

beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2024/01/11

Abstract

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

Contents

1	Installation	2
1.1	Package manager	2
1.2	Manual installation	3
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 case	5
3.3.2	List specifiers	5
4	Resolution of <code>?(...)</code> query expressions	5
4.1	Number and range overlay queries	6
4.2	Counters	7
4.3	Dotted paths	8
4.4	The beamerpauses counter	8
4.5	Multiple queries	9
4.6	Frame id	9
5	Support	9

6	Implementation	9
6.1	Package declarations	10
6.2	Facility layer: definitions and naming	10
6.3	logging	11
6.4	Facility layer: Variables	12
6.4.1	Regex	17
6.4.2	Token lists	19
6.4.3	Strings	22
6.4.4	Sequences	24
6.4.5	Integers	25
6.4.6	Prop	25
6.5	Debug facilities	26
6.6	Debug messages	26
6.7	Variable facilities	26
6.8	Testing facilities	26
6.9	Local variables	26
6.10	Infinite loop management	27
6.11	Overlay specification	28
6.12	Basic functions	28
6.13	Functions with cache	30
6.14	Implicit value counter	32
6.15	Implicit index counter	35
6.16	Regular expressions	36
6.17	beamer.cls interface	39
6.18	Defining named slide ranges	39
6.19	Scanning named overlay specifications	49
6.20	Resolution	54
6.21	Evaluation bricks	62
6.22	Index counter	74
6.23	Value counter	76
6.24	Evaluation	81
6.25	Functions for the resolution	81
6.26	Resetting counters	105
6.27	Saving the beamer pauses counter	105

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 installs 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 at <https://github.com/jlaurens/beanoves>. There is also <https://tan.org/pkg/beanoves>.

1.3 Usage

The `beanoves` package is imported by putting `\RequirePackage{beanoves}` in the preamble of a `LATEX` document that uses the `beamer` class.

The features of `beanoves` can be temporarily deactivated with simple commands `\BeanoverOff` and `\BeanoverOn`.

2 Minimal example

The document below is a contrived example to show how the `beamer` overlay specifications have been extended.

```
1 \documentclass {beamer}
2 \RequirePackage {beanoves}
3 \begin{document}
4 \Beanoves {
5   A = 1:4,
6   B = A.last::3,
7   C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \insertframenumber}
11 {\Large Slide \insertslidenum}
12 \visible<?(A.1)> {Only on slide 1}\\
13 \visible<?(B.range)> {Only on slide 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 slide 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`.

$\langle \text{frame id} \rangle ! \langle \text{short name} \rangle \langle \text{dotted path} \rangle$: 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 $\langle \text{component} \rangle$ 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 `beanoves` option of this environment. The value of the `beanoves` option is similar to the argument of the `\Beanoves` commands, but the latter takes precedence on the former. This behaviour may be useful to input the very same source code into different frames and have different combinations of slides.

`beanoves` `beanoves = { \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 }`

`\Beanoves` `\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 }`

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.

3.3.1 Basic case

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

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

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

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

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

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside \LaTeX environments, each 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 $\langle \text{path} \rangle = \langle \text{spec} \rangle$ definitions. The definition

$\langle \text{ref} \rangle = \{ \{ \langle \text{path}_1 \rangle = \langle \text{spec}_1 \rangle, \langle \text{path}_2 \rangle = \langle \text{spec}_2 \rangle, \dots, \langle \text{path}_j \rangle = \langle \text{spec}_j \rangle \}$

is a convenient shortcut for

$\langle \text{ref} \rangle . \langle \text{path}_1 \rangle = \langle \text{spec}_1 \rangle,$
 $\langle \text{ref} \rangle . \langle \text{path}_2 \rangle = \langle \text{spec}_2 \rangle,$
 $\dots,$
 $\langle \text{ref} \rangle . \langle \text{path}_j \rangle = \langle \text{spec}_j \rangle.$

The rules above can apply individually to each line.

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

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

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

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

'?(<queries>)' Each one is then evaluated and replaced by its resolved static counterpart. The overall result is finally forwarded to the **beamer** class.

The <queries> argument is a comma separated list of individual <query>'s processed from left to right as explained below. Notice that nesting a ?(...) 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 ?(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 from below to feed **beamer** with either a range or a number. When the *query reference* is a previously declared <ref>, like X after X=1, it is simply replaced by the corresponding <value>. Otherwise, we use *implicit overlay queries* and their *resolution rules* depending on the definition of the named overlay set. Here <i> denotes a signed integer whereas <value>, <first>, <last> and <length> stand for raw integers or more general numerical expressions.

4.1 Number and range overlay queries

<ref> = <value> For an unlimited range

overlay query	resolution
<ref>.1	<value>
<ref>.2	<value> + 1
<ref>.<i>	<value> + <i> - 1

<ref> = <first>: as well as <first>:: For a range limited from below:

overlay query	resolution
<ref>.1	<first>
<ref>.2	<first> + 1
<ref>.<i>	<first> + <i> - 1
<ref>.previous	<first> - 1

Notice that <ref>.previous and <ref>.0 are sometimes synonyms.

<ref> = :<last> For a range limited from above:

overlay query	resolution
<ref>.1	<last>
<ref>.0	<last> - 1
<ref>.<i>	<last> + <i> - 1
<ref>.last	<last>
<ref>.next	<last> + 1

<ref> = <first>:<last> as well as variants <first>::<length>, ::<length>:<last> or :<last>::<length>, which are equivalent provided <first>+<length> = <last>+1.

For a range limited from both above and below:

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

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.last)   == 8,
10 \BeanovesResolve[show] (A.next)   == 9,
11 \BeanovesResolve[show] (A.length) == 6,
12 \BeanovesResolve[show] (A.range)  == 3-8,
13 \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.

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 $\langle \text{ref} \rangle$ overlay query is resolved into the position of this value counter. For each frame, this variable is initialized to the first available resolution amongst $\langle \text{value} \rangle$, $\langle \text{name} \rangle.1$, $\langle \text{name} \rangle.\text{first}$ 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.

²The parser for algebraic expression is very rudimentary.

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

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

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

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

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

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

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

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

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

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

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

4.3 Dotted paths

In previous overlay queries, when $\langle name \rangle$ is replaced by $\langle name \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_j \rangle$, the longest $\langle name \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_k \rangle$ 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.

4.4 The beamerpauses counter

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 `\BeamerSavePauses{\ref}` or in a query `?(\dots(\ref)=pauses)\dots`. Then later on, we can use `?(\ref)` to refer to this saved value in the same frame³. Next frame is an example.

³See <https://tex.stackexchange.com/questions/34458/reference-overlay-numbers-with-names> for an alternative that needs two passes.


```

1 \begin{frame}
2 \visible<+>{A}\\
3 \visible<+>{B\BeanovesSavePauses{afterB}}\\
4 \visible<+>{C}\\
5 \visible<?(afterB)>{C'}\\
6 \visible<?(afterB.previous)>{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.

4.5 Multiple queries

It is possible to replace the comma separated list $\langle query_1 \rangle, \dots \langle query_1 \rangle$ with the shorter $\langle query_1 \rangle, \dots \langle query_1 \rangle$.

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,`

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

5 Support

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

6 Implementation

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

`1 \@@=bnvs`

Reserved namespace: identifiers containing the case insensitive string `beanoves` or 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 $\text{\LaTeX}3$. The `c` and `v` argument specifiers take a slightly different meaning when used in a function which name contains with `bnvs` or `BNVS`. Where $\text{\LaTeX}3$ would transform `l__bnvs_ref_tl` into `\l__bnvs_ref_tl`, `bnvs` will directly transform `ref` into `\l__bnvs_ref_tl`. The type of the local variable used depends on the context and may be `seq` or `int` for example. There are however a pair of exceptions mentioned below. For a better reading experience, ‘`ref`’ will generally stand for `\l__bnvs_ref_tl`, whereas ‘`path sequence`’ will generally stand for `\l__bnvs_path_seq`. Other similar shortcuts are used as well.

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

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

These are naming functions.

```

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

```

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

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

```

11 \cs_new:Npn \BNVS_use_raw:N #1 { #1 }
12 \cs_new:Npn \BNVS_use_raw:c #1 {
13   \exp_last_unbraced:No
14   \BNVS_use_raw:N { \cs:w #1 \cs_end: }
15 }
16 \cs_new:Npn \BNVS_use:c #1 {
17   \BNVS_use_raw:c { \BNVS:c { #1 } }
18 }

```

```

19 \cs_new:Npn \BNVS_use_raw:NN #1 #2 {
20   #1 #2
21 }
22 \cs_new:Npn \BNVS_use_raw:nN #1 #2 {
23   #1 #2
24 }
25 \cs_new:Npn \BNVS_use_raw:Nc #1 #2 {
26   \exp_last_unbraced:NNo
27   \BNVS_use_raw:NN #1 { \cs:w #2 \cs_end: }
28 }
29 \cs_new:Npn \BNVS_use_raw:nc #1 #2 {
30   \exp_last_unbraced:Nno
31   \BNVS_use_raw:nN { #1 } { \cs:w #2 \cs_end: }
32 }
33 \cs_new:Npn \BNVS_use:Nc #1 #2 {
34   \BNVS_use_raw:Nc #1 { \BNVS:c { #2 } }
35 }
36 \cs_new:Npn \BNVS_use:nc #1 #2 {
37   \BNVS_use_raw:nc { #1 } { \BNVS:c { #2 } }
38 }
39 \cs_new:Npn \BNVS_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 \BNVS_new:cpn { warning:n } {
61   \msg_warning:nnn { beanoves } { :n }

```

```

62 }
63 \BNVS_generate_variant:cn { warning:n } { x }
64 \cs_new:Npn \BNVS_error:n {
65   \msg_error:nnn { beanoves } { :n }
66 }
67 \cs_new:Npn \BNVS_error:x {
68   \msg_error:nnx { beanoves } { :n }
69 }
70 \cs_new:Npn \BNVS_fatal:n {
71   \msg_fatal:nnn { beanoves } { :n }
72 }
73 \cs_new:Npn \BNVS_fatal:x {
74   \msg_fatal:nnx { beanoves } { :n }
75 }

```

6.4 Facility layer: Variables

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

`\BNVS_v_new:c`

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

```

76 \cs_new:Npn \BNVS_N_new:c #1 {
77   \cs_new:cpn { BNVS_#1:c } ##1 {
78     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
79   }
80   \cs_new:cpn { BNVS_#1_new:c } ##1 {
81     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
82   }
83   \cs_new:cpn { BNVS_#1_use:c } ##1 {
84     \use:c { \use:c { BNVS_#1:c } { ##1 } }
85   }
86   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
87     \BNVS_use_raw:Nc
88     ##1 { \use:c { BNVS_#1:c } { ##2 } }
89   }
90   \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
91     \BNVS_use_raw:nc
92     { ##1 } { \use:c { BNVS_#1:c } { ##2 } }
93   }
94 }
95 \cs_new:Npn \BNVS_v_new:c #1 {
96   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
97     \BNVS_use_raw:nc
98     { \exp_args:Nv ##1 }
99     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
100  }
101  \cs_new:cpn { BNVS_#1_use:cv } ##1 ##2 {
102    \BNVS_use_raw:nc
103    { \exp_args:NnV \BNVS_use:c { ##1 } }
104    { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
105  }
106  \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {

```

```

107     \BNVS_use_raw:nc
108     { \exp_args:NnV \use:n { ##1 } }
109     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
110 }
111 }
112 \BNVS_N_new:c { bool }
113 \BNVS_N_new:c { int }
114 \BNVS_v_new:c { int }
115 \BNVS_N_new:c { tl }
116 \BNVS_v_new:c { tl }
117 \BNVS_N_new:c { str }
118 \BNVS_v_new:c { str }
119 \BNVS_N_new:c { seq }
120 \BNVS_v_new:c { seq }
121 \cs_undefine:N \BNVS_N_new:c

```

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

```

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

```

```

158 }
159 \cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
160   \BNVS_use:ncn {
161     \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
162   }
163 }

```

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

```

164 \cs_new:Npn \BNVS_new_c:nc #1 #2 {
165   \BNVS_new_cpn { #1_#2:c } {
166     \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
167   }
168 }
169 \cs_new:Npn \BNVS_new_cn:nc #1 #2 {
170   \BNVS_new_cpn { #1_#2:cn } ##1 {
171     \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
172   }
173 }
174 \cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
175   \BNVS_new_cpn { #2:cnn } ##1 {
176     \BNVS_use:Ncn { #3 } { ##1 } { #1 }
177   }
178 }
179 \cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
180   \BNVS_use_raw:nc {
181     \BNVS_new_cnn:ncN { #1 } { #1_#2 }
182   } { #1_#2:Nnn }
183 }
184 \cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
185   \BNVS_new_cpn { #2:cnv } ##1 ##2 {
186     \BNVS_tl_use:nv {
187       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
188     }
189   }
190 }
191 \cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
192   \BNVS_use_raw:nc {
193     \BNVS_new_cnv:ncN { #1 } { #1_#2 }
194   } { #1_#2:Nnn }
195 }
196 \cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
197   \BNVS_new_cpn { #2:cnx } ##1 ##2 {
198     \exp_args:Nnx \use:n {
199       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
200     }
201   }
202 }
203 \cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
204   \BNVS_use_raw:nc {
205     \BNVS_new_cnx:ncN { #1 } { #1_#2 }
206   } { #1_#2:Nnn }
207 }
208 \cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {

```

```

209 \BNVS_new:cpn { #2:cc } ##1 ##2 {
210     \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
211 }
212 }
213 \cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
214     \BNVS_use_raw:nc {
215         \BNVS_new_cc:ncNn { #1 } { #1_#2 }
216     } { #1_#2:NN }
217 }
218 \cs_new:Npn \BNVS_new_cc:nc #1 #2 {
219     \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
220 }
221 \cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
222     \BNVS_new:cpn { #2:cn } ##1 {
223         \BNVS_use:Ncn #3 { ##1 } { #1 }
224     }
225 }
226 \cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
227     \BNVS_use_raw:nc {
228         \BNVS_new_cn:ncNn { #1 } { #1_#2 }
229     } { #1_#2:Nn }
230 }
231 \cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
232     \BNVS_new:cpn { #2:cv } ##1 ##2 {
233         \BNVS_use:nvn {
234             \BNVS_use:Ncn #3 { ##1 } { #1 }
235         } { ##2 } { #4 }
236     }
237 }
238 \cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
239     \BNVS_use_raw:nc {
240         \BNVS_new_cv:ncNn { #1 } { #1_#2 }
241     } { #1_#2:Nn }
242 }
243 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
244     \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
245 }
246 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
247     \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
248 }
249 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
250     \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
251 }
252 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
253     \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
254 }
255 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
256     \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
257 }
258 \cs_new:Npn \BNVS_g_prop_use:Nc #1 #2 {
259     \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { prop } }
260 }
261 \cs_new:Npn \BNVS_g_prop_use:nc #1 #2 {
262     \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { prop } }

```

```

263 }
264 \cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
265   \BNVS_use:ncncncn { \exp_args:NVVV #1 }
266   { #2 } { t1 } { #3 } { t1 } { #4 } { t1 }
267 }

```

```

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

```

```

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

```



```

314 \BNVS_use:nc {
315   \BNVS_new_conditional_vc:cNn { #1 }
316 } { #1:ncTF }
317 }

```

6.4.1 Regex

```

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

```

<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

```

321 \BNVS_new_conditional:cpnn { if_extract_once:Ncn } #1 #2 #3 { T, F, TF } {
322   \BNVS_use:ncn {
323     \regex_extract_once:NnNTF #1 { #3 }
324   } { #2 } { seq } {
325     \prg_return_true:
326   } {
327     \prg_return_false:
328   }
329 }
330 \BNVS_new_conditional:cpnn { match_if_once:Nn } #1 #2 { T, F, TF } {
331   \BNVS_use:ncn {
332     \regex_extract_once:NnNTF #1 { #2 }
333   } { match } { seq } {
334     \prg_return_true:
335   } {
336     \prg_return_false:
337   }
338 }
339 \BNVS_new_conditional:cpnn { if_extract_once:Ncv } #1 #2 #3 { T, F, TF } {
340   \BNVS_seq_use:nc {
341     \BNVS_tl_use:nv {
342       \regex_extract_once:NnNTF #1
343     } { #3 }
344   } { #2 } {
345     \prg_return_true:
346   } {

```

```

347     \prg_return_false:
348   }
349 }
350 \BNVS_new_conditional:cpnn { match_if_once:Nv } #1 #2 { T, F, TF } {
351   \BNVS_seq_use:nc {
352     \BNVS_tl_use:nv {
353       \regex_extract_once:NnNTF #1
354     } { #2 }
355   } { match } {
356     \prg_return_true:
357   } {
358     \prg_return_false:
359   }
360 }
361 \BNVS_new_conditional:cpnn { match_if_once:nn } #1 #2 { T, F, TF } {
362   \BNVS_seq_use:nc {
363     \regex_extract_once:nnNTF { #1 } { #2 }
364   } { match } {
365     \prg_return_true:
366   } {
367     \prg_return_false:
368   }
369 }
370 \BNVS_new_conditional:cpnn { if_regex_split:cnc } #1 #2 #3 { T, F, TF } {
371   \BNVS_seq_use:nc {
372     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
373   } { #3 } {
374     \prg_return_true:
375   } {
376     \prg_return_false:
377   }
378 }
379 \BNVS_new_conditional:cpnn { if_regex_split:cn } #1 #2 { T, F, TF } {
380   \BNVS_seq_use:nc {
381     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
382   } { split } {
383     \prg_return_true:
384   } {
385     \prg_return_false:
386   }
387 }

```

6.4.2 Token lists

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

These are shortcuts to

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

`\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.

```

388 \cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
389   \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
390     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
391       \prg_return_true:
392     } {
393       \prg_return_false:
394     }
395   }
396 }
397 \cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
398   \BNVS_use:nc {
399     \BNVS_new_conditional_vnc:cNn { #1 }
400   } { #1:nncTF }
401 }
```

`\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.

```

402 \cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
403   \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
404     \BNVS_tl_use:nv {
405       \BNVS_tl_use:Nv #2 { ##1 }
406     } { ##2 } { ##3 } { ##4 } {
407       \prg_return_true:
408     } {
409       \prg_return_false:
410     }
411   }
412 }
413 \cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
414   \BNVS_use:nc {
415     \BNVS_new_conditional_vvnc:cNn { #1 }
416   } { #1:nncTF }
417 }
418 \cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
419   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
420     \BNVS_tl_use:nv {
421       \BNVS_tl_use:Nv #2 { ##1 }
422     } { ##2 } { ##3 } {
423       \prg_return_true:
424     } {
425       \prg_return_false:
426     }
427   }
428 }
429 \cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
430   \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
431     \BNVS_tl_use:nv {
432       \BNVS_tl_use:nv {
433         \BNVS_tl_use:Nv #2 { ##1 }
434       } { ##2 }
435     } { ##3 } { ##4 } {
436       \prg_return_true:
437     } {
438       \prg_return_false:
439     }
440   }
441 }
442 \cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
443   \BNVS_use:nc {
444     \BNVS_new_conditional_vvc:cNn { #1 }
445   } { #1:nncTF }
446 }
447 \cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
448   \BNVS_use:nc {
449     \BNVS_new_conditional_vvvc:cNn { #1 }
450   } { #1:nncTF }
451 }
452 \cs_new:Npn \BNVS_new_tl_c:c {
453   \BNVS_new_c:nc { tl }
454 }

```

```

455 \BNVS_new_tl_c:c { clear }
456 \BNVS_new_tl_c:c { use }
457 \BNVS_new_tl_c:c { count }
458
459 \BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
460   \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
461 }
462 \cs_new:Npn \BNVS_new_tl_cn:c {
463   \BNVS_new_cn:nc { tl }
464 }
465 \cs_new:Npn \BNVS_new_tl_cv:c #1 {
466   \BNVS_new_cv:ncn { tl } { #1 } { tl }
467 }
468 \BNVS_new_tl_cn:c { set }
469 \BNVS_new_tl_cv:c { set }
470 \BNVS_new:cpn { tl_set:cx } {
471   \exp_args:Nnx \__bnvs_tl_set:cn
472 }
473 \BNVS_new_tl_cn:c { put_right }
474 \BNVS_new_tl_cv:c { put_right }
475 % \BNVS_generate_variant:cn { tl_put_right:cn } { cx }
476 \BNVS_new:cpn { tl_put_right:cx } {
477   \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
478 }
479 \BNVS_new_tl_cn:c { put_left }
480 \BNVS_new_tl_cv:c { put_left }
481 % \BNVS_generate_variant:cn { tl_put_left:cn } { cx }
482 \BNVS_new:cpn { tl_put_left:cx } {
483   \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
484 }

```

```

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

```

These are shortcuts to

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

```

485 \cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
486   \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
487     \BNVS_use:Ncn #3 { ##1 } { #1 } {
488       \prg_return_true:
489     } {
490       \prg_return_false:
491     }
492   }
493 }
494 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
495   \BNVS_use_raw:nc {
496     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
497   } { #1_#2:NTF }

```

```

498 }
499 \BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }
500 \BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
501   \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
502     \prg_return_true:
503   } {
504     \prg_return_false:
505   }
506 }
507 \cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
508   \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
509     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
510       \prg_return_true:
511     } {
512       \prg_return_false:
513     }
514   }
515 }
516 \cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
517   \BNVS_use_raw:nc {
518     \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
519   } { #1_#2:NnTF }
520 }
521 \BNVS_new_conditional_cn:ncn { tl } { if_eq } { T, F, TF }
522 \cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
523   \BNVS_new_conditional:cpnn { #2:cv } ##1 ##2 { #4 } {
524     \BNVS_use:nvn {
525       \BNVS_use:Ncn #3 { ##1 } { #1 }
526     } { ##2 } { #1 } {
527       \prg_return_true:
528     } {
529       \prg_return_false:
530     }
531   }
532 }
533 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
534   \BNVS_use_raw:nc {
535     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
536   } { #1_#2:NnTF }
537 }
538 \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>}`

```

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

```

```

542     \prg_return_true:
543   } {
544     \prg_return_false:
545   }
546 }
547 }
548 \cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
549   \BNVS_use_raw:nc {
550     \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
551   } { #1_#2:nnTF }
552 }
553 \BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }
554 \cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
555   \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
556     \BNVS_use:nvn {
557       \BNVS_use:Nvn #3 { ##1 } { #1 }
558     } { ##2 } { #1 } {
559       \prg_return_true:
560     } {
561       \prg_return_false:
562     }
563   }
564 }
565 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
566   \BNVS_use_raw:nc {
567     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
568   } { #1_#2:nnTF }
569 }
570 \BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }

```

6.4.4 Sequences

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

These are shortcuts to

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

```

571 \BNVS_new_c:nc { seq } { count }
572 \BNVS_new_c:nc { seq } { clear }
573 \BNVS_new_cn:nc { seq } { use }
574 \BNVS_new_cn:nc { seq } { item }
575 \BNVS_new_cn:nc { seq } { remove_all }
576 \BNVS_new_cn:nc { seq } { map_inline }
577 \BNVS_new_cc:nc { seq } { set_eq }
578 \BNVS_new_cv:ncn { seq } { put_left } { tl }
579 \BNVS_new_cn:ncn { seq } { put_right } { tl }
580 \BNVS_new_cv:ncn { seq } { put_right } { tl }
581 \BNVS_new_cnn:nc { seq } { set_split }
582 \BNVS_new_cnv:nc { seq } { set_split }
583 \BNVS_new_cnx:nc { seq } { set_split }
584 \BNVS_new_cc:ncn { seq } { pop_left } { tl }
585 \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

```

```

586 \cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
587   \BNVS_new_conditional_cpnn { #1_#2:cc } ##1 ##2 { #4 } {
588     \BNVS_use:ncncn {
589       \BNVS_use_raw:c { #1_#2:NNTF }
590     } { ##1 } { #1 } { ##2 } { #3 } {
591       \prg_return_true:
592     } {
593       \prg_return_false:
594     }
595   }
596 }
597 \BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
598 \BNVS_new_conditional_cc:ncnn
599   { seq } { get_right } { tl } { T, F, TF }
600 \BNVS_new_conditional_cc:ncnn
601   { seq } { pop_left } { tl } { T, F, TF }
602 \BNVS_new_conditional_cc:ncnn
603   { seq } { pop_right } { tl } { T, F, TF }

```

6.4.5 Integers

<pre> __bnvs_int_new:c __bnvs_int_use:c __bnvs_int_zero:c __bnvs_int_inc:c __bnvs_int_decr:c __bnvs_int_set:cn __bnvs_int_set:cv </pre>	<pre> __bnvs_int_new:c __bnvs_int_use:c __bnvs_int_incr:c __bnvs_int_decr:c __bnvs_int_set:cn </pre>	<pre> {<core>} {<core>} {<core>} {<core>} {<core>} {<value>} </pre>
---	---	---

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>

```

604 \BNVS_new_c:nc { int } { new }
605 \BNVS_new_c:nc { int } { use }
606 \BNVS_new_c:nc { int } { zero }
607 \BNVS_new_c:nc { int } { incr }
608 \BNVS_new_c:nc { int } { decr }
609 \BNVS_new_cn:nc { int } { set }
610 \BNVS_new_cv:ncn { int } { set } { int }

```

6.4.6 Prop

```

\__bnvs_if_prop_get:NncTF

```

```

611 \BNVS_new_conditional:cpnn { if_prop_get:Nnc } #1 #2 #3 { T, F, TF } {
612   \BNVS_use:ncn {
613     \prop_get:NnNTF #1 { #2 }
614   } { #3 } { t1 } {
615     \prg_return_true:
616   } {
617     \prg_return_false:
618   }
619 }

```

6.5 Debug facilities

Typesetting file `beanoves.dtx` creates both `beanoves` and `beanoves-debug` style files. The former is intended for everyday use whereas the latter contains supplemental debugging and testing facilities which are intentionally left undocumented. In particular, we have aliases for `\group_begin:` and `\group_end:` to allow the display of supplemental informations while debugging.

6.6 Debug messages

6.7 Variable facilities

6.8 Testing facilities

6.9 Local variables

We make heavy use of local variables and function scopes. Many functions are executed within a \TeX group, which ensures no name collision with the caller stack. The number of variables used has not been optimized, nor the \TeX groups used. Optimization often goes against readability.

```

620 \tl_new:N \l__bnvs_id_last_tl
621 \tl_set:Nn \l__bnvs_id_last_tl { ?! }
622 \tl_new:N \l__bnvs_a_tl
623 \tl_new:N \l__bnvs_b_tl
624 \tl_new:N \l__bnvs_c_tl
625 \tl_new:N \l__bnvs_V_tl
626 \tl_new:N \l__bnvs_A_tl
627 \tl_new:N \l__bnvs_L_tl
628 \tl_new:N \l__bnvs_Z_tl
629 \tl_new:N \l__bnvs_ans_tl
630 \tl_new:N \l__bnvs_FQ_name_tl
631 \tl_new:N \l__bnvs_FQ_base_tl
632 \tl_new:N \l__bnvs_ref_tl
633 \tl_new:N \l__bnvs_ref_base_tl
634 \tl_new:N \l__bnvs_id_tl
635 \tl_new:N \l__bnvs_n_tl
636 \tl_new:N \l__bnvs_path_tl
637 \tl_new:N \l__bnvs_group_tl
638 \tl_new:N \l__bnvs_scan_tl
639 \tl_new:N \l__bnvs_query_tl
640 \tl_new:N \l__bnvs_token_tl
641 \tl_new:N \l__bnvs_root_tl

```

```

642 \tl_new:N \l__bnvs_n_incr_tl
643 \tl_new:N \l__bnvs_incr_tl
644 \tl_new:N \l__bnvs_plus_tl
645 \tl_new:N \l__bnvs_rhs_tl
646 \tl_new:N \l__bnvs_post_tl
647 \tl_new:N \l__bnvs_suffix_tl
648 \int_new:N \g__bnvs_call_int
649 \int_new:N \l__bnvs_int
650 \int_new:N \l__bnvs_i_int
651 \seq_new:N \g__bnvs_def_seq
652 \seq_new:N \l__bnvs_ans_seq
653 \seq_new:N \l__bnvs_match_seq
654 \seq_new:N \l__bnvs_split_seq
655 \seq_new:N \l__bnvs_path_seq
656 \seq_new:N \l__bnvs_path_base_seq
657 \seq_new:N \l__bnvs_path_head_seq
658 \seq_new:N \l__bnvs_path_tail_seq
659 \seq_new:N \l__bnvs_query_seq
660 \seq_new:N \l__bnvs_token_seq
661 \bool_new:N \l__bnvs_in_frame_bool
662 \bool_set_false:N \l__bnvs_in_frame_bool
663 \bool_new:N \l__bnvs_parse_bool

```

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

```

664 \bool_new:N \l__bnvs_provide_bool
665 \BNVS_new:cpn { provide_on: } {
666   \bool_set_true:N \l__bnvs_provide_bool
667 }
668 \BNVS_new:cpn { provide_off: } {
669   \bool_set_false:N \l__bnvs_provide_bool
670 }
671 \__bnvs_provide_off:

```

```

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

```

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

```

672 \BNVS_new_conditional:cpnn { if_provide: } { p, T, F, TF } {
673   \bool_if:NTF \l__bnvs_provide_bool {
674     \prg_return_true:
675   } {
676     \prg_return_false:
677   }
678 }

```

6.10 Infinite loop management

Unending recursivity is managed here.

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

(End of definition for `\g__bnvs_call_int`.)

```

679 \int_const:Nn \c__bnvs_max_call_int { 8192 }

```

`__bnvs_call_greset:` `__bnvs_call_greset:`

Reset globally the call stack counter to its maximum value.

```

680 \BNVS_new:cpn { call_greset: } {
681   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
682 }

```

`__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.

```

683 \BNVS_new_conditional:cpnn { if_call: } { T, F, TF } {
684   \int_gdecr:N \g__bnvs_call_int
685   \int_compare:nNnTF \g__bnvs_call_int > 0 {
686     \prg_return_true:
687   } {
688     \prg_return_false:
689   }
690 }

```

6.11 Overlay specification

6.12 Basic functions

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

`<FQ name>/V` for the value

`<FQ name>/A` for the first index

`<FQ name>/L` for the length when provided

`<FQ name>/Z` for the last index when provided

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

```

691 \prop_new:N \g__bnvs_prop

```

(End of definition for `\g__bnvs_prop`.)

<code>__bnvs_gput:nnn</code>	<code>__bnvs_gput:nnn {<subkey>} {<FQ name>} {<integer spec>}</code>
<code>__bnvs_gput:(nvn nnv)</code>	<code>__bnvs_item:nn {<subkey>} {<FQ name>}</code>
<code>__bnvs_item:nn</code>	<code>__bnvs_gremove:nn {<subkey>} {<FQ name>}</code>
<code>__bnvs_gremove:nn</code>	<code>__bnvs_gclear:n {<FQ name>}</code>
<code>__bnvs_gclear:n</code>	<code>__bnvs_gclear:</code>

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L^AT_EX3 eponym functions. The key used in `\g__bnvs_prop` is `<FQ name>/<subkey>`. In practice, `<subkey>` is one of V, A, L, Z. `fq` means “fully qualified”.

```

692 \BNVS_new:cpn { gput:nnn } #1 #2 {
693   \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
694 }

```

```

695 \BNVS_new:cpn { gput:nvn } #1 {
696   \BNVS_tl_use:nv {
697     \__bnvs_gput:nnn { #1 }
698   }
699 }
700 \BNVS_new:cpn { gput:nnv } #1 #2 {
701   \BNVS_tl_use:nv {
702     \__bnvs_gput:nnn { #1 } { #2 }
703   }
704 }
705 \BNVS_new:cpn { item:nn } #1 #2 {
706   \prop_item:Nn \g__bnvs_prop { #2 / #1 }
707 }
708 \BNVS_new:cpn { gremove:nn } #1 #2 {
709   \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
710 }
711 \BNVS_new:cpn { gclear:n } #1 {
712   \clist_map_inline:nn { V, A, Z, L } {
713     \__bnvs_gremove:nn { ##1 } { #1 }
714   }
715   \__bnvs_cache_gclear:n { #1 }
716 }
717 \BNVS_new:cpn { gclear: } {
718   \prop_gclear:N \g__bnvs_prop
719 }
720 \BNVS_generate_variant:cn { gclear:n } { V }
721 \BNVS_new:cpn { gclear:v } {
722   \BNVS_tl_use:Nc \__bnvs_gclear:V
723 }

```

```

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

```

Convenient shortcuts to test for the existence of $\langle FQ\ name \rangle / \langle subkey \rangle$, 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.

```

724 \BNVS_new_conditional:cpnn { if_in:nn } #1 #2 { p, T, F, TF } {
725   \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
726     \prg_return_true:
727   } {
728     \prg_return_false:
729   }
730 }
731 \BNVS_new_conditional:cpnn { if_in:n } #1 { p, T, F, TF } {
732   \bool_if:nTF {
733     \__bnvs_if_in_p:nn V { #1 }
734     || \__bnvs_if_in_p:nn A { #1 }
735     || \__bnvs_if_in_p:nn Z { #1 }
736   } {
737     \prg_return_true:
738   } {

```

```

739     \prg_return_false:
740   }
741 }
742 \BNVS_new_conditional:cpnn { if_in:v } #1 { p, T, F, TF } {
743   \BNVS_tl_use:Nv \__bnvs_if_in:nTF { #1 }
744   { \prg_return_true: } { \prg_return_false: }
745 }

```

__bnvs_gprovide:nnnT __bnvs_gprovide:nnnT {<subkey>} {<FQ name>} {<value>} {<yes precode>}

Execute <yes precode> before providing.

```

746 \BNVS_new:cpn { gprovide:nnnT } #1 #2 #3 #4 {
747   \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
748     #4
749     \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
750   }
751 }

```

__bnvs_if_get:nncTF __bnvs_if_get:nncTF {<subkey>} {<FQ name>} {<ans>} {<yes code>} {<no code>}

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

```

752 \BNVS_new_conditional:cpnn { if_get:nnc } #1 #2 #3 { T, F, TF } {
753   \BNVS_tl_use:nc {
754     \prop_get:NnNTF \g__bnvs_prop { #2 / #1 }
755   } { #3 } {
756     \prg_return_true:
757   } {
758     \prg_return_false:
759   }
760 }
761 \BNVS_new_conditional:cpnn { if_get:nvc } #1 #2 #3 { T, F, TF } {
762   \BNVS_tl_use:nv {
763     \__bnvs_if_get:nncTF { #1 }
764   } { #2 } { #3 } {
765     \prg_return_true:
766   } {
767     \prg_return_false:
768   }
769 }

```

6.13 Functions with cache

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

<FQ name>/V for the cached static value of the value

<FQ name>/A for the cached static value of the first index

$\langle FQ \text{ name} \rangle/L$ for the cached static value of the length

$\langle FQ \text{ name} \rangle/Z$ for the cached static value of the last index

$\langle FQ \text{ name} \rangle/P$ for the cached static value of the previous index

$\langle FQ \text{ name} \rangle/N$ for the cached static value of the next index

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

```
770 \prop_new:N \g__bnvs_cache_prop
```

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

$\backslash_bnvs_cache_gput:nnn$	$\backslash_bnvs_cache_gput:nnn \{ \langle subkey \rangle \} \{ \langle FQ \text{ name} \rangle \} \{ \langle value \rangle \}$
$\backslash_bnvs_cache_gput:(nnv nvn nvv)$	$\backslash_bnvs_cache_item:nn \{ \langle subkey \rangle \} \{ \langle FQ \text{ name} \rangle \}$
$\backslash_bnvs_cache_item:nn$	$\backslash_bnvs_cache_gremove:nn \{ \langle subkey \rangle \} \{ \langle FQ \text{ name} \rangle \}$
$\backslash_bnvs_cache_gremove:nn$	$\backslash_bnvs_cache_gclear:n \{ \langle FQ \text{ name} \rangle \}$
$\backslash_bnvs_cache_gclear:n$	$\backslash_bnvs_cache_gclear:$
$\backslash_bnvs_cache_gclear:$	

Wrapper over the functions above for $\langle FQ \text{ name} \rangle / \langle subkey \rangle$.

```
771 \BNVS_new:cpn { cache_gput:nnn } #1 #2 {
772   \prop_gput:Nnn \g__bnvs_cache_prop { #2 / #1 }
773 }
774 \BNVS_new:cpn { cache_gput:nvn } #1 {
775   \BNVS_tl_use:nv {
776     \__bnvs_cache_gput:nnn { #1 }
777   }
778 }
779 \BNVS_new:cpn { cache_gput:nnv } #1 #2 {
780   \BNVS_tl_use:nv {
781     \__bnvs_cache_gput:nnn { #1 } { #2 }
782   }
783 }
784 \BNVS_new:cpn { cache_gput:nvv } #1 #2 {
785   \BNVS_tl_use:nv {
786     \__bnvs_cache_gput:nvn { #1 } { #2 }
787   }
788 }
789 \BNVS_new:cpn { cache_item:nn } #1 #2 {
790   \prop_item:Nn \g__bnvs_cache_prop { #2 / #1 }
791 }
792 \BNVS_new:cpn { cache_gremove:nn } #1 #2 {
793   \prop_gremove:Nn \g__bnvs_cache_prop { #2 / #1 }
794 }
795 \BNVS_new:cpn { cache_gclear:n } #1 {
796   \clist_map_inline:nn { V, A, Z, L, P, N } {
797     \prop_gremove:Nn \g__bnvs_cache_prop { #1 / ##1 }
798   }
799 }
800 \BNVS_new:cpn { cache_gclear: } {
801   \prop_gclear:N \g__bnvs_cache_prop
802 }
```

```

\__bnvs_cache_if_in_p:nn * \__bnvs_cache_if_in_p:n {<subkey>} {<FQ name>}
\__bnvs_cache_if_in:nnTF * \__bnvs_cache_if_in:nTF {<subkey>} {<FQ name>} {<yes code>} {<no code>}

```

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

```

803 \prg_new_conditional:Npnn \__bnvs_cache_if_in:nn #1 #2 { p, T, F, TF } {
804   \prop_if_in:NnTF \g__bnvs_cache_prop { #2 / #1 } {
805     \prg_return_true:
806   } {
807     \prg_return_false:
808   }
809 }

```

```

\__bnvs_cache_if_get:nncTF \__bnvs_cache_if_get:nncTF {<subkey>} {<FQ name>} {<tl core>} {<yes code>} {<no code>}

```

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

```

810 \BNVS_new_conditional:cpnn { cache_if_get:nnc } #1 #2 #3 { T, F, TF } {
811   \BNVS_tl_use:nc {
812     \prop_get:NnTF \g__bnvs_cache_prop { #2 / #1 }
813   } { #3 } {
814     \prg_return_true:
815   } {
816     \prg_return_false:
817   }
818 }

```

6.14 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.

$\backslash g_bnvs_v_prop$ $\langle key \rangle$ – $\langle value \rangle$ property list to store the contents or the named value counters. The keys are fully qualified names $\langle id \rangle ! \langle name \rangle$ denoted as $\langle FQ name \rangle$.

```

819 \prop_new:N \g__bnvs_v_prop

```

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

```

\__bnvs_v_gput:nn \__bnvs_v_gput:nn {<Q name>} {<value>}
\__bnvs_v_gput:(nv|vn) \__bnvs_v_item:n {<Q name>}
\__bnvs_v_item:n \__bnvs_v_gremove:n {<Q name>}
\__bnvs_v_gremove:n \__bnvs_v_gclear:
\__bnvs_v_gclear:

```

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

```

820 \BNVS_new:cpn { v_gput:nn } {
821   \prop_gput:Nnn \g__bnvs_v_prop
822 }

```



```

823 \BNVS_new:cpn { v_gput:nv } #1 {
824   \BNVS_tl_use:nv {
825     \__bnvs_v_gput:nn { #1 }
826   }
827 }
828 \BNVS_new:cpn { v_item:n } #1 {
829   \prop_item:Nn \g__bnvs_v_prop { #1 }
830 }
831 \BNVS_new:cpn { v_gremove:n } {
832   \prop_gremove:Nn \g__bnvs_v_prop
833 }
834 \BNVS_new:cpn { v_gclear: } {
835   \prop_gclear:N \g__bnvs_v_prop
836 }

```

```

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

```

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

```

837 \BNVS_new_conditional:cpnn { v_if_in:n } #1 { p, T, F, TF } {
838   \prop_if_in:NnTF \g__bnvs_v_prop { #1 } {
839     \prg_return_true:
840   } {
841     \prg_return_false:
842   }
843 }

```

```

\__bnvs_v_if_get:ncTF \__bnvs_v_if_get:ncTF {<Q name>} {<tl core>} {<yes code>} {<no code>}

```

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

```

844 \BNVS_new_conditional:cpnn { v_if_get:nc } #1 #2 { T, F, TF } {
845   \BNVS_tl_use:nc {
846     \prop_get:NnNTF \g__bnvs_v_prop { #1 }
847   } { #2 } {
848     \prg_return_true:
849   } {
850     \prg_return_false:
851   }
852 }

```

```

\__bnvs_v_if_greset:nnTF \__bnvs_v_if_greset:nnTF {<FQ name>} {<initial value>} {<yes code>} {<no
\__bnvs_v_if_greset:(vn|nv)TF code>}
\__bnvs_if_greset_all:nnTF \__bnvs_if_greset_all:nnTF {<FQ name>} {<initial value>} {<yes code>} {<no
\__bnvs_if_greset_all:vnTF code>}

```

If the $\langle FQ\ name \rangle$ is known, reset the value counter to the given $\langle initial\ value \rangle$ otherwise and execute $\langle yes\ code \rangle$ otherwise $\langle no\ code \rangle$ is executed. The \dots_all variant also cleans the cached values.

```

853 \BNVS_new_conditional:cpnn { v_if_greset:nn } #1 #2 { T, F, TF } {
854   \__bnvs_v_if_in:nTF { #1 } {
855     \__bnvs_v_gremove:n { #1 }
856     \tl_if_empty:nF { #2 } {
857       \__bnvs_v_gput:nn { #1 } { #2 }
858     }
859     \prg_return_true:
860   } {
861     \prg_return_false:
862   }
863 }
864 \BNVS_new_conditional:cpnn { v_if_greset:nv } #1 #2 { T, F, TF } {
865   \BNVS_tl_use:Nv { \__bnvs_v_if_greset:nnTF { #1 } } { #2 }
866   { \prg_return_true: } { \prg_return_false: }
867 }
868 \BNVS_new_conditional:cpnn { v_if_greset:vn } #1 #2 { T, F, TF } {
869   \BNVS_tl_use:Nv \__bnvs_v_if_greset:nnTF { #1 } { #2 }
870   { \prg_return_true: } { \prg_return_false: }
871 }
872 \BNVS_new_conditional:cpnn { if_greset_all:nn } #1 #2 { T, F, TF } {
873   \__bnvs_if_in:nTF { #1 } {
874     \BNVS_begin:
875     \clist_map_inline:nn { V, A, Z, L } {
876       \__bnvs_if_get:nncT { ##1 } { #1 } { a } {
877         \__bnvs_quark_if_nil:cT { a } {
878           \__bnvs_cache_if_get:nncTF { ##1 } { #1 } { a } {
879             \__bnvs_gput:nnv { ##1 } { #1 } { a }
880           } {
881             \__bnvs_gput:nnn { ##1 } { #1 } { 1 }
882           }
883         }
884       }
885     }
886     \BNVS_end:
887     \__bnvs_cache_gclear:n { #1 }
888     \__bnvs_v_if_greset:nnT { #1 } { #2 } {}
889     \prg_return_true:
890   } {
891     \prg_return_false:
892   }
893 }
894 \BNVS_new_conditional:cpnn { if_greset_all:vn } #1 #2 { T, F, TF } {
895   \BNVS_tl_use:Nv \__bnvs_if_greset_all:nnTF { #1 } { #2 }
896   { \prg_return_true: } { \prg_return_false: }
897 }

```

```

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

```

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

```

898 \BNVS_new:cpn { gclear_all: } {
899   \__bnvs_gclear:
900   \__bnvs_cache_gclear:
901   \__bnvs_n_gclear:
902   \__bnvs_v_gclear:
903 }
904 \BNVS_new:cpn { gclear_all:n } #1 {
905   \__bnvs_gclear:n { #1 }
906   \__bnvs_cache_gclear:n { #1 }
907   \__bnvs_n_gremove:n { #1 }
908   \__bnvs_v_gremove:n { #1 }
909 }

```

6.15 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 names $\langle id \rangle!$ $\langle short name \rangle$.

```

910 \prop_new:N \g__bnvs_n_prop

```

(End of definition for `\g__bnvs_n_prop`.)

<code>__bnvs_n_gput:nn</code>	<code>__bnvs_n_gput:nn {$\langle Q name \rangle$} {$\langle value \rangle$}</code>
<code>__bnvs_n_gput:(nv vn)</code>	<code>__bnvs_n_item:n {$\langle Q name \rangle$}</code>
<code>__bnvs_n_gprovide:nn</code>	<code>__bnvs_n_gremove:n {$\langle Q name \rangle$}</code>
<code>__bnvs_n_item:n</code>	<code>__bnvs_n_gclear:</code>
<code>__bnvs_n_gremove:n</code>	
<code>__bnvs_n_gremove:v</code>	
<code>__bnvs_n_gclear:</code>	

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

```

911 \BNVS_new:cpn { n_gput:nn } {
912   \prop_gput:Nnn \g__bnvs_n_prop
913 }
914 \cs_generate_variant:Nn \__bnvs_n_gput:nn { nV }
915 \BNVS_new:cpn { n_gput:nv } #1 {
916   \BNVS_tl_use:nc {
917     \__bnvs_n_gput:nV { #1 }
918   }
919 }
920 \BNVS_new:cpn { n_gprovide:nn } #1 #2 {
921   \prop_if_in:NnF \g__bnvs_n_prop { #1 } {
922     \prop_gput:Nnn \g__bnvs_n_prop { #1 } { #2 }
923   }
924 }
925 \BNVS_new:cpn { n_item:n } #1 {
926   \prop_item:Nn \g__bnvs_n_prop { #1 }
927 }
928 \BNVS_new:cpn { n_gremove:n } {
929   \prop_gremove:Nn \g__bnvs_n_prop
930 }

```

```

931 \BNVS_generate_variant:cn { n_gremove:n } { V }
932 \BNVS_new:cpn { n_gremove:v } {
933   \BNVS_tl_use:nc {
934     \__bnvs_n_gremove:V
935   }
936 }
937 \BNVS_new:cpn { n_gclear: } {
938   \prop_gclear:N \g__bnvs_n_prop
939 }
940 \cs_generate_variant:Nn \__bnvs_n_gremove:n { V }

```

```

\__bnvs_n_if_in_p:n * \__bnvs_n_if_in_p:nn {<Q name>}
\__bnvs_n_if_in:nTF * \__bnvs_n_if_in:nTF {<Q name>} {<yes code>} {<no code>}

```

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

```

941 \prg_new_conditional:Npnn \__bnvs_n_if_in:n #1 { p, T, F, TF } {
942   \prop_if_in:NnTF \g__bnvs_n_prop { #1 } {
943     \prg_return_true:
944   } {
945     \prg_return_false:
946   }
947 }

```

```

\__bnvs_n_if_get:ncTF \__bnvs_n_if_get:ncTF {<Q name>} <tl core> {<yes code>} {<no code>}

```

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

```

948 \prg_new_conditional:Npnn \__bnvs_n_if_get:nc #1 #2 { T, F, TF } {
949   \__bnvs_if_prop_get:NncTF \g__bnvs_n_prop { #1 } { #2 } {
950     \prg_return_true:
951   } {
952     \prg_return_false:
953   }
954 }

```

6.16 Regular expressions

$\backslash\text{c_bnvs_short_regex}$ 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.

```

955 \regex_const:Nn \c__bnvs_short_regex {
956   [[:alpha:]]_ [[:alnum:]]_*
957 }

```

(End of definition for $\backslash\text{c_bnvs_short_regex}$.)

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

```

958 \regex_const:Nn \c__bnvs_id_regex {
959   (?: \ur{c__bnvs_short_regex} | [?] )? !
960 }

```

(End of definition for \c__bnvs_id_regex.)

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

```

961 \regex_const:Nn \c__bnvs_path_regex {
962   (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )*
963 }

```

(End of definition for \c__bnvs_path_regex.)

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

(End of definition for \c__bnvs_A_FQ_name_Z_regex.)

```

964 \regex_const:Nn \c__bnvs_A_FQ_name_Z_regex {
    1: The range name including the frame ⟨id⟩ and exclamation mark if any
    2: frame ⟨id⟩ including the exclamation mark
965   \A ( ( \ur{c__bnvs_id_regex} ? ) \ur{c__bnvs_short_regex} )
    3: the path, if any.
966   ( \ur{c__bnvs_path_regex} ) \Z
967 }

```

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

(End of definition for \c__bnvs_A_FQ_name_n_Z_regex.)

```

968 \regex_const:Nn \c__bnvs_A_FQ_name_n_Z_regex {
    1: The full match
    2: The fully qualified name including the frame ⟨id⟩ and exclamation mark if any,
      the dotted path but excluding the trailing .n (this is \c__bnvs_path_regex with
      a trailing ?).
    3: frame ⟨id⟩ including the exclamation mark

```

```

969   \A ( ( \ur{c__bnvs_id_regex} ? )
970         \ur{c__bnvs_short_regex}
971         (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )*(? )

```

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

```

972   ( \. n )? \Z
973 }

```

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

```
974 \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.

```
975 \regex_const:Nn \c__bnvs_split_regex {
976   \s* ( ? :
```

We start with ‘++’ instrussions⁴.

1 incrementation prefix

```
977   \+\+
```

1.1: *⟨qualified name⟩* of an overlay set

1.2: *⟨id⟩* of a an overlay set including the exclamation mark

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

1.3: optionally followed by a dotted path

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

2: without incement prefix

2.1: *⟨qualified name⟩* of an overlay set

2.2: *⟨id⟩* of a slide range including the exclamation mark

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

2.3: optionally followed by a dotted path

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

We continue with other expressions

2.4: the *⟨++n⟩* attribute

```
982   (?: \.(\+)\+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.

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

2.7: the post increment

```
984   | (\+)\+
```

```
985   )?
```

```
986   ) \s*
```

```
987 }
```

(End of definition for `\c__bnvs_split_regex`.)

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

6.17 beamer.cls interface

Work in progress.

```

988 \RequirePackage{keyval}
989 \define@key{beamerframe}{beanoves~id}[] {
990   \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }
991 }
992 \AddToHook{env/beamer@frameslide/before}{
993   \__bnvs_call_greset:
994   \__bnvs_n_gclear:
995   \__bnvs_v_gclear:
996   \bool_set_true:N \l__bnvs_in_frame_bool
997 }
998 \AddToHook{env/beamer@frameslide/after}{
999   \bool_set_false:N \l__bnvs_in_frame_bool
1000 }

```

6.18 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 `__bnvs_colons_regex` and set the variables accordingly. $\langle tl \rangle$ is expected to only contain colons and integers.

```

1001 \BNVS_new_conditional:cpnn { split_if_pop_left:c } #1 { T, F, TF } {
1002   \__bnvs_seq_pop_left:ccTF { split } { #1 } {
1003     \prg_return_true:
1004   } {
1005     \prg_return_false:
1006   }
1007 }
1008 \exp_args_generate:n { VVV }
1009 \BNVS_new_conditional:cpnn { range_if_set:cccn } #1 #2 #3 #4 { T, F, TF } {
1010   \BNVS_begin:
1011   \__bnvs_tl_clear:c { a }
1012   \__bnvs_tl_clear:c { b }
1013   \__bnvs_tl_clear:c { c }
1014   \__bnvs_if_regex_split:cnTF { colons } { #4 } {
1015     \__bnvs_seq_pop_left:ccT { split } { a } {

```

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

```

1016     \__bnvs_seq_pop_left:ccT { split } { b } {
1017       \__bnvs_tl_if_empty:cTF { b } {

```

This is a one colon range.

```

1018       \__bnvs_split_if_pop_left:cTF { b } {

```

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

```

1019       \__bnvs_seq_pop_left:ccT { split } { c } {
1020       \__bnvs_tl_if_empty:cTF { c } {

```

A :: was expected:

```

1021         \BNVS_error:n { Invalid-range-expression(1):~#4 }
1022     } {
1023         \int_compare:nNtT { \__bnvs_tl_count:c { c } } > { 1 } {
1024             \BNVS_error:n { Invalid-range-expression(2):~#4 }
1025         }
1026         \__bnvs_split_if_pop_left:cTF { c } {

```

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

```

1027         \__bnvs_seq_if_empty:cF { split } {
1028             \BNVS_error:n { Invalid-range-expression(3):~#4 }
1029         }
1030     } {
1031         \BNVS_error:n { Internal-error }
1032     }
1033 }
1034 }
1035 } {
1036 }
1037 } {

```

This is a two colon range component.

```

1038         \int_compare:nNtT { \__bnvs_tl_count:c { b } } > { 1 } {
1039             \BNVS_error:n { Invalid-range-expression(4):~#4 }
1040         }
1041         \__bnvs_seq_pop_left:ccT { split } { c } {

```

c contains the $\langle length \rangle$.

```

1042         \__bnvs_split_if_pop_left:cTF { b } {
1043             \__bnvs_tl_if_empty:cTF { b } {
1044                 \__bnvs_seq_pop_left:cc { split } { b }

```

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

```

1045         \__bnvs_seq_if_empty:cF { split } {
1046             \BNVS_error:n { Invalid-range-expression(5):~#4 }
1047         }
1048     } {
1049         \BNVS_error:n { Invalid-range-expression(6):~#4 }
1050     }
1051 } {
1052     \__bnvs_tl_clear:c { b }
1053 }
1054 }
1055 }
1056 }
1057 }

```

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.

```

1058     \cs_set:Npn \BNVS_next: { }
1059     \__bnvs_tl_if_empty:cT { a } {
1060         \__bnvs_tl_if_empty:cT { b } {
1061             \__bnvs_tl_if_empty:cT { c } {
1062                 \cs_set:Npn \BNVS_next: {
1063                     \BNVS_error:n { Invalid-range-expression(7):~#3 }

```



```

1064     }
1065   }
1066 }
1067 }
1068 \BNVS_next:
1069 \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1070   \BNVS_end:
1071   \__bnvs_tl_set:cn { #1 } { ##1 }
1072   \__bnvs_tl_set:cn { #2 } { ##2 }
1073   \__bnvs_tl_set:cn { #3 } { ##3 }
1074 }
1075 \BNVS_exp_args:Nvvv \BNVS:nnn { a } { b } { c }
1076 \prg_return_true:
1077 } {
1078   \BNVS_end:
1079   \prg_return_false:
1080 }
1081 }

```

`__bnvs_range:nnnn` `__bnvs_range:nnnn {<FQ name>} {<start>} {<end>} {<length>}`
`__bnvs_range:nvvv` Auxiliary function called within a group. Setup the model to define a range.

```

1082 \BNVS_new:cpn { range:nnnn } #1 {
1083   \__bnvs_if_provide:TF {
1084     \__bnvs_if_in:nnTF A { #1 } {
1085       \use_none:nnn
1086     } {
1087       \__bnvs_if_in:nnTF Z { #1 } {
1088         \use_none:nnn
1089       } {
1090         \__bnvs_if_in:nnTF L { #1 } {
1091           \use_none:nnn
1092         } {
1093           \__bnvs_do_range:nnnn { #1 }
1094         }
1095       }
1096     }
1097   } {
1098     \__bnvs_do_range:nnnn { #1 }
1099   }
1100 }
1101 \BNVS_new:cpn { range:nvvv } #1 #2 #3 #4 {
1102   \BNVS_tl_use:nv {
1103     \BNVS_tl_use:nv {
1104       \BNVS_tl_use:nv {
1105         \BNVS_use:c { range:nnnn } { #1 }
1106       } { #2 }
1107     } { #3 }
1108   } { #4 }
1109 }

```

<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:(xn vn)</code>	<code>__bnvs_n_parse_record:nn {⟨Q?F name⟩} {⟨value⟩}</code>
<code>__bnvs_n_parse_record:n</code>	
<code>__bnvs_n_parse_record:v</code>	
<code>__bnvs_n_parse_record:nn</code>	
<code>__bnvs_n_parse_record:(xn vn)</code>	

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

```

1110 \BNVS_new:cpn { parse_record:n } #1 {
1111   \__bnvs_if_provide:TF {
1112     \__bnvs_gprovide:nnnT V { #1 } { 1 } {
1113       \__bnvs_gclear:n { #1 }
1114     }
1115   } {
1116     \__bnvs_gclear:n { #1 }
1117     \__bnvs_gput:nnn V { #1 } { 1 }
1118   }
1119 }
1120 \BNVS_new:cpn { parse_record:v } {
1121   \BNVS_tl_use:nv {
1122     \__bnvs_parse_record:n
1123   }
1124 }
1125 \BNVS_new:cpn { parse_record:nn } #1 #2 {
1126   \__bnvs_range_if_set:cccnTF { a } { b } { c } { #2 } {
1127     \__bnvs_range:nvvv { #1 } { a } { b } { c }
1128   } {
1129     \__bnvs_if_provide:TF {
1130       \__bnvs_gprovide:nnnT V { #1 } { #2 } {
1131         \__bnvs_gclear_all:n { #1 }
1132       }
1133     } {
1134       \__bnvs_gclear_all:n { #1 }
1135       \__bnvs_gput:nnn V { #1 } { #2 }
1136     }
1137   }
1138 }
1139 \cs_generate_variant:Nn \__bnvs_parse_record:nn { x }
1140 \BNVS_new:cpn { parse_record:vn } {
1141   \BNVS_tl_use:nv {
1142     \__bnvs_parse_record:nn
1143   }
1144 }
1145 \BNVS_new:cpn { n_parse_record:n } #1 {
1146   \bool_if:NTF \l__bnvs_n_provide_bool {
1147     \__bnvs_n_gprovide:nn
1148   } {
1149     \__bnvs_n_gput:nn
1150   }
1151   { #1 } { 1 }

```

```

1152 }
1153 \BNVS_new:cpn { n_parse_record:v } {
1154   \BNVS_tl_use:cv { n_parse_record:n }
1155 }
1156 \BNVS_new:cpn { n_parse_record:nn } #1 #2 {
1157   \__bnvs_range_if_set:ccnTF { a } { b } { c } { #2 } {
1158     \BNVS_error:n { Unexpected~range:~#2 }
1159   } {
1160     \__bnvs_if_provide:TF {
1161       \__bnvs_n_gprovide:nn { #1 } { #2 }
1162     } {
1163       \__bnvs_n_gput:nn { #1 } { #2 }
1164     }
1165   }
1166 }
1167 \BNVS_new:cpn { n_parse_record:vn } {
1168   \BNVS_tl_use:cv { n_parse_record:nn }
1169 }

```

```

\__bnvs_if_id_FQ_name_n_get:nTF \__bnvs_id_name_n_set:nTF {<ref>} {<yes code>} {<no code>}
\__bnvs_if_id_FQ_name_n_get:vTF

```

If *<ref>* is a fully qualified name, put the frame id it defines into *id* and the fully qualified name into *QF_name*, then execute *<yes code>*. The *n tl* variable is empty except when *<ref>* ends with *.n*. Otherwise execute *<no code>*. If *<ref>* is only a qualified name, put it in *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.

```

1170 \BNVS_new_conditional:cpnn { if_id_FQ_name_n_get:n } #1 { T, F, TF } {
1171   \BNVS_begin:
1172     \__bnvs_match_if_once:NnTF \c__bnvs_A_FQ_name_n_Z_regex { #1 } {
1173       \__bnvs_if_match_pop_left:cTF { n } {
1174         \__bnvs_if_match_pop_left:cTF { FQ_name } {
1175           \__bnvs_if_match_pop_left:cTF { id } {
1176             \__bnvs_if_match_pop_left:cTF { n } {
1177               \cs_set:Npn \BNVS:nnn ##1 ##2 ##3 {
1178                 \BNVS_end:
1179                 \__bnvs_tl_set:cn { id } { ##1 }
1180                 \__bnvs_tl_set:cn { FQ_name } { ##2 }
1181                 \__bnvs_tl_set:cn { n } { ##3 }
1182               }
1183               \__bnvs_tl_if_empty:cTF { id } {
1184                 \BNVS_exp_args:Nvvv
1185                 \BNVS:nnn { id_last } { FQ_name } { n }
1186                 \__bnvs_tl_put_left:cv { FQ_name } { id_last }
1187               } {
1188                 \BNVS_exp_args:Nvvv
1189                 \BNVS:nnn { id } { FQ_name } { n }
1190                 \__bnvs_tl_set:cv { id_last } { id }
1191               }
1192             }
1193           } {
1194             \BNVS_end:
1195             \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/n }

```

```

1196         \prg_return_false:
1197     }
1198 } {
1199     \BNVS_end:
1200     \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/id }
1201     \prg_return_false:
1202 }
1203 } {
1204     \BNVS_end:
1205     \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/FQ_name }
1206     \prg_return_false:
1207 }
1208 } {
1209     \BNVS_end:
1210     \BNVS_error:n { LOGICALLY_UNREACHABLE_A_FQ_name_n_Z/n }
1211     \prg_return_false:
1212 }
1213 } {
1214     \BNVS_end:
1215     \prg_return_false:
1216 }
1217 }
1218 \BNVS_new_conditional:cpnn { if_id_FQ_name_n_get:v } #1 { T, F, TF } {
1219     \BNVS_tl_use:nv { \BNVS_use:c { if_id_FQ_name_n_get:nTF } } { #1 } {
1220         \prg_return_true:
1221     } {
1222         \prg_return_false:
1223     }
1224 }

```

__bnvs_parse:n	__bnvs_parse:n {<F/Q name>}
__bnvs_parse:nn	__bnvs_parse:nn {<F/Q name>} {<definition>}

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

\l__bnvs_match_seq Local storage for the match result.

(End of definition for \l__bnvs_match_seq.)

```

1225 \BNVS_new:cpn { parse:n } #1 {
1226     \peek_remove_spaces:n {
1227         \peek_catcode:NTF \c_group_begin_token {
1228             \__bnvs_tl_if_empty:cTF { root } {
1229                 \BNVS_error:n { Unexpected-list-at-top-level. }
1230             } {
1231                 \BNVS_begin:
1232                 \__bnvs_int_incr:c { i }
1233                 \__bnvs_tl_put_right:cx { root } { \__bnvs_int_use:c { i } . }
1234                 \cs_set:Npn \bnvs:w #####1 #####2 \s_stop {
1235                     \regex_match:nnT { \S* } { #####2 } {
1236                         \BNVS_error:n { Unexpected~#####2 }
1237                     }
1238                     \keyval_parse:nnn {

```

```

1239         \__bnvs_parse:n
1240     } {
1241         \__bnvs_parse:nn
1242     } { ###1 }
1243     \BNVS_end:
1244 }
1245 \bnvs:w #1 \s_stop
1246 }
1247 } {
1248     \__bnvs_tl_if_empty:cTF { root } {
1249         \__bnvs_if_id_FQ_name_n_get:nTF { #1 } {
1250             \__bnvs_tl_if_empty:cTF { n } {
1251                 \__bnvs_parse_record:v
1252             } {
1253                 \__bnvs_n_parse_record:v
1254             }
1255             { FQ_name }
1256         } {
1257             \BNVS_error:n { Unexpected-name:~#1 }
1258         }
1259     } {
1260         \__bnvs_int_incr:c { i }
1261         \__bnvs_tl_if_empty:cTF { n } {
1262             \__bnvs_parse_record:xn
1263         } {
1264             \__bnvs_n_parse_record:xn
1265         } {
1266             \__bnvs_tl_use:c { root } \__bnvs_int_use:c { i }
1267         } { #1 }
1268     }
1269 }
1270 }
1271 }
1272 \BNVS_new:cpn { do_range:nnnn } #1 #2 #3 #4 {
1273     \__bnvs_gclear_all:n { #1 }
1274     \tl_if_empty:nTF { #4 } {
1275         \tl_if_empty:nTF { #2 } {
1276             \tl_if_empty:nTF { #3 } {
1277                 \BNVS_error:n { Not-a-range:~:~#1 }
1278             } {
1279                 \__bnvs_gput:nnn Z { #1 } { #3 }
1280                 \__bnvs_gput:nnn V { #1 } { \q_nil }
1281             }
1282         } {
1283             \__bnvs_gput:nnn A { #1 } { #2 }
1284             \__bnvs_gput:nnn V { #1 } { \q_nil }
1285             \tl_if_empty:nF { #3 } {
1286                 \__bnvs_gput:nnn Z { #1 } { #3 }
1287                 \__bnvs_gput:nnn L { #1 } { \q_nil }
1288             }
1289         }
1290     } {
1291         \tl_if_empty:nTF { #2 } {
1292             \__bnvs_gput:nnn L { #1 } { #4 }

```

```

1293     \tl_if_empty:nF { #3 } {
1294         \__bnvs_gput:nnn Z { #1 } { #3 }
1295         \__bnvs_gput:nnn A { #1 } { \q_nil }
1296         \__bnvs_gput:nnn V { #1 } { \q_nil }
1297     }
1298 } {
1299     \__bnvs_gput:nnn A { #1 } { #2 }
1300     \__bnvs_gput:nnn L { #1 } { #4 }
1301     \__bnvs_gput:nnn Z { #1 } { \q_nil }
1302     \__bnvs_gput:nnn V { #1 } { \q_nil }
1303 }
1304 }
1305 }
1306 \cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
1307     \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
1308     { #4 }
1309 }
1310 \cs_new:Npn \BNVS_tl_set_after:ncv #1 #2 {
1311     \BNVS_tl_use:nv {
1312         #1 \__bnvs_tl_set:cn { #2 }
1313     }
1314 }
1315 \cs_new:Npn \BNVS_end_tl_set:cv #1 {
1316     \BNVS_tl_use:nv {
1317         \BNVS_end: \__bnvs_tl_set:cn { #1 }
1318     }
1319 }
1320 \BNVS_new:cpn { parse:nn } #1 #2 {
1321     \BNVS_begin:
1322     \__bnvs_tl_set:cn { a } { #1 }

```

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

```

1323     \__bnvs_tl_put_left:cv { a } { root }
1324     \__bnvs_if_id_FQ_name_n_get:vTF { a } {
1325         \regex_match:nnTF { \S } { #2 } {
1326             \peek_remove_spaces:n {
1327                 \peek_catcode:NTF \c_group_begin_token {

```

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

```

1328         \__bnvs_tl_if_empty:cF { n } {
1329     \__bnvs_warning:n { Ignoring~unexpected~suffix~.n:~#1 }
1330     }

```

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

```

1331     \BNVS_begin:
1332     \__bnvs_gput:nvn V { FQ_name } { \q_nil }
1333     \__bnvs_tl_set:cv { root } { FQ_name }
1334     \__bnvs_tl_put_right:cn { root } { . }
1335     \__bnvs_int_set:cn { i } { 0 }

```

```

1336     \cs_set:Npn \BNVS:w ##1 ##2 \s_stop {
1337       \regex_match:nnT { \S } { ##2 } {
1338         \BNVS_error:n { Unexpected~value~#2 }
1339       }
1340       \keyval_parse:nnn {
1341         \__bnvs_parse:n
1342       } {
1343         \__bnvs_parse:nn
1344       } { ##1 }
1345       \BNVS_end:
1346     }
1347     \BNVS:w
1348   } {

```

Next character is not a group begin token.

```

1349     \__bnvs_tl_if_empty:cTF { n } {
1350       \__bnvs_parse_record:vn
1351     } {
1352       \__bnvs_n_parse_record:vn
1353     }
1354     { FQ_name } { #2 }
1355     \use_none_delimit_by_s_stop:w
1356   }
1357 }
1358 #2 \s_stop
1359 } {

```

Empty value given: remove the reference.

```

1360     \__bnvs_tl_if_empty:cTF { n } {
1361       \__bnvs_gclear:v
1362     } {
1363       \__bnvs_n_gremove:v
1364     }
1365     { FQ_name }
1366   }
1367 } {
1368   \BNVS_error:n { Invalid~name:~#2 }
1369 }

```

We export \l__bnvs_id_last_tl:

```

1370   \BNVS_end_tl_set:cv { id_last } { id_last }
1371 }

1372 \BNVS_new:cpn { parse_prepare:N } #1 {
1373   \tl_set:Nx #1 #1
1374   \bool_set_false:N \l__bnvs_parse_bool
1375   \bool_do_until:Nn \l__bnvs_parse_bool {
1376     \tl_if_in:NnTF #1 {%---[
1377   ]} {
1378     \regex_replace_all:nnNF { \[ ([^\]]%---)
1379   ]*%---[(
1380   ) \] } { { { \1 } } } #1 {
1381     \bool_set_true:N \l__bnvs_parse_bool
1382   }
1383 } {

```

```

1384     \bool_set_true:N \l__bnvs_parse_bool
1385   }
1386 }
1387 \tl_if_in:NnTF #1 {%---[
1388 ]] {
1389   \BNVS_error:n { Unbalanced~%---[
1390   ]}
1391 } {
1392   \tl_if_in:NnT #1 { [%---]
1393   } {
1394     \BNVS_error:n { Unbalanced~[ %---]
1395     }
1396   }
1397 }
1398 }

```

\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.

```

1399 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1400   \BNVS_tl_use:nv {
1401     \BNVS_end:
1402     \__bnvs_tl_put_right:cn { #1 }
1403   } { #2 }
1404 }
1405 \cs_new:Npn \BNVS_end_v_gput:nv #1 {
1406   \BNVS_tl_use:nv {
1407     \BNVS_end:
1408     \__bnvs_v_gput:nn { #1 }
1409   }
1410 }
1411 \NewDocumentCommand \Beanoves { sm } {
1412   \tl_if_empty:NTF \@currentenvir {

```

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

```

1413   \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
1414 } {
1415   \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, clear everything.

```

1416   \__bnvs_gclear:
1417 }
1418 \BNVS_begin:
1419 \__bnvs_tl_clear:c { root }
1420 \__bnvs_int_zero:c { i }
1421 \__bnvs_tl_set:cn { a } { #2 }
1422 \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, use the global definitions.


```

1423     \seq_if_empty:NF \g__bnvs_def_seq {
1424         \__bnvs_tl_put_left:cx { a } {
1425             \seq_use:Nn \g__bnvs_def_seq , ,
1426         }
1427     }
1428 }
1429 \__bnvs_parse_prepare:N \l__bnvs_a_tl
1430 \IfBooleanTF {#1} {
1431     \__bnvs_provide_on:
1432 } {
1433     \__bnvs_provide_off:
1434 }
1435 \BNVS_tl_use:nv {
1436     \keyval_parse:nmn { \__bnvs_parse:n } { \__bnvs_parse:nn }
1437 } { a }
1438 \BNVS_end_tl_set:cv { id_last } { id_last }
1439 \ignorespaces
1440 }
1441 }

```

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

```

1442 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}

```

6.19 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.

```

1443 \makeatletter
1444 \cs_set:Npn \__bnvs@frame < #1 > {
1445     \BNVS_begin:
1446     \__bnvs_tl_clear:c { ans }
1447     \__bnvs_scan:nNc { #1 } \__bnvs_if_resolve:ncTF { ans }
1448     \BNVS_set:cpn { :n } ##1 { \BNVS_end: \__bnvs_saved@frame < ##1 > }
1449     \BNVS_tl_use:cv { :n } { ans }
1450 }
1451 \cs_set:Npn \__bnvs@masterdecode #1 {
1452     \BNVS_begin:
1453     \__bnvs_tl_clear:c { ans }
1454     \__bnvs_scan:nNc { #1 } \__bnvs_if_resolve_queries:ncTF { ans }

```

```

1455 \BNVS_tl_use:nv {
1456   \BNVS_end:
1457   \__bnvs_saved@masterdecode
1458 } { ans }
1459 }
1460 \cs_new:Npn \BeanovesOff {
1461   \cs_set_eq:NN \beamer@frame \__bnvs_saved@frame
1462   \cs_set_eq:NN \beamer@masterdecode \__bnvs_saved@masterdecode
1463 }
1464 \cs_new:Npn \BeanovesOn {
1465   \cs_set_eq:NN \beamer@frame \__bnvs@frame
1466   \cs_set_eq:NN \beamer@masterdecode \__bnvs@masterdecode
1467 }
1468 \AddToHook{begindocument/before}{
1469   \cs_if_exist:NTF \beamer@frame {
1470     \cs_set_eq:NN \__bnvs_saved@frame \beamer@frame
1471     \cs_set_eq:NN \__bnvs_saved@masterdecode \beamer@masterdecode
1472   } {
1473     \cs_set:Npn \__bnvs_saved@frame < #1 > {
1474       \BNVS_error:n {Missing-package-beamer}
1475     }
1476     \cs_set:Npn \__bnvs_saved@masterdecode < #1 > {
1477       \BNVS_error:n {Missing-package-beamer}
1478     }
1479   }
1480   \BeanovesOn
1481 }
1482 \makeatother

```

__bnvs_scan:nNc __bnvs_scan:nNc {*overlay query*} {*resolve*} {*ans*}

Scan the *overlay query* argument and feed the *ans* `tl` variable replacing `?(...)` instructions by their static counterpart with help from the *resolve* function, which is `__bnvs_if_resolve:nCTF`. A group is created to use local variables:

\l__bnvs_ans_tl The token list that will be appended to *tl variable* on return.

(End of definition for `\l__bnvs_ans_tl`.)

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

(End of definition for `\l__bnvs_int`.)

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

(End of definition for `\l__bnvs_query_tl`.)

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

(End of definition for `\l__bnvs_token_seq`.)

\l__bnvs_token_tl Storage for just one token.

(End of definition for `\l__bnvs_token_tl`.)

```

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

```

Next are helpers.

```

__bnvs_scan_for_query_then_end_return: __bnvs_scan_for_query_then_end_return:

```

At top level state, scan the tokens of the *<named overlay expression>* looking for a ‘?’ character. If a ‘?(...)’ is found, then the *<code>* is executed.

```

1483 \BNVS_new:cpn { scan_for_query_then_end_return: } {
1484   __bnvs_seq_pop_left:ccTF { token } { token } {
1485     __bnvs_tl_if_eq:cnTF { token } { ? } {
1486       __bnvs_scan_require_open_then_end_return:
1487     } {
1488       __bnvs_tl_put_right:cv { ans } { token }
1489       __bnvs_scan_for_query_then_end_return:
1490     }
1491   } {
1492     __bnvs_scan_end_return_true:
1493   }
1494 }

```

```

__bnvs_scan_require_open_then_end_return: __bnvs_scan_require_open_then_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.

```

1495 \BNVS_new:cpn { scan_require_open_then_end_return: } {

```

Get next token.

```

1496   __bnvs_seq_pop_left:ccTF { token } { token } {
1497     \str_if_eq:VnTF \l__bnvs_token_tl { ( % )
1498   } {

```

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

```

1499     __bnvs_int_set:cn { } { 1 }

```

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

```

1500     __bnvs_tl_clear:c { query }
1501     __bnvs_scan_require_close_and_return:
1502   } {

```

Ignore this token and loop.

```

1503     __bnvs_scan_require_open_then_end_return:
1504   }
1505 } {

```

Get next token.

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

```

1506     \BNVS_error:x {Missing~'('%---)
1507     ~after~a~? }
1508     \__bnvs_scan_end_return_true:
1509 }
1510 }

```

```

\__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 ‘)’.

```

1511 \BNVS_new:cpn { scan_require_close_and_return: } {

```

Get next token.

```

1512 \__bnvs_seq_pop_left:ccTF { token } { token } {
1513   \str_case:VnF \l__bnvs_token_tl {
1514     { ( %---)
1515   } {

```

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

```

1516     \__bnvs_int_incr:c { }
1517     \__bnvs_tl_put_right:cv { query } { token }
1518     \__bnvs_scan_require_close_and_return:
1519   }
1520   { %(---
1521     )
1522   } {

```

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

```

1523     \__bnvs_int_decr:c { }
1524     \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 ?.

```

1525     \__bnvs_scan_handle_query_then_end_return:
1526   } {

```

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

```

1527     \__bnvs_tl_put_right:cv { query } { token }
1528     \__bnvs_scan_require_close_and_return:
1529   }
1530   }
1531   } {

```

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

```

1532     \__bnvs_tl_put_right:cv { query } { token }
1533     \__bnvs_scan_require_close_and_return:
1534   }
1535   } {

```

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 ‘)’:

```

1536     \BNVS_error:x { Missing-%(---
1537     `)' }
1538     \__bnvs_tl_put_right:cx { query } {
1539         \prg_replicate:nn { \l__bnvs_int } {%(---
1540         )}
1541     }
1542     \__bnvs_scan_end_return_true:
1543 }
1544 }

1545 \BNVS_new_conditional:cpnn { scan:nNc } #1 #2 #3 { T, F, TF } {
1546     \BNVS_begin:
1547     \BNVS_set:cpn { error:x } ##1 {
1548         \msg_error:nnx { beanoves } { :n }
1549         { \tl_to_str:n { #1 } :~##1}
1550     }
1551     \__bnvs_tl_set:cn { scan } { #1 }
1552     \__bnvs_tl_clear:c { ans }
1553     \__bnvs_seq_clear:c { token }

```

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

```

1554     \regex_split:nnN { } { #1 } \l__bnvs_token_seq

```

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

```

1555     \BNVS_set:cpn { scan_handle_query_then_end_return: } {
1556         \BNVS_tl_use:Nv #2 { query } { ans } {
1557             \__bnvs_scan_for_query_then_end_return:
1558         } {
1559             \BNVS_end_tl_put_right:cv { #3 } { ans }

```

Stop on the first error.

```

1560         \prg_return_false:
1561     }
1562 }
1563 \BNVS_set:cpn { scan_end_return_true: } {
1564     \BNVS_end_tl_put_right:cv { #3 } { ans }
1565     \prg_return_true:
1566 }
1567 \BNVS_set:cpn { scan_end_return_false: } {
1568     \BNVS_end_tl_put_right:cv { #3 } { ans }
1569     \prg_return_false:
1570 }
1571 \__bnvs_scan_for_query_then_end_return:
1572 }
1573 \BNVS_new:cpn { scan:nNc } #1 #2 #3 {
1574     \BNVS_use:c { scan:nNcTF } { #1 } #2 { #3 } {} {}
1575 }

```

6.20 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_Fip:cccTF __bnvs_if_Fip:cccTF {<FQ name>} {<id>} {<path>} {<yes code>} {<no code>}
```

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

```
1576 \BNVS_new_conditional:cpnn { if_Fip:ccc } #1 #2 #3 { T, F, TF } {
1577   \BNVS_begin:
1578   __bnvs_match_if_once:NvTF \c__bnvs_A_FQ_name_Z_regex { #1 } {
```

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

```
1579   __bnvs_if_match_pop_Fip:cccTF { #1 } { #2 } { #3 } {
1580     __bnvs_end_Fip_export:ccc { #1 } { #2 } { #3 }
1581     \prg_return_true:
1582   } {
1583     \BNVS_end:
1584     \prg_return_false:
1585   }
1586 } {
1587   \BNVS_end:
1588   \prg_return_false:
1589 }
1590 }
1591 \quark_new:N \q__bnvs
1592 \BNVS_new:cpn { end_Fip_export:ccc } #1 #2 #3 {
1593   \exp_args:Nnx
1594   \use:n {
1595     \BNVS_tl_use:nv {
1596       \BNVS_tl_use:cv { end_Fip_export:nnnccc } { #1 }
1597     } { #2 }
1598   } { __bnvs_seq_use:cn { #3 } { \q__bnvs } } { #1 } { #2 } { #3 }
1599 }
1600 \BNVS_new:cpn { end_Fip_export:nnnccc } #1 #2 #3 #4 #5 #6 {
1601   \BNVS_end:
1602   \tl_if_empty:nTF { #2 } {
1603     __bnvs_tl_set:cn { #4 } { #1 }
1604     __bnvs_tl_put_left:cv { #4 } { #5 }
1605   } {
1606     __bnvs_tl_set:cn { #4 } { #1 }
1607     __bnvs_tl_set:cn { #5 } { #2 }
1608   }
1609   __bnvs_seq_set_split:cn { #6 } { \q__bnvs } { #3 }
1610   __bnvs_seq_remove_all:cn { #6 } { }
1611 }
```

Sets the `FQ_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.

```

1612 \BNVS_new_conditional:cpnn { if_match_pop_Fip:ccc } #1 #2 #3 { TF } {
1613   \_bnvs_if_match_pop_left:cTF { #1 } {
1614     \_bnvs_if_match_pop_left:cTF { #1 } {
1615       \_bnvs_if_match_pop_left:cTF { #2 } {
1616         \_bnvs_if_match_pop_left:cTF { #3 } {
1617           \_bnvs_seq_set_split:cnv { #3 } { . } { #3 }
1618           \_bnvs_seq_remove_all:cn { #3 } { }
1619           \prg_return_true:
1620         } {
1621           \prg_return_false:
1622         }
1623       } {
1624         \prg_return_false:
1625       }
1626     } {
1627       \prg_return_false:
1628     }
1629   } {
1630     \prg_return_false:
1631   }
1632 }

```

<code>_bnvs_if_resolve_Fip_n:TF</code>	<code>_bnvs_if_resolve_Fip_n:TF {<yes code>} {<no code>}</code>
<code>_bnvs_if_resolve_Fip:TF</code>	<code>_bnvs_if_resolve_Fip:TF {<yes code>} {<no code>}</code>
<code>_bnvs_if_resolve_Fip_x_path:TF</code>	<code>_bnvs_if_resolve_Fip_x_path:TF {<yes code>} {<no code>}</code>

{<yes code>} will be executed once resolution has occurred, {<no code>} otherwise. The key and id variables as well as the `path` sequence are meant to contain proper information on input and on output as well. On input, `\l__bnvs_FQ_name_tl` contains an overlay set name, `\l__bnvs_id_tl` contains a frame id and `\l__bnvs_path_seq` contains the components of a dotted path, possibly empty. On return, the variable `\l__bnvs_FQ_name_tl` contains the resolved range name, `\l__bnvs_id_tl` contains the frame id used and `\l__bnvs_path_seq` contains the list of path components that could not be resolved.

To resolve one level of a named one slide specification like $\langle name \rangle . \langle c_1 \rangle \dots \langle c_j \rangle$, we replace the longest $\langle name \rangle . \langle c_1 \rangle \dots \langle c_k \rangle$ where $0 \leq k \leq n$ by its definition $\langle name' \rangle . \langle c'_1 \rangle \dots \langle c'_l \rangle$ if any.

1. If `\l__bnvs_FQ_name_tl` content is the name of an unlimited set, and the first item of this range is exactly another name range with eventually a heading frame identifier or a trailing dotted path, then `\l__bnvs_FQ_name_tl` is replaced by this name, the `\l__bnvs_id_tl` and `\l__bnvs_id_tl` are updated accordingly and the $\langle path \ seq \ var \rangle$ is prepended with the dotted path.
2. If $\langle path \ seq \ var \rangle$ is not empty, append to the right of `\l__bnvs_FQ_name_tl` after a separating dot, all its left elements but the last one and loop. Otherwise return.

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

In the `_x` variant, the resolution is driven one step further: if $\langle path \ seq \ var \rangle$ is empty, $\langle name \ tl \ var \rangle$ can contain anything, including an integer for example.

```

__bnvs_if_resolve_Fip_x_path:TFF __bnvs_if_resolve_Fip_x_path:TFF {<yes code>} {<no code 1>} {<no
code 2>}

```

```

1633 \BNVS_new:cpn { if_resolve_Fip_x_path:TFF } #1 #2 {
1634   __bnvs_if_resolve_Fip_x_path:TF {
1635     __bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
1636   }
1637 }

```

Local variables:

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

```

1638 \BNVS_new:cpn { end_Fip_export: } {
1639   __bnvs_end_Fip_export:ccc { FQ_name } { id } { path }
1640 }
1641 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
1642   __bnvs_seq_if_empty:cF { #2 } {
1643     __bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
1644       __bnvs_seq_use:cn { #1 } { \q__bnvs }
1645       \exp_not:n { \q__bnvs }
1646       __bnvs_seq_use:cn { #2 } { \q__bnvs }
1647     }
1648     __bnvs_seq_remove_all:cn { #1 } { }
1649   }
1650 }
1651 \BNVS_new:cpn { if_resolve_Fip_x_path:nFF } #1 #2 #3 {
1652   __bnvs_if_get:nvcTF #1 { a } { b } {
1653     __bnvs_if_Fip:cccTF { b } { id } { path } {
1654       __bnvs_tl_set_eq:cc { FQ_name } { b }
1655       __bnvs_seq_merge:cc { path } { path_tail }
1656       __bnvs_seq_clear:c { path_tail }
1657       __bnvs_seq_set_eq:cc { path_head } { path }
1658       __bnvs_if_resolve_Fip_x_path_loop_or_end_return:
1659     } {
1660       __bnvs_seq_if_empty:cTF { path_tail } {
1661         __bnvs_tl_set_eq:cc { FQ_name } { b }
1662         __bnvs_seq_clear:c { path }
1663         __bnvs_seq_clear:c { path_head }
1664         __bnvs_if_resolve_Fip_x_path_loop_or_end_return:
1665       } {
1666         #2
1667       }
1668     }
1669   } {
1670     #3
1671   }
1672 }

```



```

1673 \BNVS_new:cpn { if_resolve_Fip_x_path_VAL_loop_or_end_return:F } #1 {
1674   \__bnvs_if_resolve_Fip_x_path:nFF V { #1 } {
1675     \__bnvs_if_resolve_Fip_x_path:nFF A { #1 } {
1676       \__bnvs_if_resolve_Fip_x_path:nFF L { #1 } { #1 }
1677     }
1678   }
1679 }
1680 \BNVS_new:cpn { if_resolve_Fip_x_path_end_return_true: } {
1681   \__bnvs_seq_pop_left:ccTF { path } { a } {
1682     \__bnvs_seq_if_empty:cTF { path } {
1683       \__bnvs_tl_clear:c { b }
1684       \__bnvs_index_can:vTF { FQ_name } {
1685         \__bnvs_if_append_index:vcTF { FQ_name } { a } { b } {
1686           \__bnvs_tl_set:cv { FQ_name } { b }
1687         } {
1688           \__bnvs_tl_set:cv { FQ_name } { a }
1689         }
1690       } {
1691         \__bnvs_tl_set:cv { FQ_name } { a }
1692       }
1693     } {
1694       \BNVS_error:x { Path~too~long~.\BNVS_tl_use:c { a }
1695         .\__bnvs_seq_use:cn { path } . }
1696     }
1697   } {
1698     \__bnvs_if_resolve_V:vcT { FQ_name } { FQ_name } {}
1699   }
1700   \__bnvs_end_Fip_export:ccc { FQ_name } { id } { path }
1701   \prg_return_true:
1702 }
1703 \BNVS_new_conditional:cpnn { if_resolve_Fip_x_path: } { T, F, TF } {
1704   \BNVS_begin:
1705     \__bnvs_seq_set_eq:cc { path_head } { path }
1706     \__bnvs_seq_clear:c { path_tail }
1707     \__bnvs_if_resolve_Fip_x_path_loop_or_end_return:
1708 }
1709 \BNVS_new:cpn { if_resolve_Fip_x_path_loop_or_end_return: } {
1710   \__bnvs_if_call:TF {
1711     \__bnvs_tl_set_eq:cc { a } { FQ_name }
1712     \__bnvs_seq_if_empty:cTF { path_head } {
1713       \__bnvs_if_resolve_Fip_x_path_VAL_loop_or_end_return:F {
1714         \__bnvs_if_resolve_Fip_x_path_end_return_true:
1715       }
1716     } {
1717       \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { path_head } . }
1718       \__bnvs_if_resolve_Fip_x_path_VAL_loop_or_end_return:F {
1719         \__bnvs_seq_pop_right:ccT { path_head } { c } {
1720           \__bnvs_seq_put_left:cv { path_tail } { c }
1721         }
1722       }
1723     }
1724   }
1725 } {
1726   \BNVS_end:

```

```

1727     \prg_return_false:
1728   }
1729 }

1730 \BNVS_new:cpn { if_resolve_Fip_or_end_return:nTF } #1 #2 #3 {
1731   \__bnvs_if_get:nvcTF { #1 } { a } { b } {

```

The `a` `tl` variable is known, its value is in `b`. We check if it is exactly an overlay set name. If true, the new `FQ_name` is `b`.

```

1732   \__bnvs_if_Fip:cccTF { b } { id } { path } {
1733     \__bnvs_tl_set_eq:cc { FQ_name } { b }
1734     \__bnvs_seq_merge:cc { path } { path_tail }
1735     \__bnvs_seq_set_eq:cc { path_head } { path }
1736     \__bnvs_seq_clear:c { path_tail }
1737     \__bnvs_if_resolve_Fip_loop_or_end_return:
1738   } {
1739     \__bnvs_seq_pop_right:ccTF { path_head } { b } {
1740       \__bnvs_seq_put_left:cv { path_tail } { b }
1741       \__bnvs_if_resolve_Fip_loop_or_end_return:
1742     } {
1743       \__bnvs_if_resolve_Fip_end_return_true:
1744     }
1745   }
1746 } {
1747   #3
1748 }
1749 }

1750 \BNVS_new:cpn { if_resolve_Fip_or_end_return:nF } #1 #2 {
1751   \__bnvs_if_get:nvcTF { #1 } { a } { b } {

```

The `a` `tl` variable is known, its value is in `b`. We check if it is exactly an overlay set name. If true, the new `FQ_name` is `b`.

```

1752   \__bnvs_if_Fip:cccTF { b } { id } { path } {
1753     \__bnvs_tl_set_eq:cc { FQ_name } { b }
1754     \__bnvs_seq_merge:cc { path } { path_tail }
1755     \__bnvs_seq_set_eq:cc { path_head } { path }
1756     \__bnvs_seq_clear:c { path_tail }
1757     \__bnvs_if_resolve_Fip_loop_or_end_return:
1758   } {
1759     \__bnvs_seq_pop_right:ccTF { path_head } { b } {
1760       \__bnvs_seq_put_left:cv { path_tail } { b }
1761       \__bnvs_if_resolve_Fip_loop_or_end_return:
1762     } {
1763       \__bnvs_if_resolve_Fip_end_return_true:
1764     }
1765   }
1766 } {
1767   #2
1768 }
1769 }

1770 \BNVS_new:cpn { if_resolve_Fip_n_or_end_return:nF } #1 #2 {
1771   \__bnvs_if_get:nvcTF { #1 } { a } { b } {

```

The **a** **tl** variable is known, its value is in **b**. We check if it is exactly an overlay set name. If true, the new **FQ_name** is **b**.

```

1772     \__bnvs_if_Fip:cccTF { b } { id } { path } {
1773         \__bnvs_tl_set_eq:cc { FQ_name } { b }
1774         \__bnvs_seq_merge:cc { path } { path_tail }
1775         \__bnvs_seq_set_eq:cc { path_head } { path }
1776         \__bnvs_seq_clear:c { path_tail }
1777         \__bnvs_if_resolve_Fip_n_loop_or_end_return:
1778     } {
1779         \__bnvs_seq_pop_right:ccTF { path_head } { c } {
1780             \__bnvs_seq_put_left:cv { path_tail } { c }
1781             \__bnvs_if_resolve_Fip_n_loop_or_end_return:
1782         } {
1783             \__bnvs_if_resolve_Fip_end_return_true:
1784         }
1785     }
1786 } {
1787     #2
1788 }
1789 }

```

The **b** sequence is not empty, the **a** sequence may be.

```

1790 \BNVS_new:cpn { if_resolve_Fip_VAL_loop_or_end_return: } {
1791     \__bnvs_if_resolve_Fip_end_return_or:T {
1792         \__bnvs_seq_pop_right:ccTF { path_head } { c } {

```

Move the rightmost dotted path component of **a** to the left of **b**. Then loop.

```

1793         \__bnvs_seq_put_left:cv { path_tail } { c }
1794         \__bnvs_if_resolve_Fip_loop_or_end_return:
1795     } {

```

The **a** sequence is empty.

```

1796         \__bnvs_if_resolve_Fip_end_return_true:
1797     }
1798 }
1799 }

```

```

1800 \BNVS_new:cpn { if_resolve_Fip_n_VAL_loop_or_end_return: } {
1801     \__bnvs_if_resolve_Fip_n_or_end_return:nF V {
1802         \__bnvs_if_resolve_Fip_n_or_end_return:nF A {
1803             \__bnvs_if_resolve_Fip_n_or_end_return:nF L {
1804                 \__bnvs_seq_pop_right:ccTF { path_head } { c } {

```

Move the rightmost dotted path component of **a** to the left of **b**. Then loop.

```

1805                 \__bnvs_seq_put_left:cv { path_tail } { c }
1806                 \__bnvs_if_resolve_Fip_n_loop_or_end_return:
1807             } {

```

The **a** sequence is empty.

```

1808                 \__bnvs_if_resolve_Fip_end_return_true:
1809             }
1810         }
1811     }
1812 }
1813 }
1814 \BNVS_new:cpn { if_resolve_Fip_end_return_false: } {
1815     \BNVS_end:

```

```

1816 \prg_return_false:
1817 }
1818 \BNVS_new:cpn { if_resolve_Fip_end_return_true: } {
1819   \__bnvs_end_Fip_export:ccc { FQ_name } { id } { path }
1820   \prg_return_true:
1821 }

```

__bnvs_if_resolve_Fip_n_loop_or_end_return:

Loop body to resolve the path.

```

1822 \BNVS_new:cpn { if_resolve_Fip_loop_or_end_return: } {
1823   \__bnvs_if_call:TF {

```

Copy FQ_name to a.

```

1824   \__bnvs_tl_set_eq:cc { a } { FQ_name }
1825   \__bnvs_seq_if_empty:cTF { path_head } {
1826     \__bnvs_seq_if_empty:cTF { path_tail } {

```

path_head and path_tail sequences are empty. The path is resolved, we return immediately.

```

1827     \__bnvs_if_resolve_Fip_end_return_true:
1828   } {

```

a sequence is empty, b sequence is not.

```

1829     \__bnvs_if_resolve_Fip_VAL_loop_or_end_return:
1830   }
1831 } {

```

a, b sequences are not empty. Append the a sequence to the a tl variable as dotted path.

```

1832   \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { path_head } . }
1833   \__bnvs_if_resolve_Fip_VAL_loop_or_end_return:
1834 }
1835 } {
1836   \BNVS_end:
1837   \prg_return_false:
1838 }
1839 }

```

```

1840 \BNVS_new:cpn { if_resolve_Fip_n_loop_or_end_return: } {
1841   \__bnvs_if_call:TF {

```

Copy FQ_name to a.

```

1842   \__bnvs_tl_set_eq:cc { a } { FQ_name }
1843   \__bnvs_seq_if_empty:cTF { path_head } {
1844     \__bnvs_seq_if_empty:cTF { path_tail } {

```

path_head and path_tail sequences are empty. The path is resolved, we return immediately.

```

1845     \__bnvs_if_resolve_Fip_end_return_true:
1846   } {

```

a sequence is empty, b sequence is not.

```

1847         \__bnvs_if_resolve_Fip_n_VAL_loop_or_end_return:
1848     }
1849 } {

```

a, b sequences are not empty. Append the a sequence to the a tl variable as dotted path.

```

1850         \__bnvs_tl_put_right:cx { a } { . \__bnvs_seq_use:cn { path_head } . }
1851         \__bnvs_if_resolve_Fip_n_VAL_loop_or_end_return:
1852     }
1853 } {
1854     \BNVS_end:
1855     \prg_return_false:
1856 }
1857 }

```

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

- FQ_name, id, path path sequence contain the resolution.
 - ...a_tl contains the name with a partial dotted path currently resolved.
 - \...head_seq contains the dotted path components to be resolved. It equals \...path_seq at the beginning
 - \...tail_seq is used as well. Initially empty.
- Used by ...if_resolve_path_n:TFF.

```

1858 \BNVS_new_conditional:cpnn { if_resolve_Fip: } { TF } {
1859     \BNVS_begin:

```

Initialize a to path sequence, clears the b sequence.

```

1860     \__bnvs_seq_set_eq:cc { path_head } { path }
1861     \__bnvs_seq_clear:c { path_tail }
1862     \__bnvs_tl_clear:c { a }
1863     \__bnvs_if_resolve_Fip_loop_or_end_return:
1864 }

```

```

1865 \BNVS_new_conditional:cpnn { if_resolve_Fip_n: } { TF } {
1866     \BNVS_begin:

```

Initialize a to path sequence, clears the b sequence.

```

1867     \__bnvs_seq_set_eq:cc { path_head } { path }
1868     \__bnvs_seq_clear:c { path_tail }
1869     \__bnvs_tl_clear:c { a }
1870     \__bnvs_if_resolve_Fip_n_loop_or_end_return:
1871 }

```

6.21 Evaluation bricks

We start by helpers.

```

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

```

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

```

1872 \BNVS_new:cpn { round:ans:n } #1 {
1873   \tl_if_empty:nTF { #1 } {
1874     \__bnvs_tl_put_right:cn { ans } { 0 }
1875   } {
1876     \__bnvs_tl_put_right:cx { ans } { \fp_eval:n { round(#1) } }
1877   }
1878 }
1879 \BNVS_new:cpn { round:N } #1 {
1880   \tl_if_empty:NTF #1 {
1881     \tl_set:Nn #1 { 0 }
1882   } {
1883     \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
1884   }
1885 }
1886 \BNVS_new:cpn { round:c } {
1887   \BNVS_tl_use:Nc \__bnvs_round:N
1888 }

```

```

\BNVS_end_return_false:  \BNVS_end_return_false:x  \__bnvs_end_return_false:
\__bnvs_end_return_false:x {<message>}

```

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

```

1889 \cs_new:Npn \BNVS_end_return_false: {
1890   \BNVS_end:
1891   \prg_return_false:
1892 }
1893 \cs_new:Npn \BNVS_end_return_false:x #1 {
1894   \BNVS_error:x { #1 }
1895   \BNVS_end_return_false:
1896 }

```

```

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

```

```

1897 \BNVS_new_conditional:cpnn { if_assign_value:nn } #1 #2 { T, F, TF } {
1898   \BNVS_begin:

```

```

1899 \__bnvs_if_resolve:ncTF { #2 } { a } {
1900   \__bnvs_gclear_all:n { #1 }
1901   \__bnvs_gput:nnv V { #1 } { a }
1902   \BNVS_end:
1903   \prg_return_true:
1904 } {
1905   \BNVS_end:
1906   \prg_return_false:
1907 }
1908 }
1909 \BNVS_new_conditional:cpnn { if_assign_value:nv } #1 #2 { T, F, TF } {
1910   \BNVS_tl_use:nv {
1911     \BNVS_use:c { if_assign_value:nnTF } { #1 }
1912   } { #2 } {
1913     \prg_return_true:
1914   } {
1915     \prg_return_false:
1916   }
1917 }
1918 \BNVS_new_conditional:cpnn { if_assign_value:vv } #1 #2 { T, F, TF } {
1919   \BNVS_tl_use:nv {
1920     \BNVS_tl_use:cv { if_assign_value:nnTF } { #1 }
1921   } { #2 } {
1922     \prg_return_true:
1923   } {
1924     \prg_return_false:
1925   }
1926 }

```

<u>__bnvs_if_resolve_V:ncTF</u>	__bnvs_if_resolve_V:ncTF { $\langle FQ_n ame \rangle$ } $\langle ans \rangle$ { $\langle yes code \rangle$ } { $\langle no code \rangle$ }
<u>__bnvs_if_resolve_V:vcTF</u>	__bnvs_if_append_V:ncTF { $\langle FQ_n ame \rangle$ } $\langle ans \rangle$ { $\langle yes code \rangle$ } { $\langle no code \rangle$ }
<u>__bnvs_if_append_V:ncTF</u>	
<u>__bnvs_if_append_V:(xc vc)TF</u>	

Resolve the content of the $\langle FQ_n ame \rangle$ value counter into the $\langle ans \rangle$ t1 variable or append this value to the right of the variable. Execute $\langle yes code \rangle$ when there is a $\langle value \rangle$, $\langle no code \rangle$ otherwise. Inside the $\langle no code \rangle$ branch, the content of the $\langle ans \rangle$ t1 variable is undefined. Implementation detail: in $\langle ans \rangle$ we return the first in the cache for subkey V and in the general prop for subkey V (once resolved). Once we have found a value, we feed the previous items such that the next search stops at the first item. The cache contains an integer which is the computed value from the general prop. A local group is created while appending but not while resolving.

```

1927 \BNVS_new:cpn { if_resolve_V_return:nnnT } #1 #2 #3 #4 {
1928   \__bnvs_tl_if_empty:cTF { #3 } {
1929     \prg_return_false:
1930   } {
1931     \__bnvs_cache_gput:nnv V { #2 } { #3 }
1932     #4
1933     \prg_return_true:
1934   }
1935 }

```

```

1936 \BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
1937   \BNVS_tl_use:Nc \quark_if_nil:NTF { #1 } {
1938     \prg_return_true:
1939   } {
1940     \prg_return_false:
1941   }
1942 }
1943 \BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
1944   \BNVS_tl_use:Nc \quark_if_no_value:NTF { #1 } {
1945     \prg_return_true:
1946   } {
1947     \prg_return_false:
1948   }
1949 }
1950 \makeatletter
1951 \BNVS_new_conditional:cpnn { if_resolve_V:nc } #1 #2 { T, F, TF } {
1952   \__bnvs_cache_if_get:nncTF V { #1 } { #2 } {
1953     \prg_return_true:
1954   } {
1955     \__bnvs_if_get:nncTF V { #1 } { #2 } {
1956       \__bnvs_quark_if_nil:cTF { #2 } {

```

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

```

1957   \__bnvs_gput:nnn V { #1 } { \q_no_value }
1958   \__bnvs_if_resolve_first:ncTF { #1 } { #2 } {
1959     \__bnvs_if_resolve_V_return:nnnT A { #1 } { #2 } {
1960       \__bnvs_gput:nnn V { #1 } { \q_nil }
1961     }
1962   } {
1963     \__bnvs_if_resolve_last:ncTF { #1 } { #2 } {
1964       \__bnvs_if_resolve_V_return:nnnT Z { #1 } { #2 } {
1965         \__bnvs_gput:nnn V { #1 } { \q_nil }
1966       }
1967     } {
1968       \__bnvs_gput:nnn V { #1 } { \q_nil }
1969       \prg_return_false:
1970     }
1971   }
1972 } {
1973   \__bnvs_quark_if_no_value:cTF { #2 } {
1974     \BNVS_fatal:n {Circular~definition:~#1}
1975   } {

```

Possible recursive call.

```

1976   \__bnvs_if_resolve:vcTF { #2 } { #2 } {
1977     \__bnvs_if_resolve_V_return:nnnT V { #1 } { #2 } {
1978       \__bnvs_gput:nnn V { #1 } { \q_nil }
1979     }
1980   } {
1981     \__bnvs_gput:nnn V { #1 } { \q_nil }
1982     \prg_return_false:
1983   }
1984 }
1985 }

```



```

1986   } {
1987     \str_if_eq:nnTF { #1 } { ?!pauses } {
1988       \cs_if_exist:NTF \c@beamerpauses {
1989         \exp_args:Nnx \__bnvs_tl_set:cn { #2 } { \the\c@beamerpauses }
1990         \prg_return_true:
1991       } {
1992         \prg_return_false:
1993       }
1994     } {
1995       \prg_return_false:
1996     }
1997   }
1998 }
1999 }
2000 \makeatother
2001 \BNVS_new_conditional_vc:cn { if_resolve_V } { T, F, TF }
2002 \BNVS_new_cpn { end_put_right:vc } #1 #2 {
2003   \BNVS_tl_use:nv {
2004     \BNVS_end:
2005     \__bnvs_tl_put_right:cn { #2 }
2006   } { #1 }
2007 }
2008 \BNVS_new_conditional_cpnn { if_append_V:nc } #1 #2 { T, F, TF } {
2009   \BNVS_begin:
2010   \__bnvs_if_resolve_V:ncTF { #1 } { #2 } {
2011     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2012     \prg_return_true:
2013   } {
2014     \BNVS_end:
2015     \prg_return_true:
2016   }
2017 }
2018 \BNVS_new_conditional_vc:cn { if_append_V } { T, F, TF }

```

```

\__bnvs_if_resolve_first:ncTF      \__bnvs_if_resolve_first:ncTF {<FQ name>} <tl core> {<yes code>} {<no
\__bnvs_if_resolve_first:(xc|vc)TF code>}
\__bnvs_if_append_first:ncTF      \__bnvs_if_append_first:ncTF {<FQ name>} <tl core> {<yes code>} {<no
\__bnvs_if_append_first:(xc|vc)TF code>}

```

Resolve the first index of the $\langle FQ \text{ name} \rangle$ slide range into the $\langle tl \text{ variable} \rangle$ or append the first index of the $\langle FQ \text{ name} \rangle$ 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.

```

2019 \BNVS_new_conditional_cpnn { if_resolve_first:nc } #1 #2 { T, F, TF } {
2020   \__bnvs_cache_if_get:nncTF A { #1 } { #2 } {
2021     \prg_return_true:
2022   } {
2023     \__bnvs_if_get:nncTF A { #1 } { #2 } {
2024       \__bnvs_quark_if_nil:cTF { #2 } {
2025         \__bnvs_gput:nnn A { #1 } { \q_no_value }

```

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

```

2026     \__bnvs_if_resolve_last:ncTF { #1 } { #2 } {
2027         \__bnvs_tl_put_right:cn { #2 } { - }
2028         \__bnvs_if_append_length:ncTF { #1 } { #2 } {
2029             \__bnvs_tl_put_right:cn { #2 } { + 1 }
2030             \__bnvs_round:c { #2 }
2031             \__bnvs_tl_if_empty:cTF { #2 } {
2032                 \__bnvs_gput:nnn A { #1 } { \q_nil }
2033                 \prg_return_false:
2034             } {
2035                 \__bnvs_gput:nnn A { #1 } { \q_nil }
2036                 \__bnvs_cache_gput:nnv A { #1 } { #2 }
2037                 \prg_return_true:
2038             }
2039         } {
2040             \BNVS_error:n {
2041 Unavailable~length~for~#1~(\token_to_str:N\__bnvs_if_resolve_first:ncTF/2) }
2042             \__bnvs_gput:nnn A { #1 } { \q_nil }
2043             \prg_return_false:
2044         }
2045     } {
2046         \BNVS_error:n {
2047 Unavailable~last~for~#1~(\token_to_str:N\__bnvs_if_resolve_first:ncTF/1) }
2048         \__bnvs_gput:nnn A { #1 } { \q_nil }
2049         \prg_return_false:
2050     }
2051 } {
2052     \__bnvs_quark_if_no_value:cTF { a } {
2053         \BNVS_fatal:n {Circular~definition:~#1}
2054     } {
2055         \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2056             \__bnvs_cache_gput:nnv A { #1 } { #2 }
2057             \prg_return_true:
2058         } {
2059             \prg_return_false:
2060         }
2061     }
2062 }
2063 } {
2064     \prg_return_false:
2065 }
2066 }
2067 }
2068 \BNVS_new_conditional_vc:cn { if_resolve_first } { T, F, TF }
2069 \BNVS_new_conditional_cpnn { if_append_first:nc } #1 #2 { T, F, TF } {
2070     \BNVS_begin:
2071     \__bnvs_if_resolve_first:ncTF { #1 } { #2 } {
2072         \BNVS_end_tl_put_right:cv { #2 } { #2 }
2073         \prg_return_true:
2074     } {
2075         \prg_return_false:
2076     }
2077 }

```

```

2078 \__bnvs_if_resolve_last:ncTF \__bnvs_if_resolve_last:ncTF {<FQ name>} <ans> {<yes code>} {<no code>}
2079 \__bnvs_if_append_last:ncTF \__bnvs_if_append_last:ncTF {<FQ name>} <ans> {<yes code>} {<no code>}

```

Resolve the last index of the fully qualified *<FQ name>* range into or to the right of the right of the *<tl variable>*, when possible. Execute *<yes code>* when a last index was given, *<no code>* otherwise.

```

2078 \BNVS_new_conditional:cpnn { if_resolve_last:nc } #1 #2 { T, F, TF } {
2079   \__bnvs_cache_if_get:nncTF Z { #1 } { #2 } {
2080     \prg_return_true:
2081   } {
2082     \__bnvs_if_get:nncTF Z { #1 } { #2 } {
2083       \__bnvs_quark_if_nil:cTF { #2 } {
2084         \__bnvs_gput:nnn Z { #1 } { \q_no_value }

```

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

```

2085   \__bnvs_if_resolve_first:ncTF { #1 } { #2 } {
2086     \__bnvs_tl_put_right:cn { #2 } { + }
2087     \__bnvs_if_append_length:ncTF { #1 } { #2 } {
2088       \__bnvs_tl_put_right:cn { #2 } { - 1 }
2089       \__bnvs_round:c { #2 }
2090       \__bnvs_cache_gput:nnv Z { #1 } { #2 }
2091       \__bnvs_gput:nnn Z { #1 } { \q_nil }
2092       \prg_return_true:
2093     } {
2094       \BNVS_error:x {
2095         Unavailable~length~for~#1~(\token_to_str:N \__bnvs_if_resolve_last:ncTF/1) }
2096         \__bnvs_gput:nnn Z { #1 } { \q_nil }
2097         \prg_return_false:
2098       }
2099     } {
2100       \BNVS_error:x {
2101         Unavailable~first~for~#1~(\token_to_str:N \__bnvs_if_resolve_last:ncTF/1) }
2102         \__bnvs_gput:nnn Z { #1 } { \q_nil }
2103         \prg_return_false:
2104       }
2105     } {
2106       \__bnvs_quark_if_no_value:cTF { #2 } {
2107         \BNVS_fatal:n {Circular~definition:~#1}
2108       } {
2109         \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2110           \__bnvs_cache_gput:nnv Z { #1 } { #2 }
2111           \prg_return_true:
2112         } {
2113           \prg_return_false:
2114         }
2115       }
2116     }
2117   } {
2118     \prg_return_false:
2119   }
2120 }
2121 }
2122 \BNVS_new_conditional_vc:cn { if_resolve_last } { T, F, TF }

```

```

2123 \BNVS_new_conditional:cpnn { if_append_last:nc } #1 #2 { T, F, TF } {
2124   \BNVS_begin:
2125     \__bnvs_if_resolve_last:ncTF { #1 } { #2 } {
2126       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2127       \prg_return_true:
2128     } {
2129       \BNVS_end:
2130       \prg_return_false:
2131     }
2132   }
2133 \BNVS_new_conditional_vc:cn { if_append_last } { T, F, TF }

```

```

\__bnvs_if_resolve_length:ncTF \__bnvs_if_resolve_length:ncTF {<FQ name>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_length:ncTF \__bnvs_if_append_length:ncTF {<FQ name>} <ans> {<yes code>} {<no code>}

```

Resolve the length of the *<FQ name>* slide range into *<ans>* tl variable, or append the length of the *<key>* slide range to this variable. Execute *<yes code>* when there is a *<length>*, *<no code>* otherwise.

```

2134 \BNVS_new_conditional:cpnn { if_resolve_length:nc } #1 #2 { T, F, TF } {
2135   \__bnvs_cache_if_get:nncTF L { #1 } { #2 } {
2136     \prg_return_true:
2137   } {
2138     \__bnvs_if_get:nncTF L { #1 } { #2 } {
2139       \__bnvs_quark_if_nil:cTF { #2 } {
2140         \__bnvs_gput:nnn L { #1 } { \q_no_value }

```

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

```

2141   \__bnvs_if_resolve_last:ncTF { #1 } { #2 } {
2142     \__bnvs_tl_put_right:cn { #2 } { - }
2143     \__bnvs_if_append_first:ncTF { #1 } { #2 } {
2144       \__bnvs_tl_put_right:cn { #2 } { + 1 }
2145       \__bnvs_round:c { #2 }
2146       \__bnvs_gput:nnn L { #1 } { \q_nil }
2147       \__bnvs_cache_gput:nnv L { #1 } { #2 }
2148       \prg_return_true:
2149     } {
2150       \BNVS_error:n {
2151         Unavailable~first~for~#1~(\__bnvs_if_resolve_length:ncTF/2) }
2152       \return_false:
2153     }
2154   } {
2155     \BNVS_error:n {
2156       Unavailable~last~for~#1~(\__bnvs_if_resolve_length:ncTF/1) }
2157     \return_false:
2158   }
2159 } {
2160   \__bnvs_quark_if_no_value:cTF { #2 } {
2161     \BNVS_fatal:n {Circular~definition:~#1}
2162   } {
2163     \__bnvs_if_resolve:vcTF { #2 } { #2 } {
2164       \__bnvs_cache_gput:nnv L { #1 } { #2 }
2165       \prg_return_true:
2166     } {

```

```

2167         \prg_return_false:
2168     }
2169 }
2170 }
2171 } {
2172     \prg_return_false:
2173 }
2174 }
2175 }
2176 \BNVS_new_conditional_vc:cn { if_resolve_length } { T, F, TF }
2177 \BNVS_new_conditional_cpnn { if_append_length:nc } #1 #2 { T, F, TF } {
2178     \BNVS_begin:
2179     \__bnvs_if_resolve_length:ncTF { #1 } { #2 } {
2180         \BNVS_end_tl_put_right:cv { #2 } { #2 }
2181         \prg_return_true:
2182     } {
2183         \prg_return_false:
2184     }
2185 }
2186 \BNVS_new_conditional_vc:cn { if_append_length } { T, F, TF }

```

```

\__bnvs_if_resolve_range:ncTF \__bnvs_if_resolve_range:ncTF {<FQ name>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_range:ncTF \__bnvs_if_append_range:ncTF {<FQ name>} <ans> {<yes code>} {<no code>}

```

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

```

2187 \BNVS_new_conditional_cpnn { if_append_range:nc } #1 #2 { T, F, TF } {
2188     \BNVS_begin:
2189     \__bnvs_if_resolve_first:ncTF { #1 } { a } {
2190         \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
2191             \__bnvs_tl_set:cn { a } { 0 }
2192         }
2193         \__bnvs_if_resolve_last:ncTF { #1 } { b } {

```

Limited from above and below.

```

2194         \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2195             \__bnvs_tl_set:cn { b } { 0 }
2196         }
2197         \__bnvs_tl_put_right:cn { a } { - }
2198         \__bnvs_tl_put_right:cv { a } { b }
2199         \BNVS_end_tl_put_right:cv { #2 } { a }
2200         \prg_return_true:
2201     } {

```

Limited from below.

```

2202         \BNVS_end_tl_put_right:cv { #2 } { a }
2203         \__bnvs_tl_put_right:cn { #2 } { - }
2204         \prg_return_true:
2205     }
2206 } {
2207     \__bnvs_if_resolve_last:ncTF { #1 } { b } {

```

Limited from above.

```

2208 \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2209   \__bnvs_tl_set:cn { b } { 0 }
2210 }
2211 \__bnvs_tl_put_left:cn { b } { - }
2212 \BNVS_end_tl_put_right:cv { #2 } { b }
2213 \prg_return_true:
2214 } {
2215   \__bnvs_if_resolve_V:ncTF { #1 } { b } {
2216     \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2217       \__bnvs_tl_set:cn { b } { 0 }
2218     }

```

Unlimited range.

```

2219   \BNVS_end_tl_put_right:cv { #2 } { b }
2220   \__bnvs_tl_put_right:cn { #2 } { - }
2221   \prg_return_true:
2222 } {
2223   \BNVS_end:
2224   \prg_return_false:
2225 }
2226 }
2227 }
2228 }
2229 \BNVS_new_conditional_vc:cn { if_append_range } { T, F, TF }
2230 \BNVS_new_conditional:cpnn { if_resolve_range:nc } #1 #2 { T, F, TF } {
2231   \__bnvs_tl_clear:c { #2 }
2232   \__bnvs_if_append_range:ncTF { #1 } { #2 } {
2233     \prg_return_true:
2234   } {
2235     \prg_return_false:
2236   }
2237 }
2238 \BNVS_new_conditional_vc:cn { if_resolve_range } { T, F, TF }

```

```

\__bnvs_if_resolve_previous:ncTF \__bnvs_if_append_previous:ncTF {<FQ 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.

```

2239 \BNVS_new_conditional:cpnn { if_resolve_previous:nc } #1 #2 { T, F, TF } {
2240   \__bnvs_cache_if_get:nncTF P { #1 } { #2 } {
2241     \prg_return_true:
2242   } {
2243     \__bnvs_if_resolve_first:ncTF { #1 } { #2 } {
2244       \__bnvs_tl_put_right:cn { #2 } { -1 }
2245       \__bnvs_round:c { #2 }
2246       \__bnvs_cache_gput:nnv P { #1 } { #2 }
2247       \prg_return_true:
2248     } {

```

```

2249     \prg_return_false:
2250   }
2251 }
2252 }
2253 \BNVS_new_conditional_vc:cn { if_resolve_previous } { T, F, TF }
2254 \BNVS_new_conditional_cpnn { if_append_previous:nc } #1 #2 { T, F, TF } {
2255   \BNVS_begin:
2256   \__bnvs_if_resolve_previous:ncTF { #1 } { #2 } {
2257     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2258     \prg_return_true:
2259   } {
2260     \BNVS_end:
2261     \prg_return_false:
2262   }
2263 }
2264 \BNVS_new_conditional_vc:cn { if_append_previous } { T, F, TF }

```

```

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

```

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

```

2265 \BNVS_new_conditional_cpnn { if_resolve_next:nc } #1 #2 { T, F, TF } {
2266   \__bnvs_cache_if_get:nncTF N { #1 } { #2 } {
2267     \prg_return_true:
2268   } {
2269     \__bnvs_if_resolve_last:ncTF { #1 } { #2 } {
2270       \__bnvs_tl_put_right:cn { #2 } { +1 }
2271       \__bnvs_round:c { #2 }
2272       \__bnvs_cache_gput:nnv N { #1 } { #2 }
2273       \prg_return_true:
2274     } {
2275       \prg_return_false:
2276     }
2277   }
2278 }
2279 \BNVS_new_conditional_vc:cn { if_resolve_next } { T, F, TF }
2280 \BNVS_new_conditional_cpnn { if_append_next:nc } #1 #2 { T, F, TF } {
2281   \BNVS_begin:
2282   \__bnvs_if_resolve_next:ncTF { #1 } { #2 } {
2283     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2284     \prg_return_true:
2285   } {
2286     \BNVS_end:
2287     \prg_return_true:
2288   }
2289 }
2290 \BNVS_new_conditional_vc:cn { if_append_next } { T, F, TF }

```

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

Resolve the value of the $\langle FQ\ name \rangle$ 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 `__bnvs_if_resolve_V:ncTF`.

```

2291 \BNVS_new_conditional:cpnn { if_resolve_v:nc } #1 #2 { T, F, TF } {
2292   \__bnvs_v_if_get:ncTF { #1 } { #2 } {
2293     \__bnvs_quark_if_no_value:cTF { #2 } {
2294       \BNVS_fatal:n {Circular~definition:~#1}
2295       \prg_return_false:
2296     } {
2297       \prg_return_true:
2298     }
2299   } {
2300     \__bnvs_v_gput:nn { #1 } { \q_no_value }
2301     \__bnvs_if_resolve_V:ncTF { #1 } { #2 } {
2302       \__bnvs_v_gput:nv { #1 } { #2 }
2303       \prg_return_true:
2304     } {
2305       \__bnvs_if_resolve_first:ncTF { #1 } { #2 } {
2306         \__bnvs_v_gput:nv { #1 } { #2 }
2307         \prg_return_true:
2308       } {
2309         \__bnvs_if_resolve_last:ncTF { #1 } { #2 } {
2310           \__bnvs_v_gput:nv { #1 } { #2 }
2311           \prg_return_true:
2312         } {
2313           \__bnvs_v_gremove:n { #1 }
2314           \prg_return_false:
2315         }
2316       }
2317     }
2318   }
2319 }
2320 \BNVS_new_conditional_vc:cn { if_resolve_v } { T, F, TF }
2321 \BNVS_new_conditional:cpnn { if_append_v:nc } #1 #2 { T, F, TF } {
2322   \BNVS_begin:
2323   \__bnvs_if_resolve_v:ncTF { #1 } { #2 } {
2324     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2325     \prg_return_true:
2326   } {
2327     \BNVS_end:
2328     \prg_return_false:
2329   }
2330 }
2331 \BNVS_new_conditional_vc:cn { if_append_v } { T, F, TF }

```

```

2332 \__bnvs_index_can:nTF      \__bnvs_index_can:nTF {<FQ name>} {<yes code>} {<no code>}
2333 \__bnvs_index_can:vTF      \__bnvs_if_resolve_index:nncTF {<FQ name>} {<integer>} <ans> {<yes code>}
2334 \__bnvs_if_resolve_index:nncTF {<no code>}
2335 \__bnvs_if_resolve_index:vvvTF \__bnvs_if_append_index:nncTF {<FQ name>} {<integer>} <ans> {<yes code>}
2336 \__bnvs_if_append_index:nncTF {<no code>}
2337 \__bnvs_if_append_index:vvvTF

```

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

```

2332 \BNVS_new_conditional:cpnn { index_can:n } #1 { p, T, F, TF } {
2333   \bool_if:nTF {
2334     \__bnvs_if_in_p:nn V { #1 }
2335     || \__bnvs_if_in_p:nn A { #1 }
2336     || \__bnvs_if_in_p:nn Z { #1 }
2337   } {
2338     \prg_return_true:
2339   } {
2340     \prg_return_false:
2341   }
2342 }
2343 \BNVS_new_conditional:cpnn { index_can:v } #1 { p, T, F, TF } {
2344   \BNVS_tl_use:Nv \__bnvs_index_can:nTF { #1 } {
2345     \prg_return_true:
2346   } {
2347     \prg_return_false:
2348   }
2349 }
2350 \BNVS_new_conditional:cpnn { if_resolve_index:nnc } #1 #2 #3 { T, F, TF } {
2351   \exp_args:Nx \__bnvs_if_resolve_V:ncTF { #1.#2 } { #3 } {
2352     \prg_return_true:
2353   } {
2354     \__bnvs_if_resolve_first:ncTF { #1 } { #3 } {
2355       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2356       \__bnvs_round:c { #3 }
2357     } \prg_return_true:

```

Limited overlay set.

```

2358   } {
2359     \__bnvs_if_resolve_last:ncTF { #1 } { #3 } {
2360       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2361       \__bnvs_round:c { #3 }
2362     } \prg_return_true:
2363   } {
2364     \__bnvs_if_resolve_V:ncTF { #1 } { #3 } {
2365       \__bnvs_tl_put_right:cn { #3 } { + #2 - 1 }
2366       \__bnvs_round:c { #3 }
2367     } \prg_return_true:
2368   } {

```

```

2369         \prg_return_false:
2370     }
2371 }
2372 }
2373 }
2374 }
2375 \BNVS_new_conditional:cpnn { if_resolve_index:nvc } #1 #2 #3 { T, F, TF } {
2376     \BNVS_tl_use:nv {
2377         \__bnvs_if_resolve_index:nncTF { #1 }
2378     } { #2 } { #3 } {
2379         \prg_return_true:
2380     } {
2381         \prg_return_false:
2382     }
2383 }
2384 \BNVS_new_conditional:cpnn { if_resolve_index:vvc } #1 #2 #3 { T, F, TF } {
2385     \BNVS_tl_use:nv {
2386         \BNVS_tl_use:Nv \__bnvs_if_resolve_index:nncTF { #1 }
2387     } { #2 } { #3 } {
2388         \prg_return_true:
2389     } {
2390         \prg_return_false:
2391     }
2392 }
2393 \BNVS_new_conditional:cpnn { if_append_index:nnc } #1 #2 #3 { T, F, TF } {
2394     \BNVS_begin:
2395     \__bnvs_if_resolve_index:nncTF { #1 } { #2 } { #3 } {
2396         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2397         \prg_return_true:
2398     } {
2399         \BNVS_end:
2400         \prg_return_false:
2401     }
2402 }
2403 \BNVS_new_conditional:cpnn { if_append_index:vvc } #1 #2 #3 { T, F, TF } {
2404     \BNVS_tl_use:nv {
2405         \BNVS_tl_use:Nv \__bnvs_if_append_index:nncTF { #1 }
2406     } { #2 } { #3 } {
2407         \prg_return_true:
2408     } {
2409         \prg_return_false:
2410     }
2411 }

```

6.22 Index counter

__bnvs_if_resolve_n:ncTF __bnvs_if_resolve_n:ncTF {*<FQ name>*} *<ans>* {*<yes code>*} {*<no code>*}

__bnvs_if_append_n:ncTF Evaluate the n counter associated to the {*<FQ 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.

```

2412 \BNVS_new_conditional:cpnn { if_resolve_n:nc } #1 #2 { T, F, TF } {

```

```

2413 \__bnvs_n_if_get:ncF { #1 } { #2 } {
2414   \__bnvs_tl_set:cn { #2 } { 1 }
2415   \__bnvs_n_gput:nn { #1 } { 1 }
2416 }
2417 \prg_return_true:
2418 }
2419 \BNVS_new_conditional:cpnn { if_append_n:nc } #1 #2 { T, F, TF } {
2420   \BNVS_begin:
2421     \__bnvs_if_resolve_n:ncTF { #1 } { #2 } {
2422       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2423       \prg_return_true:
2424     } {
2425       \BNVS_end:
2426       \prg_return_false:
2427     }
2428 }
2429 \BNVS_new_conditional_vc:cn { if_append_n } { T, F, TF }

```

```

\__bnvs_if_resolve_n_index:ncTF \__bnvs_if_resolve_n_index:ncTF {<FQ name>} <ans> {<yes code>} {<no
\__bnvs_if_append_n_index:ncTF  code>}
\__bnvs_if_resolve_n_index:nncTF \__bnvs_if_append_n_index:ncTF {<FQ name>} <ans> {<yes code>} {<no
\__bnvs_if_append_n_index:nncTF  code>}

```

```

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

```

Resolve the index for the value of the n counter associated to the {<FQ 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.

```

2430 \BNVS_new_conditional:cpnn { if_resolve_n_index:nc } #1 #2 { T, F, TF } {
2431   \__bnvs_if_resolve_n:ncTF { #1 } { #2 } {
2432     \__bnvs_if_resolve_index:nvcTF { #1 } { #2 } { #2 } {
2433       \prg_return_true:
2434     } {
2435       \prg_return_false:
2436     }
2437   } {
2438     \prg_return_false:
2439   }
2440 }
2441 \BNVS_new_conditional:cpnn { if_resolve_n_index:nnc } #1 #2 #3 { T, F, TF } {
2442   \__bnvs_if_resolve_n:ncTF { #1 } { #3 } {
2443     \__bnvs_tl_put_left:cn { #3 } { #2. }
2444     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2445       \prg_return_true:
2446     } {
2447       \prg_return_false:
2448     }
2449   } {

```

```

2450     \prg_return_false:
2451   }
2452 }
2453 \BNVS_new_conditional:cpnn { if_append_n_index:nc } #1 #2 { T, F, TF } {
2454   \BNVS_begin:
2455   \__bnvs_if_resolve_n_index:ncTF { #1 } { #2 } {
2456     \BNVS_end_tl_put_right:cv { #2 } { #2 }
2457     \prg_return_true:
2458   } {
2459     \BNVS_end:
2460     \prg_return_false:
2461   }
2462 }
2463 \BNVS_new_conditional:cpnn { if_append_n_index:nnc } #1 #2 #3 { T, F, TF } {
2464   \BNVS_begin:
2465   \__bnvs_if_resolve_n_index:nncTF { #1 } { #2 } { #3 } {
2466     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2467     \prg_return_true:
2468   } {
2469     \BNVS_end:
2470     \prg_return_false:
2471   }
2472 }
2473 \BNVS_new_conditional_vc:cn { if_append_n_index } { T, F, TF }
2474 \BNVS_new_conditional_vvc:cn { if_append_n_index } { T, F, TF }

```

6.23 Value counter

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

```

2475 \BNVS_new_conditional:cpnn { if_resolve_v_incr:nnc } #1 #2 #3 { T, F, TF } {
2476   \__bnvs_if_resolve:ncTF { #2 } { #3 } {
2477     \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2478       \__bnvs_if_resolve_v:ncTF { #1 } { #3 } {
2479         \prg_return_true:
2480       } {
2481         \prg_return_false:
2482       }
2483     } {
2484       \__bnvs_tl_put_right:cn { #3 } { + }
2485       \__bnvs_if_append_v:ncTF { #1 } { #3 } {
2486         \__bnvs_round:c { #3 }
2487         \__bnvs_v_gput:nv { #1 } { #3 }
2488         \prg_return_true:
2489       } {

```

```

2490         \prg_return_false:
2491     }
2492 }
2493 } {
2494     \prg_return_false:
2495 }
2496 }
2497 \BNVS_new_conditional:cpnn { if_append_v_incr:nnc } #1 #2 #3 { T, F, TF } {
2498     \BNVS_begin:
2499     \__bnvs_if_resolve_v_incr:nncTF { #1 } { #2 } { #3 } {
2500         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2501         \prg_return_true:
2502     } {
2503         \BNVS_end:
2504         \prg_return_false:
2505     }
2506 }
2507 \BNVS_new_conditional_vnc:cn { if_append_v_incr } { T, F, TF }
2508 \BNVS_new_conditional_vvc:cn { if_append_v_incr } { T, F, TF }
2509 \BNVS_new_conditional:cpnn { if_resolve_v_post:nnc } #1 #2 #3 { T, F, TF } {
2510     \__bnvs_if_resolve_v:ncTF { #1 } { #3 } {
2511         \BNVS_begin:
2512         \__bnvs_if_resolve:ncTF { #2 } { a } {
2513             \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
2514                 \BNVS_end:
2515                 \prg_return_true:
2516             } {
2517                 \__bnvs_tl_put_right:cn { a } { + }
2518                 \__bnvs_tl_put_right:cv { a } { #3 }
2519                 \__bnvs_round:c { a }
2520                 \BNVS_end_v_gput:nv { #1 } { a }
2521                 \prg_return_true:
2522             }
2523         } {
2524             \BNVS_end:
2525             \prg_return_false:
2526         }
2527     } {
2528         \prg_return_false:
2529     }
2530 }
2531 \BNVS_new_conditional_vvc:cn { if_resolve_v_post } { T, F, TF }
2532 \BNVS_new_conditional:cpnn { if_append_v_post:nnc } #1 #2 #3 { T, F, TF } {
2533     \BNVS_begin:
2534     \__bnvs_if_resolve_v_post:nncTF { #1 } { #2 } { #3 } {
2535         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2536         \prg_return_true:
2537     } {
2538         \prg_return_true:
2539     }
2540 }
2541 \BNVS_new_conditional_vnc:cn { if_append_v_post } { T, F, TF }
2542 \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 {<FQ name>} {<base FQ name>}</code>
<code>__bnvs_if_resolve_n_incr:vvncTF</code>	<code>{<offset>} {<ans>} {<yes code>} {<no code>}</code>
<code>__bnvs_if_resolve_n_incr:nncTF</code>	<code>__bnvs_if_resolve_n_incr:nncTF {<FQ name>} {<offset>} {<ans>} {<yes</code>
<code>__bnvs_if_append_n_incr:nncTF</code>	<code>code>} {<no code>}</code>
<code>__bnvs_if_append_n_incr:nncTF</code>	<code>__bnvs_if_append_n_incr:nncTF {<FQ name>} {<base FQ name>}</code>
<code>__bnvs_if_append_n_incr:(vnc vvc)TF</code>	<code>{<offset>} {<ans>} {<yes code>} {<no code>}</code>
<code>__bnvs_if_resolve_n_post:nncTF</code>	<code>__bnvs_if_append_n_incr:nncTF {<FQ name>} {<offset>} {<ans>} {<yes</code>
<code>__bnvs_if_append_n_post:nncTF</code>	<code>code>} {<no code>}</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.

2543 `\BNVS_new_conditional:cpnn { if_resolve_n_incr:nnc } #1 #2 #3 #4 { T, TF } {`
 Resolve the `<offset>` into the `<ans>` variable.

2544 `__bnvs_if_resolve:ncTF { #3 } { #4 } {`
 2545 `\BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {`

The offset is resolved to 0, we just have to resolve the ...n

2546 `__bnvs_if_resolve_n:ncTF { #1 } { #4 } {`
 2547 `__bnvs_if_resolve_index:nvcTF { #1 } { #4 } { #4 } {`
 2548 `\prg_return_true:`
 2549 `} {`
 2550 `\prg_return_false:`
 2551 `}`
 2552 `} {`
 2553 `\prg_return_false:`
 2554 `}`
 2555 `} {`

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

2556 `__bnvs_tl_put_right:cn { #4 } { + }`
 2557 `__bnvs_if_append_n:ncTF { #1 } { #4 } {`
 2558 `__bnvs_round:c { #4 }`
 2559 `__bnvs_n_gput:nv { #1 } { #4 }`
 2560 `__bnvs_if_resolve_index:nvcTF { #2 } { #4 } { #4 } {`
 2561 `\prg_return_true:`
 2562 `} {`
 2563 `\prg_return_false:`
 2564 `}`
 2565 `} {`
 2566 `\prg_return_false:`
 2567 `}`
 2568 `}`
 2569 `} {`
 2570 `\prg_return_false:`
 2571 `}`
 2572 `}`
 2573 `\BNVS_new_conditional:cpnn { if_resolve_n_incr:nnc } #1 #2 #3 { T, F, TF } {`
 2574 `__bnvs_if_resolve:ncTF { #2 } { #3 } {`
 2575 `\BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {`
 2576 `__bnvs_if_resolve_n:ncTF { #1 } { #3 } {`
 2577 `__bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {`
 2578 `\prg_return_true:`
 2579 `} {`

```

2580         \prg_return_false:
2581     }
2582 } {
2583     \prg_return_false:
2584 }
2585 } {
2586     \__bnvs_tl_put_right:cn { #3 } { + }
2587     \__bnvs_if_append_n:ncTF { #1 } { #3 } {
2588         \__bnvs_round:c { #3 }
2589         \__bnvs_n_gput:nv { #1 } { #3 }
2590         \__bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {
2591             \prg_return_true:
2592         } {
2593             \prg_return_false:
2594         }
2595     } {
2596         \prg_return_false:
2597     }
2598 }
2599 } {
2600     \prg_return_false:
2601 }
2602 }
2603 \BNVS_new_conditional_vnc:cn { if_resolve_n_incr } { T, F, TF }
2604 \BNVS_new_conditional_vvc:cn { if_resolve_n_incr } { T, F, TF }
2605 \BNVS_new_conditional_vvnc:cn { if_resolve_n_incr } { T, F, TF }
2606 % \end{BNVS/macrocode}
2607 % \begin{BNVS/macrocode}
2608 \BNVS_new_conditional:cpnn
2609 { if_append_n_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
2610     \BNVS_begin:
2611     \__bnvs_if_resolve_n_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
2612         \BNVS_end_tl_put_right:cv { #4 } { #4 }
2613         \prg_return_true:
2614     } {
2615         \BNVS_end:
2616         \prg_return_false:
2617     }
2618 }
2619 \BNVS_new_conditional_vvnc:cn { if_append_n_incr } { T, F, TF }
2620 \BNVS_new_conditional_vvvc:cn { if_append_n_incr } { T, F, TF }
2621 \BNVS_new_conditional:cpnn { if_append_n_incr:nnc } #1 #2 #3 { T, F, TF } {
2622     \BNVS_begin:
2623     \__bnvs_if_resolve_n_incr:nncTF { #1 } { #2 } { #3 } {
2624         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2625         \prg_return_true:
2626     } {
2627         \BNVS_end:
2628         \prg_return_false:
2629     }
2630 }
2631 \BNVS_new_conditional_vnc:cn { if_append_n_incr } { T, F, TF }
2632 \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 {<FQ name>} {<offset>} <ans> {<yes</code> <code>code>} {<no code>}</code> <code>_bnvs_if_append_v_post:nncTF {<FQ name>} {<offset>} <ans> {<yes</code> <code>code>} {<no code>}</code>
--	---

Resolve the value of the free counter for the given *<FQ name>* into the *<ans>* t1 variable then increment this free counter position accordingly. The append version, appends the value to the right of the *<ans>* t1 variable. The content of *<ans>* is undefined while in the *{<no code>}* branch and on resolution only.

```

2633 \BNVS_new_conditional:cpnn { if_resolve_n_post:nnc } #1 #2 #3 { T, F, TF } {
2634   \\_bnvs_if_resolve_n:ncTF { #1 } { #3 } {
2635     \BNVS_begin:
2636     \\_bnvs_if_resolve:ncTF { #2 } { #3 } {
2637       \BNVS_tl_use:Nv \int_compare:nNnTF { #3 } = 0 {
2638         \BNVS_end:
2639         \\_bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {
2640           \prg_return_true:
2641         } {
2642           \prg_return_false:
2643         }
2644       } {
2645         \\_bnvs_tl_put_right:cn { #3 } { + }
2646         \\_bnvs_if_append_n:ncTF { #1 } { #3 } {
2647           \\_bnvs_round:c { #3 }
2648           \\_bnvs_n_gput:nv { #1 } { #3 }
2649           \BNVS_end:
2650           \\_bnvs_if_resolve_index:nvcTF { #1 } { #3 } { #3 } {
2651             \prg_return_true:
2652           } {
2653             \prg_return_false:
2654           }
2655         } {
2656           \BNVS_end:
2657           \prg_return_false:
2658         }
2659       }
2660     } {
2661       \BNVS_end:
2662       \prg_return_false:
2663     }
2664   } {
2665     \prg_return_false:
2666   }
2667 }
2668 \BNVS_new_conditional:cpnn { if_append_n_post:nnc } #1 #2 #3 { T, F, TF } {
2669   \BNVS_begin:
2670   \\_bnvs_if_resolve_n_post:nncTF { #1 } { #2 } { #3 } {
2671     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2672     \prg_return_true:
2673   } {
2674     \BNVS_end:

```



```

2675     \prg_return_false:
2676   }
2677 }
2678 \BNVS_new_conditional_vnc:cn { if_append_n_post } { T, F, TF }
2679 \BNVS_new_conditional_vvc:cn { if_append_n_post } { T, F, TF }

```

6.24 Evaluation

_bnvs_round_ans: _bnvs_rslv_round:

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

```

2680 \BNVS_set:cpn { round_ans: } {
2681   \_bnvs_round:c { ans }
2682 }

```

6.25 Functions for the resolution

They manily start with _bnvs_if_resolve_

_bnvs_if_resolve_end_return_false:n _bnvs_if_resolve_end_return_false:n {<message>}

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

_bnvs_if_resolve_path_n:TFF _bnvs_if_resolve_path_n:TFF {<yes code>} {<no code 1>} {<no code 2>}

Called by function ...if_resolve_path_n_end_return_false_or:T. Calls function ...if_resolve_Fip_n:TF.

```

2683 \BNVS_new:cpn { if_resolve_path_n:TFF } #1 #2 {
2684   \_bnvs_if_resolve_Fip_n:TF {
2685     \_bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
2686   }
2687 }
2688 \BNVS_new:cpn { if_resolve_path:TFF } #1 #2 {
2689   \_bnvs_if_resolve_Fip:TF {
2690     \_bnvs_seq_if_empty:cTF { path } { #1 } { #2 }
2691   }
2692 }

```

_bnvs_if_resolve_path_n_end_return_false_or:T _bnvs_if_resolve_path_n_end_return_false_or:T {<yes code>}

Resolve the path and execute <yes code> on success. On failure, ends a T_EX block and returns false by calling \if_resolve_end_return_false:n.

```

2693 \BNVS_new:cpn { if_resolve_end_return_false:n } #1 {
2694   \BNVS_end:
2695   \prg_return_false:
2696 }

```

```

2697 \BNVS_new:cpn { if_resolve_path_end_return_false_or:T } #1 {
2698   \__bnvs_if_resolve_Fip:TF {
2699     \__bnvs_seq_if_empty:cTF { path } {
2700       #1
2701     } {
2702       \__bnvs_if_resolve_end_return_false:n {
2703         Too~many~dotted~components
2704       }
2705     }
2706   } {
2707     \__bnvs_if_resolve_end_return_false:n {
2708       Unknown~dotted~path
2709     }
2710   }
2711 }
2712 \BNVS_new:cpn { if_resolve_path_n_end_return_false_or:T } #1 {
2713   \__bnvs_if_resolve_path_n:TFF {
2714     #1
2715   } {
2716     \__bnvs_if_resolve_end_return_false:n {
2717       Too~many~dotted~components
2718     }
2719   } {
2720     \__bnvs_if_resolve_end_return_false:n {
2721       Unknown~dotted~path
2722     }
2723   }
2724 }
2725 \BNVS_set:cpn { resolve_x:T } #1 {
2726   \__bnvs_if_resolve_Fip_x_path:TFF {
2727     #1
2728   } {
2729     \__bnvs_if_resolve_end_return_false:n {
2730       Too~many~dotted~components
2731     }
2732   } {
2733     \__bnvs_if_resolve_end_return_false:n {
2734       Unknown~dotted~path
2735     }
2736   }
2737 }

```

__bnvs_if_path_suffix:nTF __bnvs_if_path_suffix:nTF {<tl>} {<yes code>} {<no code>}

If the last item of \l__bnvs_path_seq is <suffix>, then execute <yes code> otherwise execute <no code>. The suffix is n in the second case.

```

2738 \BNVS_new_conditional:cpnn { if_path_pop_right_n:c } #1 { T, F, TF } {
2739   \__bnvs_seq_pop_right:ccTF { path } { #1 }
2740   { \prg_return_true: } { \prg_return_false: }
2741 }

```

```

__bnvs_if_resolve_pop_Fip:TTF                                __bnvs_if_resolve_pop_Fip:TTF {(<blank code>)}
__bnvs_if_resolve_end_return_or_pop_complete_white:T        {(<blank code>)} {(<end code>)}
__bnvs_if_resolve_end_return_or_pop_complete_black:T        __bnvs_if_resolve_end_return_or_pop_complete_-
                                                             white:T {(<blank code>)}
                                                             __bnvs_if_resolve_end_return_or_pop_complete_-
                                                             black:T {(<blank code>)}

```

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

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

For `__bnvs_if_resolve_end_return_or_pop_complete_black:T`: pops the seven heading items of the `split` sequence then execute `<blank code>`.

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

```

2742 \BNVS_new:cpn { if_resolve_pop_Fip:TTF } #1 #2 #3 {
2743   __bnvs_split_if_pop_left:cTF { FQ_name } {
2744     __bnvs_split_if_pop_left:cTF { id } {
2745       __bnvs_split_if_pop_left:cTF { path } {
2746         __bnvs_tl_if_blank:vTF { FQ_name } {

```

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

```

2747     #1
2748   } {
2749     __bnvs_tl_if_blank:vTF { id } {
2750       __bnvs_tl_put_left:cv { FQ_name } { id_last }
2751       __bnvs_tl_set:cv { id } { id_last }
2752     } {
2753       __bnvs_tl_set:cv { id_last } { id }
2754     }
2755     __bnvs_tl_if_blank:vTF { path } {
2756       __bnvs_seq_clear:c { path }
2757     } {
2758       __bnvs_seq_set_split:cnv { path } { . } { path }
2759       __bnvs_seq_remove_all:cn { path } { }
2760     }
2761     __bnvs_tl_set_eq:cc { FQ_base } { FQ_name }
2762     __bnvs_seq_set_eq:cc { path_base } { path }
2763     #2
2764   }
2765   } {
2766   __bnvs_end_unreachable_return_false:n { if_resolve_pop_Fip:TTF/2 }
2767   }
2768   } {
2769   __bnvs_end_unreachable_return_false:n { if_resolve_pop_Fip:TTF/1 }
2770   }
2771   } { #3 }
2772 }

```

```

\__bnvs_if_resolve_pop_complete:nNT \__bnvs_if_resolve_pop_Fip:FFTF {\<no FQ_name code>} {\<no id code>}
{\<yes code>} {\<no capture code>}
\__bnvs_if_resolve_pop_complete:nNT {\<tl>} {\<tl var>} {\<yes code>}

```

$\langle tl \rangle$ and $\langle tl\ var \rangle$ are the arguments of the `__bnvs_if_resolve:ncTF` conditionals. conditional variants.

`__bnvs_if_resolve_pop_Fip:FFTF` locally sets the `FQ_name`, `id` and path `tl` variables to the 3 heading items of the split sequence, which correspond to the 3 eponym capture groups. If no capture group is available, $\langle no\ capture\ code \rangle$ is executed. If the capture group for the FQ name is empty, then $\langle empty\ FQ\ name\ code \rangle$ is executed. If there is no capture group for the id, then $\langle no\ id\ code \rangle$ is executed. Otherwise $\langle yes\ code \rangle$ is executed.

`__bnvs_rslv_pop_end:T` locally sets the four `tl` variables `n_incr`, `plus`, `rhs` and `post` to the next four heading items of the split sequence, which correspond to the last 4 eponym capture groups.

```

2773 \BNVS_new:cpn { if_resolve_end_return_or_pop_complete_white:T } #1 {
2774   \__bnvs_split_if_pop_left:cTF { n_incr } {
2775     \__bnvs_split_if_pop_left:cTF { plus } {
2776       \__bnvs_split_if_pop_left:cTF { rhs } {
2777         \__bnvs_split_if_pop_left:cTF { post } {
2778           #1
2779           } {
2780             \__bnvs_end_unreachable_return_false:n
2781             { if_resolve_end_return_or_pop_complete_white:T/4 }
2782           }
2783         } {
2784           \__bnvs_end_unreachable_return_false:n
2785           { if_resolve_end_return_or_pop_complete_white:T/3 }
2786         }
2787       } {
2788         \__bnvs_end_unreachable_return_false:n
2789         { if_resolve_end_return_or_pop_complete_white:T/2 }
2790       }
2791     } {
2792       \__bnvs_end_unreachable_return_false:n
2793       { if_resolve_end_return_or_pop_complete_white:T/1 }
2794     }
2795   }
2796 \BNVS_new:cpn { if_resolve_end_return_or_pop_complete_black:T } #1 {
2797   \__bnvs_split_if_pop_left:cTF { a } {
2798     \__bnvs_split_if_pop_left:cTF { a } {
2799       \__bnvs_split_if_pop_left:cTF { a } {
2800         \__bnvs_split_if_pop_left:cTF { a } {
2801           \__bnvs_split_if_pop_left:cTF { a } {
2802             \__bnvs_split_if_pop_left:cTF { a } {
2803               \__bnvs_split_if_pop_left:cTF { a } {
2804                 #1
2805                 } {
2806                   \__bnvs_end_unreachable_return_false:n
2807                   { if_resolve_end_return_or_pop_complete_black:T/7 }
2808                 }
2809               } {
2810                 \__bnvs_end_unreachable_return_false:n

```

```

2811         { if_resolve_end_return_or_pop_complete_black:T/6 }
2812     }
2813 } {
2814     \_bnvs_end_unreachable_return_false:n
2815     { if_resolve_end_return_or_pop_complete_black:T/5 }
2816 }
2817 } {
2818     \_bnvs_end_unreachable_return_false:n
2819     { if_resolve_end_return_or_pop_complete_black:T/4 }
2820 }
2821 } {
2822     \_bnvs_end_unreachable_return_false:n
2823     { if_resolve_end_return_or_pop_complete_black:T/3 }
2824 }
2825 } {
2826     \_bnvs_end_unreachable_return_false:n
2827     { if_resolve_end_return_or_pop_complete_black:T/2 }
2828 }
2829 } {
2830     \_bnvs_end_unreachable_return_false:n
2831     { if_resolve_end_return_or_pop_complete_black:T/1 }
2832 }
2833 }

```

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

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

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

(End of definition for `\l__bnvs_ans_tl`.)

`\l__bnvs_split_seq` The sequence of 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`.)

2834 `\BNVS_int_new:c { split }`

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

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

```

2835 \BNVS_new:cpn { if_resolve_Fip_x_path_or_end_return_false:nT } #1 #2 {
2836   \_bnvs_if_resolve_Fip_x_path:TFF {
2837     #2
2838   } {
2839     \BNVS_end_return_false:x { Too-many-dotted-components:~#1 }
2840   } {
2841     \BNVS_end_return_false:x { Unknown-dotted-path:~#1 }
2842   }
2843 }
2844 \BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
2845   \BNVS_begin:
2846     \_bnvs_if_resolve:ncTF { #1 } { #2 } {
2847       \BNVS_end_tl_put_right:cv { #2 } { #2 }
2848       \prg_return_true:
2849     } {
2850       \BNVS_end:
2851       \prg_return_false:
2852     }
2853 }
2854 \BNVS_new:cpn { end_unreachable_return_false:n } #1 {
2855   \BNVS_error:x { UNREACHABLE/#1 }
2856   \BNVS_end:

```

```

2857 \prg_return_false:
2858 }

2859 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { TF } {
2860   \__bnvs_if_call:TF {
2861     \BNVS_begin:

```

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

```

2862   \__bnvs_if_regex_split:cnTF { split } { #1 } {

```

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

```

2863     \BNVS_set:cpn { if_resolve_end_return_true: } {

```

Normal and unique end of the loop.

```

2864       \__bnvs_if_resolve_round_ans:
2865       \BNVS_end_tl_set:cv { #2 } { ans }
2866       \prg_return_true:
2867     }
2868     \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round_ans: }
2869     \__bnvs_tl_clear:c { ans }
2870     \__bnvs_if_resolve_loop_or_end_return:
2871   } {
2872     \__bnvs_tl_clear:c { ans }
2873     \__bnvs_round_ans:n { #1 }
2874     \BNVS_end_tl_set:cv { #2 } { ans }
2875     \prg_return_true:
2876   }
2877 } {
2878   \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
2879   \prg_return_false:
2880 }
2881 }
2882 \BNVS_new_conditional:cpnn { if_append:vc } #1 #2 { T, F, TF } {
2883   \BNVS_tl_use:Nv \__bnvs_if_append:ncTF { #1 } { #2 } {
2884     \prg_return_true:
2885   } {
2886     \prg_return_false:
2887   }
2888 }
2889 \BNVS_new_conditional:cpnn { if_resolve:vc } #1 #2 { T, F, TF } {
2890   \BNVS_tl_use:Nv \__bnvs_if_resolve:ncTF { #1 } { #2 } {
2891     \prg_return_true:
2892   } {
2893     \prg_return_false:
2894   }
2895 }

```

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

```

\__bnvs_if_resolve_loop_or_end_return: \__bnvs_if_resolve_loop_or_end_return:

```

May call itself at the end.

```

2896 \BNVS_new:cpn { if_resolve_loop_or_end_return: } {

```

```

2897  \_bnvs_tl_clear:c { suffix }
2898  \_bnvs_split_if_pop_left:cTF { a } {
2899    \_bnvs_tl_put_right:cv { ans } { a }
2900    \_bnvs_if_resolve_pop_Fip:TTF {
2901      \_bnvs_if_resolve_pop_Fip:TTF {
2902  \_bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/3 }
2903    } {
2904      \_bnvs_if_resolve_end_return_or_pop_complete_white:T {
2905        \_bnvs_tl_if_blank:vTF { n_incr } {
2906          \_bnvs_tl_if_blank:vTF { plus } {
2907            \_bnvs_tl_if_blank:vTF { rhs } {
2908              \_bnvs_tl_if_blank:vTF { post } {
2909                \_bnvs_if_resolve_V_loop_or_end_return_true:F {

```

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

```

2910      \_bnvs_seq_pop_right:ccTF { path } { suffix } {
2911        \BNVS_tl_use:Nv \str_case:nnF { suffix } {
2912 { n          } { \BNVS_use:c { if_resolve_loop_or_end_return[...n]: } }
2913 { length    } { \BNVS_use:c { if_resolve_loop_or_end_return[.length]: } }
2914 { last      } { \BNVS_use:c { if_resolve_loop_or_end_return[.last]: } }
2915 { range     } { \BNVS_use:c { if_resolve_loop_or_end_return[.range]: } }
2916 { previous  } { \BNVS_use:c { if_resolve_loop_or_end_return[.previous]: } }
2917 { next      } { \BNVS_use:c { if_resolve_loop_or_end_return[.next]: } }
2918 { reset     } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset]: } }
2919 { reset_all } { \BNVS_use:c { if_resolve_loop_or_end_return[.reset_all]: } }
2920      } {
2921  \BNVS_use:c { if_resolve_loop_or_end_return[...<integer>]: }
2922      }
2923      } {

```

No dotted path.

```

2924  \BNVS_use:c { if_resolve_loop_or_end_return[...]: }
2925      }
2926      }
2927      } {
2928  \BNVS_use:c { if_resolve_loop_or_end_return[...++]: }
2929      }
2930      } {
2931      \_bnvs_if_path_suffix:nTF { n } {
2932  \BNVS_use:c { if_resolve_loop_or_end_return[...n=...]: }
2933      } {
2934  \BNVS_use:c { if_resolve_loop_or_end_return[...=...]: }
2935      }
2936      }
2937      } {
2938      \_bnvs_if_path_suffix:nTF { n } {
2939  \BNVS_use:c { if_resolve_loop_or_end_return[...n+=...]: }
2940      } {
2941  \BNVS_use:c { if_resolve_loop_or_end_return[...+=...]: }
2942      }
2943      }
2944      } {
2945  \BNVS_use:c { if_resolve_loop_or_end_return[...++n]: }
2946      }

```



```

2947     }
2948   } {
2949   % split sequence empty
2950   \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/2 }
2951   }
2952   } {
2953     \__bnvs_if_resolve_end_return_or_pop_complete_black:T {
2954       \__bnvs_if_path_suffix:nTF { n } {
2955         \BNVS_use:c { if_resolve_loop_or_end_return[+...n]: }
2956       } {
2957         \BNVS_use:c { if_resolve_loop_or_end_return[+...]: }
2958       }
2959     }
2960   } {
2961     \__bnvs_if_resolve_end_return_true:
2962   }
2963   } {
2964     \__bnvs_end_unreachable_return_false:n { if_resolve_loop_or_end_return:/1 }
2965   }
2966 }

```

Implementation detail: tl variable a is used.

```

2967 \BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {
2968   \__bnvs_tl_set:cx { a } {
2969     \BNVS_tl_use:c { FQ_name } \BNVS_tl_use:c { path }
2970   }
2971   \__bnvs_if_resolve_v:vcTF { a } { a } {
2972     \__bnvs_tl_put_right:cv { ans } { a }
2973     \__bnvs_if_resolve_loop_or_end_return:
2974   } {
2975     \__bnvs_if_resolve_V:vcTF { a } { a } {
2976       \__bnvs_tl_put_right:cv { ans } { a }
2977       \__bnvs_if_resolve_loop_or_end_return:
2978     } {
2979       #1
2980     }
2981   }
2982 }
2983 \BNVS_new:cpn { end_return_error:n } #1 {
2984   \BNVS_error:n { #1 }
2985   \BNVS_end:
2986   \prg_return_false:
2987 }
2988 \BNVS_new:cpn { if_resolve_loop_or_end_return[...n]: } {

```

- Case ...n. The .n suffix is consumed. It is no longer at the end of the dotted path.

```

2989 \__bnvs_if_resolve_path_n_end_return_false_or:T {
2990   \__bnvs_resolve_base_n:
2991   \__bnvs_if_append_n_index:vcTF { FQ_name } { FQ_base } { ans } {

```

```

2992     \__bnvs_if_resolve_loop_or_end_return:
2993   } {
2994     \__bnvs_end_return_error:n {
2995       Undefined~n~index
2996     }
2997   }
2998 }
2999 }

```

```

3000 \BNVS_new_conditional:cpnn { if_path_suffix:n } #1 { T, F, TF } {
3001   \__bnvs_seq_get_right:ccTF { path } { suffix } {
3002     \__bnvs_tl_if_eq:cnTF { suffix } { #1 } {
3003       \__bnvs_seq_pop_right:ccT { path } { suffix } { }
3004       \prg_return_true:
3005     } {
3006       \prg_return_false:
3007     }
3008   } {
3009     \__bnvs_tl_clear:c { suffix }
3010     \prg_return_false:
3011   }
3012 }
3013 \BNVS_new:cpn { if_resolve_loop_or_end_return[.length]: } {

```

- Case ...length.

```

3014   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3015     \__bnvs_if_append_length:vcTF { FQ_name } { ans } {
3016       \__bnvs_if_resolve_loop_or_end_return:
3017     } {
3018       \__bnvs_if_resolve_end_return_false:n { NO~length }
3019     }
3020   }
3021 }
3022 \BNVS_new:cpn { if_resolve_loop_or_end_return[.last]: } {

```

- Case ...last.

```

3023   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3024     \__bnvs_if_append_last:vcTF { FQ_name } { ans } {
3025       \__bnvs_if_resolve_loop_or_end_return:
3026     } {
3027       \BNVS_end_return_false:x { NO~last }
3028     }
3029   }
3030 }
3031 \BNVS_new:cpn { if_resolve_loop_or_end_return[.range]: } {

```

- Case ...range.

```

3032   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3033     \__bnvs_if_append_range:vcTF { FQ_name } { ans } {
3034       \BNVS_set:cpn { if_resolve_round_ans: } { \prg_do_nothing: }

```

```

3035     \__bnvs_if_resolve_loop_or_end_return:
3036   } {
3037     \__bnvs_if_resolve_end_return_false:n { NO~range }
3038   }
3039 }
3040 }
3041 \BNVS_new:cpn { if_resolve_loop_or_end_return[.previous]: } {

  • Case ...previous.

3042   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3043     \__bnvs_if_append_previous:vcTF { FQ_name } { ans } {
3044       \__bnvs_if_resolve_loop_or_end_return:
3045     } {
3046       \__bnvs_if_resolve_end_return_false:n { NO~previous }
3047     }
3048   }
3049 }
3050 \BNVS_new:cpn { if_resolve_loop_or_end_return[.next]: } {

  • Case ...next.

3051   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3052     \__bnvs_if_append_next:vcTF { FQ_name } { ans } {
3053       \__bnvs_if_resolve_loop_or_end_return:
3054     } {
3055       \__bnvs_if_resolve_end_return_false:n { NO~next }
3056     }
3057   }
3058 }
3059 \BNVS_new:cpn { if_resolve_loop_or_end_return[.reset]: } {

  • Case ...reset.

3060   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3061     \__bnvs_v_if_greset:vnT { FQ_name } { } { }
3062     \__bnvs_if_append_V:vcTF { FQ_name } { ans } {
3063       \__bnvs_if_resolve_loop_or_end_return:
3064     } {
3065       \__bnvs_if_resolve_end_return_false:n { NO~reset }
3066     }
3067   }
3068 }
3069 \BNVS_new:cpn { if_resolve_loop_or_end_return[.reset_all]: } {

  • Case ...reset_all.

3070   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3071     \__bnvs_if_greset_all:vnT { FQ_name } { } { }
3072     \__bnvs_if_append_V:vcTF { FQ_name } { ans } {
3073       \__bnvs_if_resolve_loop_or_end_return:
3074     } {
3075       \__bnvs_if_resolve_end_return_false:n { NO~reset }
3076     }
3077   }
3078 }

```

```

3079 \BNVS_new:cpn { if_resolve_Fip_end_return_or[...<integer>]:T } #1 {
3080   \BNVS_use:c { if_resolve_Fip_or_end_return[...<integer>]:nF } V {
3081     \BNVS_use:c { if_resolve_Fip_or_end_return[...<integer>]:nF } A {
3082       \BNVS_use:c { if_resolve_Fip_or_end_return[...<integer>]:nF } L {
3083         #1
3084       }
3085     }
3086   }
3087 }

```

```

3088 \BNVS_new:cpn { if_resolve_Fip_or_end_return[...<integer>]:FF } #1 #2 {
3089   \_bnvs_if_get:nvcTF V { a } { b } {

```

The `a` `tl` variable is known, its value is in `b`. We check if it is exactly an overlay set name. If true, the new `FQ_name` is `b`.

```

3090   \_bnvs_if_Fip:cccTF { b } { id } { path } {
3091     \_bnvs_tl_set_eq:cc { FQ_name } { b }
3092     \_bnvs_seq_merge:cc { path } { path_tail }
3093     \_bnvs_seq_set_eq:cc { path_head } { path }
3094     \_bnvs_seq_clear:c { path_tail }
3095     \_bnvs_if_resolve_Fip_loop_or_end_return:
3096   } {
3097     \_bnvs_if_resolve:vcTF { b } { b } {
3098   %%% return resolve the index
3099     \BNVS_use:c {
3100       if_resolve_Fip_or_end_return_true[...<integer>]:v
3101     } { b }
3102   } {
3103     #1
3104   }
3105 } {
3106 } {
3107   #2
3108 }
3109 }

```

```

3110 \BNVS_new:cpn { if_resolve_loop_or_end_return[...<integer>]: } {

```

- Case `...<integer>`.

```

3111 \BNVS_set:cpn { if_resolve_Fip_end_return_or:T } {
3112   \BNVS_use:c { if_resolve_Fip_end_return_or[...<integer>]:T }
3113 }
3114 \_bnvs_if_resolve_path_end_return_false_or:T {
3115   \_bnvs_if_append_index:vvcTF { FQ_name } { suffix } { ans } {
3116     \_bnvs_if_resolve_loop_or_end_return:
3117   } {
3118     \_bnvs_if_resolve_end_return_false:n { NO~integer }
3119   }
3120 }
3121 }
3122 \BNVS_set:cpn { if_resolve_loop_or_end_return[...]: } {

```

- Case

```

3123 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3124   \__bnvs_if_append_V:vcTF { FQ_name } { ans } {
3125     \__bnvs_if_resolve_loop_or_end_return:
3126   } {
3127     \__bnvs_if_resolve_end_return_false:n { NO~value }
3128   }
3129 }
3130 }
3131 \BNVS_set:cpn { if_resolve_loop_or_end_return[...++]: } {

```

- Case ...++.

```

3132 \__bnvs_if_path_suffix:nTF { reset } {
3133   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3134     \__bnvs_v_if_greset:vnT { FQ_name } { } { }
3135     \__bnvs_if_append_v_post:vncTF { FQ_name } { 1 } { ans } {
3136       \__bnvs_if_resolve_loop_or_end_return:
3137     } {
3138       \__bnvs_if_resolve_end_return_false:n { NO~post }
3139     }
3140   }
3141 } {
3142   \__bnvs_if_path_suffix:nTF { reset_all } {
3143     \__bnvs_if_resolve_path_n_end_return_false_or:T {
3144       \__bnvs_if_greset_all:vnT { FQ_name } { } { }
3145       \__bnvs_if_append_v_post:vncTF { FQ_name } { 1 } { ans } {
3146         \__bnvs_if_resolve_loop_or_end_return:
3147       } {
3148         \__bnvs_if_resolve_end_return_false:n { NO~post }
3149       }
3150     }
3151   } {
3152     \__bnvs_if_resolve_path_n_end_return_false_or:T {
3153       \__bnvs_if_append_v_post:vncTF { FQ_name } { 1 } { ans } {
3154         \__bnvs_if_resolve_loop_or_end_return:
3155       } {
3156         \__bnvs_if_resolve_end_return_false:n { NO~post }
3157       }
3158     }
3159   }
3160 }
3161 }
3162 \BNVS_set:cpn { if_resolve_loop_or_end_return[...n=...]: } {

```

- Case ...n=<integer>.

```

3163 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3164   \__bnvs_resolve_base_n:
3165   \__bnvs_if_assign_value:vvTF { FQ_base } { rhs } {
3166     \__bnvs_if_resolve_n_incr:vvncTF {
3167       FQ_name } { FQ_base } { 0 } { ans } {

```

```

3168     \__bnvs_if_resolve_loop_or_end_return:
3169   } {
3170     \__bnvs_if_resolve_end_return_false:n {
3171       NO~n~assignment
3172     }
3173   }
3174 } {
3175   \__bnvs_if_resolve_end_return_false:n {
3176     NO~n~assignment
3177   }
3178 }
3179 }
3180 }
3181 \BNVS_set:cpn { if_resolve_loop_or_end_return[...n+=...]: } {

```

- Case ...n+=*<integer>*.

```

3182 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3183   \__bnvs_resolve_base:n:
3184   \__bnvs_if_append_n_incr:vvvcTF {
3185     FQ_name } { FQ_base } { rhs } { ans } {
3186     \__bnvs_if_resolve_loop_or_end_return:
3187   } {
3188     \__bnvs_if_resolve_end_return_false:n {
3189       NO~n~incrementation
3190     }
3191   }
3192 }
3193 }
3194 \BNVS_set:cpn { if_resolve_loop_or_end_return[...=...]: } {

```

- Case A=*<integer>*. Resolves rhs, on success put the result as value in the cache, then append this value in the ans variable.

```

3195 %%
3196 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3197   \__bnvs_if_assign_value:vvTF { FQ_name } { rhs } {
3198     \__bnvs_if_resolve:vcTF { FQ_name } { ans } {
3199   %   \begin{macrocode}
3200     \__bnvs_if_resolve_loop_or_end_return:
3201   } {
3202     \__bnvs_if_resolve_end_return_false:n {
3203       NO~assignment
3204     }
3205   }
3206 } {
3207   \__bnvs_if_resolve_end_return_false:n {
3208     NO~assignment
3209   }
3210 }
3211 }
3212 }
3213 \BNVS_set:cpn { if_resolve_loop_or_end_return[...+=...]: } {

```

- Case A+=*<integer>*.

```

3214 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3215   \__bnvs_if_append_v_incr:vvctf { FQ_name } { rhs } { ans } {
3216     \__bnvs_if_resolve_loop_or_end_return:
3217   } {
3218     \__bnvs_if_resolve_end_return_false:n {
3219       NO~incremented~value
3220     }
3221   }
3222 }
3223 }

```

```

\__bnvs_resolve_base: \__bnvs_resolve_base:
\__bnvs_resolve_base_n: \__bnvs_resolve_base_n:

```

If the `path_base` sequence is not empty, pops the rightmost item of `path_base` which is `n` then appends to the `FQ_base` variable the remaining components of `path_base` as a dotted path suffix. Implementation detail: use local `tl` variable `a`.

```

3224 \BNVS_new:cpn { resolve_base: } {
3225   \__bnvs_seq_if_empty:cF { path_base } {
3226     \__bnvs_tl_put_right:cx { FQ_base } {
3227       . \__bnvs_seq_use:cn { path_base } { . }
3228     }
3229   }
3230 }
3231 \BNVS_new:cpn { resolve_base_n: } {
3232   \__bnvs_seq_if_empty:cF { path_base } {
3233     \__bnvs_seq_pop_right:cc { path_base } { a }
3234     \__bnvs_seq_if_empty:cF { path_base } {
3235       \__bnvs_tl_put_right:cx { FQ_base } {
3236         . \__bnvs_seq_use:cn { path_base } { . }
3237       }
3238     }
3239   }
3240 }
3241 \BNVS_new:cpn { if_resolve_loop_or_end_return[...++n]: } {

```

- Case ...++n.

```

3242 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3243   \__bnvs_resolve_base:
3244   \__bnvs_if_append_n_incr:vvncTF { FQ_name } { FQ_base } { 1 } { ans } {
3245     \__bnvs_if_resolve_loop_or_end_return:
3246   } {
3247     \__bnvs_if_resolve_end_return_false:n { NO/...++n }
3248   }
3249 }
3250 }
3251 \BNVS_set:cpn { if_resolve_loop_or_end_return[++...n]: } {

```

- Case ++...n.

```

3252 \__bnvs_if_resolve_path_n_end_return_false_or:T {
3253   \__bnvs_resolve_base_n:
3254   \__bnvs_if_append_n_incr:vnctf { FQ_name } { FQ_base } { 1 } { ans } {
3255     \__bnvs_if_resolve_loop_or_end_return:
3256   } {
3257     \__bnvs_if_resolve_end_return_false:n { NO~++...n }
3258   }
3259 }
3260 }
3261 \BNVS_new:cpn { if_resolve_loop_or_end_return[++...]: } {

```

- Case ++....

```

3262 \__bnvs_if_path_suffix:nTF { reset } {
3263   \__bnvs_if_resolve_path_n_end_return_false_or:T {
3264     \__bnvs_if_append_v_incr:vncTF { FQ_name } { 1 } { ans } {
3265       \__bnvs_v_if_greset:vnT { FQ_name } { } { }
3266       \__bnvs_if_resolve_loop_or_end_return:
3267     } {
3268       \__bnvs_v_if_greset:vnT { FQ_name } { } { }
3269       \__bnvs_if_resolve_end_return_false:n { No~increment }
3270     }
3271   }
3272 } {
3273   \__bnvs_if_path_suffix:nTF { reset_all } {
3274     \__bnvs_if_resolve_path_n_end_return_false_or:T {
3275       \__bnvs_if_append_v_incr:vncTF { FQ_name } { 1 } { ans } {
3276         \__bnvs_if_greset_all:vnT { FQ_name } { } { }
3277         \__bnvs_if_resolve_loop_or_end_return:
3278       } {
3279         \__bnvs_if_greset_all:vnT { FQ_name } { } { }
3280         \__bnvs_if_resolve_end_return_false:n { No~increment }
3281       }
3282     }
3283   } {
3284     \__bnvs_if_resolve_path_n_end_return_false_or:T {
3285       \__bnvs_if_append_v_incr:vncTF { FQ_name } { 1 } { ans } {
3286         \__bnvs_if_resolve_loop_or_end_return:
3287       } {
3288         \__bnvs_if_resolve_end_return_false:n { No~increment }
3289       }
3290     }
3291   }
3292 }
3293 }

```

```
\__bnvs_if_resolve_query:ncTF \__bnvs_if_resolve_query:ncTF {<overlay query>} {<tl core>} {<yes code>}
{<no code>}
```

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

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

(End of definition for `\l__bnvs_V_tl`.)

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

(End of definition for `\l__bnvs_A_tl`.)

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

(End of definition for `\l__bnvs_Z_tl`.)

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

(End of definition for `\l__bnvs_L_tl`.)

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

(End of definition for `\c__bnvs_A_cln_Z_regex`.)

```
3294 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
3295   \A \s* (?
    • 2 → V
3296     ( [^:]+? )
    • 3, 4, 5 → A : Z? or A :: L?
3297     | (? ( [^:]+? ) \s* : (? \s* ( [^:]*? ) | : \s* ( [^:]*? ) ) )
    • 6, 7 → ::(L:Z)?
3298     | (? :: \s* (? ( [^:]+? ) \s* : \s* ( [^:]+? ) )? )
    • 8, 9 → :(Z::L)?
3299     | (? : \s* (? ( [^:]+? ) \s* :: \s* ( [^:]*? ) )? )
3300   )
3301   \s* \Z
3302 }
```

```
3303 \BNVS_set:cpn { resolve_query_end_return_true: } {
3304   \BNVS_end:
3305   \prg_return_true:
3306 }
3307 \BNVS_new:cpn { resolve_query_end_return_false: } {
```

```

3308 \BNVS_end:
3309 \prg_return_false:
3310 }
3311 \BNVS_new:cpn { resolve_query_end_return_false:n } #1 {
3312 \BNVS_end:
3313 \prg_return_false:
3314 }
3315 \BNVS_new:cpn { if_resolve_query_return_false:n } #1 {
3316 \prg_return_false:
3317 }
3318 \BNVS_new:cpn { resolve_query_error_return_false:n } #1 {
3319 \BNVS_error:x { #1 }
3320 \__bnvs_if_resolve_query_return_false:
3321 }
3322 \BNVS_new:cpn { if_resolve_query_return_unreachable: } {
3323 \__bnvs_resolve_query_error_return_false:n { UNREACHABLE }
3324 }
3325 \BNVS_new:cpn { if_blank:cTF } #1 {
3326 \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3327 }
3328 \BNVS_new_conditional:cpnn { if_match_pop_left:c } #1 { T, F, TF } {
3329 \BNVS_tl_use:nc {
3330 \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
3331 } { #1 } {
3332 \prg_return_true:
3333 } {
3334 \prg_return_false:
3335 }
3336 }

```

__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.

```

3337 \BNVS_new_conditional:cpnn { if_resolve_query_branch: } { T, F, TF } {
At start, we ignore the whole match.
3338 \__bnvs_if_match_pop_left:cT V {
3339 \__bnvs_if_match_pop_left:cT V {
3340 \__bnvs_if_blank:cTF V {
3341 \__bnvs_if_match_pop_left:cT A {
3342 \__bnvs_if_match_pop_left:cT Z {
3343 \__bnvs_if_match_pop_left:cT L {
3344 \__bnvs_if_blank:cTF A {
3345 \__bnvs_if_match_pop_left:cT L {
3346 \__bnvs_if_match_pop_left:cT Z {
3347 \__bnvs_if_blank:cTF L {

```

```

3348         \__bnvs_if_match_pop_left:cT Z {
3349             \__bnvs_if_match_pop_left:cT L {
3350                 \__bnvs_if_blank:cTF L {
3351                     \BNVS_use:c { if_resolve_query_return[:Z]: }
3352                 } {
3353                     \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3354                 }
3355             }
3356         }
3357     } {
3358         \__bnvs_if_blank:cTF Z {
3359     \__bnvs_resolve_query_error_return_false:n { Missing-first~or~last }
3360         } {
3361             \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3362         }
3363     }
3364 }
3365 }
3366 } {
3367     \__bnvs_if_blank:cTF Z {
3368         \__bnvs_if_blank:cTF L {
3369             \BNVS_use:c { if_resolve_query_return[A:]: }
3370         } {
3371             \BNVS_use:c { if_resolve_query_return[A::L]: }
3372         }
3373     } {
3374         \__bnvs_if_blank:cTF L {
3375             \BNVS_use:c { if_resolve_query_return[A:Z]: }
3376         } {
3377             \__bnvs_if_resolve_query_return_unreachable:
3378         }
3379     }
3380 }
3381 }
3382 }
3383 }
3384 } {
3385     \BNVS_use:c { if_resolve_query_return[V]: }
3386 }
3387 }
3388 }
3389 }
3390 \BNVS_new:cpn { if_resolve_query_return[V]: } {

```

Single value

```

3391 \__bnvs_if_resolve:vcTF { V } { ans } {
3392     \prg_return_true:
3393 } {
3394     \prg_return_false:
3395 }
3396 }
3397 \BNVS_new:cpn { if_resolve_query_return[A:Z]: } {

```

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

```

3398 \__bnvs_if_resolve:vcTF { A } { ans } {
3399   \__bnvs_tl_put_right:cn { ans } { - }
3400   \__bnvs_if_append:vcTF { Z } { ans } {
3401     \prg_return_true:
3402   } {
3403     \prg_return_false:
3404   }
3405 } {
3406   \prg_return_false:
3407 }
3408 }
3409 \BNVS_new:cpn { if_resolve_query_return[A::L]: } {

```

☛ $\langle first \rangle :: \langle length \rangle$ range

```

3410 \__bnvs_if_resolve:vcTF { A } { A } {
3411   \__bnvs_if_resolve:vcTF { L } { ans } {
3412     \__bnvs_tl_put_right:cn { ans } { + }
3413     \__bnvs_tl_put_right:cv { ans } { A }
3414     \__bnvs_tl_put_right:cn { ans } { -1 }
3415     \__bnvs_round_ans:
3416     \__bnvs_tl_put_left:cn { ans } { - }
3417     \__bnvs_tl_put_left:cv { ans } { A }
3418   \prg_return_true:
3419 } {
3420   \prg_return_false:
3421 }
3422 } {
3423   \prg_return_false:
3424 }
3425 }
3426 \BNVS_new:cpn { if_resolve_query_return[A:]: } {

```

☛ $\langle first \rangle :$ and $\langle first \rangle ::$ range

```

3427 \__bnvs_if_resolve:vcTF { A } { ans } {
3428   \__bnvs_tl_put_right:cn { ans } { - }
3429   \prg_return_true:
3430 } {
3431   \prg_return_false:
3432 }
3433 }
3434 \BNVS_new:cpn { if_resolve_query_return[:Z::L]: } {

```

☛ $: \langle last \rangle :: \langle length \rangle$ or $:: \langle length \rangle : \langle last \rangle$ range

```

3435 \__bnvs_if_resolve:vcTF { Z } { Z } {
3436   \__bnvs_if_resolve:vcTF { L } { ans } {
3437     \__bnvs_tl_put_left:cn { ans } { 1- }
3438     \__bnvs_tl_put_right:cn { ans } { + }
3439     \__bnvs_tl_put_right:cv { ans } { Z }
3440     \__bnvs_round_ans:
3441     \__bnvs_tl_put_right:cn { ans } { - }
3442     \__bnvs_tl_put_right:cv { ans } { Z }
3443   \prg_return_true:
3444 } {

```

```

3445     \prg_return_false:
3446   }
3447 } {
3448   \prg_return_false:
3449 }
3450 }
3451 \BNVS_new:cpn { if_resolve_query_return[:]: } {
  : or :: range
3452   \__bnvs_tl_set:cn { ans } { - }
3453   \prg_return_true:
3454 }
3455 \BNVS_new:cpn { if_resolve_query_return[:Z]: } {
  : <last> range
3456   \__bnvs_tl_set:cn { ans } { - }
3457   \__bnvs_if_append:vcTF { Z } { ans } {
3458     \prg_return_true:
3459   } {
3460     \prg_return_false:
3461   }
3462 }

```

```

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

```

Evaluate only one query.

```

3463 \BNVS_new_conditional:cpnn { if_resolve_query:nc } #1 #2 { T, F, TF } {
3464   \__bnvs_call_greset:
3465   \__bnvs_match_if_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
3466     \BNVS_begin:
3467     \__bnvs_if_resolve_query_branch:TF {
3468       \BNVS_tl_set_after:ncv {
3469         \BNVS_end:
3470       } { #2 } { ans }
3471       \prg_return_true:
3472     } {
3473       \BNVS_end:
3474       \prg_return_false:
3475     }
3476   } {
3477     \BNVS_error:n { Syntax-error:~#1 }
3478     \BNVS_end:
3479     \prg_return_false:
3480   }
3481 }

```

```

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

```

```

__bnvs_resolve_queries:nc  __bnvs_resolve_queries:nc {<overlay query list>} {<ans>}

```

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

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

(End of definition for `\l__bnvs_query_seq`.)

`\l__bnvs_ans_seq` Storage for the evaluated result.

(End of definition for `\l__bnvs_ans_seq`.)

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

```

3482 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End of definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

3483 \BNVS_new_conditional:cpnn { if_resolve_queries:nc } #1 #2 { TF } {
3484   \BNVS_begin:

```

Local variables cleared

```

3485   __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`.

```

3486   \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.

```

3487   \BNVS_set:cpn { end_return: } {
3488     __bnvs_seq_if_empty:cTF { ans } {
3489       \BNVS_end:
3490     } {
3491       \exp_args:Nnx
3492       \use:n {
3493         \BNVS_end:
3494         __bnvs_tl_put_right:cn { #2 }
3495       } { \__bnvs_seq_use:cn { ans } , }
3496     }
3497     \prg_return_true:
3498   }
3499   __bnvs_seq_map_inline:cn { query } {
3500     __bnvs_tl_clear:c { ans }
3501     __bnvs_if_resolve_query:ncTF { ##1 } { ans } {
3502       __bnvs_tl_if_empty:cF { ans } {
3503         __bnvs_seq_put_right:cv { ans } { ans }
3504       }

```

```

3505     } {
3506         \seq_map_break:n {
3507             \BNVS_set:cpn { end_return: } {
3508                 \BNVS_end:
3509                 \BNVS_error:n { Circular/Undefined~dependency~in~#1}
3510                 \exp_args:Nnx
3511                 \use:n {
3512                     \BNVS_end:
3513                     \__bnvs_tl_put_right:cn { #2 }
3514                 } { \__bnvs_seq_use:cn { ans } , }
3515                 \prg_return_false:
3516             }
3517         }
3518     }
3519 }
3520 \__bnvs_end_return:

```

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

```

3521 }

3522 \BNVS_new:cpn { resolve_queries:nc } #1 #2 {
3523     \BNVS_begin:

```

Local variables cleared

```

3524     \__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 $\langle first \rangle :: \langle last \rangle$ range expressions as well. We first split the expression around commas, into `\l_query_seq`.

```

3525     \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.

```

3526     \__bnvs_seq_map_inline:cn { query } {
3527         \__bnvs_tl_clear:c { ans }
3528         \__bnvs_if_resolve_query:ncTF { ##1 } { ans } {
3529             \__bnvs_seq_put_right:cv { ans } { ans }
3530         } {
3531             \seq_map_break:n {
3532                 \BNVS_error:n { Circular/Undefined~dependency~in~#1}
3533             }
3534         }
3535     }

```

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

```

3536     \exp_args:Nnx
3537     \use:n {
3538         \BNVS_end:
3539         \__bnvs_tl_put_right:cn { #2 }
3540     } { \__bnvs_seq_use:cn { ans } , }
3541 }

```

`\BeanovesResolve` `\BeanovesResolve [⟨setup⟩] {⟨overlay queries⟩}`

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

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

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

```

3542 \NewDocumentCommand \BeanovesResolve { 0{} m } {
3543   \BNVS_begin:
3544   \keys_define:nn { BeanovesResolve } {
3545     in:N .tl_set:N = \l__bnvs_resolve_in_tl,
3546     in:N .initial:n = { },
3547     show .bool_set:N = \l__bnvs_resolve_show_bool,
3548     show .default:n = true,
3549     show .initial:n = false,
3550   }
3551   \keys_set:nn { BeanovesResolve } { #1 }
3552   \__bnvs_tl_clear:c { ans }
3553   \__bnvs_if_resolve_queries:ncTF { #2 } { ans } {
3554     \__bnvs_tl_if_empty:cTF { resolve_in } {
3555       \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3556         \BNVS_tl_use:Nv \BNVS_end: { ans }
3557       } {
3558         \BNVS_end:
3559       }
3560     } {
3561       \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3562         \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3563           \BNVS_end:
3564           \tl_set:Nn ##1 { ##2 }
3565           ##2
3566         }
3567         \BNVS_tl_use:nv {
3568           \exp_last_unbraced:Nv \BNVS_end:Nn \l__bnvs_resolve_in_tl
3569         } { ans }
3570       } {
3571         \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3572           \BNVS_end:
3573           \tl_set:Nn ##1 { ##2 }
3574         }
3575         \BNVS_tl_use:nv {
3576           \exp_last_unbraced:Nv \BNVS_end:Nn \l__bnvs_resolve_in_tl
3577         } { ans }
3578       }
3579     }
3580   } {}
3581 }

```


6.26 Resetting counters

<code>\BeanovesReset</code>	<code>\BeanovesReset</code> [<i>⟨first value⟩</i>] { <i>⟨ref⟩</i> }
<code>\BeanovesReset*</code>	<code>\BeanovesReset*</code> [<i>⟨first value⟩</i>] { <i>⟨ref⟩</i> }

Forwards to `__bnvs_v_if_greset:nnF` or `__bnvs_if_greset_all:nnF` when starred.

```

3582 \NewDocumentCommand \BeanovesReset { s O{} m } {
3583   \__bnvs_if_id_FQ_name_n_get:nTF { #3 } {
3584     \BNVS_tl_use:nv {
3585       \IfBooleanTF { #1 } {
3586         \__bnvs_if_greset_all:nnF
3587       } {
3588         \__bnvs_v_if_greset:nnF
3589       }
3590     } { FQ_name } { #2 } {
3591 %     \__bnvs_warning:n { Unknown~name:~#3 }
3592   }
3593 } {
3594   \__bnvs_warning:n { Bad~name:~#3 }
3595 }
3596 \ignorespaces
3597 }
```

6.27 Saving the beamer pauses counter

<code>\BeanovesSavePauses</code>	<code>\BeanovesSavePauses</code> { <i>⟨ref⟩</i> }
----------------------------------	---

Forwards to `__bnvs_save:n`.

```

3598 \NewDocumentCommand \BeanovesSavePauses { m } {
3599   \__bnvs_save:n { #1 }
3600 }
3601 \makeatletter
3602 \BNVS_new:cpn { save:n } #1 {
3603   \cs_if_exist:NT \c@beamerpauses {
3604     \exp_args:Nnx \__bnvs_parse:nn { #1 } { \the\c@beamerpauses }
3605   }
3606 }
3607 \makeatother
3608 \ExplSyntaxOff
```

See <https://latex.org/forum/viewtopic.php?t=25777>