_can:vvTF      __bnvs_if_resolve_index:nnncTF {<id>} {<a>} {<integer>} {<ans>} {<yes
__bnvs_if_resolve_index:nnncTF code>} {<no code>}
__bnvs_if_resolve_index:vvvcTF __bnvs_if_append_index:nnncTF {<id>} {<a>} {<integer>} {<ans>} {<yes
__bnvs_if_append_index:nnncTF code>} {<no code>}
__bnvs_if_append_index:vvvcTF

```

Resolve the index associated to the $\langle id \rangle!$ $\langle a \rangle$ set and $\langle integer \rangle$ slide range into the $\langle ans \rangle$ t1 variable or append this index to the right of that 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, $\langle yes \text{ code} \rangle$ is executed, otherwise $\langle no \text{ code} \rangle$ is executed. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only. The computation may fail when too many recursion calls are required.

```

2678 \BNVS_new_conditional:cpnn { index_can:nn } #1 #2 { T, F, TF } {
2679   __bnvs_is_gset:nnnTF V { #1 } { #2 } {
2680     \prg_return_true:
2681   } {
2682     __bnvs_is_gset:nnnTF A { #1 } { #2 } {
2683       \prg_return_true:
2684     } {
2685       __bnvs_is_gset:nnnTF Z { #1 } { #2 } {
2686         \prg_return_true:
2687       } {
2688         \prg_return_false:
2689       }
2690     }
2691   }
2692 }

2693 \BNVS_new_conditional:cpnn { index_can:vv } #1 #2 { T, F, TF } {
2694   \BNVS_t1_use:nv {
2695     \BNVS_t1_use:Nv __bnvs_index_can:nTF { #1 }
2696   } { #2 } { \prg_return_true: } { \prg_return_false: }
2697 }

2698 \BNVS_new_conditional:cpnn { if_resolve_index:nnnc } #1 #2 #3 #4 { T, F, TF } {
2699   \exp_args:Ne __bnvs_if_resolve_V:nncTF { #1 } { #2.#3 } { #4 } {
2700     \prg_return_true:
2701   } {
2702     __bnvs_if_resolve_first:nncTF { #1 } { #2 } { #4 } {
2703       __bnvs_t1_put_right:cn { #4 } { + #3 - 1 }
2704       __bnvs_round:c { #4 }
2705       \prg_return_true:

```

Limited overlay set.

```

2706   } {
2707     __bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #4 } {
2708       __bnvs_t1_put_right:cn { #4 } { + #3 - 1 }
2709       __bnvs_round:c { #4 }
2710       \prg_return_true:
2711     } {
2712       __bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
2713         __bnvs_t1_put_right:cn { #4 } { + #3 - 1 }
2714         __bnvs_round:c { #4 }

```

```

2715         \prg_return_true:
2716     } {
2717         \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
2718             \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
2719             \__bnvs_round:c { #4 }
2720             \prg_return_true:
2721         } {
2722             \prg_return_false:
2723         }
2724     }
2725 }
2726 }
2727 }
2728 }

2729 \BNVS_new_conditional:cpnn { if_resolve_index:nncv } #1 #2 #3 #4 { T, F, TF } {
2730     \BNVS_tl_use:nv {
2731         \__bnvs_if_resolve_index:nnncTF { #1 } { #2 }
2732     } { #3 } { #4 } {
2733         \prg_return_true:
2734     } {
2735         \prg_return_false:
2736     }
2737 }

2738 \BNVS_new_conditional:cpnn { if_resolve_index:vvvc } #1 #2 #3 #4 { T, F, TF } {
2739     \BNVS_tl_use:nvv {
2740         \BNVS_tl_use:Nv \__bnvs_if_resolve_index:nnncTF { #1 }
2741     } { #2 } { #3 } { #4 } {
2742         \prg_return_true:
2743     } {
2744         \prg_return_false:
2745     }
2746 }

2747 \BNVS_new_conditional:cpnn { if_append_index:nnnc } #1 #2 #3 #4 { T, F, TF } {
2748     \BNVS_begin:
2749     \__bnvs_if_resolve_index:nnncTF { #1 } { #2 } { #3 } { #4 } {
2750         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2751         \prg_return_true:
2752     } {
2753         \BNVS_end:
2754         \prg_return_false:
2755     }
2756 }

2757 \BNVS_new_conditional:cpnn { if_append_index:vvvc } #1 #2 #3 #4 { T, F, TF } {
2758     \BNVS_tl_use:nvv {
2759         \BNVS_tl_use:Nv \__bnvs_if_append_index:nnncTF { #1 }
2760     } { #2 } { #3 } { #4 } {
2761         \prg_return_true:
2762     } {
2763         \prg_return_false:
2764     }
2765 }

```

6.19 Index counter

_bnvs_n_assign:nnn _bnvs_n_assign:nnn {<id>} {<a>} {<value>}

_bnvs_n_assign:vvv

Assigns the resolved <value> to n counter <id>{<a>}. Execute <yes code> when resolution succeeds, <no code> otherwise.

```

2766 \BNVS_new:cpn { n_assign:nnn } #1 #2 #3 {
2767   \\_bnvs_if_get:nnncF V { #1 } { #2 } { a } {
2768     \BNVS_warning:n { Unkwown~ #1!#2,~defaults-to~0 }
2769     \\_bnvs_gset:nnnn V { #1 } { #2 } { 0 }
2770   }
2771   \\_bnvs_if_resolve:ncTF { #3 } { a } {
2772     \\_bnvs_gset:nnnv v { #1 } { #2 } { a }
2773   } {
2774     \BNVS_error:n { NO~resolution~of~#3,~defaults-to~0 }
2775     \\_bnvs_gset:nnnn v { #1 } { #2 } { 0 }
2776   }
2777 }

2778 \BNVS_new:cpn { n_assign:vvv } #1 {
2779   \BNVS_tl_use:nvv {
2780     \BNVS_tl_use:cv { n_assign:nn } { #1 }
2781   }
2782 }
```

_bnvs_if_resolve_n:ncTF _bnvs_if_resolve_n:nnncTF {<id>} {<a>} {<ans>} {<yes code>} {<no code>}

_bnvs_if_append_n:ncTF

_bnvs_if_append_n:vcTF

Evaluate the n counter associated to the {<id>}{<a>} overlay set into <ans> t1 variable. Initialize this counter to 1 on the first use. <no code> is never executed.

```

2783 \BNVS_new_conditional:cpnn { if_resolve_n:nnnc } #1 #2 #3 { T, F, TF } {
2784   \\_bnvs_if_get:nnncTF n { #1 } { #2 } { #3 } {
2785     \\_bnvs_if_resolve:vcTF { #3 } { #3 } {
2786       \prg_return_true:
2787     } {
2788       \prg_return_false:
2789     }
2790   } {
2791     \\_bnvs_tl_set:cn { #3 } { 1 }
2792     \\_bnvs_gset:nnnn n { #1 } { #2 } { 1 }
2793     \prg_return_true:
2794   }
2795 }

2796 \BNVS_new_conditional_vvc:cn { if_resolve_n } { T, F, TF }

2797 \BNVS_new_conditional:cpnn { if_append_n:nnnc } #1 #2 #3 { T, F, TF } {
2798   \BNVS_begin:
2799   \\_bnvs_if_resolve_n:nnncTF { #1 } { #2 } { #3 } {
2800     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2801     \prg_return_true:
2802   } {
2803     \BNVS_end:
```

```

2804     \prg_return_false:
2805   }
2806 }

2807 \BNVS_new_conditional_vvc:cn { if_append_n } { T, F, TF }

```

```

\__bnvs_if_resolve_n_index:nncTF \__bnvs_if_resolve_n_index:nncTF {<id>} {<tag>} <ans> {<yes code>} {<no
\__bnvs_if_append_n_index:nncTF code>}
\__bnvs_if_append_n_index:nncTF {<id>} {<tag>} <ans> {<yes code>} {<no
code>}

```

Resolve the index for the value of the n counter associated to the {<a>} overlay set into the <ans> t1 variable or append this value the right of that 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 <ans> t1 variable is undefined on resolution only.

```

2808 \BNVS_new_conditional:cpnn { if_resolve_n_index:nnc } #1 #2 #3 { T, F, TF } {
2809   \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #3 } {
2810     \__bnvs_t1_put_left:cn { #3 } { #1!#2. }
2811     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2812       \prg_return_true:
2813     } {
2814       \prg_return_false:
2815     }
2816   } {
2817     \prg_return_false:
2818   }
2819 }

2820 \BNVS_new_conditional:cpnn { if_append_n_index:nnc } #1 #2 #3 { T, F, TF } {
2821   \BNVS_begin:
2822   \__bnvs_if_resolve_n_index:nncTF { #1 } { #2 } { #3 } {
2823     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2824     \prg_return_true:
2825   } {
2826     \BNVS_end:
2827     \prg_return_false:
2828   }
2829 }
2830 \BNVS_new_conditional_vvc:cn { if_append_n_index } { T, F, TF }

```

6.20 Value counter

```

\__bnvs_if_resolve_v_incr:nnncTF \__bnvs_if_resolve_v_incr:nnnTF {<id>} {<a>} {<offset>} <ans> {<yes
\__bnvs_if_append_v_incr:nnncTF code>} {<no code>}
\__bnvs_if_append_v_incr:vvncTF \__bnvs_if_append_v_incr:nnncTF {<id>} {<a>} {<offset>} <ans> {<yes
code>} {<no code>}

```

Increment the value counter position accordingly. Put the result in the <ans> t1 variable.

```

2831 \BNVS_new_conditional:cpnn { if_resolve_v_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {

```

```

2832 \__bnvs_if_resolve:ncTF { #3 } { #4 } {
2833   \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
2834     \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
2835       \prg_return_true:
2836     } {
2837       \prg_return_false:
2838     }
2839   } {
2840     \__bnvs_tl_put_right:cn { #4 } { + }
2841     \__bnvs_if_append_v:nncTF { #1 } { #2 } { #4 } {
2842       \__bnvs_round:c { #4 }
2843       \__bnvs_gset:nnnv v { #1 } { #2 } { #4 }
2844       \prg_return_true:
2845     } {
2846       \prg_return_false:
2847     }
2848   }
2849 } {
2850   \prg_return_false:
2851 }
2852 }

2853 \BNVS_new_conditional:cpnn { if_append_v_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
2854   \BNVS_begin:
2855   \__bnvs_if_resolve_v_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
2856     \BNVS_end_tl_put_right:cv { #4 } { #4 }
2857     \prg_return_true:
2858   } {
2859     \BNVS_end:
2860     \prg_return_false:
2861   }
2862 }
2863 \BNVS_new_conditional_vvnc:cn { if_append_v_incr } { T, F, TF }

2864 \BNVS_new_conditional:cpnn { if_resolve_v_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
2865   \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
2866     \BNVS_begin:
2867     \__bnvs_if_resolve:ncTF { #3 } { a } {
2868       \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
2869         \BNVS_end:
2870         \prg_return_true:
2871       } {
2872         \__bnvs_tl_put_right:cn { a } { + }
2873         \__bnvs_tl_put_right:cv { a } { #4 }
2874         \__bnvs_round:c { a }
2875         \BNVS_end_gset:nnnv v { #1 } { #2 } { a }
2876         \prg_return_true:
2877       }
2878     } {
2879       \BNVS_end:
2880       \prg_return_false:
2881     }
2882   } {

```

```

2883     \prg_return_false:
2884 }
2885 }
2886 \BNVS_new_conditional_vvvc:cn { if_resolve_v_post } { T, F, TF }
2887 \BNVS_new_conditional:cpnn { if_append_v_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
2888   \BNVS_begin:
2889     \__bnvs_if_resolve_v_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
2890       \BNVS_end_tl_put_right:cv { #4 } { #4 }
2891       \prg_return_true:
2892     } {
2893       \prg_return_false:
2894     }
2895 }
2896 \BNVS_new_conditional_vvnc:cn { if_append_v_post } { T, F, TF }
2897 \BNVS_new_conditional_vvvc:cn { if_append_v_post } { T, F, TF }

```

```

\__bnvs_if_resolve_n_incr:nnncTF \__bnvs_if_resolve_n_incr:nnncTF {<id>} {<a>} {<base>} {<offset>}
\__bnvs_if_resolve_n_incr:nnncTF {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append_n_incr:nnncTF \__bnvs_if_resolve_n_incr:nnncTF {<id>} {<a>} {<offset>} {<ans>} {<yes
\__bnvs_if_append_n_incr:nnncTF code>} {<no code>}
\__bnvs_if_append_n_incr:vvncTF \__bnvs_if_append_n_incr:nnncTF {<id>} {<a>} {<base>} {<offset>}
\__bnvs_if_resolve_n_post:nnncTF {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append_n_post:nnncTF \__bnvs_if_append_n_incr:nnncTF {<id>} {<a>} {<offset>} {<ans>} {<yes
\__bnvs_if_append_n_post:vvncTF code>} {<no code>}

```

Increment the implicit *n* counter accordingly. When requested, put the resulting index in the *<ans>* tl variable or append to its right. This is not run in a group.

```

2898 \BNVS_new_conditional:cpnn { if_resolve_n_incr:nnnc } #1 #2 #3 #4 { T, TF } {
Resolve the <offset> into the <ans> variable.
2899   \__bnvs_if_resolve:ncTF { #3 } { #4 } {
2900     \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
The offset is resolved to 0, we just have to resolve the ...n
2901       \__bnvs_if_resolve_n:nnncTF { #1 } { #2 } { #4 } {
2902         \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
2903           \prg_return_true:
2904         } {
2905           \prg_return_false:
2906         }
2907       } {
2908         \prg_return_false:
2909       }
2910     } {

```

The *<offset>* does not resolve to 0.

```

2911     \__bnvs_tl_put_right:cn { #4 } { + }
2912     \__bnvs_if_append_n:nnncTF { #1 } { #2 } { #4 } {
2913       \__bnvs_round:c { #4 }
2914       \__bnvs_gset:nnnv n { #1 } { #2 } { #4 }
2915       \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
2916         \prg_return_true:
2917       } {

```

```

2918         \prg_return_false:
2919     }
2920 } {
2921     \prg_return_false:
2922 }
2923 }
2924 } {
2925     \prg_return_false:
2926 }
2927 }

2928 \BNVS_new_conditional:cpnn
2929 { if_append_n_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
2930     \BNVS_begin:
2931     \__bnvs_if_resolve_n_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
2932         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2933         \prg_return_true:
2934     } {
2935         \BNVS_end:
2936         \prg_return_false:
2937     }
2938 }
2939 \BNVS_new_conditional_vvnc:cn { if_append_n_incr } { T, F, TF }

```

<pre> __bnvs_if_resolve_v_post:nnncTF __bnvs_if_append_v_post:nnncTF __bnvs_if_append_v_post:vvncTF </pre>	<pre> __bnvs_if_resolve_v_post:nnncTF {<id>} {<a>} {<offset>} <ans> {<yes code>} {<no code>} __bnvs_if_append_v_post:nnncTF {<id>} {<a>} {<offset>} <ans> {<yes code>} {<no code>} </pre>
--	---

Resolve the value of the free counter for the given $\langle a \rangle$ into the $\langle ans \rangle$ t1 variable then increment this free counter position accordingly. The append version, appends the value to the right of the $\langle ans \rangle$ t1 variable. The content of $\langle ans \rangle$ is undefined while in the $\{ \langle no \text{ code} \rangle \}$ branch and on resolution only.

```

2940 \BNVS_new_conditional:cpnn { if_resolve_n_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
2941     \__bnvs_if_resolve_n:nnncTF { #1 } { #2 } { #4 } {
2942         \BNVS_begin:
2943         \__bnvs_if_resolve:ncTF { #3 } { #4 } {
2944             \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
2945                 \BNVS_end:
2946                 \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
2947                     \prg_return_true:
2948                 } {
2949                     \prg_return_false:
2950                 }
2951             } {
2952                 \__bnvs_tl_put_right:cn { #4 } { + }
2953                 \__bnvs_if_append_n:nnncTF { #1 } { #2 } { #4 } {
2954
2955                     \__bnvs_round:c { #4 }
2956                     \__bnvs_gset:nnnv n { #1 } { #2 } { #4 }
2957                 \BNVS_end:
2958                 \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {

```



```

2959         \prg_return_true:
2960     } {
2961         \prg_return_false:
2962     }
2963 } {
2964     \BNVS_end:
2965     \prg_return_false:
2966 }
2967 }
2968 } {
2969     \BNVS_end:
2970     \prg_return_false:
2971 }
2972 } {
2973     \prg_return_false:
2974 }
2975 }

2976 \BNVS_new_conditional:cpnn { if_append_n_post:nnnc } #1 #2 #3 #4{ T, F, TF } {
2977     \BNVS_begin:
2978     \__bnvs_if_resolve_n_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
2979         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2980         \prg_return_true:
2981     } {
2982         \BNVS_end:
2983         \prg_return_false:
2984     }
2985 }
2986 \BNVS_new_conditional_vvnc:cn { if_append_n_post } { T, F, TF }

```

6.21 Functions for the resolution

They manily start with `__bnvs_if_resolve_` or `__bnvs_split_`

<code>__bnvs_split_pop_iksp:TFF</code>	<code>__bnvs_split_pop_iksp:TFF {<i><black code></i>} {<i><blank code></i>}</code>
<code>__bnvs_split_end_return_or_pop_complete:T</code>	<code>{<i><end code></i>}</code>
<code>__bnvs_split_end_return_or_pop_void:T</code>	<code>__bnvs_split_end_return_or_pop_complete:T {<i><blank code></i>}</code>
	<code>__bnvs_split_end_return_or_pop_void:T {<i><black code></i>}</code>

For `__bnvs_split_pop_iksp:TFF`. If the `split` sequence is empty, execute *<end code>*. Otherwise pops the 4 heading items of the `split` sequence into the four `tl` variables `id`, `kri`, `short`, `path`. If `short` is blank then execute *<blank code>*, otherwise execute *<black code>*.

For `__bnvs_split_end_return_or_pop_complete:T`: pops the four heading items of the `split` sequence into the four variables `n_incr`, `plus`, `rhs`, `post`. Then execute *<black code>*.

For `__bnvs_split_end_return_or_pop_void:T`: pops the eight heading items of the `split` sequence then execute *<blank code>*.

This is called each time a `ref`, `id`, `path` has been parsed.

```

2987 \BNVS_new:cpn { split_pop_iksp:TFF } #1 #2 #3 {

```

```

2988 \__bnvs_split_if_pop_left:cTF { id } {
2989 \__bnvs_split_if_pop_left:cTF { kri } {
2990 \__bnvs_split_if_pop_left:cTF { short } {
2991 \__bnvs_split_if_pop_left:cTF { path } {
2992 \__bnvs_tl_if_blank:vTF { short } {

```

The first 4 capture groups are empty, and the 4 next ones are expected to contain the expected information.

```

2993 #2
2994 } {
2995 \BNVS_tl_use:nv {
2996 \regex_match:NnT \c__bnvs_A_reserved_Z_regex
2997 } { short } {
2998 \__bnvs_tl_if_eq:cnF { short } { pauses } {
2999 \__bnvs_tl_if_eq:cnF { short } { slideinframe } {
3000 \BNVS_error:x { Use~of~reserved~``\BNVS_tl_use:c { tag }'' }
3001 }
3002 }
3003 }
3004 \__bnvs_tl_if_blank:vTF { kri } {
3005 \__bnvs_tl_set:cv { id } { id_last }
3006 } {
3007 \__bnvs_tl_set:cv { id_last } { id }
3008 }

```

Build the path sequence and lowercase components conditionals.

```

3009 \__bnvs_seq_set_split:cnv { path } { . } { path }
3010 #1
3011 }
3012 } {
3013 \BNVS_fatal:n { split_pop_iksp:TFF/path }
3014 }
3015 } {
3016 \BNVS_fatal:n { split_pop_iksp:TFF/short }
3017 }
3018 } {
3019 \BNVS_fatal:n { split_pop_iksp:TFF/kri }
3020 }
3021 } {
3022 #3
3023 }
3024 }

```

conditional variants.

```

3025 \BNVS_new:cpn { split_end_return_or_pop_complete:T } #1 {
3026 \cs_set:Npn \BNVS_split_F:n ##1 {
3027 \BNVS_end_unreachable_return_false:n {
3028 split_end_return_or_pop_complete: ##1
3029 }
3030 }
3031 \__bnvs_split_if_pop_left_or:cT { n_incr } {
3032 \__bnvs_split_if_pop_left_or:cT { plus } {
3033 \__bnvs_split_if_pop_left_or:cT { rhs } {
3034 \__bnvs_split_if_pop_left_or:cT { post } {

```

```

3035         #1
3036     }
3037 }
3038 }
3039 }
3040 }

3041 \BNVS_new:cpn { split_end_return_or_pop_void:T } #1 {
3042     \cs_set:Npn \BNVS_split_F:n ##1 {
3043         \BNVS_end_unreachable_return_false:n {
3044             split_end_return_or_pop_void: ##1
3045         }
3046     }
3047     \__bnvs_split_if_pop_left:cTn { a } {
3048         \__bnvs_split_if_pop_left:cTn { a } {
3049             \__bnvs_split_if_pop_left:cTn { a } {
3050                 \__bnvs_split_if_pop_left:cTn { a } {
3051                     \__bnvs_split_if_pop_left:cTn { a } {
3052                         \__bnvs_split_if_pop_left:cTn { a } {
3053                             \__bnvs_split_if_pop_left:cTn { a } {
3054                                 \__bnvs_split_if_pop_left:cTn { a } {
3055                                     #1
3056                                 } { T/8 }
3057                             } { T/7 }
3058                         } { T/6 }
3059                     } { T/5 }
3060                 } { T/4 }
3061             } { T/3 }
3062         } { T/2 }
3063     } { T/1 }
3064 }

```

<code>__bnvs_if_resolve:ncTF</code>	<code>__bnvs_if_resolve:ncTF {<expression>} {<ans>} {<yes code>} {<no code>}</code>
<code>__bnvs_if_resolve:vcTF</code>	<code>__bnvs_if_append:ncTF {<expression>} {<ans>} {<yes code>} {<no code>}</code>
<code>__bnvs_if_append:ncTF</code>	Resolves the <i><expression></i> , replacing all the named overlay specifications by their static counterpart then put the rounded result in <i><ans></i> t1 variable when resolving or to the right of this variable when appending.
<code>__bnvs_if_append:vcTF</code>	

Implementation details. Executed within a group. Heavily used by `\..._if_resolve_query:ncTF`, where *<expression>* was initially enclosed inside ‘?(...)’. Local variables:

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

(End of definition for `\l__bnvs_ans_tl`.)

`\l__bnvs_split_seq` The sequence of 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`.)

3065 `\BNVS_int_new:c { split }`

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

(End of definition for `\l__bnvs_tag_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 T_EX group to define local functions and variables, sometimes another grouping level is used. The main T_EX group is closed in the various `\...end_return...` functions.

```

3066 \BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
3067   \BNVS_begin:
3068     \__bnvs_if_resolve:ncTF { #1 } { #2 } {
3069       \BNVS_end_tl_put_right:cv { #2 } { #2 }
3070       \prg_return_true:
3071     } {
3072       \BNVS_end:
3073       \prg_return_false:
3074     }
3075   }
3076 \BNVS_new_conditional_vc:cn { if_append } { T, F, TF }
```

Heavily used.

```

3077 \cs_new:Npn \BNVS_end_unreachable_return_false:n #1 {
3078   \BNVS_error:n { UNREACHABLE/#1 }
3079   \BNVS_end:
3080   \prg_return_false:
3081 }
3082 \cs_new:Npn \BNVS_end_unreachable_return_false:x #1 {
3083   \BNVS_error:x { UNREACHABLE/#1 }
3084   \BNVS_end:
3085   \prg_return_false:
```

```

3086 }
3087 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { TF } {
3088   \__bnvs_if_call:TF {
3089     \BNVS_begin:

```

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

```

3090   \__bnvs_if_regex_split:cnTF { split } { #1 } {

```

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

```

3091     \BNVS_set:cpn { if_resolve_end_return_true: } {

```

Normal and unique end of the loop.

```

3092       \__bnvs_if_resolve_round_ans:
3093       \BNVS_end_tl_set:cv { #2 } { ans }
3094       \prg_return_true:
3095     }

```

Ranges are not rounded: for them \...if_resolve_round_ans: is a noop.

```

3096     \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round:c { ans } }
3097     \__bnvs_tl_clear:c { ans }
3098     \__bnvs_split_loop_or_end_return:
3099   } {

```

There is not reference.

```

3100     \__bnvs_tl_set:cn { ans } { #1 }
3101     \__bnvs_round:c { ans }
3102     \BNVS_end_tl_set:cv { #2 } { ans }
3103     \prg_return_true:
3104   }
3105 } {
3106   \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
3107   \BNVS_end:
3108   \prg_return_false:
3109 }
3110 }
3111 \BNVS_new_conditional_vc:cn { if_resolve } { T, F, TF }
3112 \BNVS_new:cpn { build_tag: } {
3113   \__bnvs_tl_set_eq:cc { tag } { short }
3114   \__bnvs_seq_map_inline:cn { path } {
3115     \__bnvs_tl_put_right:cn { tag } { . ##1 }
3116   }
3117 }
3118 \BNVS_new:cpn { build_tag_head: } {
3119   \__bnvs_tl_set_eq:cc { tag } { short }
3120   \__bnvs_seq_map_inline:cn { path_head } {
3121     \__bnvs_tl_put_right:cn { tag } { . ##1 }
3122   }
3123 }

```

__bnvs_split_loop_or_end_return: __bnvs_split_loop_or_end_return:

Manages the split sequence created by the ...if_resolve_query:... conditional. Entry point. May call itself at the end. The first step is to collect the various information into variables. Then we separate the trailing lowercase components of the path and act accordingly.

```

3124 \clist_map_inline:nn {
3125   n, reset, reset_all, v, first, last, length,
3126   previous, next, range, assign, only
3127 } {
3128   \bool_new:c { l__bnvs_#1_bool }
3129 }

3130 \BNVS_new_conditional:cpnn { if:c } #1 { p, T, F, TF } {
3131   \bool_if:cTF { l__bnvs_#1_bool } {
3132     \prg_return_true:
3133   } {
3134     \prg_return_false:
3135   }
3136 }

3137 \BNVS_new_conditional:cpnn { bool_if_exist:c } #1 { p, T, F, TF } {
3138   \bool_if_exist:cTF { l__bnvs_#1_bool } {
3139     \prg_return_true:
3140   } {
3141     \prg_return_false:
3142   }
3143 }

3144 \BNVS_new:cpn { prepare_context:N } #1 {
3145   \clist_map_inline:nn {
3146     n, v, reset, reset_all, first, last, length,
3147     previous, next, range, assign, only
3148   } {
3149     \__bnvs_set_false:c { ##1 }
3150   }
3151   \__bnvs_seq_clear:c { path_head }
3152   \__bnvs_seq_clear:c { path_tail }
3153   \__bnvs_tl_clear:c { index }
3154   \__bnvs_tl_clear:c { suffix }
3155   \BNVS_set:cpn { :n } ##1 {
3156     \tl_if_blank:nF { ##1 } {
3157       \__bnvs_tl_if_empty:cF { index } {
3158         \__bnvs_seq_put_right:cv { path_head } { index }
3159         \__bnvs_tl_clear:c { index }
3160       }
3161       \__bnvs_seq_put_right:cn { path_head } { ##1 }
3162     }
3163   }
3164   \__bnvs_seq_map_inline:cn { path } {
3165     \__bnvs_bool_if_exist:cTF { ##1 } {
3166       \__bnvs_set_true:c { ##1 }
3167       \clist_if_in:nnF { n, v, reset, reset_all } { ##1 } {
3168         \bool_if:NT #1 {
3169           \BNVS_error:n {Unexpected~##1~in~assignment }
3170         }
3171         \__bnvs_tl_set:cn { suffix } { ##1 }
3172       }
3173       \BNVS_set:cpn { :n } #####1 {
3174         \tl_if_blank:nF { #####1 } {
3175           \BNVS_error:n {Unexpected~#####1 }

```

```

3176     }
3177   }
3178   } {
3179     \regex_match:NnTF \c__bnvs_A_index_Z_regex { ##1 } {
3180       \__bnvs_tl_if_empty:cF { index } {
3181         \__bnvs_seq_put_right:cv { path_head } { index }
3182       }
3183       \__bnvs_tl_set:cn { index } { ##1 }
3184     } {
3185       \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { ##1 } {
3186         \BNVS_error:n { Unsupported-##1 }
3187       } {
3188         \__bnvs_:n { ##1 }
3189       }
3190     }
3191   }
3192 }
3193 \__bnvs_seq_set_eq:cc { path } { path_head }
3194 }

3195 \BNVS_new:cpn { split_loop_or_end_return: } {
3196   \__bnvs_split_if_pop_left:cTF { a } {
3197     \__bnvs_tl_put_right:cv { ans } { a }
3198     \__bnvs_split_pop_iksp:TFF {
3199       \__bnvs_split_end_return_or_pop_void:T {
3200         \__bnvs_prepare_context:N \c_true_bool
3201         \__bnvs_build_tag:
3202         \__bnvs_split_loop_or_end_return_iadd:n { 1 }
3203       }
3204     } {
3205       \__bnvs_split_pop_iksp:TFF {
3206         \__bnvs_split_end_return_or_pop_complete:T {
3207           \__bnvs_tl_if_blank:vTF { n_incr } {
3208             \__bnvs_tl_if_blank:vTF { plus } {
3209               \__bnvs_tl_if_blank:vTF { rhs } {
3210                 \__bnvs_tl_if_blank:vTF { post } {
3211                   \__bnvs_prepare_context:N \c_false_bool
3212                   \__bnvs_build_tag:

```

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

```

3213         \__bnvs_tl_if_empty:cTF { index } {
3214           \__bnvs_tl_if_empty:cTF { suffix } {
3215             \__bnvs_split_loop_or_end_return_v:
3216           } {
3217             \__bnvs_split_loop_or_end_return_suffix:
3218           }
3219         } {
3220           \__bnvs_split_loop_or_end_return_index:
3221         }
3222       } {
3223         \__bnvs_prepare_context:N \c_true_bool
3224         \__bnvs_build_tag:
3225         \BNVS_use:c { split_loop_or_end_return[...++]: }
3226       }
3227     } {

```

```

3228         \__bnvs_prepare_context:N \c_true_bool
3229         \__bnvs_build_tag:
3230         \__bnvs_split_loop_or_end_return_assign:
3231     }
3232 } {
3233     \__bnvs_if_resolve:vcTF { rhs } { rhs } {
3234         \__bnvs_prepare_context:N \c_true_bool
3235         \__bnvs_build_tag:
3236         \BNVS_tl_use:Nv
3237         \__bnvs_split_loop_or_end_return_iadd:n { rhs }
3238     } {
3239         \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs }}
3240         \__bnvs_split_loop_or_end_return:
3241     }
3242 }
3243 } {
3244     \__bnvs_prepare_context:N \c_true_bool
3245     \__bnvs_build_tag:
3246     \__bnvs_set_true:c { n }
3247     \__bnvs_split_loop_or_end_return_iadd:n { 1 }
3248 }
3249 }
3250 } {
3251     \BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/3 }
3252 } {
3253     \BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/2 }
3254 }
3255 } {

```

The split sequence is empty.

```

3256     \__bnvs_if_resolve_end_return_true:
3257 }
3258 } {
3259     \BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/1 }
3260 }
3261 }
3262 \BNVS_new_conditional:cpnn { if_suffix: } { T, F, TF } {
3263     \__bnvs_tl_if_empty:cTF { suffix } {
3264         \__bnvs_seq_pop_right:ccTF { path } { suffix } {
3265             \prg_return_true:
3266         } {
3267             \prg_return_false:
3268         }
3269     } {
3270         \prg_return_true:
3271     }
3272 }

```

Implementation detail: tl variable a is used.

```

3273 \BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {
3274     \__bnvs_if:cTF n {
3275         #1
3276     } {
3277         \__bnvs_build_tag:

```



```

3278     \__bnvs_tl_set:cx { a } {
3279         \BNVS_tl_use:c { tag } . \BNVS_tl_use:c { suffix }
3280     }
3281     \__bnvs_if_resolve_v:vvctf { id } { a } { a } {
3282         \__bnvs_tl_put_right:cv { ans } { a }
3283         \__bnvs_split_loop_or_end_return:
3284     } {
3285         \__bnvs_if_resolve_V:vvctf { id } { a } { a } {
3286             \__bnvs_tl_put_right:cv { ans } { a }
3287             \__bnvs_split_loop_or_end_return:
3288         } {
3289             #1
3290         }
3291     }
3292 }
3293 }

3294 \BNVS_new:cpn { error_end_return_false:n } #1 {
3295     \BNVS_error:n { #1 }
3296     \BNVS_end:
3297     \prg_return_false:
3298 }

3299 \BNVS_new:cpn { path_branch_loop_or_end_return: } {
3300     \__bnvs_if_call:TF {
3301         \__bnvs_if_path_branch:TF {
3302             \__bnvs_path_branch_end_return:
3303         } {
3304             \__bnvs_if_get:nvctf V { id } { tag } { a } {
3305                 \__bnvs_if_TIP:cccTF { id } { a } { path } {
3306                     \__bnvs_tl_set_eq:cc { tag } { a }
3307                     \__bnvs_seq_merge:cc { path } { path_tail }
3308                     \__bnvs_seq_clear:c { path_tail }
3309                     \__bnvs_seq_set_eq:cc { path_head } { path }
3310                     \__bnvs_path_branch_TIPn_loop_or_end_return:
3311                 } {
3312                     \__bnvs_path_branch_head_to_tail_end_return:
3313                 }
3314             } {
3315                 \__bnvs_path_branch_head_to_tail_end_return:
3316             }
3317         }
3318     } {
3319         \__bnvs_path_branch_end_return_false:n {
3320             Too-many-calls.
3321         }
3322     }
3323 }

3324 \BNVS_new:cpn { path_branch_end_return: } {
3325     \__bnvs_split_loop_or_end_return:
3326 }

3327 \BNVS_new:cpn { set_if_path_branch:n } {
3328     \prg_set_conditional:Npnn \__bnvs_if_path_branch: { TF }
3329 }

```

```

3330 \BNVS_new:cpn { path_branch_head_to_tail_end_return: } {
3331   \__bnvs_seq_pop_right:ccTF { path_head } { a } {
3332     \__bnvs_seq_put_left:cv { path_tail } { a }
3333     \__bnvs_build_tag_head:
3334     \__bnvs_path_branch_TIPn_loop_or_end_return:
3335   } {
3336     \__bnvs_build_tag:
3337     \__bnvs_seq_set_eq:cc { path_head } { path_tail }
3338     \__bnvs_seq_clear:c { path_tail }
3339     \__bnvs_gset:nvvn      V { id } { tag } { 0 }
3340     \__bnvs_gset_cache:nvvn V { id } { tag } { 0 }
3341     \__bnvs_path_branch_TIPn_loop_or_end_return:
3342   }
3343 }

```

The a tl variable is used locally. Update the QD variable based on ref and path, then try to resolve it

```

3344 \BNVS_new:cpn { path_branch_TIPn_loop_or_end_return: } {
3345   \__bnvs_build_tag_head:
3346   \__bnvs_if_resolve_v:vvcTF { id } { tag } { a } {
3347     \__bnvs_tl_put_right:cv { ans } { a }
3348     \__bnvs_split_loop_or_end_return:
3349   } {
3350     \__bnvs_if_resolve_V:vvcTF { id } { tag } { a } {
3351       \__bnvs_tl_put_right:cv { ans } { a }
3352       \__bnvs_split_loop_or_end_return:
3353     } {
3354       \__bnvs_path_branch_loop_or_end_return:
3355     }
3356   }
3357 }

```

- Case*<index>*.

```

3358 \BNVS_new:cpn { split_loop_or_end_return_index: } {
3359   % known, id, tag, path, suffix
3360   \__bnvs_set_if_path_branch:n {
3361     \__bnvs_if_append_index:vvcTF { id } { tag } { index } { ans } {
3362       \prg_return_true:
3363     } {
3364       \prg_return_false:
3365     }
3366   }
3367   \__bnvs_path_branch_loop_or_end_return:
3368 }

3369 \BNVS_new:cpn { split_loop_reset: } {
3370   \__bnvs_if:cT { reset_all } {
3371     \__bnvs_set_false:c { reset }
3372     \__bnvs_if_greset_all:vvvT { id } { tag } {} {}
3373   }
3374   \__bnvs_if:cT { reset } {
3375     \BNVS_use:c {
3376       \__bnvs_if:cTF nnv _if_greset:vvvT

```

```

3377     } { id } { tag } {} {}
3378   }
3379 }

```

- Case

```

3380 \BNVS_new:cpn { split_loop_or_end_return_v: } {
3381   \__bnvs_split_loop_reset:
3382   \__bnvs_if:cTF n {
3383     \__bnvs_tl_set_eq:cc { base } { tag }
3384     \__bnvs_set_if_path_branch:n {
3385       \__bnvs_if_resolve_n:vvctf { id } { tag } { index } {
3386         \__bnvs_if_append_index:vvvctf { id } { base } { index } { ans } {
3387           \prg_return_true:
3388           } {
3389             \prg_return_false:
3390           }
3391         } {
3392           \prg_return_false:
3393         }
3394       }
3395     } {
3396       \__bnvs_set_if_path_branch:n {
3397         \__bnvs_if_append_v:vvctf { id } { tag } { ans } {
3398           \prg_return_true:
3399         } {
3400           \__bnvs_if_append_V:vvctf { id } { tag } { ans } {
3401             \prg_return_true:
3402           } {
3403             \prg_return_false:
3404           }
3405         }
3406       }
3407     }
3408   \__bnvs_path_branch_loop_or_end_return:
3409 }

```

- Case<suffix>.

```

3410 \BNVS_new:cpn { split_loop_or_end_return_suffix: } {
3411   \__bnvs_if_resolve_V_loop_or_end_return_true:F {
3412     \__bnvs_if:cTF n {
3413       \__bnvs_tl_set_eq:cc { base } { tag }
3414       \__bnvs_set_if_path_branch:n {
3415         \__bnvs_if_resolve_n:vvctf { id } { tag } { index } {
3416           \__bnvs_if_append_index:vvvctf { id } { base } { index } { ans } {
3417             \prg_return_true:
3418           } {
3419             \prg_return_false:
3420           }
3421         } {
3422           \prg_return_false:
3423         }
3424       }
3425     } {

```

```

3426     \__bnvs_set_if_path_branch:n {
3427         \BNVS_use:c {
3428             if_append_ \__bnvs_tl_use:c { suffix } :vvcTF
3429         } { id } { tag } { ans } {
3430             \__bnvs_if:cT { range } {
3431                 \BNVS_set:cpn { if_resolve_round_ans: } { }
3432             }
3433             \prg_return_true:
3434         } {
3435             \prg_return_false:
3436         }
3437     }
3438 }
3439 \__bnvs_path_branch_loop_or_end_return:
3440 }
3441 }

```

- Case ...++.

```

3442 \BNVS_new:cpn { split_loop_or_end_return[...++]: } {
3443     \__bnvs_if:cTF n {
3444         \__bnvs_if:cTF { reset } {
3445             \cs_set:Npn \BNVS_split_loop: {
3446                 \BNVS_error_ans:x { NO~....reset.n++~for~\BNVS_tl_use:c { tag } }
3447             }
3448         } {
3449             \__bnvs_if:cTF { reset_all } {

```

- Casereset_all.n++.

```

3450         \cs_set:Npn \BNVS_split_loop: {
3451             \BNVS_error_ans:x {
3452                 NO~....reset_all.n++~for
3453                 ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3454             }
3455         }
3456     } {

```

- Casen++.

```

3457         \cs_set:Npn \BNVS_split_loop: {
3458             NO~....n++~for
3459             ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3460         }
3461     }
3462 } {
3463 } {
3464     \__bnvs_if:cTF { reset } {

```

- Casereset++.

```

3465         \cs_set:Npn \BNVS_split_loop: {
3466             NO~....reset++~for
3467             ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3468         }
3469     } {
3470         \__bnvs_if:cTF n {

```

- Casereset_all.n++.

```

3471         \cs_set:Npn \BNVS_split_loop: {
3472             NO~....n(.reset_all)++~for
3473             ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3474         }
3475     } {

```

- Case ...(.reset_all)++.

```

3476         \cs_set:Npn \BNVS_split_loop: {
3477             \BNVS_error_ans:x {
3478                 NO~...(.reset_all)++~for
3479                 ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3480             }
3481         }
3482     }
3483 }
3484 }
3485 \__bnvs_build_tag:
3486 \__bnvs_split_loop_reset:
3487 \BNVS_use:c {
3488     if_append_\__bnvs_if:cTF nnv _post:vvncTF
3489 } { id } { tag } { 1 } { ans } {
3490 } {
3491     \BNVS_error_ans:x {
3492         Problem~with~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }~use.
3493     }
3494 }
3495 \__bnvs_split_loop_or_end_return:
3496 }
3497 \BNVS_new:cpn { split_loop_or_end_return_assign: } {

```

- Case ...=. Resolve the rhs, on success make the assignment and put the result to the right of the ans variable.

```

3498 \__bnvs_if_resolve:vcTF { rhs } { rhs } {
3499     \__bnvs_if:cTF n {
3500         \__bnvs_gset:nvvv n { id } { tag } { rhs }
3501         \__bnvs_if_append_index:vvvcTF { id } { tag } { rhs } { ans } {
3502         } {
3503             \BNVS_error_ans:x { No~....n=... }
3504         }
3505     } {
3506         \__bnvs_gset:nvvv v { id } { tag } { rhs }
3507         \__bnvs_if_append_v:vvvcTF { id } { tag } { ans } {
3508         } {
3509             \BNVS_error_ans:n { No~...=... }
3510         }
3511     }
3512 } {
3513     \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c { rhs }. }
3514 }
3515 \__bnvs_split_loop_or_end_return:
3516 }

```

- Case ...+=....

```

3517 \BNVS_new:cpn { split_loop_or_end_return_iadd:n } #1 {
3518   \__bnvs_if_resolve:ncTF { #1 } { rhs } {
3519     \__bnvs_split_loop_reset:
3520     \BNVS_use:c {
3521       if_append_ \__bnvs_if:cTF nnv _incr:vvncTF
3522     } { id } { tag } { #1 } { ans } {
3523     } {
3524       \BNVS_error_ans:n { No~...+=... }
3525     }
3526   } {
3527     \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs } }
3528   }
3529   \__bnvs_split_loop_or_end_return:
3530 }

```

```
\_bnvs_if_resolve_query:ncTF \_bnvs_if_resolve_query:ncTF {\langle overlay query \rangle} {\langle ans \rangle} {\langle yes code \rangle} {\langle no
code \rangle}
```

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

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

(End of definition for `\l__bnvs_V_tl`.)

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

(End of definition for `\l__bnvs_A_tl`.)

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

(End of definition for `\l__bnvs_Z_tl`.)

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

(End of definition for `\l__bnvs_L_tl`.)

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

(End of definition for `\c__bnvs_A_cln_Z_regex`.)

```
3531 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
3532   \A \s* (?
    • 2  $\rightarrow V$ 
3533     ( [^:]+? )
    • 3, 4, 5  $\rightarrow A : Z?$  or  $A :: L?$ 
3534     | (? : ( [^:]+? ) \s* : (? : \s* ( [^:]*? ) | : \s* ( [^:]*? ) ) )
    • 6, 7  $\rightarrow ::(L:Z)?$ 
3535     | (? : :: \s* (? : ( [^:]+? ) \s* : \s* ( [^:]+? ) )? )
    • 8, 9  $\rightarrow :(Z::L)?$ 
3536     | (? : : \s* (? : ( [^:]+? ) \s* :: \s* ( [^:]*? ) )? )
3537   )
3538   \s* \Z
3539 }
```

```
3540 \BNVS_new:cpn { resolve_query_end_return_true: } {
3541   \BNVS_end:
3542   \prg_return_true:
3543 }
```

```
3544 \BNVS_new:cpn { resolve_query_end_return_false: } {
```

```

3545 \BNVS_end:
3546 \prg_return_false:
3547 }

3548 \BNVS_new:cpn { resolve_query_end_return_false:n } #1 {
3549 \BNVS_end:
3550 \prg_return_false:
3551 }

3552 \BNVS_new:cpn { if_resolve_query_return_false:n } #1 {
3553 \prg_return_false:
3554 }

3555 \BNVS_new:cpn { resolve_query_error_return_false:n } #1 {
3556 \BNVS_error:n { #1 }
3557 \__bnvs_if_resolve_query_return_false:
3558 }
3559 \BNVS_generate_variant:cn { resolve_query_error_return_false:n } { x }

3560 \BNVS_new:cpn { if_resolve_query_return_unreachable: } {
3561 \__bnvs_resolve_query_error_return_false:n { UNREACHABLE }
3562 }

3563 \BNVS_new:cpn { if_blank:cTF } #1 {
3564 \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3565 }

3566 \BNVS_new_conditional:cpnn { if_match_pop_left:c } #1 { T, F, TF } {
3567 \BNVS_tl_use:nc {
3568 \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
3569 } { #1 } {
3570 \prg_return_true:
3571 } {
3572 \prg_return_false:
3573 }
3574 }

```

__bnvs_if_resolve_query_branch:TF __bnvs_if_resolve_query_branch:TF {<yes code>} {<no code>}

Called by __bnvs_if_resolve_query:ncTF that just filled \l__bnvs_match_seq after the c__bnvs_A_cln_Z_regex. Puts the proper items of \l__bnvs_match_seq into the variables \l__bnvs_V_tl, \l__bnvs_A_tl, \l__bnvs_Z_tl, \l__bnvs_L_tl then branches accordingly on one of the returning

__bnvs_if_resolve_query_return[<description>]:

functions. All these functions properly set the \l__bnvs_ans_tl variable and they end with either \prg_return_true: or \prg_return_false:. This is used only once but is not inlined for readability.

```

3575 \BNVS_new_conditional:cpnn { if_resolve_query_branch: } { T, F, TF } {
At start, we ignore the whole match.
3576 \__bnvs_if_match_pop_left:cT V {
3577 \__bnvs_if_match_pop_left:cT V {
3578 \__bnvs_if_blank:cTF V {
3579 \__bnvs_if_match_pop_left:cT A {
3580 \__bnvs_if_match_pop_left:cT Z {
3581 \__bnvs_if_match_pop_left:cT L {

```



```

3582         \_bnvs_if_blank:cTF A {
3583             \_bnvs_if_match_pop_left:cT L {
3584                 \_bnvs_if_match_pop_left:cT Z {
3585                     \_bnvs_if_blank:cTF L {
3586                         \_bnvs_if_match_pop_left:cT Z {
3587                             \_bnvs_if_match_pop_left:cT L {
3588                                 \_bnvs_if_blank:cTF L {
3589                                     \BNVS_use:c { if_resolve_query_return[:Z]: }
3590                                 } {
3591                                     \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3592                                 }
3593                             }
3594                         }
3595                     } {
3596                         \_bnvs_if_blank:cTF Z {
3597         \_bnvs_resolve_query_error_return_false:n { Missing-first~or-last }
3598             } {
3599                 \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3600             }
3601         }
3602     }
3603 }
3604 } {
3605     \_bnvs_if_blank:cTF Z {
3606         \_bnvs_if_blank:cTF L {
3607             \BNVS_use:c { if_resolve_query_return[A:]: }
3608         } {
3609             \BNVS_use:c { if_resolve_query_return[A::L]: }
3610         }
3611     } {
3612         \_bnvs_if_blank:cTF L {
3613             \BNVS_use:c { if_resolve_query_return[A:Z]: }
3614         } {
3615             \_bnvs_if_resolve_query_return_unreachable:
3616         }
3617     }
3618 }
3619 }
3620 }
3621 }
3622 } {
3623     \BNVS_use:c { if_resolve_query_return[V]: }
3624 }
3625 }
3626 }
3627 }

```

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

Single value

```

3628 \BNVS_new:cpn { if_resolve_query_return[V]: } {
3629     \_bnvs_if_resolve:vcTF { V } { ans } {
3630         \prg_return_true:
3631     } {

```

```

3632     \prg_return_false:
3633   }
3634 }

☛ <first>:<last> range
3635 \BNVS_new:cpn { if_resolve_query_return[A:Z]: } {
3636   \__bnvs_if_resolve:vcTF { A } { ans } {
3637     \__bnvs_tl_put_right:cn { ans } { - }
3638     \__bnvs_if_append:vcTF { Z } { ans } {
3639       \prg_return_true:
3640     } {
3641       \prg_return_false:
3642     }
3643   } {
3644     \prg_return_false:
3645   }
3646 }

☛ <first>::<length> range
3647 \BNVS_new:cpn { if_resolve_query_return[A::L]: } {
3648   \__bnvs_if_resolve:vcTF { A } { A } {
3649     \__bnvs_if_resolve:vcTF { L } { ans } {
3650       \__bnvs_tl_put_right:cn { ans } { + }
3651       \__bnvs_tl_put_right:cv { ans } { A }
3652       \__bnvs_tl_put_right:cn { ans } { -1 }
3653       \__bnvs_round:c { ans }
3654       \__bnvs_tl_put_left:cn { ans } { - }
3655       \__bnvs_tl_put_left:cv { ans } { A }
3656       \prg_return_true:
3657     } {
3658       \prg_return_false:
3659     }
3660   } {
3661     \prg_return_false:
3662   }
3663 }

☛ <first>: and <first>:: range
3664 \BNVS_new:cpn { if_resolve_query_return[A:]: } {
3665   \__bnvs_if_resolve:vcTF { A } { ans } {
3666     \__bnvs_tl_put_right:cn { ans } { - }
3667     \prg_return_true:
3668   } {
3669     \prg_return_false:
3670   }
3671 }

☛ :<last>::<length> or ::<length>:<last> range
3672 \BNVS_new:cpn { if_resolve_query_return[:Z::L]: } {
3673   \__bnvs_if_resolve:vcTF { Z } { Z } {
3674     \__bnvs_if_resolve:vcTF { L } { ans } {
3675       \__bnvs_tl_put_left:cn { ans } { 1- }
3676       \__bnvs_tl_put_right:cn { ans } { + }
3677       \__bnvs_tl_put_right:cv { ans } { Z }
3678       \__bnvs_round:c { ans }

```

```

3679     \_bnvs_tl_put_right:cn { ans } { - }
3680     \_bnvs_tl_put_right:cv { ans } { Z }
3681     \prg_return_true:
3682   } {
3683     \prg_return_false:
3684   }
3685 } {
3686   \prg_return_false:
3687 }
3688 }

☛ : or :: range

3689 \BNVS_new:cpn { if_resolve_query_return[:]: } {
3690   \_bnvs_tl_set:cn { ans } { - }
3691   \prg_return_true:
3692 }

☛ : <last> range

3693 \BNVS_new:cpn { if_resolve_query_return[:Z]: } {
3694   \_bnvs_tl_set:cn { ans } { - }
3695   \_bnvs_if_append:vcTF { Z } { ans } {
3696     \prg_return_true:
3697   } {
3698     \prg_return_false:
3699   }
3700 }

```

_bnvs_if_resolve_query:ncTF _bnvs_if_resolve_query:ncTF {<query>} {<tl core>} {<yes code>} {<no code>}

Evaluate only one query.

```

3701 \BNVS_new_conditional:cpnn { if_resolve_query:nc } #1 #2 { T, F, TF } {
3702   \_bnvs_greset_call:
3703   \_bnvs_match_if_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
3704     \BNVS_begin:
3705     \_bnvs_if_resolve_query_branch:TF {
3706       \BNVS_end_tl_set:cv { #2 } { ans }
3707       \prg_return_true:
3708     } {
3709       \BNVS_end:
3710       \prg_return_false:
3711     }
3712   } {
3713     \BNVS_error:n { Syntax~error:~#1 }
3714     \BNVS_end:
3715     \prg_return_false:
3716   }
3717 }

```

```

\__bnvs_if_resolve_queries:ncTF \__bnvs_if_resolve_queries:ncTF {<overlay query list>} {<ans>} {<yes
code>} {<no code>}}

```

This is called by the *named overlay specifications* scanner. Evaluates the comma separated *<overlay query list>*, replacing all the individual named overlay specifications and integer expressions by their static counterparts by calling `__bnvs_if_resolve_query:ncTF`, then append the result to the right of the *<ans>* t1 variable . This is executed within a local group. Below are local variables and constants used throughout the body of this function.

`\l__bnvs_query_seq` Storage for a sequence of *<query>*'s obtained by splitting a comma separated list.

(End of definition for `\l__bnvs_query_seq`.)

`\l__bnvs_ans_seq` Storage for the evaluated result.

(End of definition for `\l__bnvs_ans_seq`.)

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

```

3718 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End of definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

3719 \BNVS_new_conditional:cpnn { if_resolve_queries:nc } #1 #2 { TF } {
3720 \BNVS_begin:

```

Local variables cleared

```

3721 \__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`.

```

3722 \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 just cleared above.

```

3723 \BNVS_set:cpn { end_return: } {
3724 \__bnvs_seq_if_empty:cTF { ans } {
3725 \BNVS_end:
3726 } {
3727 \exp_args:Nnx
3728 \use:n {
3729 \BNVS_end:
3730 \__bnvs_tl_put_right:cn { #2 }
3731 } { \__bnvs_seq_use:cn { ans } , }
3732 }
3733 \prg_return_true:
3734 }
3735 \__bnvs_seq_map_inline:cn { query } {
3736 \__bnvs_tl_clear:c { ans }
3737 \__bnvs_if_resolve_query:ncTF { ##1 } { ans } {
3738 \__bnvs_tl_if_empty:cF { ans } {
3739 \__bnvs_seq_put_right:cv { ans } { ans }
3740 }

```

```

3741 } {
3742   \seq_map_break:n {
3743     \BNVS_set:cpn { end_return: } {
3744       \BNVS_error:n { Circular/Undefined~dependency~in~#1}
3745       \exp_args:Nnx
3746       \use:n {
3747         \BNVS_end:
3748         \__bnvs_tl_put_right:cn { #2 }
3749       } { \__bnvs_seq_use:cn { ans } , }
3750       \prg_return_false:
3751     }
3752   }
3753 }
3754 }
3755 \__bnvs_end_return:

```

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

```

3756 }

3757 \NewDocumentCommand \BeanovesResolve { 0{} m } {
3758   \BNVS_begin:
3759   \keys_define:nn { BeanovesResolve } {
3760     in:N .tl_set:N = \l__bnvs_resolve_in_tl,
3761     in:N .initial:n = { },
3762     show .bool_set:N = \l__bnvs_resolve_show_bool,
3763     show .default:n = true,
3764     show .initial:n = false,
3765   }
3766   \keys_set:nn { BeanovesResolve } { #1 }
3767   \__bnvs_tl_clear:c { ans }
3768   \__bnvs_if_resolve_queries:ncTF { #2 } { ans } {
3769     \__bnvs_tl_if_empty:cTF { resolve_in } {
3770       \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3771         \BNVS_tl_use:Nv \BNVS_end: { ans }
3772       } {
3773         \BNVS_end:
3774       }
3775     } {
3776       \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3777         \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3778           \BNVS_end:
3779           \tl_set:Nn ##1 { ##2 }
3780           ##2
3781         }
3782         \BNVS_tl_use:nv {
3783           \exp_last_unbraced:Nv \BNVS_end:Nn \l__bnvs_resolve_in_tl
3784         } { ans }
3785       } {
3786         \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3787           \BNVS_end:
3788           \tl_set:Nn ##1 { ##2 }
3789         }
3790         \BNVS_tl_use:nv {

```

```

3791         \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
3792     } { ans }
3793 }
3794 }
3795 } {}
3796 }

```

6.22 Resetting counters and values

```

3797 \BNVS_new:cpn { reset:n } #1 {
3798   \BNVS_begin:
3799   \__bnvs_set_true:c { reset }
3800   \__bnvs_set_false:c { provide }
3801   \__bnvs_tl_clear:c { root }
3802   \__bnvs_int_zero:c { i }
3803   \__bnvs_tl_set:cn { a } { #1 }
3804   \__bnvs_provide_off:
3805   \BNVS_tl_use:Nv \__bnvs_keyval_named:n { a }
3806   \BNVS_end_tl_set:cv { id_last } { id_last }
3807 }

3808 \BNVS_new:cpn { reset:v } {
3809   \BNVS_tl_use:Nv \__bnvs_reset:n
3810 }

3811 \makeatletter
3812 \NewDocumentCommand \BeanovesReset { 0{} m } {
3813   \tl_if_empty:NTF \@currentvir {

```

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

```

3814   \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}}~in~the~preamble.}
3815 } {
3816   \tl_if_eq:NnT \@currentvir { document } {

```

At the top level, clear everything.

```

3817   \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}}~at~the~top~level.}
3818 }
3819 \BNVS_begin:
3820 \__bnvs_set_true:c { reset }
3821 \__bnvs_set_false:c { provide }
3822 \keys_define:nn { BeanovesReset } {
3823   all .bool_set:N = \l__bnvs_reset_all_bool,
3824   all .default:n = true,
3825   all .initial:n = false,
3826   only .bool_set:N = \l__bnvs_only_bool,
3827   only .default:n = true,
3828   only .initial:n = false,
3829 }
3830 \keys_set:nn { BeanovesReset } { #1 }
3831 \__bnvs_tl_clear:c { root }
3832 \__bnvs_int_zero:c { i }
3833 \__bnvs_tl_set:cn { a } { #2 }

```

```
3834     \__bnvs_provide_off:
3835     \BNVS_tl_use:Nv \__bnvs_keyval_named:n { a }
3836     \BNVS_end_tl_set:cv { id_last } { id_last }
3837     \ignorespaces
3838   }
3839 }
3840 \makeatother
3841 \ExplSyntaxOff
```