

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	2
1.1	Package manager	2
1.2	Manual installation	2
1.3	Usage	3
2	Minimal example	3
3	Named overlay sets	4
3.1	Presentation	4
3.2	Named overlay reference	4
3.3	Defining named overlay sets	4
3.3.1	Basic specifiers	5
3.3.2	List specifiers	5
4	Resolution of <code>?(...)</code> query expressions	6
4.1	Number and range overlay queries	6
4.2	Counters	7
4.3	Dotted paths	9
4.4	The <code>beamerpauses</code> counter	9
4.5	Multiple queries	9
4.6	Frame id	9
4.7	Resolution command	10
5	Support	10

6	Implementation	10
6.1	Package declarations	10
6.2	Facility layer: definitions and naming	10
6.3	logging	12
6.4	Facility layer: Variables	13
6.4.1	Regex	18
6.4.2	Token lists	20
6.4.3	Strings	24
6.4.4	Sequences	25
6.4.5	Integers	26
6.4.6	Prop	27
6.5	Debug facilities	27
6.6	Debug messages	27
6.7	Testing facilities	27
6.8	Local variables	27
6.9	Infinite loop management	29
6.10	Overlay specification	29
6.10.1	Basic functions	29
6.10.2	Functions with cache	32
6.11	Implicit value counter	34
6.12	Implicit index counter	37
6.13	Regular expressions	39
6.14	beamer.cls interface	41
6.15	Defining named slide ranges	42
6.16	Scanning named overlay specifications	54
6.17	Resolution	58
6.18	Evaluation bricks	60
6.19	Index counter	74
6.20	Value counter	76
6.21	Functions for the resolution	81
6.22	Resetting counters	101

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:

- T_EX Live: `tlmgr install beanoves`
- MiK_TE_X: `mpm --admin --install=beanoves`

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 *full names* like `A.1` and `C.2`,

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

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

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 $\langle \text{ref} \rangle$ key is used multiple times, only the last one is taken into account.

Notice that $\langle \text{ref}_i \rangle = 1$ can be shortened to $\langle \text{ref}_i \rangle$. 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 $\langle first \rangle$.

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

$\langle first \rangle : \langle last \rangle$, $\langle first \rangle :: \langle length \rangle$, $: \langle last \rangle :: \langle length \rangle$, $:: \langle length \rangle : \langle last \rangle$, 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 $\langle first \rangle + \langle length \rangle = \langle last \rangle + 1$.

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. The definition

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

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

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

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

$\dots,$

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

The rules above can apply individually to each line. The $\langle ref \rangle$ counter defined below is left unmodified.

To support an array like syntax, we can omit the $\langle path \rangle$ key and only give the $\langle spec \rangle$ value. Each missing $\langle path \rangle$ key is replaced by the smallest index $\langle j \rangle$ such that $\langle ref \rangle . \langle j \rangle$ is not already defined and $\langle j \rangle \geq 1$.

Notice that you can replace each opening pair $\{ \{$ by a single $[$ and each closing pair $\} \}$ by a single $]$. Anyway, delimiters should be properly balanced.

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 $\langle \textit{ref} \rangle$, like X after $\text{X}=1$, it is simply replaced by the corresponding declared $\langle \textit{value} \rangle$. Otherwise, we use *implicit overlay queries* and their *resolution rules* depending on the definition of the named overlay set. Here $\langle i \rangle$ 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

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

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

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

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

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

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

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

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

For a range limited from both above and below:

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

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

In the frame example below, we use the `\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 \insertframenum} {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 $?(A.\text{next})$, $?(A.\text{last}+1)$, $?(A.1+A.\text{length})$ 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.

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 $\langle \text{name} \rangle$ *overlay query* is resolved into the position of this value counter. For each frame, this variable is initialized to the first available resolution amongst $\langle \text{value} \rangle$, $\langle \text{name} \rangle . \text{first}$, $\langle \text{name} \rangle . 1$ or $\langle \text{name} \rangle . \text{last}$. If none is available, the counter is initialized to 1.

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

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

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

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

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

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

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

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

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

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

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

\BeanovesReset [*options*] { $\langle \text{ref}_1 \rangle = \langle \text{spec}_1 \rangle$, $\langle \text{ref}_2 \rangle = \langle \text{spec}_2 \rangle$, ..., $\langle \text{ref}_j \rangle = \langle \text{spec}_j \rangle$ }

This command is very similar to **\Beanoves**, except that a standalone $\langle \text{ref}_i \rangle$ 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 it does not correspond to a definition or an assignment, the longest qualified full 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`. Inside a `frame` environment, when the instruction `\Beanoves{\langle ref \rangle=pauses}` is executed, it saves the current value of the `beamer` pauses counter into the $\langle ref \rangle$ counter. Later on, $?(\langle ref \rangle)$ can refer to this value.

4.4 The beamerpauses counter

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 ref \rangle=pauses}` or in a query $?(\dots \langle ref \rangle = \text{pauses}) \dots$. Then later on, we can use $?(\dots \langle ref \rangle) \dots$ 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.

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 defaults to `?`. Then it gets updated after each named reference resolution. For example, the first time `A.1` reference is resolved within a given frame, it is first translated to $\langle current\ frame\ id \rangle!A.1$, but when used just after `Y!C.2`, for example, it becomes a shortcut to `Y!A.1` because the *last frame id* is then `Y`.

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

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

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

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

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 \@@=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_t1` into `\l__bnvs_ref_t1`, `bnvs` will directly transform `ref` into `\l__bnvs_ref_t1`. The type of the local variable used depends on the context and may be `seq` or `int` for example. There are however a pair of exceptions mentionned below. For a better reading experience, ‘`ref`’ will generally stand for `\l__bnvs_ref_t1`, whereas ‘`path sequence`’ will generally stand for `\l__bnvs_path_seq`. Other similar shortcuts are used as well.

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

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

These are naming functions.

```

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

```

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

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

```

21 \cs_new:Npn \BNVS_use_raw:N #1 { #1 }
22 \cs_new:Npn \BNVS_use_raw:c #1 {
23   \exp_last_unbraced:No
24   \BNVS_use_raw:N { \cs:w #1 \cs_end: }
25 }
26 \cs_new:Npn \BNVS_use:c #1 {
27   \BNVS_use_raw:c { \BNVS:c { #1 } }
28 }
29 \cs_new:Npn \BNVS_use_raw:NN #1 #2 {
30   #1 #2
31 }
32 \cs_new:Npn \BNVS_use_raw:nN #1 #2 {
33   #1 #2
34 }
35 \cs_new:Npn \BNVS_use_raw:Nc #1 #2 {
36   \exp_last_unbraced:NNo
37   \BNVS_use_raw:NN #1 { \cs:w #2 \cs_end: }
38 }
39 \cs_new:Npn \BNVS_use_raw:nc #1 #2 {
40   \exp_last_unbraced:Nno
41   \BNVS_use_raw:nN { #1 } { \cs:w #2 \cs_end: }
42 }
43 \cs_new:Npn \BNVS_use:Nc #1 #2 {
44   \BNVS_use_raw:Nc #1 { \BNVS:c { #2 } }
45 }
46 \cs_new:Npn \BNVS_use:nc #1 #2 {
47   \BNVS_use_raw:nc { #1 } { \BNVS:c { #2 } }
48 }

```

```

39 \cs_new:Npn \BNVS_log:n #1 { }
40 \cs_generate_variant:Nn \BNVS_log:n { x }

41 \cs_new:Npn \BNVS_DEBUG_on: {
42   \cs_set:Npn \BNVS_DEBUG_log:n { \BNVS_log:n }
43 }

44 \cs_new:Npn \BNVS_DEBUG_off: {
45   \cs_set:Npn \BNVS_DEBUG_log:n { \use_none:n }
46 }
47 \BNVS_DEBUG_off:

```

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

```

48 \cs_new:Npn \BNVS_new:cpn #1 {
49   \cs_new:cpn { \BNVS:c { #1 } }
50 }

51 \cs_new:Npn \BNVS_set:cpn #1 {
52   \cs_set:cpn { \BNVS:c { #1 } }
53 }

54 \cs_generate_variant:Nn \cs_generate_variant:Nn { c }
55 \cs_new:Npn \BNVS_generate_variant:cn #1 {
56   \cs_generate_variant:cn { \BNVS:c { #1 } }
57 }

```

6.3 logging

Utility messaging.

```

58 \msg_new:nnn { beanoves } { :n } { #1 }
59 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }

60 \cs_new:Npn \BNVS_warning:n {
61   \msg_warning:nnn { beanoves } { :n }
62 }
63 \cs_new:Npn \BNVS_warning:x {
64   \msg_warning:nnx { beanoves } { :n }
65 }

66 \cs_new:Npn \BNVS_error:n {
67   \msg_error:nnn { beanoves } { :n }
68 }
69 \cs_new:Npn \BNVS_error:x {
70   \msg_error:nnx { beanoves } { :n }
71 }

72 \cs_new:Npn \BNVS_fatal:n {
73   \msg_fatal:nnn { beanoves } { :n }
74 }
75 \cs_new:Npn \BNVS_fatal:x {
76   \msg_fatal:nnx { beanoves } { :n }
77 }

```

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`...

```

78 \cs_new:Npn \BNVS_N_new:c #1 {
79   \cs_new:cpn { BNVS_#1:c } ##1 {
80     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
81   }
82   \cs_new:cpn { BNVS_#1_new:c } ##1 {
83     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
84   }
85   \cs_new:cpn { BNVS_#1_use:c } ##1 {
86     \use:c { \use:c { BNVS_#1:c } { ##1 } }
87   }
88   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
89     \BNVS_use_raw:Nc
90     ##1 { \use:c { BNVS_#1:c } { ##2 } }
91   }
92   \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
93     \BNVS_use_raw:nc
94     { ##1 } { \use:c { BNVS_#1:c } { ##2 } }
95   }
96 }

97 \cs_new:Npn \BNVS_v_new:c #1 {
98   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
99     \BNVS_use_raw:nc
100     { \exp_args:Nv ##1 }
101     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
102   }
103   \cs_new:cpn { BNVS_#1_use:cv } ##1 ##2 {
104     \BNVS_use_raw:nc
105     { \exp_args:NnV \BNVS_use:c { ##1 } }
106     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
107   }
108   \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
109     \BNVS_use_raw:nc
110     { \exp_args:NnV \use:n { ##1 } }
111     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
112   }
113 }

114 \BNVS_N_new:c { bool }
115 \BNVS_N_new:c { int }
116 \BNVS_v_new:c { int }
117 \BNVS_N_new:c { tl }
118 \BNVS_v_new:c { tl }
119 \BNVS_N_new:c { str }
120 \BNVS_v_new:c { str }
121 \BNVS_N_new:c { seq }
122 \BNVS_v_new:c { seq }
123 \cs_undefine:N \BNVS_N_new:c

```

\BNVS_use:Ncn \BNVS_use:Ncn $\langle function \rangle$ $\{\langle core name \rangle\}$ $\{\langle type \rangle\}$

```

124 \cs_new:Npn \BNVS_use:Ncn #1 #2 #3 {
125   \BNVS_use_raw:c { BNVS_#3_use:Nc }   #1   { #2 }
126 }

127 \cs_new:Npn \BNVS_use:ncn #1 #2 #3 {
128   \BNVS_use_raw:c { BNVS_#3_use:nc } { #1 } { #2 }
129 }

130 \cs_new:Npn \BNVS_use:Nvn #1 #2 #3 {
131   \BNVS_use_raw:c { BNVS_#3_use:Nv }   #1   { #2 }
132 }

133 \cs_new:Npn \BNVS_use:nvn #1 #2 #3 {
134   \BNVS_use_raw:c { BNVS_#3_use:nv } { #1 } { #2 }
135 }

136 \cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
137   \BNVS_use:ncn {
138     \BNVS_use:Ncn   #1   { #2 } { #3 }
139   }
140 }

141 \cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
142   \BNVS_use:ncn {
143     \BNVS_use:ncn { #1 } { #2 } { #3 }
144   }
145 }

146 \cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
147   \BNVS_use:ncn {
148     \BNVS_use:Nvn   #1   { #2 } { #3 }
149   }
150 }

151 \cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
152   \BNVS_use:ncn {
153     \BNVS_use:nvn { #1 } { #2 } { #3 }
154   }
155 }

156 \cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
157   \BNVS_use:ncn {
158     \BNVS_use:Ncncn   #1   { #2 } { #3 } { #4 } { #5 }
159   }
160 }

161 \cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
162   \BNVS_use:ncn {
163     \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
164   }
165 }

```

\BNVS_new_c:cn \BNVS_new_c:nc $\{\langle type \rangle\}$ $\{\langle core name \rangle\}$

```

166 \cs_new:Npn \BNVS_new_c:nc #1 #2 {
167   \BNVS_new:cpn { #1_#2:c } {
168     \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
169   }
170 }

171 \cs_new:Npn \BNVS_new_cn:nc #1 #2 {
172   \BNVS_new:cpn { #1_#2:cn } ##1 {
173     \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
174   }
175 }

176 \cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
177   \BNVS_new:cpn { #2:cnn } ##1 {
178     \BNVS_use:Ncn { #3 } { ##1 } { #1 }
179   }
180 }

181 \cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
182   \BNVS_use_raw:nc {
183     \BNVS_new_cnn:ncN { #1 } { #1_#2 }
184   } { #1_#2:Nnn }
185 }

186 \cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
187   \BNVS_new:cpn { #2:cnv } ##1 ##2 {
188     \BNVS_tl_use:nv {
189       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
190     }
191   }
192 }

193 \cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
194   \BNVS_use_raw:nc {
195     \BNVS_new_cnv:ncN { #1 } { #1_#2 }
196   } { #1_#2:Nnn }
197 }

198 \cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
199   \BNVS_new:cpn { #2:cnx } ##1 ##2 {
200     \exp_args:Nnx \use:n {
201       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
202     }
203   }
204 }

205 \cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
206   \BNVS_use_raw:nc {
207     \BNVS_new_cnx:ncN { #1 } { #1_#2 }
208   } { #1_#2:Nnn }
209 }

210 \cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
211   \BNVS_new:cpn { #2:cc } ##1 ##2 {
212     \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
213   }
214 }

```

```

215 \cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
216   \BNVS_use_raw:nc {
217     \BNVS_new_cc:ncNn { #1 } { #1_#2 }
218   } { #1_#2:NN }
219 }

220 \cs_new:Npn \BNVS_new_cc:nc #1 #2 {
221   \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
222 }

223 \cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
224   \BNVS_new_cpn { #2:cn } ##1 {
225     \BNVS_use:Ncn #3 { ##1 } { #1 }
226   }
227 }

228 \cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
229   \BNVS_use_raw:nc {
230     \BNVS_new_cn:ncNn { #1 } { #1_#2 }
231   } { #1_#2:Nn }
232 }

233 \cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
234   \BNVS_new_cpn { #2:cv } ##1 ##2 {
235     \BNVS_use:nvn {
236       \BNVS_use:Ncn #3 { ##1 } { #1 }
237     } { ##2 } { #4 }
238   }
239 }

240 \cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
241   \BNVS_use_raw:nc {
242     \BNVS_new_cv:ncNn { #1 } { #1_#2 }
243   } { #1_#2:Nn }
244 }

245 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
246   \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
247 }

248 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
249   \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
250 }

251 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
252   \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
253 }

254 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
255   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
256 }

257 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
258   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
259 }

260 \cs_new:Npn \BNVS_g_prop_use:Nc #1 #2 {
261   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { prop } }
262 }

```



```

263 \cs_new:Npn \BNVS_g_prop_use:nc #1 #2 {
264   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { prop } }
265 }

266 \cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
267   \BNVS_use:ncncncn { \exp_args:NVVV #1 }
268   { #2 } { t1 } { #3 } { t1 } { #4 } { t1 }
269 }

```

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

```

270 \cs_generate_variant:Nn \prg_new_conditional:Npnn { c }

271 \cs_new:Npn \BNVS_new_conditional:cpnn #1 {
272   \prg_new_conditional:cpnn { \BNVS:c { #1 } }
273 }

274 \cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
275 \cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
276   \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
277 }

278 \cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
279   \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
280     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
281       \prg_return_true:
282     } {
283       \prg_return_false:
284     }
285   }
286 }

287 \cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
288   \BNVS_use:nc {
289     \BNVS_new_conditional_vn:cNnn { #1 }
290   } { #1:nn TF } { #2 }
291 }

292 \cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
293   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
294     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
295       \prg_return_true:
296     } {
297       \prg_return_false:
298     }
299   }
300 }

301 \cs_new:Npn \BNVS_new_conditional_vc:cnn #1 {
302   \BNVS_use:nc {
303     \BNVS_new_conditional_vc:cNnn { #1 }
304   } { #1:ncTF }
305 }

```

```

306 \cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
307   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
308     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
309       \prg_return_true:
310     } {
311       \prg_return_false:
312     }
313   }
314 }

315 \cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
316   \BNVS_use:nc {
317     \BNVS_new_conditional_vc:cNn { #1 }
318   } { #1:ncTF }
319 }

```

6.4.1 Regex

```

320 \cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
321   \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
322 }

```

<u>_bnvs_match_if_once:NnTF</u>	_bnvs_match_if_once:NnTF <regex variable> {<expression>}
<u>_bnvs_match_if_once:NvTF</u>	{<yes code>} {<no code>}
<u>_bnvs_match_if_once:nnTF</u>	_bnvs_match_if_once:nnTF {<regex>} {<expression>}
<u>_bnvs_if_regex_split:cnTF</u>	{<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

```

323 \BNVS_new_conditional:cpnn { if_extract_once:Ncn } #1 #2 #3 { T, F, TF } {
324   \BNVS_use:ncn {
325     \regex_extract_once:NnNTF #1 { #3 }
326   } { #2 } { seq } {
327     \prg_return_true:
328   } {
329     \prg_return_false:
330   }
331 }

332 \BNVS_new_conditional:cpnn { match_if_once:Nn } #1 #2 { T, F, TF } {
333   \BNVS_use:ncn {
334     \regex_extract_once:NnNTF #1 { #2 }
335   } { match } { seq } {

```

```

336     \prg_return_true:
337   } {
338     \prg_return_false:
339   }
340 }

341 \BNVS_new_conditional:cpnn { if_extract_once:Ncv } #1 #2 #3 { T, F, TF } {
342   \BNVS_seq_use:nc {
343     \BNVS_tl_use:nv {
344       \regex_extract_once:NnNTF #1
345     } { #3 }
346   } { #2 } {
347     \prg_return_true:
348   } {
349     \prg_return_false:
350   }
351 }

352 \BNVS_new_conditional:cpnn { match_if_once:Nv } #1 #2 { T, F, TF } {
353   \BNVS_seq_use:nc {
354     \BNVS_tl_use:nv {
355       \regex_extract_once:NnNTF #1
356     } { #2 }
357   } { match } {
358     \prg_return_true:
359   } {
360     \prg_return_false:
361   }
362 }

363 \BNVS_new_conditional:cpnn { match_if_once:nn } #1 #2 { T, F, TF } {
364   \BNVS_seq_use:nc {
365     \regex_extract_once:nnNTF { #1 } { #2 }
366   } { match } {
367     \prg_return_true:
368   } {
369     \prg_return_false:
370   }
371 }

372 \BNVS_new_conditional:cpnn { if_regex_split:cnc } #1 #2 #3 { T, F, TF } {
373   \BNVS_seq_use:nc {
374     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
375   } { #3 } {
376     \prg_return_true:
377   } {
378     \prg_return_false:
379   }
380 }

381 \BNVS_new_conditional:cpnn { if_regex_split:cn } #1 #2 { T, F, TF } {
382   \BNVS_seq_use:nc {
383     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
384   } { split } {
385     \prg_return_true:
386   } {
387     \prg_return_false:

```

```

388 }
389 }

```

6.4.2 Token lists

<u>_bnvs_tl_clear:c</u>	_bnvs_tl_clear:c {<core key tl>}
_bnvs_tl_use:c	_bnvs_tl_use:c {<core>}
_bnvs_tl_set_eq:cc	_bnvs_tl_count:c {<core>}
_bnvs_tl_set:cn	_bnvs_tl_set_eq:cc {<lhs core name>} {<rhs core name>}
_bnvs_tl_set:(cv cx)	_bnvs_tl_set:cn {<core>} {<tl>}
_bnvs_tl_put_left:cn	_bnvs_tl_set:cv {<core>} {<value core name>}
_bnvs_tl_put_right:cn	_bnvs_tl_put_left:cn {<core>} {<tl>}
<u>_bnvs_tl_put_right:(cx cv)</u>	_bnvs_tl_put_right:cn {<core>} {<tl>}
	_bnvs_tl_put_right:cv {<core>} {<value core name>}

These are shortcuts to

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

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

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

```

390 \cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
391   \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
392     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
393       \prg_return_true:
394     } {
395       \prg_return_false:
396     }
397   }
398 }

399 \cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
400   \BNVS_use:nc {
401     \BNVS_new_conditional_vnc:cNn { #1 }
402   } { #1:nncTF }
403 }

```

```
\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.

```

404 \cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
405   \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
406     \BNVS_tl_use:nv {
407       \BNVS_tl_use:Nv #2 { ##1 }
408     } { ##2 } { ##3 } { ##4 } {
409       \prg_return_true:
410     } {
411       \prg_return_false:
412     }
413   }
414 }

415 \cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
416   \BNVS_use:nc {
417     \BNVS_new_conditional_vvnc:cNn { #1 }
418   } { #1:nncTF }
419 }

420 \cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
421   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
422     \BNVS_tl_use:nv {
423       \BNVS_tl_use:Nv #2 { ##1 }
424     } { ##2 } { ##3 } {
425       \prg_return_true:
426     } {
427       \prg_return_false:
428     }
429   }
430 }

431 \cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
432   \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
433     \BNVS_tl_use:nv {
434       \BNVS_tl_use:nv {
435         \BNVS_tl_use:Nv #2 { ##1 }
436       } { ##2 }
437     } { ##3 } { ##4 } {
438       \prg_return_true:
439     } {
440       \prg_return_false:
441     }
442   }
443 }

444 \cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
445   \BNVS_use:nc {
446     \BNVS_new_conditional_vvc:cNn { #1 }
447   } { #1:nncTF }
448 }

```

```

449 \cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
450   \BNVS_use:nc {
451     \BNVS_new_conditional_vvvc:cNn { #1 }
452   } { #1:nnncTF }
453 }

454 \cs_new:Npn \BNVS_new_tl_c:c {
455   \BNVS_new_c:nc { tl }
456 }
457 \BNVS_new_tl_c:c { clear }
458 \BNVS_new_tl_c:c { use }
459 \BNVS_new_tl_c:c { count }

460 \BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
461   \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
462 }

463 \cs_new:Npn \BNVS_new_tl_cn:c {
464   \BNVS_new_cn:nc { tl }
465 }

466 \cs_new:Npn \BNVS_new_tl_cv:c #1 {
467   \BNVS_new_cv:ncn { tl } { #1 } { tl }
468 }
469 \BNVS_new_tl_cn:c { set }
470 \BNVS_new_tl_cv:c { set }

471 \BNVS_new:cpn { tl_set:cx } {
472   \exp_args:Nnx \__bnvs_tl_set:cn
473 }
474 \BNVS_new_tl_cn:c { put_right }
475 \BNVS_new_tl_cv:c { put_right }
476 % \BNVS_generate_variant:cn { tl_put_right:cn } { cx }

477 \BNVS_new:cpn { tl_put_right:cx } {
478   \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
479 }
480 \BNVS_new_tl_cn:c { put_left }
481 \BNVS_new_tl_cv:c { put_left }
482 % \BNVS_generate_variant:cn { tl_put_left:cn } { cx }

483 \BNVS_new:cpn { tl_put_left:cx } {
484   \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
485 }

```

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

```

486 \cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
487   \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
488     \BNVS_use:Ncn #3 { ##1 } { #1 } {
489       \prg_return_true:
490     } {
491       \prg_return_false:
492     }
493   }
494 }

495 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
496   \BNVS_use_raw:nc {
497     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
498   } { #1_#2:NnTF }
499 }
500 \BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }

501 \BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
502   \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
503     \prg_return_true:
504   } {
505     \prg_return_false:
506   }
507 }

508 \cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
509   \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
510     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
511       \prg_return_true:
512     } {
513       \prg_return_false:
514     }
515   }
516 }

517 \cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
518   \BNVS_use_raw:nc {
519     \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
520   } { #1_#2:NnTF }
521 }
522 \BNVS_new_conditional_cn:ncn { tl } { if_eq } { T, F, TF }

523 \cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
524   \BNVS_new_conditional:cpnn { #2:cv } ##1 ##2 { #4 } {
525     \BNVS_use:nvn {
526       \BNVS_use:Ncn #3 { ##1 } { #1 }
527     } { ##2 } { #1 } {
528       \prg_return_true:
529     } {
530       \prg_return_false:
531     }
532   }
533 }

```

```

534 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
535   \BNVS_use_raw:nc {
536     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
537   } { #1_#2:NnTF }
538 }
539 \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>}

```

540 \cs_new:Npn \BNVS_new_conditional_vv:cNn #1 #2 #3 {
541   \BNVS_new_conditional:cpnn { #1:vv } ##1 ##2 { #3 } {
542     \BNVS_tl_use:nv {
543       \BNVS_tl_use:Nv #2 { ##1 }
544     } { ##2 } {
545       \prg_return_true:
546     } {
547       \prg_return_false:
548     }
549   }
550 }

551 \cs_new:Npn \BNVS_new_conditional_vv:cn #1 {
552   \BNVS_use:nc {
553     \BNVS_new_conditional_vvnc:cNn { #1 }
554   } { #1:nnTF }
555 }

556 \cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
557   \BNVS_new_conditional:cpnn { #2:vn } ##1 ##2 { #4 } {
558     \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
559       \prg_return_true:
560     } {
561       \prg_return_false:
562     }
563   }
564 }

565 \cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
566   \BNVS_use_raw:nc {
567     \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
568   } { #1_#2:nnTF }
569 }
570 \BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }

571 \cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
572   \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
573     \BNVS_use:nvn {
574       \BNVS_use:Nvn #3 { ##1 } { #1 }

```



```

575     } { ##2 } { #1 } {
576     \prg_return_true:
577     } {
578     \prg_return_false:
579     }
580   }
581 }

582 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
583   \BNVS_use_raw:nc {
584     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
585   } { #1_#2:nnTF }
586 }
587 \BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }

```

6.4.4 Sequences

<code>__bnvs_seq_count:c</code>	<code>__bnvs_seq_new:c {<core>}</code>
<code>__bnvs_seq_clear:c</code>	<code>__bnvs_seq_count:c {<core>}</code>
<code>__bnvs_seq_set_eq:cc</code>	<code>__bnvs_seq_clear:c {<core>}</code>
<code>__bnvs_seq_use:cn</code>	<code>__bnvs_seq_set_eq:cc {<core₁>} {<core₂>}</code>
<code>__bnvs_seq_item:cn</code>	<code>__bnvs_seq_use:cn {<core>} {<separator>}</code>
<code>__bnvs_seq_remove_all:cn</code>	<code>__bnvs_seq_item:cn {<core>} {<integer expression>}</code>
<code>__bnvs_seq_put_left:cv</code>	<code>__bnvs_seq_remove_all:cn {<core>} {<tl>}</code>
<code>__bnvs_seq_put_right:cn</code>	<code>__bnvs_seq_put_right:cn {<seq core>} {<tl>}</code>
<code>__bnvs_seq_put_right:cv</code>	<code>__bnvs_seq_put_right:cv {<seq core>} {<tl core>}</code>
<code>__bnvs_seq_set_split:cn</code>	<code>__bnvs_seq_set_split:cn {<seq core>} {<tl>} {<separator>}</code>
<code>__bnvs_seq_set_split:(cnv cnx)</code>	<code>__bnvs_seq_pop_left:cc {<core₁>} {<core₂>}</code>
<code>__bnvs_seq_pop_left:cc</code>	

These are shortcuts to

- `\seq_set_eq:cc {l__bnvs_<core1>_seq} {l__bnvs_<core2>_seq}`
- `\seq_count:c {l__bnvs_<core>_seq}`
- `\seq_use:cn {l__bnvs_<core>_seq}{<separator>}`
- `\seq_item:cn {l__bnvs_<core>_seq}{<integer expression>}`
- `\seq_remove_all:cn {l__bnvs_<core>_seq}{<tl>}`
- `__bnvs_seq_clear:c {l__bnvs_<core>_seq}`
- `\seq_put_right:cv {l__bnvs_<seq core>_seq} {l__bnvs_<tl core>_tl}`
- `\seq_set_split:cn {l__bnvs_<seq core>_seq}{l__bnvs_<tl core>_tl}{<tl>}`

```

588 \BNVS_new_c:nc { seq } { count }
589 \BNVS_new_c:nc { seq } { clear }
590 \BNVS_new_cn:nc { seq } { use }
591 \BNVS_new_cn:nc { seq } { item }
592 \BNVS_new_cn:nc { seq } { remove_all }
593 \BNVS_new_cn:nc { seq } { map_inline }
594 \BNVS_new_cc:nc { seq } { set_eq }

```

```

595 \BNVS_new_cv:ncn { seq } { put_left } { tl }
596 \BNVS_new_cn:ncn { seq } { put_right } { tl }
597 \BNVS_new_cv:ncn { seq } { put_right } { tl }
598 \BNVS_new_cnn:nc { seq } { set_split }
599 \BNVS_new_cnv:nc { seq } { set_split }
600 \BNVS_new_cnx:nc { seq } { set_split }
601 \BNVS_new_cc:ncn { seq } { pop_left } { tl }
602 \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

```

```

603 \cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
604   \BNVS_new_conditional:cpnn { #1_#2:cc } ##1 ##2 { #4 } {
605     \BNVS_use:ncncn {
606       \BNVS_use_raw:c { #1_#2:NNTF }
607     } { ##1 } { #1 } { ##2 } { #3 } {
608       \prg_return_true:
609     } {
610       \prg_return_false:
611     }
612   }
613 }
614 \BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
615 \BNVS_new_conditional_cc:ncnn
616 { seq } { get_right } { tl } { T, F, TF }
617 \BNVS_new_conditional_cc:ncnn
618 { seq } { pop_left } { tl } { T, F, TF }
619 \BNVS_new_conditional_cc:ncnn
620 { 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>

```

621 \BNVS_new_c:nc { int } { new }
622 \BNVS_new_c:nc { int } { use }
623 \BNVS_new_c:nc { int } { zero }
624 \BNVS_new_c:nc { int } { incr }
625 \BNVS_new_c:nc { int } { decr }
626 \BNVS_new_cn:nc { int } { set }
627 \BNVS_new_cv:ncn { int } { set } { int }

```

6.4.6 Prop

_bnvs_if_prop_get:Nnc \overline{TF}

```

628 \BNVS_new_conditional:cpnn { if_prop_get:Nnc } #1 #2 #3 { T, F, TF } {
629   \BNVS_use:ncn {
630     \prop_get:NnNTF #1 { #2 }
631   } { #3 } { t1 } {
632     \prg_return_true:
633   } {
634     \prg_return_false:
635   }
636 }

```

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 $\mathrm{T\!E\!X}$ group, which ensures no name collision with the caller stack. The number of variables used has not been optimized, nor the $\mathrm{T\!E\!X}$ groups used. Optimization often goes against readability.

```

637 \tl_new:N \l__bnvs_id_last_tl
638 \tl_set:Nn \l__bnvs_id_last_tl { ?! }
639 \tl_new:N \l__bnvs_a_tl
640 \tl_new:N \l__bnvs_b_tl
641 \tl_new:N \l__bnvs_c_tl
642 \tl_new:N \l__bnvs_V_tl
643 \tl_new:N \l__bnvs_A_tl
644 \tl_new:N \l__bnvs_L_tl
645 \tl_new:N \l__bnvs_Z_tl
646 \tl_new:N \l__bnvs_ans_tl

```

```

647 \tl_new:N \l__bnvs_Q_name_tl
648 \tl_new:N \l__bnvs_QF_name_tl
649 \tl_new:N \l__bnvs_QF_base_tl
650 \tl_new:N \l__bnvs_ref_tl
651 \tl_new:N \l__bnvs_ref_base_tl
652 \tl_new:N \l__bnvs_id_tl
653 \tl_new:N \l__bnvs_n_tl
654 \tl_new:N \l__bnvs_path_tl
655 \tl_new:N \l__bnvs_group_tl
656 \tl_new:N \l__bnvs_scan_tl
657 \tl_new:N \l__bnvs_query_tl
658 \tl_new:N \l__bnvs_token_tl
659 \tl_new:N \l__bnvs_root_tl
660 \tl_new:N \l__bnvs_n_incr_tl
661 \tl_new:N \l__bnvs_incr_tl
662 \tl_new:N \l__bnvs_plus_tl
663 \tl_new:N \l__bnvs_rhs_tl
664 \tl_new:N \l__bnvs_post_tl
665 \tl_new:N \l__bnvs_suffix_tl
666 \tl_new:N \l__bnvs_index_tl
667 \int_new:N \g__bnvs_call_int
668 \int_new:N \l__bnvs_int
669 \int_new:N \l__bnvs_i_int
670 \seq_new:N \g__bnvs_def_seq
671 \seq_new:N \l__bnvs_ans_seq
672 \seq_new:N \l__bnvs_match_seq
673 \seq_new:N \l__bnvs_split_seq
674 \seq_new:N \l__bnvs_path_seq
675 \seq_new:N \l__bnvs_path_head_seq
676 \seq_new:N \l__bnvs_path_tail_seq
677 \seq_new:N \l__bnvs_query_seq
678 \seq_new:N \l__bnvs_token_seq
679 \bool_new:N \l__bnvs_in_frame_bool
680 \bool_set_false:N \l__bnvs_in_frame_bool
681 \bool_new:N \l__bnvs_parse_bool

682 \cs_new:Npn \BNVS_error_ans:x {
683   \__bnvs_tl_put_right:cn { ans } { 0 }
684   \BNVS_error:x
685 }

```

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

```

686 \bool_new:N \l__bnvs_provide_bool

687 \BNVS_new:cpn { set_true:c } #1 {
688   \exp_args:Nc \bool_set_true:N { l__bnvs_#1_bool }
689 }

690 \BNVS_new:cpn { set_false:c } #1 {
691   \exp_args:Nc \bool_set_false:N { l__bnvs_#1_bool }
692 }

693 \BNVS_new:cpn { provide_on: } {
694   \__bnvs_set_true:c { provide }
695 }

```

```

696 \BNVS_new:cpn { provide_off: } {
697   \__bnvs_set_false:c { provide }
698 }
699 \__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.)

```

700 \int_const:Nn \c__bnvs_max_call_int { 8192 }

```

__bnvs_greset: __bnvs_greset:

Reset globally the call stack counter to its maximum value.

```

701 \BNVS_new:cpn { greset: } {
702   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
703 }

```

__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.

```

704 \BNVS_new_conditional:cpnn { if_call: } { T, F, TF } {
705   \int_gdecr:N \g__bnvs_call_int
706   \int_compare:nNnTF \g__bnvs_call_int > 0 {
707     \prg_return_true:
708   } {
709     \prg_return_false:
710   }
711 }

```

6.10 Overlay specification

6.10.1 Basic functions

\g__bnvs_prop <key>—<integer spec> property list to store the named overlay sets. The keys are constructed from fully qualified names denoted as <QF name>.

<QF name>/V for the value

<QF name>/A for the first index

<QF name>/L for the length when provided

<QF name>/Z for the last index when provided

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

712 \prop_new:N \g__bnvs_prop

(End of definition for \g__bnvs_prop.)

__bnvs_gput:nnn __bnvs_gput:(nvn nnv nvv) __bnvs_item:nn __bnvs_gremove:nn __bnvs_gclear:n __bnvs_gclear:n __bnvs_gclear:v __bnvs_gclear:	__bnvs_gput:nnn {<subkey>} {<QF name>} {<integer spec>} __bnvs_item:nn {<subkey>} {<QF name>} __bnvs_gremove:nn {<subkey>} {<QF name>} __bnvs_gclear:n {<QF name>} __bnvs_gclear: Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L ^A T _E X3 eponym functions. The key used in \g__bnvs_prop is <QF name>/<subkey>. In practice, <subkey> is one of V, A, L, Z. fq means “fully qualified”.
--	--

713 \BNVS_new:cpn { gput:nnn } #1 #2 {
714 \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
715 }

716 \BNVS_new:cpn { gput:nvn } #1 {
717 \BNVS_tl_use:nv {
718 __bnvs_gput:nnn { #1 }
719 }
720 }

721 \BNVS_new:cpn { gput:nnv } #1 #2 {
722 \BNVS_tl_use:nv {
723 __bnvs_gput:nnn { #1 } { #2 }
724 }
725 }

726 \BNVS_new:cpn { gput:nvv } #1 #2 {
727 \BNVS_tl_use:nv {
728 __bnvs_gput:nvn { #1 } { #2 }
729 }
730 }

731 \BNVS_new:cpn { item:nn } #1 #2 {
732 \prop_item:Nn \g__bnvs_prop { #2 / #1 }
733 }

734 \BNVS_new:cpn { gremove:nn } #1 #2 {
735 \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
736 }

737 \BNVS_new:cpn { gclear:n } #1 {
738 \clist_map_inline:nn { V, A, Z, L } {
739 __bnvs_gremove:nn { ##1 } { #1 }
740 }
741 __bnvs_c_gclear:n { #1 }
742 }

743 \BNVS_new:cpn { gclear: } {
744 \prop_gclear:N \g__bnvs_prop
745 }

746 \BNVS_generate_variant:cn { gclear:n } { V }

```

747 \BNVS_new:cpn { gclear:v } {
748     \BNVS_tl_use:Nc \_bnvs_gclear:V
749 }

```

```

\_bnvs_if_in_p:nn * \_bnvs_if_in_p:nn {<subkey>} {<QF name>}
\_bnvs_if_in:nnTF * \_bnvs_if_in:nnTF {<subkey>} {<QF name>} {<yes code>} {<no code>}
\_bnvs_if_in_p:n * \_bnvs_if_in_p:n {<QF name>}
\_bnvs_if_in:nTF * \_bnvs_if_in:nTF {<QF name>} {<yes code>} {<no code>}

```

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

```

750 \BNVS_new_conditional:cpnn { if_in:nn } #1 #2 { p, T, F, TF } {
751     \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
752         \prg_return_true:
753     } {
754         \prg_return_false:
755     }
756 }

757 \BNVS_new_conditional:cpnn { if_in:n } #1 { p, T, F, TF } {
758     \bool_if:nTF {
759         \_bnvs_if_in_p:nn V { #1 }
760         || \_bnvs_if_in_p:nn A { #1 }
761         || \_bnvs_if_in_p:nn Z { #1 }
762     } {
763         \prg_return_true:
764     } {
765         \prg_return_false:
766     }
767 }

768 \BNVS_new_conditional:cpnn { if_in:v } #1 { p, T, F, TF } {
769     \BNVS_tl_use:Nv \_bnvs_if_in:nTF { #1 }
770     { \prg_return_true: } { \prg_return_false: }
771 }

```

```

\_bnvs_gprovide:nnnT \_bnvs_gprovide:nnnT {<subkey>} {<QF name>} {<value>} {<yes precode>}

```

Execute $\langle yes \text{ precode} \rangle$ before providing.

```

772 \BNVS_new:cpn { gprovide:nnnT } #1 #2 #3 #4 {
773     \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
774         #4
775     \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
776 }
777 }

```

```

\_bnvs_if_get:nncTF \_bnvs_if_get:nncTF {<subkey>} {<QF name>} {<ans>} {<yes code>} {<no code>}

```

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

```

778 \BNVS_new_conditional:cpnn { if_get:nnc } #1 #2 #3 { T, F, TF } {
779   \BNVS_tl_use:nc {
780     \prop_get:NnNTF \g__bnvs_prop { #2 / #1 }
781   } { #3 } {
782     \prg_return_true:
783   } {
784     \prg_return_false:
785   }
786 }

787 \BNVS_new_conditional:cpnn { if_get:nvc } #1 #2 #3 { T, F, TF } {
788   \BNVS_tl_use:nv {
789     \__bnvs_if_get:nncTF { #1 }
790   } { #2 } { #3 } {
791     \prg_return_true:
792   } {
793     \prg_return_false:
794   }
795 }

```

6.10.2 Functions with cache

`\g__bnvs_c_prop` $\langle key \rangle$ – $\langle value \rangle$ property list to store the named overlay sets. Other keys are eventually used to cache results when some attributes are defined from other slide ranges.

$\langle QF\ name \rangle/V$ for the cached static value of the value

$\langle QF\ name \rangle/A$ for the cached static value of the first index

$\langle QF\ name \rangle/L$ for the cached static value of the length

$\langle QF\ name \rangle/Z$ for the cached static value of the last index

$\langle QF\ name \rangle/P$ for the cached static value of the previous index

$\langle QF\ name \rangle/N$ for the cached static value of the next index

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

```

796 \prop_new:N \g__bnvs_c_prop

```

(End of definition for `\g__bnvs_c_prop`.)

<code>__bnvs_c_gput:nnn</code>	<code>__bnvs_c_gput:nnn {$\langle subkey \rangle$} {$\langle QF\ name \rangle$} {$\langle value \rangle$}</code>
<code>__bnvs_c_gput:(nnv nvn nvv)</code>	<code>__bnvs_c_item:nn {$\langle subkey \rangle$} {$\langle QF\ name \rangle$}</code>
<code>__bnvs_c_item:nn</code>	<code>__bnvs_c_gremove:nn {$\langle subkey \rangle$} {$\langle QF\ name \rangle$}</code>
<code>__bnvs_c_gremove:nn</code>	<code>__bnvs_c_gclear:n {$\langle QF\ name \rangle$}</code>
<code>__bnvs_c_gclear:n</code>	<code>__bnvs_c_gclear:</code>
<code>__bnvs_c_gclear:</code>	Wrapper over the functions above for $\langle QF\ name \rangle/\langle subkey \rangle$.

```

797 \BNVS_new:cpn { c_gput:nnn } #1 #2 {
798   \prop_gput:Nnn \g__bnvs_c_prop { #2 / #1 }
799 }

```



```

800 \BNVS_new:cpn { c_gput:nvn } #1 {
801   \BNVS_tl_use:nv {
802     \__bnvs_c_gput:nnn { #1 }
803   }
804 }

805 \BNVS_new:cpn { c_gput:nnv } #1 #2 {
806   \BNVS_tl_use:nv {
807     \__bnvs_c_gput:nnn { #1 } { #2 }
808   }
809 }

810 \BNVS_new:cpn { c_gput:nvv } #1 #2 {
811   \BNVS_tl_use:nv {
812     \__bnvs_c_gput:nvn { #1 } { #2 }
813   }
814 }

815 \BNVS_new:cpn { c_item:nn } #1 #2 {
816   \prop_item:Nn \g__bnvs_c_prop { #2 / #1 }
817 }

818 \BNVS_new:cpn { c_gremove:nn } #1 #2 {
819   \prop_gremove:Nn \g__bnvs_c_prop { #2 / #1 }
820 }

821 \BNVS_new:cpn { c_gclear:n } #1 {
822   \clist_map_inline:nn { V, A, Z, L, P, N } {
823     \prop_gremove:Nn \g__bnvs_c_prop { #1 / ##1 }
824   }
825 }

826 \BNVS_new:cpn { c_gclear: } {
827   \prop_gclear:N \g__bnvs_c_prop
828 }

```

```

\__bnvs_c_if_in_p:nn * \__bnvs_c_if_in_p:n {<subkey>} {<QF name>}
\__bnvs_c_if_in:nnTF * \__bnvs_c_if_in:nTF {<subkey>} {<QF name>} {<yes code>} {<no code>}

```

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

```

829 \prg_new_conditional:Npnn \__bnvs_c_if_in:nn #1 #2 { p, T, F, TF } {
830   \prop_if_in:NnTF \g__bnvs_c_prop { #2 / #1 } {
831     \prg_return_true:
832   } {
833     \prg_return_false:
834   }
835 }

```

```

\__bnvs_c_if_get:nncTF \__bnvs_c_if_get:nncTF {<subkey>} {<QF name>} {<ans>} {<yes code>} {<no code>}

```

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

```

836 \BNVS_new_conditional:cpnn { c_if_get:nnc } #1 #2 #3 { T, F, TF } {
837   \BNVS_tl_use:nc {
838     \prop_get:NnNTF \g__bnvs_c_prop { #2 / #1 }
839   } { #3 } {
840     \prg_return_true:
841   } {
842     \prg_return_false:
843   }
844 }

```

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 fully qualified names $\langle id \rangle! \langle name \rangle . \langle c_1 \rangle \dots \langle c_j \rangle$ denoted as $\langle QF name \rangle$.

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

(End of definition for `\g__bnvs_v_prop`.)

<code>__bnvs_v_gput:nn</code>	<code>__bnvs_v_gput:nn { \langle Q name \rangle } { \langle value \rangle }</code>
<code>__bnvs_v_gput:(nv vv)</code>	<code>__bnvs_v_gremove:n { \langle Q name \rangle }</code>
<code>__bnvs_v_gremove:n</code>	<code>__bnvs_v_gclear:</code>
<code>__bnvs_v_gclear:</code>	

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

```

846 \BNVS_new:cpn { v_gput:nn } {
847   \prop_gput:Nnn \g__bnvs_v_prop
848 }
849 \BNVS_new:cpn { v_gput:nv } #1 {
850   \BNVS_tl_use:nv {
851     \__bnvs_v_gput:nn { #1 }
852   }
853 }
854 \BNVS_new:cpn { v_gput:vv } {
855   \BNVS_tl_use:Nv \__bnvs_v_gput:nv
856 }
857 \BNVS_new:cpn { v_gremove:n } {
858   \prop_gremove:Nn \g__bnvs_v_prop
859 }
860 \BNVS_new:cpn { v_gclear: } {
861   \prop_gclear:N \g__bnvs_v_prop
862 }

```

<code>__bnvs_v_if_in_p:n *</code>	<code>__bnvs_v_if_in_p:n { \langle QF name \rangle }</code>
<code>__bnvs_v_if_in:nTF *</code>	<code>__bnvs_v_if_in:nTF { \langle QF name \rangle } { \langle yes code \rangle } { \langle no code \rangle }</code>

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

```

863 \BNVS_new_conditional:cpnn { v_if_in:n } #1 { p, T, F, TF } {
864   \prop_if_in:NnTF \g__bnvs_v_prop { #1 } {
865     \prg_return_true:
866   } {
867     \prg_return_false:
868   }
869 }

```

__bnvs_v_if_get:ncTF __bnvs_v_if_get:ncTF {<Q name>} {<ans>} {<yes code>} {<no code>}

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

```

870 \BNVS_new_conditional:cpnn { v_if_get:nc } #1 #2 { T, F, TF } {
871   \BNVS_tl_use:nc {
872     \prop_get:NnTF \g__bnvs_v_prop { #1 }
873   } { #2 } {
874     \prg_return_true:
875   } {
876     \prg_return_false:
877   }
878 }

```

__bnvs_v_if_greset:nnTF	__bnvs_v_if_greset:nnTF {<QF name>} {<initial value>} {<yes code>} {<no
__bnvs_v_if_greset:(vn nv vv)TF	code>}
__bnvs_n_if_greset:nnTF	__bnvs_n_if_greset:nnTF {<QF name>} {<initial value>} {<yes code>} {<no
__bnvs_n_if_greset:(vn nv vv)TF	code>}
__bnvs_if_greset_all:nnTF	__bnvs_if_greset_all:nnTF {<QF name>} {<initial value>} {<yes code>}
__bnvs_if_greset_all:vnTF	{<no code>}

If the <QF name> is known, reset the value counter or the n counter to the given <initial value> and execute <yes code> otherwise <no code> is executed. The ..._all variant also cleans the cached values.

```

879 \BNVS_new_conditional:cpnn { v_if_greset:nn } #1 #2 { T, F, TF } {
880   \__bnvs_v_if_in:nTF { #1 } {
881     \__bnvs_v_gremove:n { #1 }
882     \tl_if_empty:nF { #2 } {
883       \__bnvs_v_gput:nn { #1 } { #2 }
884     }
885     \prg_return_true:
886   } {
887     \prg_return_false:
888   }
889 }
890 \BNVS_new_conditional:cpnn { v_if_greset:nv } #1 #2 { T, F, TF } {
891   \BNVS_tl_use:nv { \__bnvs_v_if_greset:nnTF { #1 } } { #2 }
892   { \prg_return_true: } { \prg_return_false: }
893 }

```

```

894 \BNVS_new_conditional:cpnn { v_if_greset:vn } #1 #2 { T, F, TF } {
895   \BNVS_tl_use:Nv \_bnvs_v_if_greset:nnTF { #1 } { #2 }
896   { \prg_return_true: } { \prg_return_false: }
897 }
898 \BNVS_new_conditional:cpnn { v_if_greset:vv } #1 #2 { T, F, TF } {
899   \BNVS_tl_use:Nv \_bnvs_v_if_greset:nnTF { #1 } { #2 }
900   { \prg_return_true: } { \prg_return_false: }
901 }
902 \BNVS_new_conditional:cpnn { n_if_greset:nn } #1 #2 { T, F, TF } {
903   \_bnvs_n_if_in:nTF { #1 } {
904     \_bnvs_n_gremove:n { #1 }
905     \tl_if_empty:nF { #2 } {
906       \_bnvs_n_gput:nn { #1 } { #2 }
907     }
908     \prg_return_true:
909   } {
910     \prg_return_false:
911   }
912 }
913 \BNVS_new_conditional:cpnn { n_if_greset:nv } #1 #2 { T, F, TF } {
914   \BNVS_tl_use:nv { \_bnvs_n_if_greset:nnTF { #1 } } { #2 }
915   { \prg_return_true: } { \prg_return_false: }
916 }
917 \BNVS_new_conditional:cpnn { n_if_greset:vn } #1 #2 { T, F, TF } {
918   \BNVS_tl_use:Nv \_bnvs_n_if_greset:nnTF { #1 } { #2 }
919   { \prg_return_true: } { \prg_return_false: }
920 }
921 \BNVS_new_conditional:cpnn { n_if_greset:vv } #1 #2 { T, F, TF } {
922   \BNVS_tl_use:Nv \_bnvs_n_if_greset:nnTF { #1 } { #2 }
923   { \prg_return_true: } { \prg_return_false: }
924 }
925 \BNVS_new_conditional:cpnn { if_greset_all:nn } #1 #2 { T, F, TF } {
926   \_bnvs_if_in:nTF { #1 } {
927     \BNVS_begin:
928     \clist_map_inline:nn { V, A, Z, L } {
929       \_bnvs_if_get:nncT { ##1 } { #1 } { a } {
930         \_bnvs_quark_if_nil:cT { a } {
931           \_bnvs_c_if_get:nncTF { ##1 } { #1 } { a } {
932             \_bnvs_gput:nnv { ##1 } { #1 } { a }
933           } {
934             \_bnvs_gput:nnn { ##1 } { #1 } { 1 }
935           }
936         }
937       }
938     }
939     \BNVS_end:
940     \_bnvs_c_gclear:n { #1 }
941     \_bnvs_n_gremove:n { #1 }
942     \_bnvs_v_if_greset:nnT { #1 } { #2 } {}
943     \prg_return_true:

```

```

944 } {
945   \prg_return_false:
946 }
947 }

948 \BNVS_new_conditional:cpnn { if_greset_all:vn } #1 #2 { T, F, TF } {
949   \BNVS_tl_use:Nv \__bnvs_if_greset_all:nnTF { #1 } { #2 }
950   { \prg_return_true: } { \prg_return_false: }
951 }

```

```

\__bnvs_gclear_all:n \__bnvs_gclear_all:n {\<QF name>}
\__bnvs_gclear_all: \__bnvs_gclear_all:

```

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

```

952 \BNVS_new:cpn { gclear_all: } {
953   \__bnvs_gclear:
954   \__bnvs_c_gclear:
955   \__bnvs_n_gclear:
956   \__bnvs_v_gclear:
957 }

958 \BNVS_new:cpn { gclear_all:n } #1 {
959   \__bnvs_gclear:n { #1 }
960   \__bnvs_c_gclear:n { #1 }
961   \__bnvs_n_gremove:n { #1 }
962   \__bnvs_v_gremove:n { #1 }
963 }

```

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.

`\g__bnvs_n_prop` $\langle key \rangle$ – $\langle value \rangle$ property list to store the contents of the named index counters. The keys are qualified full names.

```

964 \prop_new:N \g__bnvs_n_prop

```

(End of definition for `\g__bnvs_n_prop`.)

```

\__bnvs_n_gput:nn \__bnvs_n_gput:nn {\<QF name>} {\<value>}
\__bnvs_n_gput:(nv|vv) \__bnvs_n_item:n {\<QF name>}
\__bnvs_n_gprovide:nn \__bnvs_n_gremove:n {\<QF name>}
\__bnvs_n_item:n \__bnvs_n_gclear:
\__bnvs_n_gremove:n
\__bnvs_n_gremove:v
\__bnvs_n_gclear:

```

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

```

965 \BNVS_new:cpn { n_gput:nn } {
966   \prop_gput:Nnn \g__bnvs_n_prop
967 }

```

```

968 \BNVS_new:cpn { n_gput:nv } #1 {
969   \BNVS_t1_use:nv {
970     \__bnvs_n_gput:nn { #1 }
971   }
972 }

973 \BNVS_new:cpn { n_gput:vv } {
974   \BNVS_t1_use:Nv \__bnvs_n_gput:nv
975 }

976 \BNVS_new:cpn { n_gprovide:nn } #1 #2 {
977   \prop_if_in:NnF \g__bnvs_n_prop { #1 } {
978     \prop_gput:Nnn \g__bnvs_n_prop { #1 } { #2 }
979   }
980 }

981 \BNVS_new:cpn { n_item:n } #1 {
982   \prop_item:Nn \g__bnvs_n_prop { #1 }
983 }

984 \BNVS_new:cpn { n_gremove:n } {
985   \prop_gremove:Nn \g__bnvs_n_prop
986 }
987 \BNVS_generate_variant:cn { n_gremove:n } { V }

988 \BNVS_new:cpn { n_gremove:v } {
989   \BNVS_t1_use:nc {
990     \__bnvs_n_gremove:V
991   }
992 }

993 \BNVS_new:cpn { n_gclear: } {
994   \prop_gclear:N \g__bnvs_n_prop
995 }
996 \cs_generate_variant:Nn \__bnvs_n_gremove:n { V }

```

```

\__bnvs_n_if_in_p:n ★ \__bnvs_n_if_in_p:nn {⟨QF name⟩}
\__bnvs_n_if_in:nTF ★ \__bnvs_n_if_in:nTF {⟨QF name⟩} {⟨yes code⟩} {⟨no code⟩}

```

Convenient shortcuts to test for the existence of the $\langle QF \text{ name} \rangle$ value counter.

```

997 \prg_new_conditional:Npnn \__bnvs_n_if_in:n #1 { p, T, F, TF } {
998   \prop_if_in:NnTF \g__bnvs_n_prop { #1 } {
999     \prg_return_true:
1000   } {
1001     \prg_return_false:
1002   }
1003 }

```

```

\__bnvs_n_if_get:ncTF \__bnvs_n_if_get:ncTF {⟨QF name⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}

```

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute $\langle \text{yes code} \rangle$ when the item is found, $\langle \text{no code} \rangle$ otherwise. In the latter case, the content of the $\langle \text{ans} \rangle$ t1 variable is undefined. NB: the predicate won't work because $\backslash\text{prop_get:NnNTF}$ is not expandable.

```

1004 \prg_new_conditional:Npnn \__bnvs_n_if_get:nc #1 #2 { T, F, TF } {
1005   \__bnvs_if_prop_get:NncTF \g__bnvs_n_prop { #1 } { #2 } {
1006     \prg_return_true:
1007   } {
1008     \prg_return_false:
1009   }
1010 }

```

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.

```

1011 \regex_const:Nn \c__bnvs_short_regex {
1012   [[:alpha:]]_ [[:alnum:]]_*
1013 }

```

(End of definition for `\c__bnvs_short_regex`.)

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

```

1014 \regex_const:Nn \c__bnvs_id_regex {
1015   (?: \ur{c__bnvs_short_regex} | [?] )? !
1016 }

```

(End of definition for `\c__bnvs_id_regex`.)

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

```

1017 \regex_const:Nn \c__bnvs_path_regex {
1018   (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )*
1019 }

```

(End of definition for `\c__bnvs_path_regex`.)

`\c__bnvs_A_index_Z_regex`

(End of definition for `\c__bnvs_A_index_Z_regex`.)

```

1020 \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`.)

```

1021 \regex_const:Nn \c__bnvs_A_reserved_Z_regex {
1022   \A_* [a-z] [_a-z0-9]* \Z
1023 }

```

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

(End of definition for `\c__bnvs_A_QF_name_Z_regex`.)

```

1024 \regex_const:Nn \c__bnvs_A_QF_name_Z_regex {
    1: The range name including the frame  $\langle id \rangle$  and exclamation mark if any
    2: frame  $\langle id \rangle$  including the exclamation mark
1025 \A ( ( \ur{c__bnvs_id_regex} ? ) \ur{c__bnvs_short_regex} )
    3: the path, if any.
1026 ( \ur{c__bnvs_path_regex} ) \Z
1027 }

```

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

(End of definition for \c__bnvs_A_QF_name_n_Z_regex.)

```

1028 \regex_const:Nn \c__bnvs_A_QF_name_n_Z_regex {
    1: The full match
    2: The fully qualified name including the frame  $\langle id \rangle$  and exclamation mark if any,
       the dotted path but excluding the trailing .n (this is \c__bnvs_path_regex with
       a trailing ?).
    3: frame  $\langle id \rangle$  including the exclamation mark
1029 \A ( ( \ur{c__bnvs_id_regex} ? )
1030 \ur{c__bnvs_short_regex}
1031 (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )*(? )
    4: the last .n component if any.
1032 ( \. n )? \Z
1033 }

```

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

```

1034 \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 10 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.

```

1035 \regex_const:Nn \c__bnvs_split_regex {
1036 \s* ( ? :

```

We start with ‘++’ instrussions⁴.

1 incrementation prefix

```

1037 \+\+

```


1.1: $\langle \textit{qualified name} \rangle$ of an overlay set

1.2: $\langle id \rangle$ of a an overlay set including the exclamation mark

```
1038      ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_short_regex} )
```

1.3: optionally followed by a dotted path

```
1039      ( \ur{c__bnvs_path_regex} )
```

2: without incement prefix

2.1: $\langle \textit{qualified name} \rangle$ of an overlay set

2.2: $\langle id \rangle$ of a slide range including the exclamation mark

```
1040      | ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_short_regex} )
```

2.3: optionally followed by a dotted path

```
1041      ( \ur{c__bnvs_path_regex} )
```

We continue with other expressions

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

```
1042      (?: \.(\+)\++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.

```
1043      | \s* (\+?)= \s* ( \S+ )
```

2.7: the post increment

```
1044      | (\+)\+
```

```
1045      )?
```

```
1046      ) \s*
```

```
1047      }
```

(End of definition for `c__bnvs_split_regex`.)

6.14 beamer.cls interface

Work in progress.

```
1048 \RequirePackage{keyval}
```

```
1049 \define@key{beamerframe}{beanoves~id}[] {
1050   \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }
1051 }
```

```
1052 \AddToHook{env/beamer@frameslide/before}{
1053   \__bnvs_greset:
1054   \__bnvs_n_gclear:
1055   \__bnvs_v_gclear:
1056   \bool_set_true:N \l__bnvs_in_frame_bool
1057 }
```

```
1058 \AddToHook{env/beamer@frameslide/after}{
1059   \bool_set_false:N \l__bnvs_in_frame_bool
1060 }
```

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

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.

```

1061 \BNVS_new_conditional:cpnn { split_if_pop_left:c } #1 { T, F, TF } {
1062   \__bnvs_seq_pop_left:ccTF { split } { #1 } {
1063     \prg_return_true:
1064   } {
1065     \prg_return_false:
1066   }
1067 }

1068 \BNVS_new:cpn { split_if_pop_left:cTn } #1 #2 #3 {
1069   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS:n { #3 } }
1070 }

1071 \BNVS_new:cpn { split_if_pop_left_or:cT } #1 #2 {
1072   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS:n { #1 } }
1073 }
1074 \exp_args_generate:n { VVV }

1075 \BNVS_new_conditional:cpnn { range_if_set:cccn } #1 #2 #3 #4 { T, F, TF } {
1076   \BNVS_begin:
1077   \__bnvs_tl_clear:c { a }
1078   \__bnvs_tl_clear:c { b }
1079   \__bnvs_tl_clear:c { c }
1080   \__bnvs_if_regex_split:cnTF { colons } { #4 } {
1081     \__bnvs_seq_pop_left:ccT { split } { a } {

```

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

```

1082   \__bnvs_seq_pop_left:ccT { split } { b } {
1083     \__bnvs_tl_if_empty:cTF { b } {

```

This is a one colon range.

```

1084   \__bnvs_split_if_pop_left:cTF { b } {

```

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

```

1085   \__bnvs_seq_pop_left:ccT { split } { c } {
1086     \__bnvs_tl_if_empty:cTF { c } {

```

A :: was expected:

```

1087   \BNVS_error:n { Invalid-range-expression(1):~#4 }
1088 } {
1089   \int_compare:nNnT { \__bnvs_tl_count:c { c } } > { 1 } {
1090     \BNVS_error:n { Invalid-range-expression(2):~#4 }
1091   }
1092   \__bnvs_split_if_pop_left:cTF { c } {

```

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

```

1093         \__bnvs_seq_if_empty:cF { split } {
1094             \BNVS_error:n { Invalid-range-expression(3):~#4 }
1095         }
1096     } {
1097         \BNVS_error:n { Internal-error }
1098     }
1099 }
1100 }
1101 } {
1102 }
1103 } {

```

This is a two colon range component.

```

1104     \int_compare:nNtT { \__bnvs_tl_count:c { b } } > { 1 } {
1105         \BNVS_error:n { Invalid-range-expression(4):~#4 }
1106     }
1107     \__bnvs_seq_pop_left:ccT { split } { c } {

```

c contains the $\langle length \rangle$.

```

1108         \__bnvs_split_if_pop_left:cTF { b } {
1109             \__bnvs_tl_if_empty:cTF { b } {
1110                 \__bnvs_seq_pop_left:cc { split } { b }

```

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

```

1111         \__bnvs_seq_if_empty:cF { split } {
1112             \BNVS_error:n { Invalid-range-expression(5):~#4 }
1113         }
1114     } {
1115         \BNVS_error:n { Invalid-range-expression(6):~#4 }
1116     }
1117     } {
1118         \__bnvs_tl_clear:c { b }
1119     }
1120 }
1121 }
1122 }
1123 }

```

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

```

1124     \cs_set:Npn \BNVS_next: { }
1125     \__bnvs_tl_if_empty:cT { a } {
1126         \__bnvs_tl_if_empty:cT { b } {
1127             \__bnvs_tl_if_empty:cT { c } {
1128                 \cs_set:Npn \BNVS_next: {
1129                     \BNVS_error:n { Invalid-range-expression(7):~#3 }
1130                 }
1131             }
1132         }
1133     }
1134     \BNVS_next:
1135     \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1136         \BNVS_end:
1137         \__bnvs_tl_set:cn { #1 } { ##1 }

```

```

1138     \__bnvs_tl_set:cn { #2 } { ##2 }
1139     \__bnvs_tl_set:cn { #3 } { ##3 }
1140   }
1141   \BNVS_exp_args:Nvvv \BNVS:nnn { a } { b } { c }
1142   \prg_return_true:
1143 } {
1144   \BNVS_end:
1145   \prg_return_false:
1146 }
1147 }

```

`__bnvs_range:nnnn` `__bnvs_range:nnnn {<Q?F name>} {<start>} {<end>} {<length>}`
`__bnvs_range:nvvv`

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

```

1148 \BNVS_new:cpn { range:nnnn } #1 {
1149   \__bnvs_if:cTF { provide } {
1150     \__bnvs_if_in:nnTF A { #1 } {
1151       \use_none:nnn
1152     } {
1153       \__bnvs_if_in:nnTF Z { #1 } {
1154         \use_none:nnn
1155       } {
1156         \__bnvs_if_in:nnTF L { #1 } {
1157           \use_none:nnn
1158         } {
1159           \__bnvs_do_range:nnnn { #1 }
1160         }
1161       }
1162     }
1163   } {
1164     \__bnvs_do_range:nnnn { #1 }
1165   }
1166 }

1167 \BNVS_new:cpn { range:nvvv } #1 #2 #3 #4 {
1168   \BNVS_tl_use:nv {
1169     \BNVS_tl_use:nv {
1170       \BNVS_tl_use:nv {
1171         \BNVS_use:c { range:nnnn } { #1 }
1172       } { #2 }
1173     } { #3 }
1174   } { #4 }
1175 }

```

<code>__bnvs_parse_record:n</code>	<code>__bnvs_parse_record:n {<Q?F name>}</code>
<code>__bnvs_parse_record:v</code>	<code>__bnvs_parse_record:nn {<Q?F name>} {<value>}</code>
<code>__bnvs_parse_record:nn</code>	<code>__bnvs_n_parse_record:n {<Q?F name>}</code>
<code>__bnvs_parse_record:vn</code>	<code>__bnvs_n_parse_record:nn {<Q?F name>} {<value>}</code>
<code>__bnvs_n_parse_record:n</code>	Auxiliary function for <code>__bnvs_parse:n</code> and <code>__bnvs_parse:nn</code> below. If <code><value></code> does not correspond to a range, the V key is used. The <code>_n</code> variant concerns the index counter. These are bottlenecks.
<code>__bnvs_n_parse_record:v</code>	
<code>__bnvs_n_parse_record:nn</code>	
<code>__bnvs_n_parse_record:vn</code>	

```

1176 \BNVS_new:cpn { parse_record:n } #1 {
1177   \__bnvs_if:cTF { provide } {
1178     \__bnvs_gprovide:nnnT V { #1 } { 1 } {
1179       \__bnvs_gclear:n { #1 }
1180     }
1181   } {
1182     \__bnvs_if:cTF { reset } {
1183       \__bnvs_if:cT { reset_all } {
1184         \__bnvs_if_greset_all:nnT { #1 } { } { }
1185       }
1186       \__bnvs_if:cF { only } {
1187         \prop_map_inline:Nn \g__bnvs_c_prop {
1188           \str_if_in:nnT { . ##1 } { .#1. } {
1189             \prop_gremove:Nn \g__bnvs_c_prop { ##1 }
1190           }
1191         }
1192       }
1193       \__bnvs_v_if_greset:nnT { #1 } { } { }
1194     } {
1195       \__bnvs_gclear:n { #1 }
1196       \__bnvs_gput:nnn V { #1 } { 1 }
1197     }
1198   }
1199 }

1200 \BNVS_new:cpn { parse_record:v } {
1201   \BNVS_tl_use:nv {
1202     \__bnvs_parse_record:n
1203   }
1204 }

1205 \BNVS_new:cpn { parse_record:nn } #1 #2 {
1206   \__bnvs_range_if_set:cccnTF { a } { b } { c } { #2 } {
1207     \__bnvs_range:nvvv { #1 } { a } { b } { c }
1208   } {
1209     \__bnvs_if:cTF { provide } {
1210       \__bnvs_gprovide:nnnT V { #1 } { #2 } {
1211         \__bnvs_gclear_all:n { #1 }
1212       }
1213     } {
1214       \__bnvs_if:cTF { reset } {
1215         \__bnvs_if:cT { reset_all } {
1216           \__bnvs_if_greset_all:nnT { #1 } { #2 } { }
1217         }
1218         \__bnvs_v_if_greset:nnT { #1 } { #2 } { }
1219       } {
1220         \__bnvs_gclear_all:n { #1 }
1221         \__bnvs_gput:nnn V { #1 } { #2 }
1222       }
1223     }
1224   }
1225 }

```

```

1226 \BNVS_new:cpn { parse_record:vn } {
1227   \BNVS_tl_use:nv {
1228     \__bnvs_parse_record:nn
1229   }
1230 }

1231 \BNVS_new:cpn { n_parse_record:n } #1 {
1232   \__bnvs_if:cTF { provide } {
1233     \__bnvs_n_gprovide:nn { #1 } { 1 }
1234   } {
1235     \__bnvs_if:cTF { reset } {
1236       \__bnvs_if:cTF { reset_all } {
1237         \__bnvs_if_greset_all:nnT { #1 } { } { }
1238       }
1239       \__bnvs_n_if_greset:nnT { #1 } { } { }
1240     } {
1241       \__bnvs_n_gput:nn { #1 } { 1 }
1242     }
1243   }
1244 }

1245 \BNVS_new:cpn { n_parse_record:v } {
1246   \BNVS_tl_use:cv { n_parse_record:n }
1247 }

1248 \BNVS_new:cpn { n_parse_record:nn } #1 #2 {
1249   \__bnvs_range_if_set:cccnTF { a } { b } { c } { #2 } {
1250     \BNVS_error:n { Unexpected~range:~#2 }
1251   } {
1252     \__bnvs_if:cTF { provide } {
1253       \__bnvs_n_gprovide:nn
1254     } {
1255       \__bnvs_n_gput:nn
1256     } { #1 } { #2 }
1257   }
1258 }

1259 \BNVS_new:cpn { n_parse_record:vn } {
1260   \BNVS_tl_use:cv { n_parse_record:nn }
1261 }

```

```

\__bnvs_if_id_QF_name_n:n $\overline{TF}$  \__bnvs_id_name_n_set:nTF {<ref>} {<yes code>} {<no code>}
\__bnvs_if_id_QF_name_n:v $\overline{TF}$ 

```

If $\langle ref \rangle$ is a fully qualified name, put the frame id it defines into `id` and the fully qualified name into `QF_name`, then execute $\langle yes\ code \rangle$. The `n tl` variable is empty except when $\langle ref \rangle$ ends with `.n`. Otherwise execute $\langle no\ code \rangle$. If $\langle ref \rangle$ is only a qualified name, put it in `QF_name`, once prepended with `id_last`, and set `id` to `id_last`. `id_last` is not modified, but this must be discussed further on.

```

1262 \BNVS_new_conditional:cpnn { if_id_QF_name_n:n } #1 { T, F, TF } {
1263   \BNVS_begin:

```

```

1264 \__bnvs_match_if_once:NnTF \c__bnvs_A_QF_name_n_Z_regex { #1 } {
1265   \__bnvs_if_match_pop_left:cTF { n } {
1266     \__bnvs_if_match_pop_left:cTF { QF_name } {
1267       \__bnvs_if_match_pop_left:cTF { id } {
1268         \__bnvs_if_match_pop_left:cTF { n } {
1269           \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1270             \BNVS_end:
1271             \__bnvs_tl_set:cn { id } { ##1 }
1272             \__bnvs_tl_set:cn { QF_name } { ##2 }
1273             \__bnvs_tl_set:cn { n } { ##3 }
1274           }
1275           \__bnvs_tl_if_empty:cTF { id } {
1276             \BNVS_exp_args:Nvvv
1277             \BNVS:nnn { id_last } { QF_name } { n }
1278             \__bnvs_tl_put_left:cv { QF_name } { id_last }
1279           } {
1280             \BNVS_exp_args:Nvvv
1281             \BNVS:nnn { id } { QF_name } { n }
1282             \__bnvs_tl_set:cv { id_last } { id }
1283           }
1284           \prg_return_true:
1285         } {
1286           \BNVS_end:
1287           \BNVS_error:n { LOGICALLY_UNREACHABLE_A_QF_name_n_Z/n }
1288           \prg_return_false:
1289         }
1290       } {
1291         \BNVS_end:
1292         \BNVS_error:n { LOGICALLY_UNREACHABLE_A_QF_name_n_Z/id }
1293         \prg_return_false:
1294       }
1295     } {
1296       \BNVS_end:
1297       \BNVS_error:n { LOGICALLY_UNREACHABLE_A_QF_name_n_Z/QF_name }
1298       \prg_return_false:
1299     }
1300   } {
1301     \BNVS_end:
1302     \BNVS_error:n { LOGICALLY_UNREACHABLE_A_QF_name_n_Z/n }
1303     \prg_return_false:
1304   }
1305 } {
1306   \BNVS_end:
1307   \prg_return_false:
1308 }
1309 }

1310 \BNVS_new_conditional:cpnn { if_id_QF_name_n:v } #1 { T, F, TF } {
1311   \BNVS_tl_use:nv { \__bnvs_if_id_QF_name_n:nTF } { #1 } {
1312     \prg_return_true:
1313   } {
1314     \prg_return_false:
1315   }
1316 }

```

```

__bnvs_parse:n  __bnvs_parse:n {<QF name>}
__bnvs_parse:nn __bnvs_parse:nn {<QF name>} {<definition>}

```

Auxiliary functions called within a group by \keyval_parse:nnn. <QF name> is the overlay (eventually fully) qualified name, including eventually a dotted path and a frame identifier, <definition> is the corresponding definition.

\l__bnvs_match_seq Local storage for the match result.

(End of definition for \l__bnvs_match_seq.)

```

1317 \BNVS_new:cpn { parse:n } #1 {
1318   \peek_remove_spaces:n {
1319     \peek_catcode:NTF \c_group_begin_token {
1320       \__bnvs_tl_if_empty:cTF { root } {
1321         \BNVS_error:n { Unexpected-list-at-top-level. }
1322       } {
1323         \BNVS_begin:
1324         \__bnvs_int_incr:c { i }
1325         \__bnvs_tl_put_right:cx { root } { \__bnvs_int_use:c { i } . }
1326         \cs_set:Npn \BNVS:w #####1 #####2 \s_stop {
1327           \regex_match:nnT { \S* } { #####2 } {
1328             \BNVS_error:n { Unexpected-#####2 }
1329           }
1330           \keyval_parse:nnn {
1331             \__bnvs_parse:n
1332           } {
1333             \__bnvs_parse:nn
1334           } { #####1 }
1335           \BNVS_end:
1336         }
1337         \BNVS:w #1 \s_stop
1338       }
1339     } {
1340       \__bnvs_tl_if_empty:cTF { root } {
1341         \__bnvs_if_id_QF_name_n:nTF { #1 } {
1342           \__bnvs_tl_if_empty:cTF { n } {
1343             \__bnvs_parse_record:v
1344           } {
1345             \__bnvs_n_parse_record:v
1346           }
1347           { QF_name }
1348         } {
1349           \BNVS_error:n { Unexpected-name:~#1 }
1350         }
1351       } {

```

Find the first free index.

```

1352   \cs_set:Npn \BNVS: {
1353     \__bnvs_int_incr:c { i }
1354     \exp_args:NNx \prop_if_in:NnT \g__bnvs_prop {
1355       \__bnvs_tl_use:c { root } \__bnvs_int_use:c { i } / V
1356     } { \BNVS: }
1357   }
1358   \BNVS:

```



```

1359     \exp_args:Nnx \use:n {
1360         \__bnvs_tl_if_empty:cTF { n } {
1361             \int_compare:nNnT { \__bnvs_int_use:c { i } } = 1 {
1362                 }
1363             \__bnvs_parse_record:nn
1364         } {
1365             \__bnvs_n_parse_record:nn
1366         }
1367     } {
1368         \__bnvs_tl_use:c { root } \__bnvs_int_use:c { i }
1369     } { #1 }
1370 }
1371 }
1372 }
1373 }

1374 \BNVS_new:cpn { do_range:nnnn } #1 #2 #3 #4 {
1375     \__bnvs_gclear_all:n { #1 }
1376     \tl_if_empty:nTF { #4 } {
1377         \tl_if_empty:nTF { #2 } {
1378             \tl_if_empty:nTF { #3 } {
1379                 \BNVS_error:n { Not~a~range::~~#1 }
1380             } {
1381                 \__bnvs_gput:nnn Z { #1 } { #3 }
1382                 \__bnvs_gput:nnn V { #1 } { \q_nil }
1383             }
1384         } {
1385             \__bnvs_gput:nnn A { #1 } { #2 }
1386             \__bnvs_gput:nnn V { #1 } { \q_nil }
1387             \tl_if_empty:nF { #3 } {
1388                 \__bnvs_gput:nnn Z { #1 } { #3 }
1389                 \__bnvs_gput:nnn L { #1 } { \q_nil }
1390             }
1391         }
1392     } {
1393         \tl_if_empty:nTF { #2 } {
1394             \__bnvs_gput:nnn L { #1 } { #4 }
1395             \tl_if_empty:nF { #3 } {
1396                 \__bnvs_gput:nnn Z { #1 } { #3 }
1397                 \__bnvs_gput:nnn A { #1 } { \q_nil }
1398                 \__bnvs_gput:nnn V { #1 } { \q_nil }
1399             }
1400         } {
1401             \__bnvs_gput:nnn A { #1 } { #2 }
1402             \__bnvs_gput:nnn L { #1 } { #4 }
1403             \__bnvs_gput:nnn Z { #1 } { \q_nil }
1404             \__bnvs_gput:nnn V { #1 } { \q_nil }
1405         }
1406     }
1407 }

1408 \cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
1409     \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
1410     { #4 }
1411 }

```

```

1412 \cs_new:Npn \BNVS_end_tl_set:cv #1 {
1413   \BNVS_tl_use:nv {
1414     \BNVS_end: \__bnvs_tl_set:cn { #1 }
1415   }
1416 }

```

Helper for `\keyval_parse:nnn` used in `\Beanoves` command.

```

1417 \regex_const:Nn \c__bnvs_one_suffix_regex { \A(.*)\.(?:1|first)\Z }

1418 \BNVS_new:cpn { parse:nn } #1 #2 {
1419   \BNVS_begin:
1420   \__bnvs_tl_set:cn { a } { #1 }

```

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

```

1421   \__bnvs_tl_put_left:cv { a } { root }
1422   \__bnvs_if_id_QF_name_n:vTF { a } {
1423     \regex_match:nnTF { \S } { #2 } {
1424       \peek_remove_spaces:n {
1425         \peek_catcode:NTF \c_group_begin_token {

```

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

```

1426         \__bnvs_tl_if_empty:cF { n } {
1427   \BNVS_warning:n { Ignoring~unexpected~suffix~.n:~#1 }
1428   }

```

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

```

1429   \BNVS_begin:
1430   \__bnvs_reset:v { QF_name }
1431   \__bnvs_gput:nvn V { QF_name } { \q_nil }
1432   \__bnvs_tl_set:cv { root } { QF_name }
1433   \__bnvs_tl_put_right:cn { root } { . }
1434   \__bnvs_int_set:cn { i } { 0 }
1435   \cs_set:Npn \BNVS:w ##1 ##2 \s_stop {
1436     \regex_match:nnT { \S } { ##2 } {
1437       \BNVS_error:n { Unexpected~value~#2 }
1438     }
1439     \keyval_parse:nnn {
1440       \__bnvs_parse:n
1441     } {
1442       \__bnvs_parse:nn
1443     } { ##1 }
1444     \BNVS_end:

```

If there is no `#1` counter but `#1.1` or `#1.first` is defined, define the counter.

```

1445   \cs_set:Npn \BNVS: {
1446     \cs_set:Npn \BNVS: {
1447       \__bnvs_gput:nvn V { QF_name } {a }

```

```

1448     }
1449     \exp_args:Nnx \__bnvs_if_get:nncF V {
1450       \__bnvs_tl_use:c { QF_name } .1
1451     } {a } {
1452       \exp_args:Nnx \__bnvs_if_get:nncF V {
1453         \__bnvs_tl_use:c { QF_name } .first
1454       } {a } {
1455         \cs_set:Npn \BNVS: { }
1456       }
1457     }
1458     \BNVS:
1459   }
1460   \__bnvs_if_get:nncT V { #1 } {a } {
1461     \__bnvs_quark_if_nil:cF {a } {
1462       \cs_set:Npn \BNVS: { }
1463     }
1464   }
1465   \BNVS:
1466 }
1467 \BNVS:w
1468 } {

```

Next character is not a group begin token.

```

1469     \__bnvs_tl_if_empty:cTF { n } {
1470       \__bnvs_parse_record:vn
1471     } {
1472       \__bnvs_n_parse_record:vn
1473     }
1474     { QF_name } { #2 }
1475     \use_none_delimit_by_s_stop:w
1476   }
1477 }
1478 #2 \s_stop
1479 } {

```

Empty value given: remove the reference.

```

1480     \__bnvs_tl_if_empty:cTF { n } {
1481       \__bnvs_gclear:v
1482     } {
1483       \__bnvs_n_gremove:v
1484     }
1485     { QF_name }
1486   }
1487 } {
1488   \BNVS_error:n { Invalid-name:~#2 }
1489 }

```

We export \l__bnvs_id_last_tl:

```

1490   \__bnvs_match_if_once:NvT \c__bnvs_one_suffix_regex { QF_name } {
1491     \__bnvs_if_match_pop_left:cTF { a } {
1492       \__bnvs_if_match_pop_left:cTF { a } {
1493         \cs_set:Npn \BNVS: {
1494           \__bnvs_gput:nvn V { a } { #2 }
1495         }
1496       }

```

```

1497         \__bnvs_quark_if_nil:cF { b } {
1498             \cs_set:Npn \BNVS: { }
1499         }
1500     }
1501     \BNVS:
1502 } {
1503     \BNVS_error:n { Ubnreachable~2 }
1504 }
1505 } {
1506     \BNVS_error:n { Ubnreachable~1 }
1507 }
1508 }
1509 \BNVS_end_tl_set:cv { id_last } { id_last }
1510 }

1511 \BNVS_new:cpn { parse_prepare:N } #1 {
1512     \tl_set:Nx #1 #1
1513     \bool_set_false:N \l__bnvs_parse_bool
1514     \bool_do_until:Nn \l__bnvs_parse_bool {
1515         \tl_if_in:NnTF #1 {%---[
1516     ]} {
1517         \regex_replace_all:nnNF { \[ ( [^\]]%---)
1518     ]*%---[(
1519     ) \] } { { { \1 } } } #1 {
1520         \bool_set_true:N \l__bnvs_parse_bool
1521     }
1522     } {
1523         \bool_set_true:N \l__bnvs_parse_bool
1524     }
1525 }
1526 \tl_if_in:NnTF #1 {%---[
1527 ]} {
1528     \BNVS_error:n { Unbalanced~%---[
1529 ]}
1530 } {
1531     \tl_if_in:NnT #1 { [%---]
1532     } {
1533         \BNVS_error:n { Unbalanced~[ %---]
1534     }
1535     }
1536 }
1537 }

```

\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.

```

1538 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1539     \BNVS_tl_use:nv {
1540         \BNVS_end:
1541         \__bnvs_tl_put_right:cn { #1 }
1542     } { #2 }
1543 }

```

```

1544 \cs_new:Npn \BNVS_end_v_gput:nv #1 {
1545   \BNVS_tl_use:nv {
1546     \BNVS_end:
1547     \__bnvs_v_gput:nn { #1 }
1548   }
1549 }

1550 \NewDocumentCommand \Beanoves { sm } {
1551   \__bnvs_set_false:c { reset }
1552   \__bnvs_set_false:c { reset_all }
1553   \__bnvs_set_false:c { only }
1554   \tl_if_empty:NTF \@currentenvir {

```

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

```

1555   \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
1556 } {
1557   \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, clear everything.

```

1558   \__bnvs_gclear:
1559 }
1560 \BNVS_begin:
1561 \__bnvs_tl_clear:c { root }
1562 \__bnvs_int_zero:c { i }
1563 \__bnvs_tl_set:cn { a } { #2 }
1564 \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, use the global definitions.

```

1565   \seq_if_empty:NF \g__bnvs_def_seq {
1566     \__bnvs_tl_put_left:cx { a } {
1567       \seq_use:Nn \g__bnvs_def_seq , ,
1568     }
1569   }
1570 }
1571 \__bnvs_parse_prepare:N \l__bnvs_a_tl
1572 \IfBooleanTF {#1} {
1573   \__bnvs_provide_on:
1574 } {
1575   \__bnvs_provide_off:
1576 }
1577 \BNVS_tl_use:nv {
1578   \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn }
1579 } { a }
1580 \BNVS_end_tl_set:cv { id_last } { id_last }
1581 \ignorespaces
1582 }
1583 }

```

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

```

1584 \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@masterdecode</code>	<code>_bnvs@frame {⟨overlay specification⟩}</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.

```

1585 \makeatletter
1586 \cs_set:Npn \_bnvs@frame < #1 > {
1587   \BNVS_begin:
1588   \_bnvs_tl_clear:c { ans }
1589   \_bnvs_scan:nNc { #1 } \_bnvs_if_resolve:ncTF { ans }
1590   \BNVS_set:cpn { :n } ##1 { \BNVS_end: \_bnvs_saved@frame < ##1 > }
1591   \BNVS_tl_use:cv { :n } { ans }
1592 }

1593 \cs_set:Npn \_bnvs@masterdecode #1 {
1594   \BNVS_begin:
1595   \_bnvs_tl_clear:c { ans }
1596   \_bnvs_scan:nNc { #1 } \_bnvs_if_resolve_queries:ncTF { ans }
1597   \BNVS_tl_use:nv {
1598     \BNVS_end:
1599     \_bnvs_saved@masterdecode
1600   } { ans }
1601 }

1602 \cs_new:Npn \BeanovesOff {
1603   \cs_set_eq:NN \beamer@frame \_bnvs_saved@frame
1604   \cs_set_eq:NN \beamer@masterdecode \_bnvs_saved@masterdecode
1605 }

1606 \cs_new:Npn \BeanovesOn {
1607   \cs_set_eq:NN \beamer@frame \_bnvs@frame
1608   \cs_set_eq:NN \beamer@masterdecode \_bnvs@masterdecode
1609 }

1610 \AddToHook{begindocument/before}{
1611   \cs_if_exist:NTF \beamer@frame {
1612     \cs_set_eq:NN \_bnvs_saved@frame \beamer@frame
1613     \cs_set_eq:NN \_bnvs_saved@masterdecode \beamer@masterdecode
1614   } {
1615     \cs_set:Npn \_bnvs_saved@frame < #1 > {
1616       \BNVS_error:n {Missing~package~beamer}
1617     }
1618     \cs_set:Npn \_bnvs_saved@masterdecode < #1 > {
1619       \BNVS_error:n {Missing~package~beamer}
1620     }
1621   }
1622   \BeanovesOn

```

```

1623 }
1624 \makeatother

```

```

\__bnvs_scan:nNc \__bnvs_scan:nNc {\<overlay query>} {\<resolve>} {\<ans>}

```

Scan the $\langle overlay\ query \rangle$ argument and feed the $\langle ans \rangle$ `tl` variable replacing $?(\dots)$ instructions by their static counterpart with help from the $\langle resolve \rangle$ function, which is $\backslash_bnvs_if_resolve:nCTF$. 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 $\backslash_bnvs_ans_tl$.)

```

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

```

(End of definition for \backslash_bnvs_int .)

```

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

```

(End of definition for $\backslash_bnvs_query_tl$.)

```

\l__bnvs_token_seq The \<overlay expression> is split into the sequence of its tokens.

```

(End of definition for $\backslash_bnvs_token_seq$.)

```

\l__bnvs_token_tl Storage for just one token.

```

(End of definition for $\backslash_bnvs_token_tl$.)

```

\__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 $\langle named\ overlay\ expression \rangle$ looking for a ‘?’ character. If a ‘ $?(\dots)$ ’ is found, then the $\langle code \rangle$ is executed.

```

1625 \BNVS_new:cpn { scan_for_query_then_end_return: } {
1626   \__bnvs_seq_pop_left:ccTF { token } { token } {
1627     \__bnvs_tl_if_eq:cnTF { token } { ? } {
1628       \__bnvs_scan_require_open_end_return:
1629     } {
1630       \__bnvs_tl_put_right:cv { ans } { token }
1631       \__bnvs_scan_for_query_then_end_return:
1632     }
1633   } {
1634     \__bnvs_scan_end_return_true:
1635   }
1636 }

```

__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.

```
1637 \BNVS_new:cpn { scan_require_open_end_return: } {
```

Get next token.

```
1638 \__bnvs_seq_pop_left:ccTF { token } { token } {
1639   \str_if_eq:VnTF \l__bnvs_token_tl { ( % )
1640   } {
```

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

```
1641   \__bnvs_int_set:cn { } { 1 }
```

Record the forthcoming content in the \l__bnvs_query_tl variable, up to the next balancing ')':

```
1642   \__bnvs_tl_clear:c { query }
1643   \__bnvs_scan_require_close_and_return:
1644   }
```

Ignore this token and loop.

```
1645   \__bnvs_scan_require_open_end_return:
1646   }
1647 }
```

Get next token.

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

```
1648 \BNVS_error:x {Missing~'('%---)
1649   ~after~a~? }
1650 \__bnvs_scan_end_return_true:
1651 }
1652 }
```

__bnvs_scan_require_close_and_return: __bnvs_scan_require_close_and_return:

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

```
1653 \BNVS_new:cpn { scan_require_close_and_return: } {
```

Get next token.

```
1654 \__bnvs_seq_pop_left:ccTF { token } { token } {
1655   \str_case:VnF \l__bnvs_token_tl {
1656     { ( %---)
1657   } {
```

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

```
1658   \__bnvs_int_incr:c { }
1659   \__bnvs_tl_put_right:cv { query } { token }
1660   \__bnvs_scan_require_close_and_return:
1661   }
1662   { %(---
1663   )
1664   }
```


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

```
1665         \__bnvs_int_decr:c { }
1666         \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 ‘?’.

```
1667         \__bnvs_scan_handle_query_then_end_return:
1668     } {
```

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

```
1669         \__bnvs_tl_put_right:cv { query } { token }
1670         \__bnvs_scan_require_close_and_return:
1671     }
1672 }
1673 }
```

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

```
1674         \__bnvs_tl_put_right:cv { query } { token }
1675         \__bnvs_scan_require_close_and_return:
1676     }
1677 }
```

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 ‘)’.

```
1678     \BNVS_error:x { Missing~%(---
1679     `)' }
1680     \__bnvs_tl_put_right:cx { query } {
1681     \prg_replicate:nn { \l__bnvs_int } {%(---
1682     )}
1683 }
1684     \__bnvs_scan_end_return_true:
1685 }
1686 }
```

```
1687 \BNVS_new_conditional:cpnn { scan:nNc } #1 #2 #3 { T, F, TF } {
1688     \BNVS_begin:
1689     \BNVS_set:cpn { error:x } ##1 {
1690         \msg_error:nnx { beanoves } { :n }
1691         { \tl_to_str:n { #1 } :~##1 }
1692     }
1693     \__bnvs_tl_set:cn { scan } { #1 }
1694     \__bnvs_tl_clear:c { ans }
1695     \__bnvs_seq_clear:c { token }
```

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

```
1696     \regex_split:nnN { } { #1 } \l__bnvs_token_seq
```

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

```
1697     \BNVS_set:cpn { scan_handle_query_then_end_return: } {
1698     \BNVS_tl_use:Nv #2 { query } { ans } {
```

```

1699     \__bnvs_scan_for_query_then_end_return:
1700   } {
1701     \BNVS_end_tl_put_right:cv { #3 } { ans }

```

Stop on the first error.

```

1702     \prg_return_false:
1703   }
1704 }
1705 \BNVS_set:cpn { scan_end_return_true: } {
1706   \BNVS_end_tl_put_right:cv { #3 } { ans }
1707   \prg_return_true:
1708 }
1709 \BNVS_set:cpn { scan_end_return_false: } {
1710   \BNVS_end_tl_put_right:cv { #3 } { ans }
1711   \prg_return_false:
1712 }
1713 \__bnvs_scan_for_query_then_end_return:
1714 }
1715 \BNVS_new:cpn { scan:nNc } #1 #2 #3 {
1716   \BNVS_use:c { scan:nNcTF } { #1 } #2 { #3 } {} {}
1717 }

```

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_Qip:cccTF \__bnvs_if_Qip:cccTF {<QF name>} {<id>} {<path>} {<yes code>} {<no code>}

```

Auxiliary function. On input, the $\langle QF \text{ name} \rangle$ tl variable contains a set name whereas the $\langle id \rangle$ tl variable contains a frame id. If $\langle name \rangle$ tl variable contents is a recorded set, on return, $\langle QF \text{ name} \rangle$ tl variable contains the resolved name, $\langle id \rangle$ tl variable contains the used frame id, $\langle path \rangle$ seq variable is prepended with new dotted path components, $\langle yes \text{ code} \rangle$ is executed, otherwise variables are left untouched and $\langle no \text{ code} \rangle$ is executed.

```

1718 \BNVS_new_conditional:cpnn { if_Qip:ccc } #1 #2 #3 { T, F, TF } {
1719   \BNVS_begin:
1720   \__bnvs_match_if_once:NvTF \c__bnvs_A_QF_name_Z_regex { #1 } {

```

This is a correct QF name, update the path sequence accordingly.

```

1721   \__bnvs_if_match_pop_Qip:cccTF { #1 } { #2 } { #3 } {
1722     \__bnvs_end_Qip_export:ccc { #1 } { #2 } { #3 }
1723     \prg_return_true:
1724   } {
1725     \BNVS_end:
1726     \prg_return_false:
1727   }
1728 } {
1729   \BNVS_end:
1730   \prg_return_false:
1731 }
1732 }
1733 \quark_new:N \q__bnvs

```

```

1734 \BNVS_new:cpn { end_Qip_export:ccc } #1 #2 #3 {
1735   \exp_args:Nnx
1736   \use:n {
1737     \BNVS_tl_use:nv {
1738       \BNVS_tl_use:cv { end_Qip_export:nnccc } { #1 }
1739     } { #2 }
1740   } { \_bnvs_seq_use:cn { #3 } { \q__bnvs } } { #1 } { #2 } { #3 }
1741 }

1742 \BNVS_new:cpn { end_Qip_export:nnccc } #1 #2 #3 #4 #5 #6 {
1743   \BNVS_end:
1744   \tl_if_empty:nTF { #2 } {
1745     \_bnvs_tl_set:cn { #4 } { #1 }
1746     \_bnvs_tl_put_left:cv { #4 } { #5 }
1747   } {
1748     \_bnvs_tl_set:cn { #4 } { #1 }
1749     \_bnvs_tl_set:cn { #5 } { #2 }
1750   }
1751   \_bnvs_seq_set_split:cn { #6 } { \q__bnvs } { #3 }
1752   \_bnvs_seq_remove_all:cn { #6 } { }
1753 }

```

Sets the QF_name and id to the heading items of the match sequence. Sets the path sequence to the components of the path variable as dotted path.

```

1754 \BNVS_new_conditional:cpnn { if_match_pop_Qip:ccc } #1 #2 #3 { TF } {
1755   \_bnvs_if_match_pop_left:cTF { #1 } {
1756     \_bnvs_if_match_pop_left:cTF { #1 } {
1757       \_bnvs_if_match_pop_left:cTF { #2 } {
1758         \_bnvs_if_match_pop_left:cTF { #3 } {
1759           \_bnvs_seq_set_split:cnv { #3 } { . } { #3 }
1760           \_bnvs_seq_remove_all:cn { #3 } { }
1761           \prg_return_true:
1762         } {
1763           \prg_return_false:
1764         }
1765       } {
1766         \prg_return_false:
1767       }
1768     } {
1769       \prg_return_false:
1770     }
1771   } {
1772     \prg_return_false:
1773   }
1774 }

```

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.

```

1775 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
1776   \__bnvs_seq_if_empty:cF { #2 } {
1777     \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
1778       \__bnvs_seq_use:cn { #1 } { \q__bnvs }
1779       \exp_not:n { \q__bnvs }
1780       \__bnvs_seq_use:cn { #2 } { \q__bnvs }
1781     }
1782     \__bnvs_seq_remove_all:cn { #1 } { }
1783   }
1784 }

```

6.18 Evaluation bricks

We start by helpers.

```

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

```

Replaces the variable content with its rounded floating point evaluation.

```

1785 \BNVS_new:cpn { round:N } #1 {
1786   \tl_if_empty:NTF #1 {
1787     \tl_set:Nn #1 { 0 }
1788   } {
1789     \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
1790   }
1791 }

1792 \BNVS_new:cpn { round:c } {
1793   \BNVS_tl_use:Nc \__bnvs_round:N
1794 }

```

```

\__bnvs_if_assign_value:nnTF \__bnvs_if_assign_value:nnTF <{QF_name}> <value> <{yes code}> <{no
\__bnvs_if_assign_value:(nv|vv)TF code>

```

```

1795 \BNVS_new_conditional:cpnn { if_assign_value:nn } #1 #2 { T, F, TF } {
1796   \BNVS_begin:
1797   \__bnvs_if_resolve:ncTF { #2 } { a } {
1798     \__bnvs_gclear_all:n { #1 }
1799     \__bnvs_gput:nnv V { #1 } { a }
1800     \__bnvs_c_gput:nnv V { #1 } { a }
1801     \__bnvs_v_gput:nv { #1 } { a }
1802     \BNVS_end:
1803     \prg_return_true:
1804   } {
1805     \BNVS_end:
1806     \prg_return_false:
1807   }
1808 }

```

```

1809 \BNVS_new_conditional:cpnn { if_assign_value:nv } #1 #2 { T, F, TF } {
1810   \BNVS_tl_use:nv {
1811     \BNVS_use:c { if_assign_value:nnTF } { #1 }
1812   } { #2 } {
1813     \prg_return_true:
1814   } {
1815     \prg_return_false:
1816   }
1817 }

1818 \BNVS_new_conditional:cpnn { if_assign_value:vv } #1 #2 { T, F, TF } {
1819   \BNVS_tl_use:nv {
1820     \BNVS_tl_use:cv { if_assign_value:nnTF } { #1 }
1821   } { #2 } {
1822     \prg_return_true:
1823   } {
1824     \prg_return_false:
1825   }
1826 }

```

```

\__bnvs_if_resolve_V:ncTF \__bnvs_if_resolve_V:ncTF {<QFname>} <ans> {<yes code>} {<no code>}
\__bnvs_if_resolve_V:vcTF \__bnvs_if_append_V:ncTF {<QFname>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_V:ncTF
\__bnvs_if_append_V:(xc|vc)TF

```

Resolve the content of the $\langle QF_n \text{ame} \rangle$ value counter into the $\langle \text{ans} \rangle$ t1 variable or append this value to the right of the variable. Execute $\langle \text{yes code} \rangle$ when there is a $\langle \text{value} \rangle$, $\langle \text{no code} \rangle$ otherwise. Inside the $\langle \text{no code} \rangle$ branch, the content of the $\langle \text{ans} \rangle$ t1 variable is undefined. Implementation detail: in $\langle \text{ans} \rangle$ 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.

```

1827 \BNVS_new:cpn { if_resolve_V_return:nncT } #1 #2 #3 #4 {
1828   \__bnvs_tl_if_empty:cTF { #3 } {
1829     \prg_return_false:
1830   } {
1831     \__bnvs_c_gput:nnv V { #2 } { #3 }
1832     #4
1833     \prg_return_true:
1834   }
1835 }

1836 \BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
1837   \BNVS_tl_use:Nc \quark_if_nil:NTF { #1 } {
1838     \prg_return_true:
1839   } {
1840     \prg_return_false:
1841   }
1842 }

```

```

1843 \BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
1844   \BNVS_tl_use:Nc \quark_if_no_value:NTF { #1 } {
1845     \prg_return_true:
1846   } {
1847     \prg_return_false:
1848   }
1849 }

1850 \makeatletter
1851 \BNVS_new_conditional:cpnn { if_resolve_V:nc } #1 #2 { T, F, TF } {
1852   \__bnvs_c_if_get:nncTF V { #1 } { #2 } {
1853     \prg_return_true:
1854   } {
1855     \__bnvs_if_get:nncTF V { #1 } { #2 } {
1856       \__bnvs_quark_if_nil:cTF { #2 } {

```

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

```

1857   \__bnvs_gput:nnn V { #1 } { \q_no_value }
1858   \__bnvs_if_resolve_A:ncTF { #1 } { #2 } {
1859     \__bnvs_if_resolve_V_return:nncT A { #1 } { #2 } {
1860       \__bnvs_gput:nnn V { #1 } { \q_nil }
1861     }
1862   } {
1863     \__bnvs_if_resolve_Z:ncTF { #1 } { #2 } {
1864       \__bnvs_if_resolve_V_return:nncT Z { #1 } { #2 } {
1865         \__bnvs_gput:nnn V { #1 } { \q_nil }
1866       }
1867     } {
1868       \__bnvs_gput:nnn V { #1 } { \q_nil }
1869       \prg_return_false:
1870     }
1871   }
1872 } {
1873   \__bnvs_quark_if_no_value:cTF { #2 } {
1874     \BNVS_fatal:n {Circular~definition:~#1}
1875   } {

```

Possible recursive call.

```

1876   \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1877     \__bnvs_if_resolve_V_return:nncT V { #1 } { #2 } {
1878       \__bnvs_gput:nnn V { #1 } { \q_nil }
1879     }
1880   } {
1881     \__bnvs_gput:nnn V { #1 } { \q_nil }
1882     \prg_return_false:
1883   }
1884 }
1885 }
1886 } {
1887   \str_if_eq:nnTF { #1 } { ?!pauses } {
1888     \cs_if_exist:NTF \c@beamerpauses {
1889       \exp_args:Nnx \__bnvs_tl_set:cn { #2 } { \the\c@beamerpauses }
1890       \prg_return_true:
1891     } {

```

```

1892         \prg_return_false:
1893     }
1894 } {
1895     \prg_return_false:
1896 }
1897 }
1898 }
1899 }
1900 \makeatother
1901 \BNVS_new_conditional_vc:cn { if_resolve_V } { T, F, TF }

1902 \BNVS_new:cpn { end_put_right:vc } #1 #2 {
1903     \BNVS_tl_use:nv {
1904         \BNVS_end:
1905         \__bnvs_tl_put_right:cn { #2 }
1906     } { #1 }
1907 }

1908 \BNVS_new_conditional:cpnn { if_append_V:nc } #1 #2 { T, F, TF } {
1909     \BNVS_begin:
1910     \__bnvs_if_resolve_V:ncTF { #1 } { #2 } {
1911         \BNVS_end_tl_put_right:cv { #2 } { #2 }
1912         \prg_return_true:
1913     } {
1914         \BNVS_end:
1915         \prg_return_false:
1916     }
1917 }
1918 \BNVS_new_conditional_vc:cn { if_append_V } { T, F, TF }

```

```

\__bnvs_if_resolve_A:ncTF \__bnvs_if_resolve_A:ncTF {<QF name>} <tl core> {<yes code>} {<no code>}
\__bnvs_if_append_A:ncTF \__bnvs_if_append_A:ncTF {<QF name>} <tl core> {<yes code>} {<no code>}

```

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

```

1919 \BNVS_new_conditional:cpnn { if_resolve_A:nc } #1 #2 { T, F, TF } {
1920     \__bnvs_c_if_get:nncTF A { #1 } { #2 } {
1921         \prg_return_true:
1922     } {
1923         \__bnvs_if_get:nncTF A { #1 } { #2 } {
1924             \__bnvs_quark_if_nil:cTF { #2 } {
1925                 \__bnvs_gput:nnn A { #1 } { \q_no_value }

```

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

```

1926     \__bnvs_if_resolve_Z:ncTF { #1 } { #2 } {
1927         \__bnvs_tl_put_right:cn { #2 } { - }
1928         \__bnvs_if_append_L:ncTF { #1 } { #2 } {
1929             \__bnvs_tl_put_right:cn { #2 } { + 1 }
1930             \__bnvs_round:c { #2 }
1931             \__bnvs_tl_if_empty:cTF { #2 } {

```

```

1932         \_bnvs_gput:nnn A { #1 } { \q_nil }
1933         \prg_return_false:
1934     } {
1935         \_bnvs_gput:nnn A { #1 } { \q_nil }
1936         \_bnvs_c_gput:nnv A { #1 } { #2 }
1937         \prg_return_true:
1938     }
1939 } {
1940     \BNVS_error:n {
1941 Unavailable~length~for~#1~(\token_to_str:N\_bnvs_if_resolve_A:ncTF/2) }
1942         \_bnvs_gput:nnn A { #1 } { \q_nil }
1943         \prg_return_false:
1944     }
1945 } {
1946     \BNVS_error:n {
1947 Unavailable~last~for~#1~(\token_to_str:N\_bnvs_if_resolve_A:ncTF/1) }
1948         \_bnvs_gput:nnn A { #1 } { \q_nil }
1949         \prg_return_false:
1950     }
1951 } {
1952     \_bnvs_quark_if_no_value:cTF { a } {
1953         \BNVS_fatal:n {Circular~definition:~#1}
1954     } {
1955         \_bnvs_if_resolve:vcTF { #2 } { #2 } {
1956             \_bnvs_c_gput:nnv A { #1 } { #2 }
1957             \prg_return_true:
1958         } {
1959             \prg_return_false:
1960         }
1961     }
1962 }
1963 } {
1964     \prg_return_false:
1965 }
1966 }
1967 }

1968 \BNVS_new_conditional:cpnn { if_append_A:nc } #1 #2 { T, F, TF } {
1969     \BNVS_begin:
1970     \_bnvs_if_resolve_A:ncTF { #1 } { #2 } {
1971         \BNVS_end_tl_put_right:cv { #2 } { #2 }
1972         \prg_return_true:
1973     } {
1974         \BNVS_end:
1975         \prg_return_false:
1976     }
1977 }

```

$\backslash_bnvs_if_resolve_Z:ncTF$ $\backslash_bnvs_if_append_Z:ncTF$	$\backslash_bnvs_if_resolve_Z:ncTF \{ \langle QF \ name \rangle \} \langle ans \rangle \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$ $\backslash_bnvs_if_append_Z:ncTF \{ \langle QF \ name \rangle \} \langle ans \rangle \{ \langle yes \ code \rangle \} \{ \langle no \ code \rangle \}$
---	---

Resolve the last index of the fully qualified $\langle QF \ name \rangle$ range into or to the right of the right of the $\langle tl \ variable \rangle$, when possible. Execute $\langle yes \ code \rangle$ when a last index was given, $\langle no \ code \rangle$ otherwise.


```

1978 \BNVS_new_conditional:cpnn { if_resolve_Z:nc } #1 #2 { T, F, TF } {
1979   \__bnvs_c_if_get:nncTF Z { #1 } { #2 } {
1980     \prg_return_true:
1981   } {
1982     \__bnvs_if_get:nncTF Z { #1 } { #2 } {
1983       \__bnvs_quark_if_nil:cTF { #2 } {
1984         \__bnvs_gput:nnn Z { #1 } { \q_no_value }

```

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

```

1985   \__bnvs_if_resolve_A:ncTF { #1 } { #2 } {
1986     \__bnvs_tl_put_right:cn { #2 } { + }
1987     \__bnvs_if_append_L:ncTF { #1 } { #2 } {
1988       \__bnvs_tl_put_right:cn { #2 } { - 1 }
1989       \__bnvs_round:c { #2 }
1990       \__bnvs_c_gput:nnv Z { #1 } { #2 }
1991       \__bnvs_gput:nnn Z { #1 } { \q_nil }
1992       \prg_return_true:
1993     } {
1994       \BNVS_error:x {
1995   Unavailable~last~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
1996       \__bnvs_gput:nnn Z { #1 } { \q_nil }
1997       \prg_return_false:
1998     }
1999   } {
2000     \BNVS_error:x {
2001   Unavailable~first~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
2002     \__bnvs_gput:nnn Z { #1 } { \q_nil }
2003     \prg_return_false:
2004   }
2005   } {
2006     \__bnvs_quark_if_no_value:cTF { #2 } {
2007       \BNVS_fatal:n {Circular~definition:~#1}
2008       \prg_return_false:
2009     } {
2010       \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2011         \__bnvs_c_gput:nnv Z { #1 } { #2 }
2012         \prg_return_true:
2013       } {
2014         \prg_return_false:
2015       }
2016     }
2017   }
2018   } {
2019     \prg_return_false:
2020   }
2021 }
2022 }
2023 \BNVS_new_conditional_vc:cn { if_resolve_Z } { T, F, TF }
2024 \BNVS_new_conditional:cpnn { if_append_Z:nc } #1 #2 { T, F, TF } {
2025   \BNVS_begin:
2026   \__bnvs_if_resolve_Z:ncTF { #1 } { #2 } {
2027     \BNVS_end_tl_put_right:cv { #2 } { #2 }

```

```

2028     \prg_return_true:
2029 } {
2030     \BNVS_end:
2031     \prg_return_false:
2032 }
2033 }
2034 \BNVS_new_conditional_vc:cn { if_append_Z } { T, F, TF }

```

```

\__bnvs_if_resolve_L:ncTF \__bnvs_if_resolve_L:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_L:ncTF \__bnvs_if_append_L:ncTF {<QF name>} <ans> {<yes code>} {<no code>}

```

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

```

2035 \BNVS_new_conditional:cpnn { if_resolve_L:nc } #1 #2 { T, F, TF } {
2036     \__bnvs_c_if_get:nncTF L { #1 } { #2 } {
2037         \prg_return_true:
2038     } {
2039         \__bnvs_if_get:nncTF L { #1 } { #2 } {
2040             \__bnvs_quark_if_nil:cTF { #2 } {
2041                 \__bnvs_gput:nnn L { #1 } { \q_no_value }

```

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

```

2042     \__bnvs_if_resolve_Z:ncTF { #1 } { #2 } {
2043         \__bnvs_tl_put_right:cn { #2 } { - }
2044         \__bnvs_if_append_A:ncTF { #1 } { #2 } {
2045             \__bnvs_tl_put_right:cn { #2 } { + 1 }
2046             \__bnvs_round:c { #2 }
2047             \__bnvs_gput:nnn L { #1 } { \q_nil }
2048             \__bnvs_c_gput:nnv L { #1 } { #2 }
2049             \prg_return_true:
2050         } {
2051             \BNVS_error:n {
2052 Unavailable~first~for~#1~(\__bnvs_if_resolve_L:ncTF/2) }
2053             \prg_return_false:
2054         }
2055     } {
2056         \BNVS_error:n {
2057 Unavailable~last~for~#1~(\__bnvs_if_resolve_L:ncTF/1) }
2058         \prg_return_false:
2059     }
2060 } {
2061     \__bnvs_quark_if_no_value:cTF { #2 } {
2062         \BNVS_fatal:n {Circular~definition:~#1}
2063     } {
2064         \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2065             \__bnvs_c_gput:nnv L { #1 } { #2 }
2066             \prg_return_true:
2067         } {
2068             \prg_return_false:
2069         }
2070     }
2071 }
2072 } {

```

```

2073     \prg_return_false:
2074   }
2075 }
2076 }
2077 \BNVS_new_conditional_vc:cn { if_resolve_L } { T, F, TF }
2078 \BNVS_new_conditional_cpnn { if_append_L:nc } #1 #2 { T, F, TF } {
2079   \BNVS_begin:
2080     \__bnvs_if_resolve_L:ncTF { #1 } { #2 } {
2081       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2082       \prg_return_true:
2083     } {
2084       \prg_return_false:
2085     }
2086   }
2087 \BNVS_new_conditional_vc:cn { if_append_L } { T, F, TF }

```

```

\__bnvs_if_resolve_previous:ncTF \__bnvs_if_append_previous:ncTF {<QF name>} <ans> {<yes code>} {<no
\__bnvs_if_append_previous:ncTF 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.

```

\__bnvs_if_resolve_first:ncTF \__bnvs_if_resolve_first:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\__bnvs_if_resolve_first:vcTF \__bnvs_if_append_first:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_first:ncTF
\__bnvs_if_append_first:vcTF

```

Resolve the first index starting the *<QF name>* slide range into the *<ans>* tl variable, or append this index to that variable. Execute *<yes code>* when there is a *<first>* index, *<no code>* otherwise. In the latter case, the content of the *<ans>* tl variable is undefined, on resolution only.

```

2088 \BNVS_new_conditional_cpnn { if_resolve_first:nc } #1 #2 { T, F, TF } {
2089   \exp_args:Nx \__bnvs_if_resolve_V:ncTF {
2090     \__bnvs_tl_use:c { QF_name }.first
2091   } { #2 } { \prg_return_true: } {
2092     \__bnvs_if_resolve_A:ncTF { #1 } { #2 } { \prg_return_true: } {
2093       \exp_args:Nx \__bnvs_if_resolve_v:ncTF {
2094         \__bnvs_tl_use:c { QF_name }.1
2095       } { #2 } { \prg_return_true: } { \prg_return_false: }
2096     }
2097   }
2098 }
2099 \BNVS_new_conditional_vc:cn { if_resolve_first } { T, F, TF }
2100 \BNVS_new_conditional_cpnn { if_append_first:nc } #1 #2 { T, F, TF } {
2101   \__bnvs_if_append_A:ncTF { #1 } { #2 } { \prg_return_true: } {
2102     \exp_args:Nx \__bnvs_if_append_v:ncTF {
2103       \BNVS_tl_use:c { QF_name }.1
2104     } { #2 } { \prg_return_true: } { \prg_return_false: }
2105   }
2106 }

```

```
2107 \BNVS_new_conditional_vc:cn { if_append_first } { T, F, TF }
```

```
\_bnvs_if_resolve_last:ncTF \_bnvs_if_resolve_last:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\_bnvs_if_resolve_last:vcTF \_bnvs_if_append_last:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\_bnvs_if_append_last:ncTF
\_bnvs_if_append_last:vcTF
```

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

```
2108 \BNVS_new_conditional:cpnn { if_resolve_last:nc } #1 #2 { T, F, TF } {
2109   \_bnvs_if_resolve_Z:ncTF { #1 } { #2 }
2110   { \prg_return_true: } { \prg_return_false: }
2111 }
2112 \BNVS_new_conditional_vc:cn { if_resolve_last } { T, F, TF }

2113 \BNVS_new_conditional:cpnn { if_append_last:nc } #1 #2 { T, F, TF } {
2114   \_bnvs_if_append_Z:ncTF { #1 } { #2 }
2115   { \prg_return_true: } { \prg_return_false: }
2116 }
2117 \BNVS_new_conditional_vc:cn { if_append_last } { T, F, TF }
```

```
\_bnvs_if_resolve_length:ncTF \_bnvs_if_resolve_length:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\_bnvs_if_resolve_length:vcTF \_bnvs_if_append_length:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\_bnvs_if_append_length:ncTF
\_bnvs_if_append_length:vcTF
```

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

```
2118 \BNVS_new_conditional:cpnn { if_resolve_length:nc } #1 #2 { T, F, TF } {
2119   \_bnvs_if_resolve_L:ncTF { #1 } { #2 }
2120   { \prg_return_true: } { \prg_return_false: }
2121 }
2122 \BNVS_new_conditional_vc:cn { if_resolve_length } { T, F, TF }

2123 \BNVS_new_conditional:cpnn { if_append_length:nc } #1 #2 { T, F, TF } {
2124   \_bnvs_if_append_L:ncTF { #1 } { #2 }
2125   { \prg_return_true: } { \prg_return_false: }
2126 }
2127 \BNVS_new_conditional_vc:cn { if_append_length } { T, F, TF }
```

```
\_bnvs_if_resolve_range:ncTF \_bnvs_if_resolve_range:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\_bnvs_if_append_range:ncTF \_bnvs_if_append_range:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
```

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

```
2128 \BNVS_new_conditional:cpnn { if_append_range:nc } #1 #2 { T, F, TF } {
```

```

2129 \BNVS_begin:
2130 \__bnvs_if_resolve_A:ncTF { #1 } { a } {
2131     \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
2132         \__bnvs_tl_set:cn { a } { 0 }
2133     }
2134     \__bnvs_if_resolve_Z:ncTF { #1 } { b } {

```

Limited from above and below.

```

2135         \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2136             \__bnvs_tl_set:cn { b } { 0 }
2137         }
2138         \__bnvs_tl_put_right:cn { a } { - }
2139         \__bnvs_tl_put_right:cv { a } { b }
2140         \BNVS_end_tl_put_right:cv { #2 } { a }
2141         \prg_return_true:
2142     } {

```

Limited from below.

```

2143         \BNVS_end_tl_put_right:cv { #2 } { a }
2144         \__bnvs_tl_put_right:cn { #2 } { - }
2145         \prg_return_true:
2146     }
2147 } {
2148     \__bnvs_if_resolve_Z:ncTF { #1 } { b } {

```

Limited from above.

```

2149         \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2150             \__bnvs_tl_set:cn { b } { 0 }
2151         }
2152         \__bnvs_tl_put_left:cn { b } { - }
2153         \BNVS_end_tl_put_right:cv { #2 } { b }
2154         \prg_return_true:
2155     } {
2156         \__bnvs_if_resolve_V:ncTF { #1 } { b } {
2157             \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2158                 \__bnvs_tl_set:cn { b } { 0 }
2159             }

```

Unlimited range.

```

2160         \BNVS_end_tl_put_right:cv { #2 } { b }
2161         \__bnvs_tl_put_right:cn { #2 } { - }
2162         \prg_return_true:
2163     } {
2164         \BNVS_end:
2165         \prg_return_false:
2166     }
2167 }
2168 }
2169 }
2170 \BNVS_new_conditional_vc:cn { if_append_range } { T, F, TF }

2171 \BNVS_new_conditional_cpnn { if_resolve_range:nc } #1 #2 { T, F, TF } {
2172     \__bnvs_tl_clear:c { #2 }
2173     \__bnvs_if_append_range:ncTF { #1 } { #2 } {

```

```

2174 \prg_return_true:
2175 } {
2176 \prg_return_false:
2177 }
2178 }
2179 \BNVS_new_conditional_vc:cn { if_resolve_range } { T, F, TF }

```

```

\__bnvs_if_resolve_previous:ncTF \__bnvs_if_append_previous:ncTF {<QF name>} <ans> {<yes code>} {<no
\__bnvs_if_append_previous:ncTF code>}

```

Resolve the index after the *<key>* 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 *<tl variable>* is undefined on resolution only.

```

2180 \BNVS_new_conditional:cpnn { if_resolve_previous:nc } #1 #2 { T, F, TF } {
2181 \__bnvs_c_if_get:nncTF P { #1 } { #2 } {
2182 \prg_return_true:
2183 } {
2184 \__bnvs_if_resolve_A:ncTF { #1 } { #2 } {
2185 \__bnvs_tl_put_right:cn { #2 } { -1 }
2186 \__bnvs_round:c { #2 }
2187 \__bnvs_c_gput:nnv P { #1 } { #2 }
2188 \prg_return_true:
2189 } {
2190 \prg_return_false:
2191 }
2192 }
2193 }
2194 \BNVS_new_conditional_vc:cn { if_resolve_previous } { T, F, TF }

2195 \BNVS_new_conditional:cpnn { if_append_previous:nc } #1 #2 { T, F, TF } {
2196 \BNVS_begin:
2197 \__bnvs_if_resolve_previous:ncTF { #1 } { #2 } {
2198 \BNVS_end_tl_put_right:cv { #2 } { #2 }
2199 \prg_return_true:
2200 } {
2201 \BNVS_end:
2202 \prg_return_false:
2203 }
2204 }
2205 \BNVS_new_conditional_vc:cn { if_append_previous } { T, F, TF }

```

```

\__bnvs_if_resolve_next:ncTF \__bnvs_if_resolve_next:ncTF {<QF name>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_next:ncTF \__bnvs_if_append_next:ncTF {<QF name>} <ans> {<yes code>} {<no code>}

```

Resolve the index after the *<key>* 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 *<tl variable>* is undefined, on resolution only.

```

2206 \BNVS_new_conditional:cpnn { if_resolve_next:nc } #1 #2 { T, F, TF } {
2207 \__bnvs_c_if_get:nncTF N { #1 } { #2 } {

```

```

2208 \prg_return_true:
2209 } {
2210 \__bnvs_if_resolve_Z:ncTF { #1 } { #2 } {
2211 \__bnvs_tl_put_right:cn { #2 } { +1 }
2212 \__bnvs_round:c { #2 }
2213 \__bnvs_c_gput:nnv N { #1 } { #2 }
2214 \prg_return_true:
2215 } {
2216 \prg_return_false:
2217 }
2218 }
2219 }
2220 \BNVS_new_conditional_vc:cn { if_resolve_next } { T, F, TF }
2221 \BNVS_new_conditional_cpnn { if_append_next:nc } #1 #2 { T, F, TF } {
2222 \BNVS_begin:
2223 \__bnvs_if_resolve_next:ncTF { #1 } { #2 } {
2224 \BNVS_end_tl_put_right:cv { #2 } { #2 }
2225 \prg_return_true:
2226 } {
2227 \BNVS_end:
2228 \prg_return_true:
2229 }
2230 }
2231 \BNVS_new_conditional_vc:cn { if_append_next } { T, F, TF }

```

<pre> __bnvs_if_resolve_v:ncTF __bnvs_if_resolve_v:vcTF __bnvs_if_append_v:nc </pre>	<pre> __bnvs_if_resolve_v:ncTF {<QF name>} <ans> {<yes code>} {<no code>} __bnvs_if_append_v:ncTF {<QF name>} <ans> {<yes code>} {<no code>} </pre>
---	---

Resolve the value of the $\langle QF\ name \rangle$ overlay set 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. In the latter case, the content of the $\langle tl\ variable \rangle$ is undefined, on resolution only. Calls $\backslash_bnvs_if_resolve_V:ncTF$.

```

2232 \BNVS_new_conditional_cpnn { if_resolve_v:nc } #1 #2 { T, F, TF } {
2233 \__bnvs_v_if_get:ncTF { #1 } { #2 } {
2234 \__bnvs_quark_if_no_value:cTF { #2 } {
2235 \BNVS_fatal:n {Circular~definition:~#1}
2236 \prg_return_false:
2237 } {
2238 \prg_return_true:
2239 }
2240 } {
2241 \__bnvs_v_gput:nn { #1 } { \q_no_value }
2242 \__bnvs_if_resolve_V:ncTF { #1 } { #2 } {
2243 \__bnvs_v_gput:nv { #1 } { #2 }
2244 \prg_return_true:
2245 } {
2246 \__bnvs_if_resolve_A:ncTF { #1 } { #2 } {
2247 \__bnvs_v_gput:nv { #1 } { #2 }
2248 \prg_return_true:
2249 } {
2250 \__bnvs_if_resolve_Z:ncTF { #1 } { #2 } {

```

```

2251         \__bnvs_v_gput:nv { #1 } { #2 }
2252         \prg_return_true:
2253     } {
2254         \__bnvs_v_gremove:n { #1 }
2255         \prg_return_false:
2256     }
2257 }
2258 }
2259 }
2260 }
2261 \BNVS_new_conditional_vc:cn { if_resolve_v } { T, F, TF }
2262 \BNVS_new_conditional:cpnn { if_append_v:nc } #1 #2 { T, F, TF } {
2263     \BNVS_begin:
2264     \__bnvs_if_resolve_v:ncTF { #1 } { #2 } {
2265         \BNVS_end_t1_put_right:cv { #2 } { #2 }
2266         \prg_return_true:
2267     } {
2268         \BNVS_end:
2269         \prg_return_false:
2270     }
2271 }
2272 \BNVS_new_conditional_vc:cn { if_append_v } { T, F, TF }

```

<u>__bnvs_index_can:nTF</u> <u>__bnvs_index_can:vTF</u> <u>__bnvs_if_resolve_index:nncTF</u> <u>__bnvs_if_resolve_index:vvcTF</u> <u>__bnvs_if_append_index:nncTF</u> <u>__bnvs_if_append_index:vvcTF</u>	__bnvs_index_can:nTF {<QF name>} {<yes code>} {<no code>} __bnvs_if_resolve_index:nncTF {<QF name>} {<integer>} {<ans>} {<yes code>} {<no code>} __bnvs_if_append_index:nncTF {<QF name>} {<integer>} {<ans>} {<yes code>} {<no code>} __bnvs_if_append_index:vvcTF
--	--

Resolve the index associated to the *<QF name>* and *<integer>* slide range into the *<ans>* t1 variable or append this index to the right of that variable. When *<integer>* is 1, this is the first index, when *<integer>* is 2, this is the second index, and so on. When *<integer>* is 0, this is the index, before the first one, and so on. If the computation is possible, *<yes code>* is executed, otherwise *<no code>* is executed. In the latter case, the content of the *<ans>* t1 variable is undefined, on resolution only. The computation may fail when too many recursion calls are required.

```

2273 \BNVS_new_conditional:cpnn { index_can:n } #1 { p, T, F, TF } {
2274     \bool_if:nTF {
2275         \__bnvs_if_in_p:nn V { #1 }
2276         || \__bnvs_if_in_p:nn A { #1 }
2277         || \__bnvs_if_in_p:nn Z { #1 }
2278     } {
2279         \prg_return_true:
2280     } {
2281         \prg_return_false:
2282     }
2283 }

```



```

2284 \BNVS_new_conditional:cpnn { index_can:v } #1 { p, T, F, TF } {
2285   \BNVS_tl_use:Nv \_bnvs_index_can:nTF { #1 } {
2286     \prg_return_true:
2287   } {
2288     \prg_return_false:
2289   }
2290 }

2291 \BNVS_new_conditional:cpnn { if_resolve_index:nnc } #1 #2 #3 { T, F, TF } {
2292   \exp_args:Nx \_bnvs_if_resolve_V:ncTF { #1.#2 } { #3 } {
2293     \prg_return_true:
2294   } {
2295     \_bnvs_if_resolve_first:ncTF { #1 } { #3 } {
2296       \_bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2297       \_bnvs_round:c { #3 }
2298     } \prg_return_true:

```

Limited overlay set.

```

2299   } {
2300     \_bnvs_if_resolve_Z:ncTF { #1 } { #3 } {
2301       \_bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2302       \_bnvs_round:c { #3 }
2303     } \prg_return_true:
2304   } {
2305     \_bnvs_if_resolve_V:ncTF { #1 } { #3 } {
2306       \_bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2307       \_bnvs_round:c { #3 }
2308     } \prg_return_true:
2309   } {
2310     \_bnvs_if_resolve_v:ncTF { #1 } { #3 } {
2311       \_bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2312       \_bnvs_round:c { #3 }
2313     } \prg_return_true:
2314   } {
2315     \prg_return_false:
2316   }
2317   }
2318 }
2319 }
2320 }
2321 }

2322 \BNVS_new_conditional:cpnn { if_resolve_index:nvc } #1 #2 #3 { T, F, TF } {
2323   \BNVS_tl_use:nv {
2324     \_bnvs_if_resolve_index:nncTF { #1 }
2325   } { #2 } { #3 } {
2326     \prg_return_true:
2327   } {
2328     \prg_return_false:
2329   }
2330 }

```

```

2331 \BNVS_new_conditional:cpnn { if_resolve_index:vvc } #1 #2 #3 { T, F, TF } {
2332   \BNVS_tl_use:nv {
2333     \BNVS_tl_use:Nv \__bnvs_if_resolve_index:nncTF { #1 }
2334   } { #2 } { #3 } {
2335     \prg_return_true:
2336   } {
2337     \prg_return_false:
2338   }
2339 }

2340 \BNVS_new_conditional:cpnn { if_append_index:nnc } #1 #2 #3 { T, F, TF } {
2341   \BNVS_begin:
2342   \__bnvs_if_resolve_index:nncTF { #1 } { #2 } { #3 } {
2343     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2344     \prg_return_true:
2345   } {
2346     \BNVS_end:
2347     \prg_return_false:
2348   }
2349 }

2350 \BNVS_new_conditional:cpnn { if_append_index:vvc } #1 #2 #3 { T, F, TF } {
2351   \BNVS_tl_use:nv {
2352     \BNVS_tl_use:Nv \__bnvs_if_append_index:nncTF { #1 }
2353   } { #2 } { #3 } {
2354     \prg_return_true:
2355   } {
2356     \prg_return_false:
2357   }
2358 }

```

6.19 Index counter

```

\__bnvs_n_assign:nn \__bnvs_n_assign:nn {<QF name>} {<value>}

```

`__bnvs_n_assign:vv` Assigns the resolved `<value>` to `n` counter `<QF name>`. Execute `<yes code>` when resolution succeeds, `<no code>` otherwise.

```

2359 \BNVS_new:cpn { n_assign:nn } #1 #2 {
2360   \__bnvs_if_get:nncF V { #1 } { a } {
2361     \BNVS_warning:n { Unkwown~ #1,~defaults~to~-1000 }
2362     \__bnvs_gput:nnn V { #1 } {-1000 }
2363   }
2364   \__bnvs_if_resolve:ncTF { #2 } { a } {
2365     \__bnvs_n_gput:nv { #1 } { a }
2366   } {
2367     \__bnvs_error:n { NO~resolution~of~#2,~defaults~to~0 }
2368     \__bnvs_n_gput:nn { #1 } { 0 }
2369   }
2370 }

```

```

2371 \BNVS_new:cpn { n_assign:vv } #1 {
2372   \BNVS_tl_use:nv {
2373     \BNVS_tl_use:cv { n_assign:nn } { #1 }
2374   }
2375 }

```

```

\__bnvs_if_resolve_n:ncTF \__bnvs_if_resolve_n:ncTF {<QF name>} {<ans>} {<yes code>} {<no code>}

```

```

\__bnvs_if_append_n:ncTF Evaluate the n counter associated to the {<QF name>} overlay set into <ans> tl variable.

```

```

\__bnvs_if_append_n:vcTF Initialize this counter to 1 on the first use. <no code> is never executed.

```

```

2376 \BNVS_new_conditional:cpnn { if_resolve_n:nc } #1 #2 { T, F, TF } {
2377   \__bnvs_n_if_get:ncTF { #1 } { #2 } {
2378     \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2379       \prg_return_true:
2380     } {
2381       \prg_return_false:
2382     }
2383   } {
2384     \__bnvs_tl_set:cn { #2 } { 1 }
2385     \__bnvs_n_gput:nn { #1 } { 1 }
2386     \prg_return_true:
2387   }
2388 }

2389 \BNVS_new_conditional:cpnn { if_append_n:nc } #1 #2 { T, F, TF } {
2390   \BNVS_begin:
2391     \__bnvs_if_resolve_n:ncTF { #1 } { #2 } {
2392       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2393       \prg_return_true:
2394     } {
2395       \BNVS_end:
2396       \prg_return_false:
2397     }
2398   }

2399 \BNVS_new_conditional_vc:cn { if_append_n } { T, F, TF }

```

```

\__bnvs_if_resolve_n_index:nncTF \__bnvs_if_resolve_n_index:nncTF {<QF name>} {<QF base>} <ans> {<yes
\__bnvs_if_append_n_index:nncTF code>} {<no code>}
\__bnvs_if_append_n_index:nncTF {<QF name>} {<QF base>} <ans> {<yes
code>} {<no code>}

```

Resolve the index for the value of the n counter associated to the {<QF name>} overlay set into the <ans> tl 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> tl variable is undefined on resolution only.

```

2400 \BNVS_new_conditional:cpnn { if_resolve_n_index:nnc } #1 #2 #3 { T, F, TF } {
2401   \__bnvs_if_resolve_n:ncTF { #1 } { #3 } {
2402     \__bnvs_tl_put_left:cn { #3 } { #1. }
2403     \__bnvs_if_resolve:vcTF { #3 } { #3 } {

```

```

2404     \prg_return_true:
2405   } {
2406     \prg_return_false:
2407   }
2408 } {
2409   \prg_return_false:
2410 }
2411 }

2412 \BNVS_new_conditional:cpnn { if_append_n_index:nnc } #1 #2 #3 { T, F, TF } {
2413   \BNVS_begin:
2414   \__bnvs_if_resolve_n_index:nncTF { #1 } { #2 } { #3 } {
2415     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2416     \prg_return_true:
2417   } {
2418     \BNVS_end:
2419     \prg_return_false:
2420   }
2421 }
2422 \BNVS_new_conditional_vvc:cn { if_append_n_index } { T, F, TF }

```

6.20 Value counter

<u>__bnvs_if_resolve_v_incr:nncTF</u> <u>__bnvs_if_append_v_incr:nncTF</u> <u>__bnvs_if_append_v_incr:(vnc vvc)TF</u>	__bnvs_if_resolve_v_incr:nncTF {<QF name>} {<offset>} {<ans>} {<yes code>} {<no code>} __bnvs_if_append_v_incr:nncTF {<QF name>} {<offset>} {<ans>} {<yes code>} {<no code>}
--	---

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

```

2423 \BNVS_new_conditional:cpnn { if_resolve_v_incr:nnc } #1 #2 #3 { T, F, TF } {
2424   \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2425     \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2426       \__bnvs_if_resolve_v:ncTF { #1 } { #3 } {
2427         \prg_return_true:
2428       } {
2429         \prg_return_false:
2430       }
2431     } {
2432       \__bnvs_tl_put_right:cn { #3 } { + }
2433       \__bnvs_if_append_v:ncTF { #1 } { #3 } {
2434         \__bnvs_round:c { #3 }
2435         \__bnvs_v_gput:nv { #1 } { #3 }
2436         \prg_return_true:
2437       } {
2438         \prg_return_false:
2439       }
2440     }
2441   } {
2442     \prg_return_false:
2443   }
2444 }

```

```

2445 \BNVS_new_conditional:cpnn { if_append_v_incr:nnc } #1 #2 #3 { T, F, TF } {
2446   \BNVS_begin:
2447   \__bnvs_if_resolve_v_incr:nncTF { #1 } { #2 } { #3 } {
2448     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2449     \prg_return_true:
2450   } {
2451     \BNVS_end:
2452     \prg_return_false:
2453   }
2454 }
2455 \BNVS_new_conditional_vnc:cn { if_append_v_incr } { T, F, TF }
2456 \BNVS_new_conditional_vvc:cn { if_append_v_incr } { T, F, TF }

2457 \BNVS_new_conditional:cpnn { if_resolve_v_post:nnc } #1 #2 #3 { T, F, TF } {
2458   \__bnvs_if_resolve_v:ncTF { #1 } { #3 } {
2459     \BNVS_begin:
2460     \__bnvs_if_resolve:ncTF { #2 } { a } {
2461       \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
2462         \BNVS_end:
2463         \prg_return_true:
2464       } {
2465         \__bnvs_tl_put_right:cn { a } { + }
2466         \__bnvs_tl_put_right:cv { a } { #3 }
2467         \__bnvs_round:c { a }
2468         \BNVS_end_v_gput:nv { #1 } { a }
2469         \prg_return_true:
2470       }
2471     } {
2472       \BNVS_end:
2473       \prg_return_false:
2474     }
2475   } {
2476     \prg_return_false:
2477   }
2478 }
2479 \BNVS_new_conditional_vvc:cn { if_resolve_v_post } { T, F, TF }

2480 \BNVS_new_conditional:cpnn { if_append_v_post:nnc } #1 #2 #3 { T, F, TF } {
2481   \BNVS_begin:
2482   \__bnvs_if_resolve_v_post:nncTF { #1 } { #2 } { #3 } {
2483     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2484     \prg_return_true:
2485   } {
2486     \prg_return_false:
2487   }
2488 }
2489 \BNVS_new_conditional_vnc:cn { if_append_v_post } { T, F, TF }
2490 \BNVS_new_conditional_vvc:cn { if_append_v_post } { T, F, TF }

```

<code>_bnvs_if_resolve_n_incr:nncTF</code>	<code>_bnvs_if_resolve_n_incr:nncTF {<QF name>} {<QF base>} {<offset>}</code>
<code>_bnvs_if_resolve_n_incr:vvncTF</code>	<code>{<ans>} {<yes code>} {<no code>}</code>
<code>_bnvs_if_resolve_n_incr:nncTF</code>	<code>_bnvs_if_resolve_n_incr:nncTF {<QF name>} {<offset>} {<ans>} {<yes</code>
<code>_bnvs_if_resolve_n_incr:vncTF</code>	<code>code>} {<no code>}</code>
<code>_bnvs_if_append_n_incr:nncTF</code>	<code>_bnvs_if_append_n_incr:nncTF {<QF name>} {<QF base>} {<offset>}</code>
<code>_bnvs_if_append_n_incr:nncTF</code>	<code>{<ans>} {<yes code>} {<no code>}</code>
<code>_bnvs_if_append_n_incr:(vnc vvc)TF</code>	<code>_bnvs_if_append_n_incr:nncTF {<QF name>} {<offset>} {<ans>} {<yes</code>
<code>_bnvs_if_resolve_n_post:nncTF</code>	<code>code>} {<no code>}</code>
<code>_bnvs_if_append_n_post:nncTF</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.

2491 `\BNVS_new_conditional:cpnn { if_resolve_n_incr:nnc } #1 #2 #3 #4 { T, TF } {`
Resolve the `<offset>` into the `<ans>` variable.

2492 `_bnvs_if_resolve:ncTF { #3 } { #4 } {`
2493 `\BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {`
The offset is resolved to 0, we just have to resolve the ...
2494 `_bnvs_if_resolve_n:ncTF { #1 } { #4 } {`
2495 `_bnvs_if_resolve_index:nvcTF { #1 } { #4 } { #4 } {`
2496 `\prg_return_true:`
2497 `} {`
2498 `\prg_return_false:`
2499 `}`
2500 `} {`
2501 `\prg_return_false:`
2502 `}`
2503 `} {`

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

2504 `_bnvs_tl_put_right:cn { #4 } { + }`
2505 `_bnvs_if_append_n:ncTF { #1 } { #4 } {`
2506 `_bnvs_round:c { #4 }`
2507 `_bnvs_n_gput:nv { #1 } { #4 }`
2508 `_bnvs_if_resolve_index:nvcTF { #2 } { #4 } { #4 } {`
2509 `\prg_return_true:`
2510 `} {`
2511 `\prg_return_false:`
2512 `}`
2513 `} {`
2514 `\prg_return_false:`
2515 `}`
2516 `}`
2517 `} {`
2518 `\prg_return_false:`
2519 `}`
2520 `}`

2521 `\BNVS_new_conditional:cpnn { if_resolve_n_incr:nnc } #1 #2 #3 { T, F, TF } {`
2522 `_bnvs_if_resolve:ncTF { #2 } { #3 } {`
2523 `\BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {`
2524 `_bnvs_if_resolve_n:ncTF { #1 } { #3 } {`
2525 `_bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {`
2526 `\prg_return_true:`
2527 `} {`

```

2528         \prg_return_false:
2529     }
2530 } {
2531     \prg_return_false:
2532 }
2533 } {
2534     \__bnvs_tl_put_right:cn { #3 } { + }
2535     \__bnvs_if_append_n:ncTF { #1 } { #3 } {
2536         \__bnvs_round:c { #3 }
2537         \__bnvs_n_gput:nv { #1 } { #3 }
2538         \__bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {
2539             \prg_return_true:
2540         } {
2541             \prg_return_false:
2542         }
2543     } {
2544         \prg_return_false:
2545     }
2546 }
2547 } {
2548     \prg_return_false:
2549 }
2550 }
2551 \BNVS_new_conditional_vnc:cn { if_resolve_n_incr } { T, F, TF }
2552 \BNVS_new_conditional_vvc:cn { if_resolve_n_incr } { T, F, TF }
2553 \BNVS_new_conditional_vvnc:cn { if_resolve_n_incr } { T, F, TF }
2554 \BNVS_new_conditional:cpnn
2555 { if_append_n_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
2556     \BNVS_begin:
2557     \__bnvs_if_resolve_n_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
2558         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2559         \prg_return_true:
2560     } {
2561         \BNVS_end:
2562         \prg_return_false:
2563     }
2564 }
2565 \BNVS_new_conditional_vvnc:cn { if_append_n_incr } { T, F, TF }
2566 \BNVS_new_conditional_vvvc:cn { if_append_n_incr } { T, F, TF }
2567 \BNVS_new_conditional:cpnn { if_append_n_incr:nnc } #1 #2 #3 { T, F, TF } {
2568     \BNVS_begin:
2569     \__bnvs_if_resolve_n_incr:nncTF { #1 } { #2 } { #3 } {
2570         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2571         \prg_return_true:
2572     } {
2573         \BNVS_end:
2574         \prg_return_false:
2575     }
2576 }
2577 \BNVS_new_conditional_vnc:cn { if_append_n_incr } { T, F, TF }
2578 \BNVS_new_conditional_vvc:cn { if_append_n_incr } { T, F, TF }

```

<code>_bnvs_if_resolve_v_post:nncTF</code> <code>_bnvs_if_resolve_v_post:vvcTF</code> <code>_bnvs_if_append_v_post:nncTF</code> <code>_bnvs_if_append_v_post:(vnN vvN)TF</code>	<code>_bnvs_if_resolve_v_post:nncTF {<QF name>} {<offset>} <ans> {<yes</code> <code>code>} {<no code>}</code> <code>_bnvs_if_append_v_post:nncTF {<QF name>} {<offset>} <ans> {<yes</code> <code>code>} {<no code>}</code>
--	---

Resolve the value of the free counter for the given $\langle QF \text{ name} \rangle$ 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.

```

2579 \BNVS_new_conditional:cpnn { if_resolve_n_post:nnc } #1 #2 #3 { T, F, TF } {
2580   \_bnvs_if_resolve_n:ncTF { #1 } { #3 } {
2581     \BNVS_begin:
2582     \_bnvs_if_resolve:ncTF { #2 } { #3 } {
2583       \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2584         \BNVS_end:
2585         \_bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {
2586           \prg_return_true:
2587           } {
2588           \prg_return_false:
2589           }
2590         } {
2591           \_bnvs_tl_put_right:cn { #3 } { + }
2592           \_bnvs_if_append_n:ncTF { #1 } { #3 } {
2593             \_bnvs_round:c { #3 }
2594             \_bnvs_n_gput:nv { #1 } { #3 }
2595             \BNVS_end:
2596             \_bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {
2597               \prg_return_true:
2598               } {
2599               \prg_return_false:
2600               }
2601             } {
2602               \BNVS_end:
2603               \prg_return_false:
2604             }
2605           }
2606         } {
2607           \BNVS_end:
2608           \prg_return_false:
2609         }
2610       } {
2611         \prg_return_false:
2612       }
2613     }
2614 \BNVS_new_conditional:cpnn { if_append_n_post:nnc } #1 #2 #3 { T, F, TF } {
2615   \BNVS_begin:
2616   \_bnvs_if_resolve_n_post:nncTF { #1 } { #2 } { #3 } {
2617     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2618     \prg_return_true:
2619   } {
2620     \BNVS_end:

```



```

2621     \prg_return_false:
2622   }
2623 }
2624 \BNVS_new_conditional_vnc:cn { if_append_n_post } { T, F, TF }
2625 \BNVS_new_conditional_vvc:cn { if_append_n_post } { T, F, TF }

```

6.21 Functions for the resolution

They manily start with __bnvs_if_resolve_ or __bnvs_split_

__bnvs_split_pop_Qip:TFF	__bnvs_split_pop_Qip:TFF {<black code>} {<blank code>}
__bnvs_split_end_return_or_pop_complete:T	{<end code>}
__bnvs_split_end_return_or_pop_void:T	__bnvs_split_end_return_or_pop_complete:T {<blank code>}
	__bnvs_split_end_return_or_pop_void:T {<black code>}

For __bnvs_split_pop_Qip:TFF. If the split sequence is empty, execute <end code>. Otherwise pops the 3 heading items of the split sequence into the three tl variables Q_name, id, path. If Q_name 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 seven heading items of the split sequence then execute <blank code>.

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

```

2626 \BNVS_new:cpn { split_pop_Qip:TFF } #1 #2 #3 {
2627   \__bnvs_split_if_pop_left:cTF { Q_name } {
2628     \__bnvs_split_if_pop_left:cTF { id } {
2629       \__bnvs_split_if_pop_left:cTF { path } {
2630
2631         \__bnvs_tl_if_blank:vTF { Q_name } {

```

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

```

2631         #2
2632       } {
2633         \BNVS_tl_use:nv {
2634           \regex_match:NnT \c__bnvs_A_reserved_Z_regex
2635         } { Q_name } {
2636           \__bnvs_tl_if_eq:cnF { Q_name } { pauses } {
2637             \BNVS_error:x { Use-of~reserved~``\BNVS_tl_use:c { Q_name }'' }
2638           }
2639         }
2640         \__bnvs_tl_if_blank:vTF { id } {
2641           \__bnvs_tl_put_left:cv { Q_name } { id_last }
2642           \__bnvs_tl_set:cv { id } { id_last }
2643         } {
2644           \__bnvs_tl_set:cv { id_last } { id }
2645         }

```

Build the path sequence and lowercase components conditionals.

```

2646   \__bnvs_seq_set_split:cnv { path } { . } { path }

```

```

2647         #1
2648     }
2649 } {
2650 \_bnvs_end_unreachable_return_false:n { split_pop_Qip:TFF/2 }
2651 }
2652 } {
2653 \_bnvs_end_unreachable_return_false:n { split_pop_Qip:TFF/1 }
2654 }
2655 } { #3 }
2656 }

```

conditional variants.

```

2657 \BNVS_new:cpn { split_end_return_or_pop_complete:T } #1 {
2658   \cs_set:Npn \BNVS:n ##1 {
2659     \_bnvs_end_unreachable_return_false:n {
2660       split_end_return_or_pop_complete: ##1
2661     }
2662   }
2663   \_bnvs_split_if_pop_left_or:cT { n_incr } {
2664     \_bnvs_split_if_pop_left_or:cT { plus } {
2665       \_bnvs_split_if_pop_left_or:cT { rhs } {
2666         \_bnvs_split_if_pop_left_or:cT { post } {
2667           #1
2668         }
2669       }
2670     }
2671   }
2672 }

2673 \BNVS_new:cpn { split_end_return_or_pop_void:T } #1 {
2674   \cs_set:Npn \BNVS:n ##1 {
2675     \_bnvs_end_unreachable_return_false:n {
2676       split_end_return_or_pop_void: ##1
2677     }
2678   }
2679   \_bnvs_split_if_pop_left:cTn { a } {
2680     \_bnvs_split_if_pop_left:cTn { a } {
2681       \_bnvs_split_if_pop_left:cTn { a } {
2682         \_bnvs_split_if_pop_left:cTn { a } {
2683           \_bnvs_split_if_pop_left:cTn { a } {
2684             \_bnvs_split_if_pop_left:cTn { a } {
2685               \_bnvs_split_if_pop_left:cTn { a } {
2686                 #1
2687               } { T/7 }
2688             } { T/6 }
2689           } { T/5 }
2690         } { T/4 }
2691       } { T/3 }
2692     } { T/2 }
2693   } { T/1 }
2694 }

```

<code>__bnvs_if_resolve:ncTF</code> <code>__bnvs_if_resolve:vcTF</code> <code>__bnvs_if_append:ncTF</code> <code>__bnvs_if_append:vcTF</code>	<code>__bnvs_if_resolve:ncTF {<expression>} {<ans>} {<yes code>} {<no code>}</code> <code>__bnvs_if_append:ncTF {<expression>} {<ans>} {<yes code>} {<no code>}</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.
--	--

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`.)

2695 `\BNVS_int_new:c { split }`

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

(End of definition for `\l__bnvs_Q_name_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.

```

2696 \BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
2697   \BNVS_begin:
2698     \__bnvs_if_resolve:ncTF { #1 } { #2 } {
2699       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2700       \prg_return_true:
2701     } {
2702       \BNVS_end:
2703       \prg_return_false:
2704     }
2705   }
2706 \BNVS_new_conditional_vc:cn { if_append } { T, F, TF }

```

Heavily used.

```

2707 \BNVS_new:cpn { end_unreachable_return_false:n } #1 {
2708   \BNVS_error:x { UNREACHABLE/#1 }
2709   \BNVS_end:
2710   \prg_return_false:
2711 }

2712 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { TF } {
2713   \__bnvs_if_call:TF {
2714     \BNVS_begin:

```

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

```
2715 \__bnvs_if_regex_split:cnTF { split } { #1 } {
```

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

```
2716 \BNVS_set:cpn { if_resolve_end_return_true: } {
```

Normal and unique end of the loop.

```
2717 \__bnvs_if_resolve_round_ans:
2718 \BNVS_end_tl_set:cv { #2 } { ans }
2719 \prg_return_true:
2720 }
```

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

```
2721 \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round:c { ans } }
2722 \__bnvs_tl_clear:c { ans }
2723 \__bnvs_split_loop_or_end_return:
2724 } {
```

There is not reference.

```
2725 \__bnvs_tl_set:cn { ans } { #1 }
2726 \__bnvs_round:c { ans }
2727 \BNVS_end_tl_set:cv { #2 } { ans }
2728 \prg_return_true:
2729 }
2730 } {
2731 \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
2732 \prg_return_false:
2733 }
2734 }
2735 \BNVS_new_conditional_vc:cn { if_resolve } { T, F, TF }

2736 \BNVS_new:cpn { build_QF_name: } {
2737 \__bnvs_tl_set_eq:cc { QF_name } { Q_name }
2738 \__bnvs_seq_map_inline:cn { path } {
2739 \__bnvs_tl_put_right:cn { QF_name } { . ##1 }
2740 }
2741 }
2742 \BNVS_new:cpn { build_QF_name_head: } {
2743 \__bnvs_tl_set_eq:cc { QF_name } { Q_name }
2744 \__bnvs_seq_map_inline:cn { path_head } {
2745 \__bnvs_tl_put_right:cn { QF_name } { . ##1 }
2746 }
2747 }
```

`__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.

```

2748 \clist_map_inline:nn {
2749   n, reset, reset_all, v, first, last, length,
2750   previous, next, range, assign, only
2751 } {
2752   \bool_new:c { l__bnvs_#1_bool }
2753 }

2754 \BNVS_new_conditional:cpnn { if:c } #1 { p, T, F, TF } {
2755   \bool_if:cTF { l__bnvs_#1_bool } {
2756     \prg_return_true:
2757   } {
2758     \prg_return_false:
2759   }
2760 }

2761 \BNVS_new_conditional:cpnn { bool_if_exist:c } #1 { p, T, F, TF } {
2762   \bool_if_exist:cTF { l__bnvs_#1_bool } {
2763     \prg_return_true:
2764   } {
2765     \prg_return_false:
2766   }
2767 }

2768 \BNVS_new:cpn { prepare_context:N } #1 {
2769   \clist_map_inline:nn {
2770     n, v, reset, reset_all, first, last, length,
2771     previous, next, range, assign, only
2772   } {
2773     \__bnvs_set_false:c { ##1 }
2774   }
2775   \__bnvs_seq_clear:c { path_head }
2776   \__bnvs_seq_clear:c { path_tail }
2777   \__bnvs_tl_clear:c { index }
2778   \__bnvs_tl_clear:c { suffix }
2779   \BNVS_set:cpn { :n } ##1 {
2780     \tl_if_blank:nF { ##1 } {
2781       \__bnvs_tl_if_empty:cF { index } {
2782         \__bnvs_seq_put_right:cv { path_head } { index }
2783         \__bnvs_tl_clear:c { index }
2784       }
2785       \__bnvs_seq_put_right:cn { path_head } { ##1 }
2786     }
2787   }
2788   \__bnvs_seq_map_inline:cn { path } {
2789     \__bnvs_bool_if_exist:cTF { ##1 } {
2790       \__bnvs_set_true:c { ##1 }
2791       \clist_if_in:nnF { n, v, reset, reset_all } { ##1 } {
2792         \bool_if:NT #1 {
2793           \BNVS_error:n {Unexpected~##1~in~assignment }
2794         }
2795         \__bnvs_tl_set:cn { suffix } { ##1 }
2796       }
2797       \BNVS_set:cpn { :n } #####1 {
2798         \tl_if_blank:nF { #####1 } {
2799           \BNVS_error:n {Unexpected~#####1 }

```

```

2800     }
2801   }
2802   } {
2803     \regex_match:NnTF \c__bnvs_A_index_Z_regex { ##1 } {
2804       \__bnvs_tl_if_empty:cTF { index } {
2805         \__bnvs_seq_put_right:cv { path_head } { index }
2806       }
2807       \__bnvs_tl_set:cn { index } { ##1 }
2808     } {
2809       \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { ##1 } {
2810         \BNVS_error:n { Unsupported~##1 }
2811       } {
2812         \__bnvs_:n { ##1 }
2813       }
2814     }
2815   }
2816 }
2817 \__bnvs_seq_set_eq:cc { path } { path_head }
2818 }

2819 \BNVS_new:cpn { split_loop_or_end_return: } {
2820   \__bnvs_split_if_pop_left:cTF { a } {
2821     \__bnvs_tl_put_right:cv { ans } { a }
2822     \__bnvs_split_pop_Qip:TFF {
2823       \__bnvs_split_end_return_or_pop_void:T {
2824         \__bnvs_prepare_context:N \c_true_bool
2825         \__bnvs_build_QF_name:
2826         \__bnvs_split_loop_or_end_return_iadd:n { 1 }
2827       }
2828     } {
2829       \__bnvs_split_pop_Qip:TFF {
2830         \__bnvs_split_end_return_or_pop_complete:T {
2831           \__bnvs_tl_if_blank:vTF { n_incr } {
2832             \__bnvs_tl_if_blank:vTF { plus } {
2833               \__bnvs_tl_if_blank:vTF { rhs } {
2834                 \__bnvs_tl_if_blank:vTF { post } {
2835                   \__bnvs_prepare_context:N \c_false_bool
2836                   \__bnvs_build_QF_name:

```

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

```

2837       \__bnvs_tl_if_empty:cTF { index } {
2838         \__bnvs_tl_if_empty:cTF { suffix } {
2839           \__bnvs_split_loop_or_end_return_v:
2840         } {
2841           \__bnvs_split_loop_or_end_return_suffix:
2842         }
2843       } {
2844         \__bnvs_split_loop_or_end_return_index:
2845       }
2846     } {
2847       \__bnvs_prepare_context:N \c_true_bool
2848       \__bnvs_build_QF_name:
2849       \BNVS_use:c { split_loop_or_end_return[...++]: }
2850     }
2851   } {

```

```

2852         \__bnvs_prepare_context:N \c_true_bool
2853         \__bnvs_build_QF_name:
2854         \__bnvs_split_loop_or_end_return_assign:
2855     }
2856   } {
2857     \__bnvs_if_resolve:vcTF { rhs } { rhs } {
2858       \__bnvs_prepare_context:N \c_true_bool
2859       \__bnvs_build_QF_name:
2860       \BNVS_tl_use:Nv
2861       \__bnvs_split_loop_or_end_return_iadd:n { rhs }
2862     } {
2863       \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs }}
2864       \__bnvs_split_loop_or_end_return:
2865     }
2866   }
2867   } {
2868     \__bnvs_prepare_context:N \c_true_bool
2869     \__bnvs_build_QF_name:
2870     \__bnvs_set_true:c { n }
2871     \__bnvs_split_loop_or_end_return_iadd:n { 1 }
2872   }
2873 }
2874 } {
2875   \__bnvs_end_unreachable_return_false:n { split_loop_or_end_return:/3 }
2876 } {
2877   \__bnvs_end_unreachable_return_false:n { split_loop_or_end_return:/2 }
2878 }
2879 } {

```

The split sequence is empty.

```

2880   \__bnvs_if_resolve_end_return_true:
2881 }
2882 } {
2883   \__bnvs_end_unreachable_return_false:n { split_loop_or_end_return:/1 }
2884 }
2885 }

2886 \BNVS_new_conditional:cpnn { if_suffix: } { T, F, TF } {
2887   \__bnvs_tl_if_empty:cTF { suffix } {
2888     \__bnvs_seq_pop_right:ccTF { path } { suffix } {
2889       \prg_return_true:
2890     } {
2891       \prg_return_false:
2892     }
2893   } {
2894     \prg_return_true:
2895   }
2896 }

```

Implementation detail: tl variable a is used.

```

2897 \BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {
2898   \__bnvs_if:cTF { n } {
2899     #1
2900   } {
2901     \__bnvs_build_QF_name:

```

```

2902     \__bnvs_tl_set:cx { a } {
2903         \BNVS_tl_use:c { QF_name } . \BNVS_tl_use:c { suffix }
2904     }
2905     \__bnvs_if_resolve_v:vcTF { a } { a } {
2906         \__bnvs_tl_put_right:cv { ans } { a }
2907         \__bnvs_split_loop_or_end_return:
2908     } {
2909         \__bnvs_if_resolve_V:vcTF { a } { a } {
2910             \__bnvs_tl_put_right:cv { ans } { a }
2911             \__bnvs_split_loop_or_end_return:
2912         } {
2913             #1
2914         }
2915     }
2916 }
2917 }

2918 \BNVS_new:cpn { end_return_error:n } #1 {
2919     \BNVS_error:n { #1 }
2920     \BNVS_end:
2921     \prg_return_false:
2922 }

2923 \BNVS_new:cpn { path_branch_loop_or_end_return: } {
2924     \__bnvs_if_call:TF {
2925         \__bnvs_if_path_branch:TF {
2926             \__bnvs_path_branch_end_return:
2927         } {
2928             \__bnvs_if_get:nvcTF V { QF_name } { a } {
2929                 \__bnvs_if_Qip:cccTF { a } { id } { path } {
2930                     \__bnvs_tl_set_eq:cc { Q_name } { a }
2931                     \__bnvs_seq_merge:cc { path } { path_tail }
2932                     \__bnvs_seq_clear:c { path_tail }
2933                     \__bnvs_seq_set_eq:cc { path_head } { path }
2934                     \__bnvs_path_branch_QF_loop_or_end_return:
2935                 } {
2936                     \__bnvs_path_branch_head_to_tail_end_return:
2937                 }
2938             } {
2939                 \__bnvs_path_branch_head_to_tail_end_return:
2940             }
2941         }
2942     } {
2943         \__bnvs_path_branch_end_return_false:n {
2944             Too-many-calls.
2945         }
2946     }
2947 }

2948 \BNVS_new:cpn { path_branch_end_return: } {
2949     \__bnvs_split_loop_or_end_return:
2950 }

2951 \BNVS_new:cpn { set_if_path_branch:n } {
2952     \prg_set_conditional:Npnn \__bnvs_if_path_branch: { TF }
2953 }

```



```

2954 \BNVS_new:cpn { path_branch_head_to_tail_end_return: } {
2955   \__bnvs_seq_pop_right:ccTF { path_head } { a } {
2956     \__bnvs_seq_put_left:cv { path_tail } { a }
2957     \__bnvs_build_QF_name_head:
2958     \__bnvs_path_branch_QF_loop_or_end_return:
2959   } {
2960     \__bnvs_build_QF_name:
2961     \__bnvs_seq_set_eq:cc { path_head } { path_tail }
2962     \__bnvs_seq_clear:c { path_tail }
2963     \__bnvs_gput:nvn V { QF_name } {-1000 }
2964     \__bnvs_c_gput:nvn V { QF_name } {-1000 }
2965     \BNVS_warning:x {
2966       Unknown~\l__bnvs_QF_name_tl,~defaults~to~-1000
2967     }
2968     \__bnvs_path_branch_QF_loop_or_end_return:
2969   }
2970 }

```

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

```

2971 \BNVS_new:cpn { path_branch_QF_loop_or_end_return: } {
2972   \__bnvs_build_QF_name_head:
2973   \__bnvs_if_resolve_v:vcTF { QF_name } { a } {
2974     \__bnvs_tl_put_right:cv { ans } { a }
2975     \__bnvs_split_loop_or_end_return:
2976   } {
2977     \__bnvs_if_resolve_V:vcTF { QF_name } { a } {
2978       \__bnvs_tl_put_right:cv { ans } { a }
2979       \__bnvs_split_loop_or_end_return:
2980     } {
2981       \__bnvs_path_branch_loop_or_end_return:
2982     }
2983   }
2984 }

```

- Case*<index>*.

```

2985 \BNVS_new:cpn { split_loop_or_end_return_index: } {
2986   % known, id, QF_name, path, suffix
2987   \__bnvs_set_if_path_branch:n {
2988     \__bnvs_if_append_index:vvctf { QF_name } { index } { ans } {
2989       \prg_return_true:
2990     } {
2991       \prg_return_false:
2992     }
2993   }
2994   \__bnvs_path_branch_loop_or_end_return:
2995 }

2996 \BNVS_new:cpn { split_loop_reset: } {
2997   \__bnvs_if:cT { reset_all } {
2998     \__bnvs_set_false:c { reset }
2999     \__bnvs_if_greset_all:vnT { QF_name } { } { }
3000   }

```

```

3001  \__bnvs_if:cT { reset } {
3002      \BNVS_use:c {
3003          \__bnvs_if:cTF nnv _if_greset:vnT
3004      } { QF_name } { } { }
3005  }
3006 }

```

- Case

```

3007 \BNVS_new:cpn { split_loop_or_end_return_v: } {
3008     \__bnvs_split_loop_reset:
3009     \__bnvs_if:cTF { n } {
3010         \__bnvs_tl_set_eq:cc { QF_base } { QF_name }
3011         \__bnvs_set_if_path_branch:n {
3012             \BNVS_tl_use:Nv \__bnvs_if_resolve_n:ncTF { QF_name } { index } {
3013                 \__bnvs_if_append_index:vcTF { QF_base } { index } { ans } {
3014                     \prg_return_true:
3015                 } {
3016                     \prg_return_false:
3017                 }
3018             } {
3019                 \prg_return_false:
3020             }
3021         }
3022     } {
3023         \__bnvs_set_if_path_branch:n {
3024             \__bnvs_if_append_v:vcTF { QF_name } { ans } {
3025                 \prg_return_true:
3026             } {
3027                 \__bnvs_if_append_V:vcTF { QF_name } { ans } {
3028                     \prg_return_true:
3029                 } {
3030                     \prg_return_false:
3031                 }
3032             }
3033         }
3034     }
3035     \__bnvs_path_branch_loop_or_end_return:
3036 }

```

- Case<suffix>.

```

3037 \BNVS_new:cpn { split_loop_or_end_return_suffix: } {
3038     \__bnvs_if_resolve_V_loop_or_end_return_true:F {
3039         \__bnvs_if:cTF { n } {
3040             \__bnvs_tl_set_eq:cc { QF_base } { QF_name }
3041             \__bnvs_set_if_path_branch:n {
3042                 \BNVS_tl_use:Nv \__bnvs_if_resolve_n:ncTF { QF_name } { index } {
3043                     \__bnvs_if_append_index:vcTF { QF_base } { index } { ans } {
3044                         \prg_return_true:
3045                     } {
3046                         \prg_return_false:
3047                     }
3048                 } {
3049                     \prg_return_false:

```

```

3050     }
3051   }
3052 } {
3053   \__bnvs_set_if_path_branch:n {
3054     \BNVS_use:c {
3055       if_append_ \__bnvs_tl_use:c { suffix } :vcTF
3056     } { QF_name } { ans } {
3057       \__bnvs_if:cT { range } {
3058         \BNVS_set:cpn { if_resolve_round_ans: } { }
3059       }
3060       \prg_return_true:
3061     } {
3062       \prg_return_false:
3063     }
3064   }
3065 }
3066 \__bnvs_path_branch_loop_or_end_return:
3067 }
3068 }

```

- Case ...++.

```

3069 \BNVS_new:cpn { split_loop_or_end_return[...++]: } {
3070   \__bnvs_if:cTF { n } {
3071     \__bnvs_if:cTF { reset } {
3072       \cs_set:Npn \BNVS: {
3073         \BNVS_error_ans:x { NO~....reset.n++~for~\BNVS_tl_use:c { QF_name } }
3074       }
3075     } {
3076       \__bnvs_if:cTF { reset_all } {

```

- Casereset_all.n++.

```

3077       \cs_set:Npn \BNVS: {
3078         \BNVS_error_ans:x {
3079           NO~....reset_all.n++~for~\BNVS_tl_use:c { QF_name }
3080         }
3081       }
3082     } {

```

- Casen++.

```

3083       \cs_set:Npn \BNVS: {
3084         \BNVS_error_ans:x { NO~....n++~for~\BNVS_tl_use:c { QF_name } }
3085       }
3086     }
3087   }
3088 } {
3089   \__bnvs_if:cTF { reset } {

```

- Casereset++.

```

3090       \cs_set:Npn \BNVS: {
3091         \BNVS_error_ans:x { NO~....reset++~for~\BNVS_tl_use:c { QF_name } }
3092       }
3093     } {
3094       \__bnvs_if:cTF { n } {

```

- Casereset_all.n++.

```

3095         \cs_set:Npn \BNVS: {
3096             \BNVS_error_ans:x {
3097                 NO~....n(.reset_all)++~for~\BNVS_tl_use:c { QF_name }
3098             }
3099         }
3100     } {

```

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

```

3101         \cs_set:Npn \BNVS: {
3102             \BNVS_error_ans:x {
3103                 NO~...(.reset_all)++~for~\BNVS_tl_use:c { QF_name }
3104             }
3105         }
3106     }
3107 }
3108 }
3109 \__bnvs_build_QF_name:
3110 \__bnvs_split_loop_reset:
3111 \BNVS_use:c {
3112     if_append_\__bnvs_if:cTF nnv _post:vncTF
3113 } { QF_name } { 1 } { ans } {
3114 } {
3115     \BNVS_error_ans:x { Problem~with~\BNVS_tl_use:c { QF_name }~use. }
3116 }
3117 \__bnvs_split_loop_or_end_return:
3118 }
3119 \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.

```

3120 \__bnvs_if_resolve:vcTF { rhs } { rhs } {
3121     \__bnvs_if:cTF n {
3122         \__bnvs_n_gput:vv { QF_name } { rhs }
3123         \__bnvs_if_append_index:vvcTF { QF_name } { rhs } { ans } {
3124             } {
3125                 \BNVS_error_ans:x { No~....n=... }
3126             }
3127         } {
3128             \__bnvs_v_gput:vv { QF_name } { rhs }
3129             \__bnvs_if_append_v:vcTF { QF_name } { ans } {
3130                 } {
3131                     \BNVS_error_ans:x { No~...=... }
3132                 }
3133             }
3134         } {
3135             \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c { rhs }. }
3136         }
3137     \__bnvs_split_loop_or_end_return:
3138 }

```

- Case ...+=

```

3139 \BNVS_new:cpn { split_loop_or_end_return_iadd:n } #1 {
3140   \__bnvs_if_resolve:ncTF { #1 } { rhs } {
3141     \__bnvs_split_loop_reset:
3142     \BNVS_use:c {
3143       if_append_ \__bnvs_if:cTF nnv _incr:vncTF
3144     } { QF_name } { #1 } { ans } {
3145     } {
3146       \BNVS_error_ans:x { No~...+=... }
3147     }
3148   } {
3149     \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs } }
3150   }
3151   \__bnvs_split_loop_or_end_return:
3152 }

```

```
\_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`.)

```
3153 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
3154   \A \s* (?
    • 2  $\rightarrow$  V
    ( [^:]+? )
    • 3, 4, 5  $\rightarrow$  A : Z? or A :: L?
    | (? : ( [^:]+? ) \s* : (? : \s* ( [^:]*? ) | : \s* ( [^:]*? ) ) )
    • 6, 7  $\rightarrow$  ::(L:Z)?
    | (? : :: \s* (? : ( [^:]+? ) \s* : \s* ( [^:]+? ) )? )
    • 8, 9  $\rightarrow$  :(Z::L)?
    | (? : : \s* (? : ( [^:]+? ) \s* :: \s* ( [^:]*? ) )? )
    )
    \s* \Z
  }
```

```
3162 \BNVS_new:cpn { resolve_query_end_return_true: } {
3163   \BNVS_end:
3164   \prg_return_true:
3165 }
3166 \BNVS_new:cpn { resolve_query_end_return_false: } {
```

```

3167 \BNVS_end:
3168 \prg_return_false:
3169 }

3170 \BNVS_new:cpn { resolve_query_end_return_false:n } #1 {
3171 \BNVS_end:
3172 \prg_return_false:
3173 }

3174 \BNVS_new:cpn { if_resolve_query_return_false:n } #1 {
3175 \prg_return_false:
3176 }

3177 \BNVS_new:cpn { resolve_query_error_return_false:n } #1 {
3178 \BNVS_error:x { #1 }
3179 \__bnvs_if_resolve_query_return_false:
3180 }

3181 \BNVS_new:cpn { if_resolve_query_return_unreachable: } {
3182 \__bnvs_resolve_query_error_return_false:n { UNREACHABLE }
3183 }

3184 \BNVS_new:cpn { if_blank:cTF } #1 {
3185 \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3186 }

3187 \BNVS_new_conditional:cpnn { if_match_pop_left:c } #1 { T, F, TF } {
3188 \BNVS_tl_use:nc {
3189 \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
3190 } { #1 } {
3191 \prg_return_true:
3192 } {
3193 \prg_return_false:
3194 }
3195 }

```

__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.

```

3196 \BNVS_new_conditional:cpnn { if_resolve_query_branch: } { T, F, TF } {
At start, we ignore the whole match.
3197 \__bnvs_if_match_pop_left:cT V {
3198 \__bnvs_if_match_pop_left:cT V {
3199 \__bnvs_if_blank:cTF V {
3200 \__bnvs_if_match_pop_left:cT A {
3201 \__bnvs_if_match_pop_left:cT Z {
3202 \__bnvs_if_match_pop_left:cT L {
3203 \__bnvs_if_blank:cTF A {

```

```

3204         \__bnvs_if_match_pop_left:cT L {
3205             \__bnvs_if_match_pop_left:cT Z {
3206                 \__bnvs_if_blank:cTF L {
3207                     \__bnvs_if_match_pop_left:cT Z {
3208                         \__bnvs_if_match_pop_left:cT L {
3209                             \__bnvs_if_blank:cTF L {
3210                                 \BNVS_use:c { if_resolve_query_return[:Z]: }
3211                             } {
3212                                 \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3213                             }
3214                         }
3215                     }
3216                 } {
3217                     \__bnvs_if_blank:cTF Z {
3218 \__bnvs_resolve_query_error_return_false:n { Missing-first-or-last }
3219                 } {
3220                     \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3221                 }
3222             }
3223         }
3224     }
3225 } {
3226     \__bnvs_if_blank:cTF Z {
3227         \__bnvs_if_blank:cTF L {
3228             \BNVS_use:c { if_resolve_query_return[A:]: }
3229         } {
3230             \BNVS_use:c { if_resolve_query_return[A::L]: }
3231         }
3232     } {
3233         \__bnvs_if_blank:cTF L {
3234             \BNVS_use:c { if_resolve_query_return[A:Z]: }
3235         } {
3236             \__bnvs_if_resolve_query_return_unreachable:
3237         }
3238     }
3239 }
3240 }
3241 }
3242 }
3243 } {
3244     \BNVS_use:c { if_resolve_query_return[V]: }
3245 }
3246 }
3247 }
3248 }

```

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

Single value

```

3249 \BNVS_new:cpn { if_resolve_query_return[V]: } {
3250     \__bnvs_if_resolve:vcTF { V } { ans } {
3251         \prg_return_true:
3252     } {
3253         \prg_return_false:
3254     }

```



```

3255 }
3256 ❶ <first>:<last> range
3257 \BNVS_new:cpn { if_resolve_query_return[A:Z]: } {
3258   \__bnvs_if_resolve:vcTF { A } { ans } {
3259     \__bnvs_tl_put_right:cn { ans } { - }
3260     \__bnvs_if_append:vcTF { Z } { ans } {
3261       \prg_return_true:
3262     } {
3263       \prg_return_false:
3264     } {
3265       \prg_return_false:
3266     }
3267 }
3278 ❷ <first>::<length> range
3279 \BNVS_new:cpn { if_resolve_query_return[A::L]: } {
3280   \__bnvs_if_resolve:vcTF { A } { A } {
3281     \__bnvs_if_resolve:vcTF { L } { ans } {
3282       \__bnvs_tl_put_right:cn { ans } { + }
3283       \__bnvs_tl_put_right:cv { ans } { A }
3284       \__bnvs_tl_put_right:cn { ans } { -1 }
3285       \__bnvs_round:c { ans }
3286       \__bnvs_tl_put_left:cn { ans } { - }
3287       \__bnvs_tl_put_left:cv { ans } { A }
3288     } \prg_return_true:
3289   } {
3290     \prg_return_false:
3291   }
3292 } {
3293   \prg_return_false:
3294 }
3295 ❸ <first>: and <first>:: range
3296 \BNVS_new:cpn { if_resolve_query_return[A:]: } {
3297   \__bnvs_if_resolve:vcTF { A } { ans } {
3298     \__bnvs_tl_put_right:cn { ans } { - }
3299   } \prg_return_true:
3300 } {
3301   \prg_return_false:
3302 }
3303 ❹ :<last>::<length> or ::<length>:<last> range
3304 \BNVS_new:cpn { if_resolve_query_return[:Z::L]: } {
3305   \__bnvs_if_resolve:vcTF { Z } { Z } {
3306     \__bnvs_if_resolve:vcTF { L } { ans } {
3307       \__bnvs_tl_put_left:cn { ans } { 1- }
3308       \__bnvs_tl_put_right:cn { ans } { + }
3309       \__bnvs_tl_put_right:cv { ans } { Z }
3310     } \__bnvs_round:c { ans }
3311     \__bnvs_tl_put_right:cn { ans } { - }
3312     \__bnvs_tl_put_right:cv { ans } { Z }

```

```

3302     \prg_return_true:
3303   } {
3304     \prg_return_false:
3305   }
3306 } {
3307   \prg_return_false:
3308 }
3309 }

☛ : or :: range

3310 \BNVS_new:cpn { if_resolve_query_return[:]: } {
3311   \_bnvs_tl_set:cn { ans } { - }
3312   \prg_return_true:
3313 }

☛ : <last> range

3314 \BNVS_new:cpn { if_resolve_query_return[:Z]: } {
3315   \_bnvs_tl_set:cn { ans } { - }
3316   \_bnvs_if_append:vcTF { Z } { ans } {
3317     \prg_return_true:
3318   } {
3319     \prg_return_false:
3320   }
3321 }

```

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

Evaluate only one query.

```

3322 \BNVS_new_conditional:cpnn { if_resolve_query:nc } #1 #2 { T, F, TF } {
3323   \_bnvs_greset:
3324   \_bnvs_match_if_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
3325     \BNVS_begin:
3326     \_bnvs_if_resolve_query_branch:TF {
3327       \BNVS_end_tl_set:cv { #2 } { ans }
3328       \prg_return_true:
3329     } {
3330       \BNVS_end:
3331       \prg_return_false:
3332     }
3333   } {
3334     \BNVS_error:n { Syntax~error:~#1 }
3335     \BNVS_end:
3336     \prg_return_false:
3337   }
3338 }

```

```

\__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>* `tl` variable . This is executed within a local group. Below are local variables and constants used throughout the body of this function.

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

(End of definition for `\l__bnvs_query_seq`.)

`\l__bnvs_ans_seq` Storage for the evaluated result.

(End of definition for `\l__bnvs_ans_seq`.)

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

```

3339 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End of definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

3340 \BNVS_new_conditional:cpnn { if_resolve_queries:nc } #1 #2 { TF } {
3341   \BNVS_begin:

```

Local variables cleared

```

3342   \__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`.

```

3343   \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.

```

3344   \BNVS_set:cpn { end_return: } {
3345     \__bnvs_seq_if_empty:cTF { ans } {
3346       \BNVS_end:
3347     } {
3348       \exp_args:Nnx
3349       \use:n {
3350         \BNVS_end:
3351         \__bnvs_tl_put_right:cn { #2 }
3352       } { \__bnvs_seq_use:cn { ans } , }
3353     }
3354     \prg_return_true:
3355   }
3356   \__bnvs_seq_map_inline:cn { query } {
3357     \__bnvs_tl_clear:c { ans }
3358     \__bnvs_if_resolve_query:ncTF { ##1 } { ans } {
3359       \__bnvs_tl_if_empty:cF { ans } {
3360         \__bnvs_seq_put_right:cv { ans } { ans }
3361       }

```

```

3362 } {
3363   \seq_map_break:n {
3364     \BNVS_set:cpn { end_return: } {
3365       \BNVS_end:
3366       \BNVS_error:n { Circular/Undefined~dependency~in~#1}
3367       \exp_args:Nnx
3368       \use:n {
3369         \BNVS_end:
3370         \__bnvs_tl_put_right:cn { #2 }
3371       } { \__bnvs_seq_use:cn { ans } , }
3372       \prg_return_false:
3373     }
3374   }
3375 }
3376 }
3377 \__bnvs_end_return:

```

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

```

3378 }

3379 \NewDocumentCommand \BeanovesResolve { 0{} m } {
3380   \BNVS_begin:
3381   \keys_define:nn { BeanovesResolve } {
3382     in:N .tl_set:N = \l__bnvs_resolve_in_tl,
3383     in:N .initial:n = { },
3384     show .bool_set:N = \l__bnvs_resolve_show_bool,
3385     show .default:n = true,
3386     show .initial:n = false,
3387   }
3388   \keys_set:nn { BeanovesResolve } { #1 }
3389   \__bnvs_tl_clear:c { ans }
3390   \__bnvs_if_resolve_queries:ncTF { #2 } { ans } {
3391     \__bnvs_tl_if_empty:cTF { resolve_in } {
3392       \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3393         \BNVS_tl_use:Nv \BNVS_end: { ans }
3394       } {
3395         \BNVS_end:
3396       }
3397     } {
3398       \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3399         \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3400           \BNVS_end:
3401           \tl_set:Nn ##1 { ##2 }
3402           ##2
3403         }
3404         \BNVS_tl_use:nv {
3405           \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
3406         } { ans }
3407       } {
3408         \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3409           \BNVS_end:
3410           \tl_set:Nn ##1 { ##2 }
3411         }

```

```

3412         \BNVS_tl_use:nv {
3413             \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
3414         } { ans }
3415     }
3416 }
3417 } {}
3418 }

```

6.22 Resetting counters

```

3419 \BNVS_new:cpn { reset:n } #1 {
3420     \BNVS_begin:
3421     \__bnvs_set_true:c { reset }
3422     \__bnvs_set_false:c { provide }
3423     \__bnvs_tl_clear:c { root }
3424     \__bnvs_int_zero:c { i }
3425     \__bnvs_tl_set:cn { a } { #1 }
3426     \__bnvs_provide_off:
3427     \BNVS_tl_use:nv {
3428         \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn }
3429     } { a }
3430     \BNVS_end_tl_set:cv { id_last } { id_last }
3431 }

3432 \BNVS_new:cpn { reset:v } {
3433     \BNVS_tl_use:Nv \__bnvs_reset:n
3434 }

3435 \makeatletter
3436 \NewDocumentCommand \BeanovesReset { 0{ } m } {
3437     \tl_if_empty:NTF \@currentvir {

```

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

```

3438     \BNVS_error:x {No~\token_to_str:N \BeanovesReset{ }~in~the~preamble.}
3439 } {
3440     \tl_if_eq:NnT \@currentvir { document } {

```

At the top level, clear everything.

```

3441     \BNVS_error:x {No~\token_to_str:N \BeanovesReset{ }~at~the~top~level.}
3442 }
3443 \BNVS_begin:
3444 \__bnvs_set_true:c { reset }
3445 \__bnvs_set_false:c { provide }
3446 \keys_define:nn { BeanovesReset } {
3447     all .bool_set:N = \l__bnvs_reset_all_bool,
3448     all .default:n = true,
3449     all .initial:n = false,
3450     only .bool_set:N = \l__bnvs_only_bool,
3451     only .default:n = true,
3452     only .initial:n = false,
3453 }
3454 \keys_set:nn { BeanovesReset } { #1 }
3455 \__bnvs_tl_clear:c { root }
3456 \__bnvs_int_zero:c { i }
3457 \__bnvs_tl_set:cn { a } { #2 }

```

```

3458     \__bnvs_provide_off:
3459     \BNVS_tl_use:nv {
3460         \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn }
3461     } { a }
3462     \BNVS_end_tl_set:cv { id_last } { id_last }
3463     \ignorespaces
3464 }
3465 }
3466 \makeatother
3467 \ExplSyntaxOff

```