

# beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0      2024/01/11

## Abstract

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

## Contents

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

<b>6</b>	<b>Implementation</b>	<b>9</b>
6.1	Package declarations	10
6.2	Facility layer: definitions and naming	10
6.3	logging	13
6.4	Facility layer: Variables	14
6.4.1	Regex	20
6.4.2	Token lists	22
6.4.3	Strings	26
6.4.4	Sequences	27
6.4.5	Integers	28
6.5	Debug facilities	29
6.6	Debug messages	29
6.7	Testing facilities	29
6.8	Local variables	29
6.9	Infinite loop management	31
6.10	Overlay specification	31
6.10.1	Registration	31
6.10.2	Basic functions	36
6.10.3	Functions with cache	41
6.11	Implicit value counter	43
6.12	Regular expressions	44
6.13	beamer.cls interface	47
6.14	Defining named slide ranges	48
6.14.1	Square brackets	55
6.14.2	Root level	56
6.14.3	List specifiers	57
6.15	Scanning named overlay specifications	65
6.16	Resolution	70
6.17	Evaluation bricks	72
6.18	Index counter	87
6.19	Value counter	89
6.20	Functions for the resolution	92
6.21	Resetting counters and values	111

## 1 Installation

### 1.1 Package manager

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

### 1.2 Manual installation

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

### 1.3 Usage

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

## 2 Minimal example

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

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

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

## 3 Named overlay sets

### 3.1 Presentation

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

### 3.2 Named overlay reference

`A.1`, `C.2` are *named overlay references*, as well as `A` and `Y!C.2`. More precisely, they are string identifiers, each one referencing a well defined static integer or range to be used in `beamer` overlay specifications. They have 3 components:

1.  $\langle \text{frame id} \rangle !$ , like `X!`, optional
2.  $\langle \text{short name} \rangle$  like `A`, required
3.  $\langle c_1 \rangle \dots \langle c_j \rangle$  like `.B.C`, optional ( $j = 0$ ), globally denoted as *dotted path*.

The *frame ids*, *short names* and  $\langle c \rangle$ 's are alphanumerical case sensitive identifiers, with possible underscores but with no space. Unicode symbols above U+00A0 are allowed if the underlying `TEX` engine supports it. Only the *frame id* is allowed to be empty, in which case it may apply to any common frame. The *short names* must not consist of only lowercase letters.

The mapping from *named overlay references* to sets of 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 custom `beanoves` option of this environment.

---

<code>\Beanoves</code>	<code>\Beanoves{\langle ref_1 \rangle = \langle spec_1 \rangle, \dots, \langle ref_j \rangle = \langle spec_j \rangle}</code>
------------------------	---

---

<code>\Beanoves*</code>	
-------------------------	--

Each  $\langle \text{ref} \rangle$  key is a *named overlay reference* whereas each  $\langle \text{spec} \rangle$  is an *overlay set specifier*. When the same  $\langle \text{ref} \rangle$  key is used multiple times, only the last one is taken into account. Notice that  $\langle \text{ref} \rangle = 1$  can be shortened to  $\langle \text{ref} \rangle$ .

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside `LATEX` environments, each new call cumulates with the previous one. Notice that the argument of this function can contain macros: they will be exhaustively expanded at resolution time<sup>1</sup>.

---

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

---

<sup>1</sup>Precision is needed about the exact time when the expansion occurs.

The `\Beanoves` arguments take precedence over both the `\Beanoves*` arguments and the `beanoves` options. This allows to provide an overlay name only when not already defined, which is helpfull when the very same frame source is included multiple times in different contexts.

### 3.3.1 Value specifiers

Hereafter  $\langle \text{value} \rangle$  denotes a numerical expression.

$\langle \text{ref} \rangle = \langle \text{value} \rangle$ , a *value specifier* for a single number. When omitted it defaults to 1.

$\langle \text{ref} \rangle = [\langle \text{index}_1 \rangle = \langle \text{value}_1 \rangle, \dots, \langle \text{index}_j \rangle = \langle \text{value}_j \rangle]$ , a *value specifier* partially defined in extension.  $\langle \text{index} \rangle$  denotes an explicit positive integer. This removes the actual  $\langle \text{ref} \rangle$  and executes  $\langle \text{ref} \rangle . \langle \text{index}_k \rangle = \langle \text{value}_k \rangle$  for  $1 \leq k \leq j$ . When  $\langle \text{index}_k \rangle =$  is omitted,  $\langle i \rangle = \langle \text{value}_k \rangle$  is executed where  $\langle i \rangle$  is the smallest positive integer such that  $\langle \text{ref} \rangle . \langle i \rangle$  is not already defined.

The numerical expressions are evaluated and then rounded using `\fp_eval:n`. They can contain mathematical functions and *named overlay references* defined above but should not contain *named overlay references* to *value specifiers*.

The corresponding overlay set can be seen as a *value counter*.

### 3.3.2 Range specifiers

Hereafter  $\langle \text{first} \rangle$ ,  $\langle \text{last} \rangle$  and  $\langle \text{length} \rangle$  are *value specifiers*.

$\langle \text{ref} \rangle = \langle \text{first} \rangle :$ ,

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

$\langle \text{ref} \rangle = \langle \text{first} \rangle : \langle \text{last} \rangle$ ,

$\langle \text{ref} \rangle = \langle \text{first} \rangle :: \langle \text{length} \rangle$ ,

$\langle \text{ref} \rangle = : \langle \text{last} \rangle :: \langle \text{length} \rangle$ ,

$\langle \text{ref} \rangle = :: \langle \text{length} \rangle : \langle \text{last} \rangle$ , are variants for the same finite range of signed integers starting at and including  $\langle \text{first} \rangle$ , ending at and including  $\langle \text{last} \rangle$ , provided  $\langle \text{first} \rangle + \langle \text{length} \rangle = \langle \text{last} \rangle + 1$ .  $\langle \text{first} \rangle$  can be omitted, in which case it defaults to 1. Additionally  $: \langle \text{last} \rangle$  and  $:: \langle \text{length} \rangle$  are then equivalent.

$\langle \text{ref} \rangle = : \langle \text{last} \rangle$ ,

$\langle \text{ref} \rangle = :: \langle \text{length} \rangle$ , are syntactic sugar when  $\langle \text{first} \rangle$  is 1.

### 3.3.3 List specifiers

Here is the **implementation**.

$\langle \text{ref} \rangle = \{ \langle \text{value} \rangle \}$ , synonym of  $\langle \text{ref} \rangle = \langle \text{value} \rangle$ .

$\langle \text{ref} \rangle = \{ \langle \text{def}_1 \rangle, \dots, \langle \text{def}_j \rangle \}$ , where  $\langle \text{def}_k \rangle$ ,  $1 \leq k \leq j$ , is one of

- $\langle \text{index} \rangle = \langle \text{spec} \rangle$ ,
- $\langle \text{spec} \rangle$  for  $\langle i \rangle = \langle \text{spec} \rangle$ ,  $\langle i \rangle$  being the smallest positive integer such that  $\langle \text{ref} \rangle . \langle i \rangle$  is not already defined.

for value or range specifiers. The first step is to remove previous  $\langle \text{ref} \rangle$  related definitions, then execute the various  $\langle \text{ref} \rangle . \langle \text{index} \rangle = \langle \text{spec} \rangle$  definitions in the order given.

$\langle \text{ref} \rangle = \{ \{ \langle \text{def}_1 \rangle, \dots, \langle \text{def}_j \rangle \} \}$ , where  $\langle \text{def}_k \rangle$ ,  $1 \leq k \leq j$ , is one of

- $\langle \text{name} \rangle = \langle \text{spec} \rangle$ ,
- $\langle \text{name} \rangle$  for  $\langle \text{name} \rangle = 1$ ,

The first step is to remove previous  $\langle \text{ref} \rangle$  related definitions, then execute the various  $\langle \text{ref} \rangle . \langle \text{name} \rangle = \langle \text{spec} \rangle$  definitions in the order given.  $\langle \text{spec} \rangle$  is any specifier.

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

The  $\langle queries \rangle$  argument is a comma separated list of individual  $\langle query \rangle$ 's processed from left to right as explained below. Notice that nesting a  $?(...)$  query expression 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 to the nearest integer to feed **beamer** with either ranges or numbers. When the *query reference* is a previously declared  $\langle \text{ref} \rangle$ , like  $X$  after  $X=1$ , it is simply replaced by the corresponding declared  $\langle \text{value} \rangle$ , here 1. Otherwise, we use *implicit overlay queries* and their *resolution rules* depending on the definition of the named overlay set. Hereafter  $\langle i \rangle$  denotes a signed integer whereas  $\langle \text{value} \rangle$ ,  $\langle \text{first} \rangle$ ,  $\langle \text{last} \rangle$  and  $\langle \text{length} \rangle$  stand for raw integers or more general numerical expressions that are evaluated beforehand.

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

### 4.1 Range overlay queries

$\langle \text{ref} \rangle = \langle \text{first} \rangle$ : as well as  $\langle \text{first} \rangle ::$  defines a range limited from below:

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

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

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

For a range limited from both above and below:

overlay query	resolution
$\langle \text{ref} \rangle$	$\langle \text{first} \rangle - \langle \text{last} \rangle$
$\langle \text{ref} \rangle . 1$	$\langle \text{first} \rangle$
$\langle \text{ref} \rangle . 2$	$\langle \text{first} \rangle + 1$
$\langle \text{ref} \rangle . \langle i \rangle$	$\langle \text{first} \rangle + \langle i \rangle - 1$
$\langle \text{ref} \rangle . \text{previous}$	$\langle \text{first} \rangle - 1$
$\langle \text{ref} \rangle . \text{first}$	$\langle \text{first} \rangle$
$\langle \text{ref} \rangle . \text{last}$	$\langle \text{last} \rangle$
$\langle \text{ref} \rangle . \text{next}$	$\langle \text{last} \rangle + 1$
$\langle \text{ref} \rangle . \text{length}$	$\langle \text{length} \rangle$

Notice that the resolution of  $\langle \text{ref} \rangle$  is a beamer range and not an algebraic difference, 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)          == 3-8,
7 \BeanovesResolve[show] (A.1)       == 3,
8 \BeanovesResolve[show] (A.-1)      == 1,
9 \BeanovesResolve[show] (A.previous) == 2,
10 \BeanovesResolve[show] (A.first)   == 3,
11 \BeanovesResolve[show] (A.last)    == 8,
12 \BeanovesResolve[show] (A.next)    == 9,
13 \BeanovesResolve[show] (A.length)  == 6,
14 \end{frame}

```

$\langle \text{ref} \rangle = \{ \dots \}$  for an indexed list of specifications. The resolution of the overlay query  $\langle \text{ref} \rangle$  gives the resolution of each item of the list, separated by commas, for the largest consecutive indices starting a 1.

$\langle \text{ref} \rangle = \{ \{ \dots \} \}$  for a named list of specifications. The resolution of the overlay query  $\langle \text{ref} \rangle$  gives the resolution of each item of the list, in the given order, separated by commas. Notice that each item may in turn be resolved into a comma separated list of beamer ranges.

## 4.2 Value counter queries

$\langle \text{ref} \rangle = \langle \text{value} \rangle$  defines a counter value.

overlay query	resolution
$\langle \text{ref} \rangle$	$\langle \text{value} \rangle$
$\langle \text{ref} \rangle.1$	$\langle \text{value} \rangle$
$\langle \text{ref} \rangle.2$	$\langle \text{value} \rangle + 1$
$\langle \text{ref} \rangle.\langle i \rangle$	$\langle \text{value} \rangle + \langle i \rangle - 1$
$\langle \text{ref} \rangle.\text{previous}$	$\langle \text{value} \rangle - 1$
$\langle \text{ref} \rangle.\text{first}$	$\langle \text{value} \rangle$
$\langle \text{ref} \rangle.\text{last}$	$\langle \text{value} \rangle$
$\langle \text{ref} \rangle.\text{next}$	$\langle \text{value} \rangle + 1$

Additionally, resolution rules are provided for dedicated *overlay queries*, here  $\langle \text{ref} \rangle$  is considered a standard programming variable:

$\langle \text{ref} \rangle = \langle \text{integer expression} \rangle$ , resolve  $\langle \text{integer expression} \rangle$  into  $\langle \text{integer} \rangle$ , assign it to the  $\langle \text{ref} \rangle$  and use it. It defines  $\langle \text{ref} \rangle$  globally if not already done. Here  $\langle \text{integer expression} \rangle$  is the longest character sequence with no space<sup>2</sup>.

$\langle \text{ref} \rangle += \langle \text{integer expression} \rangle$ , resolve  $\langle \text{integer expression} \rangle$  into  $\langle \text{integer} \rangle$ , advance  $\langle \text{ref} \rangle$  by  $\langle \text{integer} \rangle$  and use the result.

$++\langle \text{ref} \rangle$ , increment  $\langle \text{ref} \rangle$  by 1 and use it.

$\langle \text{ref} \rangle ++$ , use  $\langle \text{ref} \rangle$  and then increment it by 1.

This can be used for an indirection.

```

1 \Beanoves {
2 A = 1,
3 B = { 10 = 100 },
4 C = 10,
5 }
6 \begin{frame} {Frame \insertframenum} {Slide \insertslidenumber}
7 \ttfamily
8 \BeanovesResolve[show](A.C) == \BeanovesResolve[show](A.10) == 10,
9 \BeanovesResolve[show](B.C) == \BeanovesResolve[show](B.10) == 100,
10 \BeanovesResolve[show](A.C+=10) == \BeanovesResolve[show](A.20) == 20,
11 \BeanovesResolve[show](B.C) == \BeanovesResolve[show](B.20) == 20,
12 \end{frame}

```

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

For each new frame, these counters are reset to the value they were initialized with. Sometimes, resetting the counter manually is necessary, for example when managing tikz overlay material.

---

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

---

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

<sup>2</sup>The parser for algebraic expression is very rudimentary.



### 4.3 Dotted paths

Previous overlay queries may fail simply because  $\langle \text{ref} \rangle$  is not defined. If the *dotted path* is empty, an error is raised and the resolution returns 0.

If the *dotted path* is not empty,  $\langle \text{ref} \rangle$  is  $\langle \text{name} \rangle.\langle c_1 \rangle \dots \langle c_j \rangle$ . The indirection resolution takes place: the path is split into  $\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_k \rangle$  and  $\langle c_{k+1} \rangle \dots \langle c_j \rangle$ , where  $0 \leq k \leq j$ ,  $\langle \text{name} \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_k \rangle$  is defined,  $\langle c_{k+1} \rangle \dots \langle c_j \rangle$  is defined as a *value counter*, and  $k$  is the largest. Then  $\langle c_{k+1} \rangle \dots \langle c_j \rangle$  is resolved into  $\langle \text{integer} \rangle$ , taking into account the *value counter* resolution rules, and  $\langle \text{name} \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_k \rangle.\langle \text{integer} \rangle$  is resolved in turn.

If no such  $k$  exists, the replacement resolution takes place: the longest reference  $\langle \text{name} \rangle.\langle c_1 \rangle.\langle c_2 \rangle \dots \langle c_k \rangle$ , where  $0 \leq k \leq j$ , is first replaced by its definition  $\langle \text{name}' \rangle.\langle c'_1 \rangle \dots \langle c'_l \rangle$  if any and then the modified overlay query is resolved with preceding rules as well as this one. For example, with `\Beanoves{A.B=D, D.C=E}`, `A.B.C` is resolved like `E`.

If this resolution still fails, an error is raised and the resolution returns 0.

Care should be taken to avoid circular dependencies.

### 4.4 The beamer counters

While inside a `frame` environment, it is possible to save the current value of the `beamerpauses` counter that controls whether elements should appear on the current slide. For that, we can execute one of `\Beanoves{\langle \text{ref} \rangle=\text{pauses}}` or in a query `?(\dots(\langle \text{ref} \rangle=\text{pauses})\dots)`. Then later on, we can use `?(\dots(\langle \text{ref} \rangle)\dots)` to refer to this saved value in the same frame<sup>3</sup>. Next frame source is an example of usage.

```

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

```

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

### 4.5 Multiple queries

It is possible to replace the comma separated list `?(\langle \text{query}_1 \rangle), \dots, ?(\langle \text{query}_j \rangle)` with the shorter `?(\langle \text{query}_1 \rangle, \dots, \langle \text{query}_j \rangle)`.

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

## 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* and if the resolution fails an empty *frame id*. At the beginning of each frame, the *last frame id* is set to the *frame id* of the current frame, which is denoted *current frame id* and is empty by default. Then it gets updated after each named reference resolution where a *frame id* is explicitly given. For example, the first time `A.1` reference is resolved within a given frame, it is first translated to `<last frame id>!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 `<frame id>`, use the new `beanoves` id option of the `beamer` frame environment.

---

```
beanoves id beanoves id=<frame id>,
```

---

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

## 4.7 Resolution command

---

```
\BeanovesResolve \BeanovesResolve [<setup>] {<queries>}
```

---

This function resolves the `<queries>`, which are like the argument of `?(...)` instructions: a comma separated list of single `<query>`'s. The optional `<setup>` is a key-value:

`show` the result is left into the input stream

`in:N=<command>` the result is stored into `<command>`.

## 5 Support

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

## 6 Implementation

Identify the internal prefix (L<sup>A</sup>T<sub>E</sub>X3 DocStrip convention, unused).

```
1 <MY=bnvs>
```

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

## 6.1 Package declarations

```

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

```

## 6.2 Facility layer: definitions and naming

In order to make the code shorter and easier to read during development, we add a layer over  $\text{\LaTeX}3$ . 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 {&lt;cs core name&gt;}</code>
<code>\BNVS_l:cn</code>	<code>\BNVS_l:cn {&lt;local variable core name&gt;} {&lt; type &gt;}</code>
<code>\BNVS_g:cn</code>	<code>\BNVS_g:cn {&lt;global variable core name&gt;} {&lt; type &gt;}</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 {&lt;cs name&gt;}</code>
<code>\BNVS_use_raw:Nc</code>	<code>\BNVS_use_raw:Nc &lt;function&gt; {&lt;cs name&gt;}</code>
<code>\BNVS_use_raw:nc</code>	<code>\BNVS_use_raw:nc {&lt;tokens&gt;} {&lt;cs name&gt;}</code>
<code>\BNVS_use:c</code>	<code>\BNVS_use:c {&lt;cs core&gt;}</code>
<code>\BNVS_use:Nc</code>	<code>\BNVS_use:Nc &lt;function&gt; {&lt;cs core&gt;}</code>
<code>\BNVS_use:nc</code>	<code>\BNVS_use:nc {&lt;tokens&gt;} {&lt;cs core&gt;}</code>

---

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

```

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

```

```

16 \cs_new:Npn \BNVS_use:c #1 {
17   \BNVS_use_raw:c { \BNVS:c { #1 } }
18 }

19 \cs_new:Npn \BNVS_use_raw:NN #1 #2 {
20   #1 #2
21 }

22 \cs_new:Npn \BNVS_use_raw:nN #1 #2 {
23   #1 #2
24 }

25 \cs_new:Npn \BNVS_use_raw:Nc #1 #2 {
26   \exp_last_unbraced:NNo
27   \BNVS_use_raw:NN #1 { \cs:w #2 \cs_end: }
28 }

29 \cs_new:Npn \BNVS_use_raw:nc #1 #2 {
30   \exp_last_unbraced:Nno
31   \BNVS_use_raw:nN { #1 } { \cs:w #2 \cs_end: }
32 }

33 \cs_new:Npn \BNVS_use:Nc #1 #2 {
34   \BNVS_use_raw:Nc #1 { \BNVS:c { #2 } }
35 }

36 \cs_new:Npn \BNVS_use:nc #1 #2 {
37   \BNVS_use_raw:nc { #1 } { \BNVS:c { #2 } }
38 }

39 \cs_new:Npn \BNVS_tl_use:nvv #1 #2 {
40   \BNVS_tl_use:nv { \BNVS_tl_use:nv { #1 } { #2 } }
41 }
42 \cs_new:Npn \BNVS_tl_use:nvvv #1 #2 {
43   \BNVS_tl_use:nvv { \BNVS_tl_use:nv { #1 } { #2 } }
44 }

45 \cs_new:Npn \BNVS_log:n #1 { }
46 \cs_generate_variant:Nn \BNVS_log:n { x }

```

---

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

---

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

```

47 \cs_new:Npn \BNVS_DEBUG:c #1 {
48   BNVS_DEBUG~#1~
49 }
50 \cs_new:Npn \BNVS_DEBUG_on:n #1 {
51   \tl_if_empty:nT { #1 } {
52     \typein { Empty~argument~not~allowed }
53   }
54   \cs_set:cpn { \BNVS_DEBUG:c { #1 } log:n } { \BNVS_log:n }
55   \cs_generate_variant:cn { \BNVS_DEBUG:c { #1 } log:n } { x }
56 }

57 \cs_new:Npn \BNVS_DEBUG_off:n #1 {
58   \cs_set:cpn { \BNVS_DEBUG:c { #1 } log:n } { \use_none:n }
59 }
60 \seq_new:N \l_BNVS_DEBUG_push_n_seq
61 \cs_new:Npn \BNVS_DEBUG_push:n #1 {
62   \tl_if_empty:nT { #1 } {
63     \typein { Empty~argument~not~allowed }
64   }
65   \tl_map_inline:nn { #1 } {
66     \BNVS_DEBUG_on:n { ##1 }
67   }
68   \seq_put_left:Nn \l_BNVS_DEBUG_push_n_seq {
69     \tl_map_inline:nn { #1 } {
70       \BNVS_DEBUG_off:n { ##1 }
71     }
72   }
73 }
74 \tl_new:N \l_BNVS_DEBUG_push_n_tl
75 \cs_new:Npn \BNVS_DEBUG_pop: {
76   \seq_pop_left:NNTF \l_BNVS_DEBUG_push_n_seq \l_BNVS_DEBUG_push_n_tl {
77     \l_BNVS_DEBUG_push_n_tl
78   } {
79     \BNVS_error:n { Unbalanced~\BNVS_DEBUG_pop: }
80   }
81 }
82 \AddToHookNext { env/BNVS.test/begin } {
83   \BNVS_DEBUG_push:n {CDBGpfarsRomqi}
84   \BNVS_DEBUG_pop:
85 }
86 \cs_new:Npn \BNVS_DEBUG_log:nn #1 {
87   \cs_if_exist_use:cF { \BNVS_DEBUG:c { #1 } log:n } {
88     \BNVS_warning:n { Undeclared~DEBUG~type:~#1}
89     \cs_new:cpn { \BNVS_DEBUG:c { #1 } log:n } { \use_none:n }
90     \use_none:n
91   }
92 }
93 \cs_new:Npn \BNVS_DEBUG_on: {
94   \BNVS_DEBUG_on:n {C}
95 }
96 \cs_new:Npn \BNVS_DEBUG_off: {
97   \BNVS_DEBUG_off:n {C}
98 }

```

---

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

---

```

99 \cs_new:Npn \BNVS_new:cpn #1 {
100   \cs_new:cpn { \BNVS:c { #1 } }
101 }

102 \cs_new:Npn \BNVS_set:cpn #1 {
103   \cs_set:cpn { \BNVS:c { #1 } }
104 }

105 \cs_generate_variant:Nn \cs_generate_variant:Nn { c }
106 \cs_new:Npn \BNVS_generate_variant:cn #1 {
107   \cs_generate_variant:cn { \BNVS:c { #1 } }
108 }

```

### 6.3 logging

Utility messaging.

```

109 \msg_new:nnn { beanoves } { :n } { #1 }
110 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }

111 \cs_new:Npn \BNVS_warning:n {
112   \msg_warning:nnn { beanoves } { :n }
113 }
114 \cs_new:Npn \BNVS_warning:x {
115   \msg_warning:nnx { beanoves } { :n }
116 }

117 \cs_new:Npn \BNVS_error:n {
118   \msg_error:nnn { beanoves } { :n }
119 }
120 \cs_new:Npn \BNVS_error:x {
121   \msg_error:nnx { beanoves } { :n }
122 }

123 \cs_new:Npn \BNVS_fatal:n {
124   \msg_fatal:nnn { beanoves } { :n }
125 }
126 \cs_new:Npn \BNVS_fatal:x {
127   \msg_fatal:nnx { beanoves } { :n }
128 }

```

## 6.4 Facility layer: Variables

---

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

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

```

129 \cs_new:Npn \BNVS_N_new:c #1 {
130   \cs_new:cpn { BNVS_#1:c } ##1 {
131     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
132   }
133   \cs_new:cpn { BNVS_#1_new:c } ##1 {
134     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
135   }
136   \cs_new:cpn { BNVS_#1_use:c } ##1 {
137     \use:c { \cs:w BNVS_#1:c \cs_end: { ##1 } }
138   }
139   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
140     \BNVS_use_raw:Nc
141     ##1 { \cs:w BNVS_#1:c \cs_end: { ##2 } }
142   }
143   \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
144     \BNVS_use_raw:nc
145     { ##1 } { \cs:w BNVS_#1:c \cs_end: { ##2 } }
146   }
147 }

148 \cs_new:Npn \BNVS_v_new:c #1 {
149   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
150     \BNVS_use_raw:nc
151     { \exp_args:Nv ##1 }
152     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
153   }
154   \cs_new:cpn { BNVS_#1_use:cv } ##1 ##2 {
155     \BNVS_use_raw:nc
156     { \exp_args:NnV \BNVS_use:c { ##1 } }
157     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
158   }
159   \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
160     \BNVS_use_raw:nc
161     { \exp_args:NnV \use:n { ##1 } }
162     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
163   }
164 }

165 \BNVS_N_new:c { bool }
166 \BNVS_N_new:c { int }
167 \BNVS_v_new:c { int }
168 \BNVS_N_new:c { tl }
169 \BNVS_v_new:c { tl }
170 \cs_new:Npn \BNVS_tl_use:Nvv #1 {
171   \BNVS_exp_args:Nvv #1
172 }

```

```

173 \BNVS_N_new:c { str }
174 \BNVS_v_new:c { str }
175 \BNVS_N_new:c { seq }
176 \BNVS_v_new:c { seq }
177 \cs_undefine:N \BNVS_N_new:c

```

---

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

---

```

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

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

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

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

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

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

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

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

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

```



```

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

```

---

\BNVS\_new\_c:cn \BNVS\_new\_c:nc {<type>} {<core name>}

---

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

299 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
300   \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
301 }

302 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
303   \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
304 }

```

```

305 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
306   \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
307 }

308 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
309   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
310 }

311 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
312   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
313 }

314 \cs_new:Npn \BNVS_exp_args:Nvv #1 #2 #3 {
315   \BNVS_use:ncncn { \exp_args:NVV #1 }
316   { #2 } { t1 } { #3 } { t1 }
317 }

318 \cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
319   \BNVS_use:ncncncn { \exp_args:NVVV #1 }
320   { #2 } { t1 } { #3 } { t1 } { #4 } { t1 }
321 }

322 \cs_new:Npn \BNVS_exp_args:Nvvvv #1 #2 #3 #4 #5 {
323   \BNVS_tl_use:nc {
324     \exp_args:NnV \use:n {
325       \BNVS_exp_args:Nvvv #1 { #2 } { #3 } { #4 }
326     }
327   } { #5 }
328 }

```

---

\BNVS\_new\_conditional:cpnn \BNVS\_new\_conditional:cpnn {<core name>} <parameter> {<conditions>} {<code>}

---

```

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

330 \cs_new:Npn \BNVS_new_conditional:cpnn #1 {
331   \prg_new_conditional:cpnn { \BNVS:c { #1 } }
332 }

333 \cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
334 \cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
335   \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
336 }

337 \cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
338   \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
339     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
340       \prg_return_true:
341     } {
342       \prg_return_false:
343     }
344   }
345 }

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

```

```

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

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

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

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

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

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

```

```

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

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

```

#### 6.4.1 Regex

```

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

```

---

```

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

```

---

These are shortcuts to

- \regex\_match\_if\_once:NnNTF with the match sequence as N argument
- \regex\_match\_if\_once:nnNTF with the match sequence as N argument
- \regex\_split:NnNTF with the split sequence as last N argument

```

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

```

```

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

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

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

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

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

```

```

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

```

### 6.4.2 Token lists

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

These are shortcuts to

- \tl\_clear:c {l\_\_bnvs\_<core>\_tl}
- \tl\_use:c {l\_\_bnvs\_<core>\_tl}
- \tl\_set\_eq:cc {l\_\_bnvs\_<lhs core>\_tl}{l\_\_bnvs\_<rhs core>\_tl}
- \tl\_set:cv {l\_\_bnvs\_<core>\_tl}{l\_\_bnvs\_<value core>\_tl}
- \tl\_set:cx {l\_\_bnvs\_<core>\_tl}{<tl>}
- \tl\_put\_left:cn {l\_\_bnvs\_<core>\_tl}{<tl>}
- \tl\_put\_right:cn {l\_\_bnvs\_<core>\_tl}{<tl>}
- \tl\_put\_right:cv {l\_\_bnvs\_<core>\_tl}{l\_\_bnvs\_<value core>\_tl}

<u>\BNVS_new_conditional_vnc:cn</u>	\BNVS_new_conditional_vnc:cn {<core>} {<conditions>}
-------------------------------------	--

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

```

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

```

```

488 }
489 }

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

```

---

\BNVS\_new\_conditional\_vnc:cn \BNVS\_new\_conditional\_vnc:cn {<core>} {<conditions>}

Forwards to \BNVS\_new\_conditional\_vnc:cNn with \<core>:nncTF as function argument. Used for testing.

```

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

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

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

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

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

```



```

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

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

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

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

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

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

```

---

```

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

```

---

These are shortcuts to

- \tl\_if\_empty:cTF {l\_\_bnvs\_<core>\_tl} {<yes code>} {<no code>}
- \tl\_if\_eq:cnTF {l\_\_bnvs\_<core>\_tl}{<tl>} {<yes code>} {<no code>}

```

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

```

```

568 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
569   \BNVS_use_raw:nc {
570     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
571   } { #1_#2:NnTF }
572 }
573 \BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }

574 \BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
575   \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
576     \prg_return_true:
577   } {
578     \prg_return_false:
579   }
580 }

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

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

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

607 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
608   \BNVS_use_raw:nc {
609     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
610   } { #1_#2:NnTF }
611 }
612 \BNVS_new_conditional_cv:ncn { tl } { if_eq } { T, F, TF }

```

### 6.4.3 Strings

---

\\_bnvs\_str\_if\_eq:vnTF \\_bnvs\_str\_if\_eq:vnTF {<core>} {<tl>} {<yes code>} {<no code>}

---

These are shortcuts to

- \str\_if\_eq:ccTF {l\\_bnvs\_<core>\_tl}{<yes code>} {<no code>}

```

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

624 \cs_new:Npn \BNVS_new_conditional_vv:cn #1 {
625   \BNVS_use:nc {
626     \BNVS_new_conditional_vvnc:cn #1 {
627     } { #1:nnTF }
628   }
629 }

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

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

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

```

655 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
656   \BNVS_use_raw:nc {
657     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
658   } { #1_#2:nnTF }
659 }
660 \BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }

```

#### 6.4.4 Sequences

---

<code>\__bnvs_seq_count:c</code>	<code>\__bnvs_seq_new:c {&lt;core&gt;}</code>
<code>\__bnvs_seq_clear:c</code>	<code>\__bnvs_seq_count:c {&lt;core&gt;}</code>
<code>\__bnvs_seq_set_eq:cc</code>	<code>\__bnvs_seq_clear:c {&lt;core&gt;}</code>
<code>\__bnvs_seq_gset_eq:cc</code>	<code>\__bnvs_seq_set_eq:cc {&lt;core<sub>1</sub>&gt;} {&lt;core<sub>2</sub>&gt;}</code>
<code>\__bnvs_seq_use:cn</code>	<code>\__bnvs_seq_use:cn {&lt;core&gt;} {&lt;separator&gt;}</code>
<code>\__bnvs_seq_item:cn</code>	<code>\__bnvs_seq_item:cn {&lt;core&gt;} {&lt;integer expression&gt;}</code>
<code>\__bnvs_seq_remove_all:cn</code>	<code>\__bnvs_seq_remove_all:cn {&lt;core&gt;} {&lt;tl&gt;}</code>
<code>\__bnvs_seq_put_left:cv</code>	<code>\__bnvs_seq_put_right:cn {&lt;seq core&gt;} {&lt;tl&gt;}</code>
<code>\__bnvs_seq_put_right:cn</code>	<code>\__bnvs_seq_put_right:cv {&lt;seq core&gt;} {&lt;tl core&gt;}</code>
<code>\__bnvs_seq_put_right:cv</code>	<code>\__bnvs_seq_set_split:cnn {&lt;seq core&gt;} {&lt;tl&gt;} {&lt;separator&gt;}</code>
<code>\__bnvs_seq_set_split:cnn</code>	<code>\__bnvs_seq_pop_left:cc {&lt;core<sub>1</sub>&gt;} {&lt;core<sub>2</sub>&gt;}</code>
<code>\__bnvs_seq_set_split:(cnv cnx)</code>	
<code>\__bnvs_seq_pop_left:cc</code>	

---

These are shortcuts to

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

```

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

```

```

672 \BNVS_new_cnn:nc { seq } { set_split }
673 \BNVS_new_cnv:nc { seq } { set_split }
674 \BNVS_new_cnx:nc { seq } { set_split }
675 \BNVS_new_cc:ncn { seq } { pop_left } { t1 }
676 \BNVS_new_cc:ncn { seq } { pop_right } { t1 }

```

---

```

\__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>} {<t1 core>} {<yes code>} {<no code>}
\__bnvs_seq_pop_left:ccTF
\__bnvs_seq_pop_right:ccTF

```

---

```

677 \cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
678   \BNVS_new_conditional:cpnn { #1_#2:cc } ##1 ##2 { #4 } {
679     \BNVS_use:ncncn {
680       \BNVS_use_raw:c { #1_#2:NNTF }
681     } { ##1 } { #1 } { ##2 } { #3 } {
682       \prg_return_true:
683     } {
684       \prg_return_false:
685     }
686   }
687 }
688 \BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
689 \BNVS_new_conditional_cc:ncnn
690 { seq } { get_right } { t1 } { T, F, TF }
691 \BNVS_new_conditional_cc:ncnn
692 { seq } { pop_left } { t1 } { T, F, TF }
693 \BNVS_new_conditional_cc:ncnn
694 { seq } { pop_right } { t1 } { T, F, TF }

```

#### 6.4.5 Integers

---

```

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

```

---

These are shortcuts to

- \int\_new:c {l\_\_bnvs\_<core>\_int}
- \int\_use:c {l\_\_bnvs\_<core>\_int}
- \int\_incr:c {l\_\_bnvs\_<core>\_int}
- \int\_idocr:c {l\_\_bnvs\_<core>\_int}
- \int\_set:cn {l\_\_bnvs\_<core>\_int} <value>

```

695 \BNVS_new_c:nc { int } { new }
696 \BNVS_new_c:nc { int } { use }
697 \BNVS_new_c:nc { int } { zero }
698 \BNVS_new_c:nc { int } { incr }
699 \BNVS_new_c:nc { int } { decr }
700 \BNVS_new_cn:nc { int } { set }
701 \BNVS_new_cv:ncn { int } { set } { int }

```

## 6.5 Debug facilities

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

## 6.6 Debug messages

## 6.7 Testing facilities

## 6.8 Local variables

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

```

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

```

```

728 \tl_new:N \l__bnvs_post_tl
729 \tl_new:N \l__bnvs_suffix_tl
730 \tl_new:N \l__bnvs_index_tl
731 \int_new:N \g__bnvs_call_int
732 \int_new:N \l__bnvs_int
733 \int_new:N \l__bnvs_i_int
734 \seq_new:N \g__bnvs_def_seq
735 \seq_new:N \l__bnvs_a_seq
736 \seq_new:N \l__bnvs_b_seq
737 \seq_new:N \l__bnvs_ans_seq
738 \seq_new:N \l__bnvs_match_seq
739 \seq_new:N \l__bnvs_split_seq
740 \seq_new:N \l__bnvs_path_seq
741 \seq_new:N \l__bnvs_path_head_seq
742 \seq_new:N \l__bnvs_path_tail_seq
743 \seq_new:N \l__bnvs_query_seq
744 \seq_new:N \l__bnvs_token_seq
745 \bool_new:N \l__bnvs_in_frame_bool
746 \bool_set_false:N \l__bnvs_in_frame_bool
747 \bool_new:N \l__bnvs_parse_bool
748 \bool_set_false:N \l__bnvs_parse_bool
749 \bool_new:N \l__bnvs_deep_bool
750 \bool_set_false:N \l__bnvs_deep_bool

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

```

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

```

755 \bool_new:N \l__bnvs_provide_bool

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

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

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

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

```

## 6.9 Infinite loop management

Unending recursivity is managed here.

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

*(End of definition for \g\_\_bnvs\_call\_int.)*

```
769 \int_const:Nn \c__bnvs_max_call_int { 8192 }
```

---

`\__bnvs_greset_call:` `\__bnvs_greset_call:`

Reset globally the call stack counter to its maximum value.

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

---

`\__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.

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

## 6.10 Overlay specification

### 6.10.1 Registration

We keep track of the `\id` `\tag` combinations and provide looping mechanisms.

---

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

---

Create a unique name from the arguments.

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

`\g__bnvs_I_seq` List of registered identifiers.

*(End of definition for \g\_\_bnvs\_I\_seq.)*

```
784 \seq_new:N \g__bnvs_I_seq
```



---

```

__bnvs_register:nn    __bnvs_register:nn    {<id>} {<tag>}
__bnvs_unregister:nn __bnvs_unregister:nn {<id>} {<tag>}
__bnvs_unregister:n  __bnvs_unregister:n   {<id>}
__bnvs_unregister:   __bnvs_unregister:

```

---

Register and unregister according to the arguments. The  $\langle id \rangle! \langle tag \rangle$  combination must be registered on definition and unregistered on disposal.

```

785 \seq_new:N \l__bnvs_register_NNnn_seq
786 \BNVS_new:cpn { register:NNnn } #1 #2 #3 #4 {
787   \cs_if_exist:NF #1 {
788     \cs_gset:Npn #1 { }
789     \seq_if_exist:NTF #2 {
790       __bnvs_seq_clear:c { register_NNnn }
791       \cs_set:Npn \BNVS_register_NNnn: {
792         __bnvs_seq_put_right:cn { register_NNnn } { #4 }
793         \cs_set:Npn \BNVS_register_NNnn: { }
794       }
795       \cs_set:Npn \BNVS_register_NNnn:w ##1 ##2 {
796         \str_compare:nNnTF { ##2 } < { #4 } {
797           __bnvs_seq_put_right:cn { register_NNnn } { ##2 }
798         } {
799           \BNVS_register_NNnn:
800           __bnvs_seq_put_right:cn { register_NNnn } { ##2 }
801           \cs_set:Npn \BNVS_register_NNnn:w ####1 ####2 {
802             __bnvs_seq_put_right:cn { register_NNnn } { ####2 }
803           }
804         }
805       }
806       __bnvs_foreach_T:nNTF { #3 } \BNVS_register_NNnn:w {
807         \BNVS_register_NNnn:
808         \seq_gset_eq:NN #2 \l__bnvs_register_NNnn_seq
809       } {
810         \BNVS_error:n { Unreachable/register:NNnn-id-#3 }
811       }
812     } {
813       \seq_new:N #2
814       \seq_gput_right:Nn #2 { #4 }
815       __bnvs_seq_clear:c { register_NNnn }
816       \cs_set:Npn \BNVS_register_NNnn: {
817         __bnvs_seq_put_right:cn { register_NNnn } { #3 }
818         \cs_set:Npn \BNVS_register_NNnn: { }
819       }
820       \cs_set:Npn \BNVS_register_NNnn:w ##1 {
821         \str_compare:nNnTF { ##1 } < { #3 } {
822           __bnvs_seq_put_right:cn { register_NNnn } { ##1 }
823         } {
824           \BNVS_register_NNnn:
825           __bnvs_seq_put_right:cn { register_NNnn } { ##1 }
826           \cs_set:Npn \BNVS_register_NNnn:w ####1 {
827             __bnvs_seq_put_right:cn { register_NNnn } { ####1 }
828           }
829         }
830       }
831       __bnvs_foreach_I:N \BNVS_register_NNnn:w

```

```

832     \BNVS_register_NNnn:
833     \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_register_NNnn_seq
834   }
835 }
836 }

837 \BNVS_new:cpn { register:nn } #1 #2 {
838   \exp_args:Ncc \__bnvs_register:NNnn
839   { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
840   { #1 } { #2 }
841 }

```

---

\\_\_bnvs\_unregister:NNnn \\_\_bnvs\_unregister:NNnn <cs> <seq> {<id>} {<tag>}

Unregistering a <id> <tag> combination is not straightforward. <cs> and <seq> are respectively the command and the sequence uniquely associated to this combination.

```

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

```

```

879 }
880 }

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

```

---

\\_\_bnvs\_if\_registered:nnTF \\_\_bnvs\_if\_register:nnTF {<id> } {<tag> } {<yes code> } {<no code> }

Execute <yes code> or <no code> depending on the <id> ! <tag> combination being registered.

```

886 \BNVS_new_conditional:cpnn { if_registered:nn } #1 #2 { T, F, TF } {
887   \cs_if_exist:cTF { \__bnvs_name:nn { #1 } { #2 } } {
888     \prg_return_true:
889   } {
890     \prg_return_false:
891   }
892 }

```

---

\\_\_bnvs\_foreach\_I:N \\_\_bnvs\_foreach\_I:N {<function:n>  
\\_\_bnvs\_foreach\_I:n \\_\_bnvs\_foreach\_I:n {<code> }

Execute the <function:n> or the <code> for each declared identifier.

```

893 \BNVS_new:cpn { foreach_I:N } {
894   \seq_map_function:NN \g__bnvs_I_seq
895 }

896 \BNVS_new:cpn { foreach_I:n } {
897   \seq_map_inline:Nn \g__bnvs_I_seq
898 }

```

---

\\_\_bnvs\_foreach\_T:nNTF \\_\_bnvs\_foreach\_T:nNTF {<id> } {<function:nn> } {<yes code> } {<no code> }  
\\_\_bnvs\_foreach\_T:nnTF \\_\_bnvs\_foreach\_T:nnTF {<id> } {<code> } {<yes code> } {<no code> }

If <id> is a declared identifier, execute <function:nn> or <code> for each combination of <id> and its associate <tag>s.

```

899 \BNVS_new_conditional:cpnn { foreach_T:nN } #1 #2 { T, F, TF } {
900   \seq_if_exist:cTF { g__bnvs_#1!_seq } {
901     \seq_map_inline:cn { g__bnvs_#1!_seq } { #2 { #1 } { ##1 } }
902     \prg_return_true:
903   } { \prg_return_false: }
904 }

905 \BNVS_new_conditional:cpnn { foreach_T:nn } #1 #2 { T, F, TF } {
906   \seq_if_exist:cTF { g__bnvs_#1!_seq } {
907     \cs_set:Npn \BNVS_foreach_T_nn:nn ##1 ##2 { #2 }
908     \seq_map_inline:cn { g__bnvs_#1!_seq }
909     { \BNVS_foreach_T_nn:nn { #1 } { ##1 } }
910     \prg_return_true:
911   } { \prg_return_false: }
912 }

```

---

```

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

```

---

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

```

913 \BNVS_new:cpn { foreach_IT:N } #1 {
914   \__bnvs_foreach_I:n {
915     \__bnvs_foreach_T:nNT { ##1 } #1 { }
916   }
917 }

918 \BNVS_new:cpn { foreach_IT:n } #1 {
919   \cs_set:Npn \BNVS_foreach_IT_n:nn ##1 ##2 { #1 }
920   \__bnvs_foreach_I:n {
921     \__bnvs_foreach_T:nNT { ##1 } \BNVS_foreach_IT_n:nn { }
922   }
923 }

```

---

```

\__bnvs_foreach_I:N \__bnvs_foreach_key:N <function:n>
\__bnvs_foreach_I:n \__bnvs_foreach_key:n <code>

```

---

```

\__bnvs_foreach_key_main:N <function:n>
\__bnvs_foreach_key_main:n <code>
\__bnvs_foreach_key_sub:N <function:n>
\__bnvs_foreach_key_sub:n <code>
\__bnvs_foreach_key_cache:N <function:n>
\__bnvs_foreach_key_cache:n <code>

```

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

```

924 \BNVS_new:cpn { foreach_key_main:N } {
925   \tl_map_function:nN { VAZL }
926 }

927 \BNVS_new:cpn { foreach_key_main:n } {
928   \tl_map_inline:nn { VAZL }
929 }

930 \BNVS_new:cpn { foreach_key_sub:N } {
931   \tl_map_function:nN { PNvn }
932 }

933 \BNVS_new:cpn { foreach_key_sub:n } {
934   \tl_map_inline:nn { PNvn }
935 }

936 \BNVS_new:cpn { foreach_key:n } #1 {
937   \__bnvs_foreach_key_main:n { #1 }
938   \__bnvs_foreach_key_sub:n { #1 }
939 }

940 \BNVS_new:cpn { foreach_key:N } #1 {
941   \__bnvs_foreach_key_main:N #1
942   \__bnvs_foreach_key_sub:N #1
943 }

```

```

944 \BNVS_new:cpn { foreach_key_cache:N } {
945   \tl_map_function:nN { {V*}{A*}{Z*}{L*}{P*}{N*} }
946 }

947 \BNVS_new:cpn { foreach_key_cache:n } {
948   \tl_map_inline:nn { {V*}{A*}{Z*}{L*}{P*}{N*} }
949 }

```

### 6.10.2 Basic functions

---

<code>\__bnvs_gset:nnnn</code>	<code>\__bnvs_gset:nnnn {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;} {&lt;spec&gt;}</code>
<code>\__bnvs_gset:(nnnv nvnn nvvn nvvv)</code>	

---

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

```

950 \BNVS_new:cpn { gset:nnnn } #1 #2 #3 {
951   \regex_match:nnTF { ^[a-z_]+$ } { #3 } {
952     \use_none:n
953   } {
954     \__bnvs_register:nn { #2 } { #3 }
955     \cs_gset:cpn { \__bnvs_name:nnn { #1 } { #2 } { #3 } }
956   }
957 }

958 \BNVS_new:cpn { gset:nvnn } #1 {
959   \BNVS_tl_use:nv { \__bnvs_gset:nnnn { #1 } }
960 }

961 \BNVS_new:cpn { gset:nvvn } #1 {
962   \BNVS_tl_use:nvv { \__bnvs_gset:nnnn { #1 } }
963 }

964 \BNVS_new:cpn { gset:nnnv } #1 #2 #3 {
965   \BNVS_tl_use:nv {
966     \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
967   }
968 }

969 \BNVS_new:cpn { gset:nvvv } #1 {
970   \BNVS_tl_use:nvvv { \__bnvs_gset:nnnn { #1 } }
971 }

```

---

<code>\__bnvs_gunset:nnn</code>	<code>\__bnvs_gunset:nnn {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;}</code>
<code>\__bnvs_gunset:nn</code>	<code>\__bnvs_gunset:nn {&lt;id&gt;} {&lt;tag&gt;}</code>
<code>\__bnvs_gunset:n</code>	<code>\__bnvs_gunset:n {&lt;id&gt;}</code>
<code>\__bnvs_gunset:</code>	<code>\__bnvs_gunset:</code>

---

Removes the specifications for the `<key>`, `<id>`, `<tag>` combination. In the variant, all possible `<key>`s and `<tag>`s are used.

```

972 \BNVS_new:cpn { gunset:nnn } #1 #2 #3 {
973   \cs_undefine:c { \__bnvs_name:nnn { #1 } { #2 } { #3 } }
974 }

```

```

975 \BNVS_new:cpn { gunset:nvv } #1 {
976   \BNVS_tl_use:nvv { \_bnvs_gunset:nnn { #1 } }
977 }

978 \BNVS_new:cpn { gunset:nn } #1 #2 {
979   \_bnvs_if_registered:nnT { #1 } { #2 } {
980     \tl_map_inline:nn {
981       \_bnvs_foreach_key_main:n
982       \_bnvs_foreach_key_sub:n
983       \_bnvs_foreach_key_cache:n
984     } {
985       ##1 {
986         \_bnvs_gunset:nnn { ####1 } { #1 } { #2 }
987       }
988     }
989     \_bnvs_unregister:nn { #1 } { #2 }
990   }
991 }

992 \BNVS_new:cpn { gunset_deep:nn } #1 #2 {
993   \_bnvs_foreach_IT:n {
994     \tl_if_eq:nnT { #1 } { ##1 } {
995       \tl_if_in:nnT { .. ##2 } { .. #2 . } {
996         \_bnvs_gunset:nn { #1 } { ##2 }
997       }
998     }
999   }
1000 }

1001 \BNVS_new:cpn { gunset:vn } {
1002   \BNVS_tl_use:Nv \_bnvs_gunset:nn
1003 }

1004 \BNVS_new:cpn { gunset:vv } {
1005   \BNVS_tl_use:Nvv \_bnvs_gunset:nn
1006 }

1007 \BNVS_new:cpn { gunset_deep:vv } {
1008   \BNVS_tl_use:Nvv \_bnvs_gunset_deep:nn
1009 }

1010 \seq_new:N \l__bnvs_gunset_n_seq
1011 \BNVS_new:cpn { gunset:n } #1 {
1012   \_bnvs_seq_clear:c { gunset_n }
1013   \_bnvs_foreach_I:n {
1014     \tl_if_eq:nnTF { ##1 } { #1 } {
1015       \_bnvs_foreach_T:nn { #1 } {
1016         \_bnvs_gunset:nn { #1 } { ####1 }
1017       }
1018     } {
1019       \_bnvs_seq_put_right:cn { gunset_n } { ##1 }
1020     }
1021   }
1022   \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_gunset_n_seq
1023 }

```

```

1024 \BNVS_new:cpn { gunset: } {
1025   \_bnvs_foreach_IT:N \_bnvs_gunset:nn
1026 }

```

---

<pre> \_bnvs_is_gset:nnnTF \_bnvs_is_gset:(nvv nvx)TF \_bnvs_is_gset:nnTF \_bnvs_if_spec:nnnTF \_bnvs_if_spec:nnTF </pre>	<pre> \_bnvs_is_gset:nnnTF {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;} {&lt;yes code&gt;} {&lt;no code&gt;} \_bnvs_is_gset:nnTF {&lt;id&gt;} {&lt;tag&gt;} {&lt;yes code&gt;} {&lt;no code&gt;} \_bnvs_if_spec:nnnTF {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;} {&lt;yes code&gt;} {&lt;no code&gt;} \_bnvs_if_spec:nnTF {&lt;id&gt;} {&lt;tag&gt;} {&lt;yes code&gt;} {&lt;no code&gt;} </pre>
---	--

---

Convenient shortcuts to test for the existence of a  $\langle spec \rangle$  for that  $\langle key \rangle$ ,  $\langle id \rangle$ ,  $\langle tag \rangle$  combination. The version with no  $\langle key \rangle$  is the or combination for keys V, A and Z.

The  $\_spec:\dots$  variant is similar except that it uses  $\langle key \rangle$ ,  $\langle id \rangle$ ,  $\langle ref \rangle$  or  $\langle key \rangle$  empty  $\langle id \rangle$ ,  $\langle tag \rangle$  combinations.

```

1027 \BNVS_new_conditional:cpnn { is_gset:nnn } #1 #2 #3 { T, F, TF } {
1028   \cs_if_exist:cTF { \_bnvs_name:nnn { #1 } { #2 } { #3 } } {
1029     \prg_return_true:
1030   } {
1031     \prg_return_false:
1032   }
1033 }

```

```

1034 \BNVS_new_conditional:cpnn { is_gset:nvx } #1 #2 #3 { T, F, TF } {
1035   \exp_args:Nnnx \BNVS_tl_use:nv {
1036     \_bnvs_is_gset:nnnTF { #1 }
1037   } { #2 } { #3 } {
1038     \prg_return_true:
1039   } {
1040     \prg_return_false:
1041   }
1042 }

```

```

1043 \BNVS_new_conditional:cpnn { is_gset:nvv } #1 #2 #3 { T, F, TF } {
1044   \BNVS_tl_use:nvv {
1045     \_bnvs_is_gset:nnnTF { #1 }
1046   } { #2 } { #3 } {
1047     \prg_return_true:
1048   } {
1049     \prg_return_false:
1050   }
1051 }

```

```

1052 \BNVS_new_conditional:cpnn { is_gset:nn } #1 #2 { T, F, TF } {
1053   \_bnvs_is_gset:nnnTF V { #1 } { #2 } {
1054     \prg_return_true:
1055   } {
1056     \_bnvs_is_gset:nnnTF A { #1 } { #2 } {
1057       \prg_return_true:
1058     } {
1059       \_bnvs_is_gset:nnnTF Z { #1 } { #2 } {
1060         \prg_return_true:
1061       } {
1062         \prg_return_false:

```

```

1063     }
1064   }
1065 }
1066 }

1067 \BNVS_new_conditional:cpnn { if_spec:nnn } #1 #2 #3 { T, F, TF } {
1068   \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
1069     \prg_return_true:
1070   } {
1071     \tl_if_empty:nTF { #2 } {
1072       \prg_return_false:
1073     } {
1074       \__bnvs_is_gset:nnnTF { #1 } { } { #3 } {
1075         \prg_return_true:
1076       } {
1077         \prg_return_false:
1078       }
1079     }
1080   }
1081 }

1082 \BNVS_new_conditional:cpnn { if_spec:nn } #1 #2 { T, F, TF } {
1083   \__bnvs_is_gset:nnTF { #1 } { #2 } {
1084     \prg_return_true:
1085   } {
1086     \tl_if_empty:nTF { #1 } {
1087       \prg_return_false:
1088     } {
1089       \__bnvs_is_gset:nnTF { } { #2 } {
1090         \prg_return_true:
1091       } {
1092         \prg_return_false:
1093       }
1094     }
1095   }
1096 }

```

---

<code>\__bnvs_if_get:nnncTF</code> <code>\__bnvs_spec:nnncTF</code>	<code>\__bnvs_if_get:nnncTF {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;} {&lt;ans&gt;}</code> <code>{&lt;yes code&gt;} {&lt;no code&gt;}</code> <code>\__bnvs_spec:nnncTF {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;} {&lt;ans&gt;}</code> <code>{&lt;yes code&gt;} {&lt;no code&gt;}</code>
--	--

---

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

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

```

1097 \BNVS_new_conditional:cpnn { if_get:nnnc } #1 #2 #3 #4 { T, F, TF } {
1098   \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
1099     \exp_args:Nnc \use:n { \exp_args:Nno \cs_set:cpn { \BNVS_1:cn { #4 } { tl } } } { \__bnv
1100     \prg_return_true:
1101   } {
1102     \prg_return_false:

```



```

1103 }
1104 }

1105 \BNVS_new_conditional:cpnn { if_get:nvvc } #1 #2 #3 #4 { T, F, TF } {
1106   \BNVS_tl_use:nvv {
1107     \__bnvs_if_get:nnncTF { #1 }
1108   } { #2 } { #3 } { #4 } {
1109     \prg_return_true:
1110   } {
1111     \prg_return_false:
1112   }
1113 }

1114 \BNVS_new_conditional:cpnn { if_spec:nnnc } #1 #2 #3 #4 { T, F, TF } {
1115   \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #4 } {
1116     \prg_return_true:
1117   } {
1118     \tl_if_empty:nTF { #2 } {
1119       \prg_return_false:
1120     } {
1121       \__bnvs_if_get:nnncTF { #1 } { } { #3 } { #4 } {
1122         \prg_return_true:
1123       } {
1124         \prg_return_false:
1125       }
1126     }
1127   }
1128 }

```

---

```

\__bnvs_is_provide_gset:nnnTF \__bnvs_is_provide_gset:nnnTF {<key>} {<id>} {<tag>} {<yes code>} {<no
\__bnvs_is_provide_gset:nvvTF code>}

```

---

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

```

1129 \BNVS_new_conditional:cpnn { is_provide_gset:nnn } #1 #2 #3 { T, F, TF } {
1130   \__bnvs_if:cTF { provide } {
1131     \cs_if_exist:cTF { \__bnvs_name:nnn { #1 } { #2 } { #3 } } {
1132       \prg_return_true:
1133     } {
1134       \prg_return_false:
1135     }
1136   } {
1137     \prg_return_false:
1138   }
1139 }

1140 \BNVS_new_conditional:cpnn { is_provide_gset:nvv } #1 #2 #3 { T, F, TF } {
1141   \BNVS_tl_use:nvv { \__bnvs_is_provide_gset:nnnTF { #1 } } { #2 } { #3 } {
1142     \prg_return_true:
1143   } {
1144     \prg_return_false:
1145   }
1146 }

```

---

```

__bnvs_gprovide:TnnnnF  __bnvs_gprovide:TnnnnF {<yes code>} {<key>} {<id>} {<tag>} {<value>} {<no code>}
__bnvs_gprovide:TvnvnF  Execute <no code> exclusively when not in provide mode. Does nothing when something
__bnvs_gprovide:TnnvnF  was set for the <id>{<tag>}/{<key>} combination. Execute <yes code> before providing.

```

```

1147 \BNVS_new:cpn { gprovide:TnnnnF } #1 #2 #3 #4 #5 {
1148   __bnvs_if:cTF { provide } {
1149     __bnvs_is_gset:nnnF { #2 } { #3 } { #4 } {
1150       #1
1151       __bnvs_gset:nnnn { #2 } { #3 } { #4 } { #5 }
1152     }
1153   }
1154 }

1155 \BNVS_new:cpn { gprovide:TvnvnF } #1 #2 {
1156   \BNVS_t1_use:nv { __bnvs_gprovide:TnnnnF { #1 } { #2 } }
1157 }
1158 \BNVS_new:cpn { gprovide:TnnvnF } #1 #2 {
1159   \BNVS_t1_use:nv { __bnvs_gprovide:TnnnnF { #1 } { #2 } }
1160 }

```

### 6.10.3 Functions with cache

---

```

__bnvs_gset_cache:nnnn  __bnvs_gset_cache:nnnn {<key>} {<id>} {<tag>} {<value>}
__bnvs_gset_cache:(nnnv|nvnv)

```

---

Wrapper over the functions above for <key>\* instead of <key>.

```

1161 \BNVS_new:cpn { gset_cache:nnnn } #1 {
1162   __bnvs_gset:nnnn { #1 * }
1163 }

1164 \BNVS_new:cpn { gset_cache:nvnv } #1 #2 {
1165   \BNVS_t1_use:nv {
1166     \BNVS_t1_use:nv {
1167       __bnvs_gset_cache:nnnn { #1 }
1168     } { #2 }
1169   }
1170 }

1171 \BNVS_new:cpn { gset_cache:nnnv } #1 #2 #3 {
1172   \BNVS_t1_use:nv {
1173     __bnvs_gset_cache:nnnn { #1 } { #2 } { #3 }
1174   }
1175 }

```

---

```

__bnvs_if_get_cache:nnncTF  __bnvs_if_get_cache:nnncTF {<key>} {<id>} {<tag>} {<ans>}
__bnvs_if_get_cache:nnncTF {<yes code>} {<false code>}

```

---

Wrapper over the functions above for <key>\* instead of <key>.

```

1176 \BNVS_new_conditional:cpnn { if_get_cache:nnnc } #1 #2 #3 #4 { T, F, TF } {
1177   \_bnvs_if_get:nnncTF { #1 * } { #2 } { #3 } { #4 } {
1178     \prg_return_true:
1179   } {
1180     \prg_return_false:
1181   }
1182 }

```

---

<pre> \_bnvs_gunset_cache:nnn \_bnvs_gunset_cache:nvv \_bnvs_gunset_cache:nn \_bnvs_gunset_cache:vv \_bnvs_gunset_cache:n \_bnvs_gunset_cache: </pre>	<pre> \_bnvs_gunset_cache:nnn {&lt;key&gt;} {&lt;id&gt;} {&lt;tag&gt;} \_bnvs_gunset_cache:nn {&lt;id&gt;} {&lt;tag&gt;} \_bnvs_gunset_cache:n {&lt;id&gt;} \_bnvs_gunset_cache: Wrapper over the functions above for &lt;key&gt;* instead of &lt;key&gt;. </pre>
---	---

---

```

1183 \BNVS_new:cpn { gunset_cache:nnn } #1 {
1184   \_bnvs_gunset:nnn { #1 * }
1185 }

1186 \BNVS_new:cpn { gunset_cache:nvv } #1 {
1187   \_bnvs_gunset:nvv { #1 * }
1188 }

1189 \BNVS_new:cpn { gunset_cache:nn } #1 #2 {
1190   \_bnvs_foreach_key_cache:n {
1191     \_bnvs_gunset:nnn { ##1 } { #1 } { #2 }
1192   }
1193 }

1194 \BNVS_new:cpn { gunset_cache:vv } {
1195   \BNVS_tl_use:Nvv \_bnvs_gunset:nn
1196 }

1197 \BNVS_new:cpn { gunset_cache:n } #1 {
1198   \_bnvs_foreach_IT:n {
1199     \tl_if_eq:nnT { #1 } { ##1 } {
1200       \_bnvs_gunset_cache:nn { ##1 } { ##2 }
1201     }
1202   }
1203 }

1204 \BNVS_new:cpn { gunset_cache: } {
1205   \_bnvs_foreach_IT:n {
1206     \_bnvs_gunset_cache:nn { ##1 } { ##2 }
1207   }
1208 }

```

## 6.11 Implicit value counter

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

---

```

\__bnvs_v_gunset:nn \__bnvs_v_gunset:n {<id>} {<tag>}
\__bnvs_v_gunset:n \__bnvs_v_gunset:n {<id>}
\__bnvs_v_gunset:   \__bnvs_v_gunset:

```

---

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L<sup>A</sup>T<sub>E</sub>X3 eponym functions.

```

1209 \BNVS_new:cpn { v_gunset: } {
1210   \__bnvs_foreach_IT:n {
1211     \__bnvs_gunset:nnn v { ##1 } { ##2 }
1212     \__bnvs_gunset_cache:nnn v { ##1 } { ##2 }
1213   }
1214 }

1215 \BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
1216   \BNVS_tl_use:nc { \exp_args:No \quark_if_nil:nTF } { #1 } {
1217     \prg_return_true:
1218   } {
1219     \prg_return_false:
1220   }
1221 }

1222 \BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
1223   \BNVS_tl_use:nc { \exp_args:No \quark_if_no_value:nTF } { #1 } {
1224     \prg_return_true:
1225   } {
1226     \prg_return_false:
1227   }
1228 }

```

---

```

\__bnvs_if_greset_all:nnnTF \__bnvs_if_greset_all:nnnTF {<id>} {<tag>} {<initial value>} {<yes code>}
\__bnvs_if_greset_all:vvnnTF {<no code>}

```

---

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

```

1229 \BNVS_new_conditional:cpnn { if_greset_all:nnn } #1 #2 #3 { T, F, TF } {
1230   \__bnvs_is_gset:nnTF { #1 } { #2 } {
1231     \BNVS_begin:
1232     \__bnvs_foreach_key_main:n {
1233       \__bnvs_if_get:nnncT { ##1 } { #1 } { #2 } { a } {
1234         \__bnvs_quark_if_nil:cT { a } {
1235           \__bnvs_if_get_cache:nnncTF { ##1 } { #1 } { #2 } { a } {
1236             \__bnvs_gset:nnnv { ##1 } { #1 } { #2 } { a }
1237           } {
1238             \__bnvs_gset:nnnn { ##1 } { #1 } { #2 } { 1 }
1239           }

```

```

1240     }
1241   }
1242 }
1243 \BNVS_end:
1244 \__bnvs_gunset_cache:nn { #1 } { #2 }
1245 \__bnvs_foreach_key_sub:n {
1246   \__bnvs_gunset:nnn { ##1 } { #1 } { #2 }
1247 }
1248 \prg_return_true:
1249 } {
1250 \prg_return_false:
1251 }
1252 }

1253 \BNVS_new_conditional:cpnn { if_greset_all:vvnn } #1 #2 #3 { T, F, TF } {
1254   \BNVS_tl_use:nv {
1255     \BNVS_tl_use:Nv \__bnvs_if_greset_all:nnnTF { #1 }
1256   } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
1257 }

```

## 6.12 Regular expressions

`\c__bnvs_short_regex` This regular expressioin 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.

```

1258 \regex_const:Nn \c__bnvs_short_regex {
1259   [[:alpha:]]_ [[:alnum:]]_*
1260 }

```

*(End of definition for \c\_\_bnvs\_short\_regex.)*

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

```

1261 \regex_const:Nn \c__bnvs_path_regex {
1262   (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ ) *
1263 }

```

*(End of definition for \c\_\_bnvs\_path\_regex.)*

`\c__bnvs_A_integer_Z_regex`

*(End of definition for \c\_\_bnvs\_A\_integer\_Z\_regex.)*

```

1264 \regex_const:Nn \c__bnvs_A_integer_Z_regex { \A[-+]? \d+ \Z }

```

`\c__bnvs_A_index_Z_regex`

*(End of definition for \c\_\_bnvs\_A\_index\_Z\_regex.)*

```

1265 \regex_const:Nn \c__bnvs_A_index_Z_regex { \A[-+]? \d+ \Z }

```

`\c__bnvs_A_reserved_Z_regex`

*(End of definition for \c\_\_bnvs\_A\_reserved\_Z\_regex.)*

```

1266 \regex_const:Nn \c__bnvs_A_reserved_Z_regex {
1267   \A_*[a-z][_a-z0-9]*\Z
1268 }

```

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

*(End of definition for \c\_\_bnvs\_A\_ref\_Z\_regex.)*

```

1269 \regex_const:Nn \c__bnvs_A_ref_Z_regex {

    1: the  $\langle frame\ id \rangle$ 

1270   \A (?: ( \ur{c__bnvs_short_regex} )? ! )?

    2: The short name.

1271   ( \ur{c__bnvs_short_regex} )

    3: the path, if any.

1272   ( \ur{c__bnvs_path_regex} ) \Z
1273 }

```

`\c__bnvs_A_IKT_Z_regex` Matches the whole string, split into  $\langle id \rangle$  and  $\langle tag \rangle$ .

*(End of definition for \c\_\_bnvs\_A\_IKT\_Z\_regex.)*

1: The full match,

```

1274 \regex_const:Nn \c__bnvs_A_IKT_Z_regex {

    2: the frame  $\langle id \rangle$ 

1275   \A (?: ( \ur{c__bnvs_short_regex} )? (!) )?

```

3: The short name

```

1276   ( \ur{c__bnvs_short_regex}
1277   (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )* ) \Z
1278 }

```

`\c__bnvs_A_ISP_Z_regex` Matches the whole string.

*(End of definition for \c\_\_bnvs\_A\_ISP\_Z\_regex.)*

1: The full match,

```

1279 \regex_const:Nn \c__bnvs_A_ISP_Z_regex {

    2: the frame  $\langle id \rangle$ 

```

```
1280      \A (?: ( \ur{c__bnvs_short_regex} )? (!) )?
```

3: The short name

```
1281      ( \ur{c__bnvs_short_regex} )
```

4: The dotted path.

```
1282      ( (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )* ) \Z
1283    }
```

`\c__bnvs_A_SP_Z_regex` Matches the whole string.

*(End of definition for \c\_\_bnvs\_A\_SP\_Z\_regex.)*

```
1284 \regex_const:Nn \c__bnvs_A_SP_Z_regex {
```

1: The full match,

2: the frame `<id>`

```
1285      \A ( \ur{c__bnvs_short_regex} | [-+]? \d+ )
```

3: The dotted path.

```
1286      ( (?: \. \ur{c__bnvs_short_regex} | \. [-+]? \d+ )* ) \Z
1287    }
```

`\c__bnvs_A_P_Z_regex` Matches the whole string.

*(End of definition for \c\_\_bnvs\_A\_P\_Z\_regex.)*

```
1288 \regex_const:Nn \c__bnvs_A_P_Z_regex {
```

```
1289   \A \ur{c__bnvs_short_regex} (?: \. \ur{c__bnvs_short_regex} )* \Z
1290 }
```

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

```
1291 \regex_const:Nn \c__bnvs_colons_regex { :(:+)? }
```

*(End of definition for \c\_\_bnvs\_colons\_regex.)*

`\c__bnvs_split_regex` Used to parse slide list overlay specifications in queries. Next are the 12 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

```
1292 \regex_const:Nn \c__bnvs_split_regex {
1293   \s* ( ? :
```

We start with ‘++’ instrussions<sup>4</sup>.

1 incrementation prefix

1294        \+\+

1.1: optional identifier: optional  $\langle frame\ id \rangle$

1.2: followed by required !

1295        (?: ( \ur{c\_\_bnvs\_short\_regex} )? (!) )?

1.3:  $\langle short\ name \rangle$

1296        ( \ur{c\_\_bnvs\_short\_regex} )

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

1297        ( \ur{c\_\_bnvs\_path\_regex} )

2: without incement prefix

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

2.2: required !

1298        | (?: ( \ur{c\_\_bnvs\_short\_regex} )? (!) )?

2.3:  $\langle short\ name \rangle$

1299        ( \ur{c\_\_bnvs\_short\_regex} )

2.4: optionally followed by a dotted path

1300        ( \ur{c\_\_bnvs\_path\_regex} )

We continue with other expressions

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.

1301        (?: \s\* (\+?)= \s\* ( \S+ )

2.7: the post increment

1302        | (\+)\+

1303        )?

1304        ) \s\*

1305        }

”

(End of definition for `\c__bnvs_split_regex`.)

---

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



## 6.13 beamer.cls interface

Work in progress.

```

1306 \RequirePackage{keyval}

1307 \define@key{beamerframe}{beanoves-id}[]{}
1308 \tl_set:Nx \l__bnvs_id_last_tl { #1 }
1309 }

1310 \AddToHook{env/beamer@frameslide/before}{
1311   \__bnvs_greset_call:
1312   \__bnvs_v_gunset:
1313   \__bnvs_set_true:c { in_frame }
1314 }

1315 \AddToHook{env/beamer@frameslide/after}{
1316   \__bnvs_set_false:c { in_frame }
1317 }
```

## 6.14 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  $\backslash c\_bnvs\_colons\_regex$  and set the variables accordingly.  $\langle tl \rangle$  is expected to only contain colons and integers.

```

1318 \BNVS_new_conditional:cpnn { split_if_pop_left:c } #1 { T, F, TF } {
1319   \__bnvs_seq_pop_left:ccTF { split } { #1 } {
1320     \prg_return_true:
1321   } {
1322     \prg_return_false:
1323   }
1324 }

1325 \BNVS_new:cpn { split_if_pop_left:cTn } #1 #2 #3 {
1326   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS_split_F:n { #3 } }
1327 }

1328 \BNVS_new:cpn { split_if_pop_left_or:cT } #1 #2 {
1329   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS_split_F:n { #1 } }
1330 }
1331 \exp_args_generate:n { VVV }

1332 \BNVS_new_conditional:cpnn { range_if_set:cccn } #1 #2 #3 #4 { T, F, TF } {
1333   \BNVS_begin:
1334   \__bnvs_tl_clear:c A
1335   \__bnvs_tl_clear:c Z
1336   \__bnvs_tl_clear:c L
1337   \__bnvs_if_regex_split:cnTF { colons } { #4 } {
1338     \__bnvs_seq_pop_left:ccTF { split } A {

A may contain the  $\langle first \rangle$ , possibly empty, kept arround.

1339   \__bnvs_split_if_pop_left:cTF Z {
1340     \__bnvs_tl_if_empty:cTF Z {
```

This is a one colon  $\langle A \rangle : [^:]^*$ .

```
1341      \_bnvs_split_if_pop_left:cTF Z {
1342      \_bnvs_split_if_pop_left:cT L {
```

Z may contain the  $\langle last \rangle$  and there is more material.

```
1343      \_bnvs_tl_if_empty:cTF L {
```

A :: was expected:

```
1344      \BNVS_error:n { Invalid-range-expression(1):~#4 }
1345    } {
1346      \int_compare:nNtT { \_bnvs_tl_count:c L } > { 1 } {
1347        \BNVS_error:n { Invalid-range-expression(2):~#4 }
1348      }
1349      \_bnvs_split_if_pop_left:cTF L {
```

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

```
1350      \_bnvs_seq_if_empty:cF { split } {
1351        \BNVS_error:n { Invalid-range-expression(3):~#4 }
1352      }
1353    } {
1354      \BNVS_error:n { Unreachable~6~(range_if_set_cccnTF:nn) }
1355    }
1356  }
1357 }
1358 } {
1359   \BNVS_error:n { Unreachable~5~(range_if_set_cccnTF:nn) }
1360 }
1361 } {
```

This is a two colons  $\langle A \rangle :: \dots$ , we expect a length.

```
1362   \int_compare:nNtT { \_bnvs_tl_count:c Z } > { 1 } {
1363     \BNVS_error:n { Too-many-colons(1):~#4 }
1364   }
1365   \_bnvs_split_if_pop_left:cTF L {
```

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

```
1366   \_bnvs_split_if_pop_left:cTF Z {
1367     \_bnvs_tl_if_empty:cF Z {
1368       \BNVS_error:n { Too-many-colons(2):~#4 }
1369     }
1370     \_bnvs_split_if_pop_left:cTF Z {
```

Z may contain the  $\langle last \rangle$ .

```
1371     \_bnvs_seq_if_empty:cF { split } {
1372       \BNVS_error:n { Invalid-range-expression(5):~#4 }
1373     }
1374   } {
1375     \BNVS_error:n { Invalid-range-expression(6):~#4 }
1376   }
1377   } {
1378     \_bnvs_tl_clear:c Z
1379   }
1380 } {
1381   \BNVS_error:n { Unreachable~3~(range_if_set_cccnTF:nn) }
1382 }
```

```

1383     }
1384   } {
1385     \BNVS_error:n { Unreachable~2~(range_if_set_cccnTF:nn) }
1386   }
1387 } {
1388   \BNVS_error:n { Unreachable~1~(range_if_set_cccnTF:nn) }
1389 }

```

Providing both the  $\langle first \rangle$ ,  $\langle last \rangle$  and  $\langle length \rangle$  of a range is not allowed, even if they happen to be consistent. If there is not enough information, use 1 as  $\langle first \rangle$ .

```

1390   \__bnvs_tl_if_empty:cT A {
1391     \__bnvs_tl_if_empty:cTF Z {
1392       \__bnvs_tl_if_empty:cTF L {
1393         \BNVS_error:n { Invalid~range~expression(7):~#3 }
1394       } {
1395         \__bnvs_tl_set:cn A 1
1396       }
1397     } {
1398       \__bnvs_tl_if_empty:cT L {
1399         \__bnvs_tl_set:cn A 1
1400       }
1401     }
1402   }
1403   \cs_set:Npn \BNVS_range_if_set_cccnTF:w ##1 ##2 ##3 {
1404     \BNVS_end:
1405     \__bnvs_tl_set:cn { #1 } { ##1 }
1406     \__bnvs_tl_set:cn { #2 } { ##2 }
1407     \__bnvs_tl_set:cn { #3 } { ##3 }
1408   }
1409   \BNVS_exp_args:Nvvv \BNVS_range_if_set_cccnTF:w A Z L
1410   \prg_return_true:
1411 } {
1412   \BNVS_end:
1413   \prg_return_false:
1414 }
1415 }

```

---

$\backslash\_bnvs\_parse\_I:nn$   $\backslash\_bnvs\_parse\_I:nn \{ \langle tag \rangle \} \{ \langle value \rangle \}$

$\backslash\_bnvs\_parse\_I:nv$   $\backslash\_bnvs\_parsed\_IT:n \{ \langle value \rangle \}$

---

$\backslash\_bnvs\_parsed\_IT:n$  Auxiliary function for  $\backslash\_bnvs\_parse:n$  and  $\backslash\_bnvs\_parse:nn$  below. If  $\langle value \rangle$  does not correspond to a range, the V key is used. The  $\_n$  variant concerns the index counter. These are bottlenecks.

---

$\backslash\_bnvs\_range:nnnnn$   $\backslash\_bnvs\_range:nnnnn \{ \langle id \rangle \} \{ \langle tag \rangle \} \{ \langle first \rangle \} \{ \langle last \rangle \} \{ \langle length \rangle \}$

---

$\backslash\_bnvs\_range:nnvvv$  Auxiliary function called within a group. Setup the model to define a range.

```

1416 \BNVS_new:cpn { range:nnnnn } #1 #2 {
1417   \__bnvs_if:cTF { provide } {
1418     \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
1419       \use_none:nnn
1420     } {
1421       \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {

```

```

1422         \use_none:nnn
1423     } {
1424         \__bnvs_is_gset:nnnTF L { #1 } { #2 } {
1425             \use_none:nnn
1426         } {
1427             \__bnvs_do_range:nnnnn { #1 } { #2 }
1428         }
1429     }
1430 }
1431 } {
1432     \__bnvs_do_range:nnnnn { #1 } { #2 }
1433 }
1434 }

1435 \BNVS_new:cpn { range:nnvvv } #1 #2 {
1436     \BNVS_tl_use:nnvv {
1437         \__bnvs_range:nnnnn { #1 } { #2 }
1438     }
1439 }

1440 \BNVS_new:cpn { do_range:nnnnn } #1 #2 #3 #4 #5 {
1441     \__bnvs_gunset_deep:nn { #1 } { #2 }
1442     \__bnvs_gunset:nn { #1 } { #2 }
1443     \tl_if_empty:nTF { #5 } {
1444         \tl_if_empty:nTF { #3 } {
1445             \tl_if_empty:nTF { #4 } {
1446                 \BNVS_error:n { Not~a~range:~#1!#2 }
1447             } {
1448                 \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
1449                 \__bnvs_gset:nnnn A { #1 } { #2 } { 1 }
1450                 \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1451             }
1452         } {
1453             \__bnvs_gset:nnnn A { #1 } { #2 } { #3 }
1454             \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1455             \tl_if_empty:nF { #4 } {
1456                 \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
1457                 \__bnvs_gset:nnnn L { #1 } { #2 } { \q_nil }
1458             }
1459         }
1460     } {
1461         \tl_if_empty:nTF { #3 } {
1462             \__bnvs_gset:nnnn L { #1 } { #2 } { #5 }
1463             \tl_if_empty:nF { #4 } {
1464                 \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
1465                 \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
1466                 \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1467             }
1468         } {
1469             \__bnvs_gset:nnnn A { #1 } { #2 } { #3 }
1470             \__bnvs_gset:nnnn L { #1 } { #2 } { #5 }
1471             \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
1472             \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
1473         }
1474     }

```

```

1475 }
1476 \BNVS_new:cpn { range_IT:vvv } {
1477   \__bnvs_range:nnvvv { id } { tag }
1478 }

```

---

\\_\_bnvs\_parsed:nnn \\_\_bnvs\_parsed:nnn {<id>} {<tag>} {<one spec>}

Sets the internal model for the given <one spec>, either a value or a range specification.

```

1479 \BNVS_new:cpn { parsed:nnn } #1 #2 #3 {
1480   \__bnvs_range_if_set:cccnTF AZL { #3 } {
1481     \__bnvs_range:nnvvv { #1 } { #2 } AZL
1482   } {
1483     \__bnvs_is_provide_gset:nnnF V { #1 } { #2 } {
1484       \__bnvs_gunset_deep:nn { #1 } { #2 }
1485       \__bnvs_gunset:nn { #1 } { #2 }
1486       \tl_if_empty:nF { #3 } {
1487         \__bnvs_gset:nnnn V { #1 } { #2 } { #3 }
1488       }
1489     }
1490   }
1491 }

1492 \BNVS_new:cpn { parsed_IT:n } {
1493   \BNVS_tl_use:Nvv \__bnvs_parsed:nnn { id } { tag }
1494 }

```

---

\__bnvs_if_ref:nTF	\__bnvs_if_ref:nTF {<name>} {<yes code>} {<no code>}
\__bnvs_if_ref:nnTF	\__bnvs_if_ref:nnTF {<root>} {<relative>} {<yes code>} {<no code>}
\__bnvs_if_ref:vnTF	\__bnvs_if_ref_relative:nnTF {<root>} {<relative>} {<yes code>} {<no code>}
\__bnvs_if_ref_relative:nnTF	

---

If <name> is a reference, put the frame id it defines into `id` the short name into `short`, the dotted path into `path`, then execute <yes code>. Otherwise execute {<no code>}.

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

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

```

1495 \BNVS_new_conditional:cpnn { if_ref:n } #1 { T, F, TF } {
1496   \BNVS_begin:
1497   \__bnvs_match_if_once:NnTF \c__bnvs_A_ISP_Z_regex { #1 } {
1498     \__bnvs_if_match_pop_left:cTF { n } {
1499       \__bnvs_if_match_pop_left:cTF { id } {
1500         \__bnvs_if_match_pop_left:cTF { kri } {
1501           \__bnvs_if_match_pop_left:cTF { short } {
1502             \__bnvs_if_match_pop_left:cTF { path } {
1503               \cs_set:Npn \BNVS_aux_if_ref_nTF:nnn ##1 ##2 ##3 {
1504                 \BNVS_end:
1505                 \__bnvs_tl_set:cn { id } { ##1 }
1506                 \__bnvs_tl_set:cn { short } { ##2 }
1507                 \__bnvs_tl_set:cn { path } { ##3 }
1508               }
1509               \__bnvs_tl_if_empty:cTF { kri } {

```

```

1510         \BNVS_exp_args:Nvvv
1511         \BNVS_aux_if_ref_nTF:nnn
1512         { id_last }
1513     } {
1514         \BNVS_exp_args:Nvvv
1515         \BNVS_aux_if_ref_nTF:nnn
1516         { id }
1517     } { short } { path }
1518     \__bnvs_tl_set:cv { tag } { path }
1519     \__bnvs_tl_put_left:cv { tag } { short }
1520     \__bnvs_tl_set:cv { id_last } { id }
1521     \prg_return_true:
1522 } {
1523     \BNVS_end_unreachable_return_false:n { A_ISP_Z/path }
1524 }
1525 } {
1526     \BNVS_end_unreachable_return_false:n { A_ISP_Z/short }
1527 }
1528 } {
1529     \BNVS_end_unreachable_return_false:n { A_ISP_Z/kri }
1530 }
1531 } {
1532     \BNVS_end_unreachable_return_false:n { A_ISP_Z/id }
1533 }
1534 } {
1535     \BNVS_end_unreachable_return_false:n { A_ISP_Z/full_match }
1536 }
1537 } {
1538     \BNVS_end:
1539     \prg_return_false:
1540 }
1541 }

1542 \BNVS_new_conditional:cpnn { if_ref_relative:nn } #1 #2 { T, F, TF } {
1543     \BNVS_begin:
1544     \__bnvs_match_if_once:NnTF \c__bnvs_A_SP_Z_regex { #2 } {
1545         \__bnvs_if_match_pop_left:cTF { n } {
1546             \__bnvs_if_match_pop_left:cTF { short } {
1547                 \__bnvs_if_match_pop_left:cTF { path } {
1548                     \cs_set:Npn \BNVS_aux_if_ref_nTF:nn ##1 ##2 {
1549                         \BNVS_end:
1550                         \__bnvs_tl_put_right:cn { path } { . ##1 ##2 }
1551                     }
1552                     \BNVS_exp_args:Nvv
1553                     \BNVS_aux_if_ref_nTF:nn { short } { path }
1554                     \__bnvs_tl_set:cv { tag } { path }
1555                     \__bnvs_tl_put_left:cv { tag } { short }
1556                     \prg_return_true:
1557                 } {
1558                     \BNVS_end_unreachable_return_false:n { A_SP_Z/path }
1559                 }
1560             } {
1561                 \BNVS_end_unreachable_return_false:n { A_SP_Z/short }
1562             }

```

```

1563     } {
1564         \BNVS_end_unreachable_return_false:n { A_SP_Z/full_match }
1565     }
1566 } {
1567     \BNVS_end:
1568     \prg_return_false:
1569 }
1570 }

1571 \BNVS_new_conditional:cpnn { if_ref:nn } #1 #2 { T, F, TF } {
1572     \tl_if_empty:nTF { #1 } {
1573         \__bnvs_if_ref:nTF { #2 } {
1574             \prg_return_true:
1575         } {
1576             \prg_return_false:
1577         }
1578     } {
1579         \__bnvs_if_ref_relative:nnTF { #1 } { #2 } {
1580             \prg_return_true:
1581         } {
1582             \prg_return_false:
1583         }
1584     }
1585 }

1586 \BNVS_new_conditional:cpnn { if_ref:vn } #1 #2 { T, F, TF } {
1587     \BNVS_tl_use:Nv \__bnvs_if_ref:nnTF { #1 } { #2 } {
1588         \prg_return_true:
1589     } {
1590         \prg_return_false:
1591     }
1592 }

```

`\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`.)

```

1593 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1594     \A \s* (?

```

- 2  $\rightarrow$  V

```

1595     ( [^:]+? )

```

- 3, 4, 5  $\rightarrow$  A : Z? or A :: L?

```

1596     | (?: ( [^:]+? ) \s* : (?: \s* ( [^:]*? ) | : \s* ( [^:]*? ) ) ) )

```

- 6, 7  $\rightarrow$  ::(L:Z)?

```

1597     | (?: :: \s* (?: ( [^:]+? ) \s* : \s* ( [^:]+? ) )? )

```

- 8, 9  $\rightarrow$  :(Z::L)?

```

1598     | (?: : \s* (?: ( [^:]+? ) \s* :: \s* ( [^:]*? ) )? )
1599   )
1600   \s* \Z
1601 }

1602 \BNVS_int_new:c { next_IT }
1603 \BNVS_tl_new:c { next_IT }
1604 \BNVS_new:cpn { next_IT: } {
1605   \__bnvs_int_incr:c { next_IT }
1606   \__bnvs_tl_set_eq:cc { next_IT } { tag }
1607   \__bnvs_tl_put_right:cn { next_IT } { . }
1608   \BNVS_int_use:nv { \__bnvs_tl_put_right:cn { next_IT } } { next_IT }
1609   \__bnvs_is_gset:nvT V { id } { next_IT } { \__bnvs_next_IT: }
1610 }

1611 \BNVS_new:cpn { next_IT:n } #1 {
1612   \BNVS_begin:
1613   \__bnvs_next_IT:
1614   \__bnvs_tl_put_right:cn { tag } { . }
1615   \BNVS_int_use:nv { \__bnvs_tl_put_right:cn { tag } } { next_IT }
1616   #1
1617   \BNVS_end_int_set:cv { next_IT } { next_IT }
1618 }

```

### 6.14.1 Square brackets

See section 3.3.1.

---

```

\__bnvs_bracket_parse_IT:n \__bnvs_bracket_keyval_IT:n {\<definitions>}
\__bnvs_bracket_parse_IT:nn \__bnvs_bracket_parse_IT:nn {\<name>} {\<spec>}
\__bnvs_bracket_keyval_IT:n \__bnvs_bracket_parse_IT:n {\<spec>}

```

---

To parse what is inside square brackets.

```

1619 \BNVS_new:cpn { bracket_assign:nnn } #1 #2 #3 {
1620   \__bnvs_gunset_deep:nn { #1 } { #2 }
1621   \__bnvs_gunset:nn { #1 } { #2 }
1622   \tl_if_empty:nF { #3 } {
1623     \__bnvs_gset:nnnn V { #1 } { #2 } { #3 }
1624   }
1625 }
1626 \BNVS_new:cpn { bracket_assign_I:cn } {
1627   \BNVS_tl_use:Nvv \__bnvs_bracket_assign:nnn { id }
1628 }
1629 \BNVS_new:cpn { bracket_assign_IT:n } {
1630   \__bnvs_bracket_assign_I:cn { tag }
1631 }
1632 \BNVS_tl_new:c { bracket_assign_IT_nn }
1633 \BNVS_new:cpn { bracket_assign_IT:nn } #1 {
1634   \__bnvs_tl_set_eq:cc { bracket_assign_IT_nn } { tag }
1635   \__bnvs_tl_put_right:cn { bracket_assign_IT_nn } { .#1 }
1636   \__bnvs_bracket_assign_I:cn { bracket_assign_IT_nn }
1637 }

```



```

1638 \BNVS_new:cpn { bracket_parse_IT:nn } #1 #2 {
1639   \__bnvs_match_if_once:NnTF \c__bnvs_colons_regex { #2 } {
1640     \BNVS_error:n { No~colon~allowed:~[...=#2...]}
1641   } {
1642     \__bnvs_match_if_once:NnTF \c__bnvs_A_integer_Z_regex { #1 } {
1643       \__bnvs_bracket_assign_IT:nn {#1 } { #2 }
1644     } {
1645       \BNVS_error:n { Not~an~integer:~#1 }
1646     }
1647   }
1648 }

1649 \BNVS_new:cpn { bracket_parse_IT:n } #1 {
1650   \__bnvs_match_if_once:NnTF \c__bnvs_colons_regex { #1 } {
1651     \BNVS_error:n { No~colon~allowed:~[...#1...]}
1652   } {

```

Find the first available index.

```

1653   \__bnvs_next_IT:n { \__bnvs_bracket_assign_IT:n { #1 } }
1654 }
1655 }

```

For X=[...].

```

1656 \BNVS_new:cpn { bracket_keyval_IT:n } #1 {

```

The root tl variable is set and not empty. Remove what is related to tag.

```

1657   \__bnvs_tl_if_empty:cTF { tag } {
1658     \BNVS_error:n { Unexpected~list~at~top~level. }
1659   } {
1660     \__bnvs_is_provide_gset:nvvF V { id } { tag } {
1661       \BNVS_begin:
1662       \__bnvs_gunset_deep:vv { id } { tag }
1663       \__bnvs_gunset:vv { id } { tag }
1664       \__bnvs_int_zero:c { next_IT }
1665       \keyval_parse:nnn
1666       \__bnvs_bracket_parse_IT:n \__bnvs_bracket_parse_IT:nn { #1 }
1667       \BNVS_end:
1668     }
1669   }
1670 }

```

### 6.14.2 Root level

---

```

\__bnvs_root_parse:n \__bnvs_root_keyval:n {\definitions}
\__bnvs_root_parse:nn \__bnvs_root_parse:nn {\name} {\spec}
\__bnvs_root_keyval:n \__bnvs_root_parse:n {\name}

```

---

To parse what is at the root level.

```

1671 \BNVS_new:cpn { root_parse:n } #1 {
1672 }

1673 \BNVS_new:cpn { root_parse:nn } #1 #2 {

```

```
1674 }
```

```
1675 \BNVS_new:cpn { root_keyval:n } #1 {
```

This is a list, possibly at the top

```
1676 \BNVS_begin:
```

```
1677 \keyval_parse:nnn \BNVS_root_parse:n \BNVS_root_parse:nn { #1 }
```

```
1678 \BNVS_end:
```

This is a list, possibly at the top

```
1679 }
```

`\l__bnvs_match_seq` Local storage for the match result.

*(End of definition for `\l__bnvs_match_seq`.)*

The `a` `tl` auxiliary variable is used.

`\c__bnvs_one_suffix_regex` To catch the suffix 1 or first.

*(End of definition for `\c__bnvs_one_suffix_regex`.)*

```
1680 \regex_const:Nn \c__bnvs_one_suffix_regex { \A(.*)\.(?:1|first)\Z }
```

### 6.14.3 List specifiers

Here is the [documentation](#). For  $\langle ref \rangle = \langle spec \rangle$ .

---

`\__bnvs_list_keyval_IT:n` `\__bnvs_list_keyval_IT:n { $\langle definitions \rangle$ }`

---

Auxiliary functions called within a group by `\keyval_parse:nnn`.  $\langle name \rangle$  is the overlay set name, including eventually a dotted path or a frame identifier,  $\langle definition \rangle$  is the corresponding definition. The `id` and `tag` `tl` variables are set beforehand.

We parse all at once, then manage what is parsed. We could avoid a grouping level.

```
1681 \BNVS_new:cpn { list_keyval_IT:n } #1 {
1682 \BNVS_begin:
1683 \__bnvs_tl_clear:c V
1684 \__bnvs_tl_clear:c a
1685 \cs_set:Npn \BNVS: {
1686 \cs_set:Npn \BNVS:n #####1 {
1687 \__bnvs_tl_put_right:cn a { \BNVS:n { #####1 } }
1688 }
1689 \cs_set:Npn \BNVS:nn #####1 #####2 {
1690 \__bnvs_tl_put_right:cn a { \BNVS:nn { #####1 } { #####2 } }
1691 }
1692 }
1693 \cs_set:Npn \BNVS:n {
1694 \BNVS:
1695 \__bnvs_tl_set:cn V
1696 }
1697 \cs_set:Npn \BNVS:nn {
1698 \BNVS:
1699 \BNVS:nn
1700 }
1701 \keyval_parse:nnn \BNVS:n \BNVS:nn { #1 }
1702 \__bnvs_tl_if_empty:cTF a {
1703 \__bnvs_tl_if_empty:cTF V {
```

```

1704     } {
1705         \BNVS_tl_use:nv { \_bnvs_range_if_set:cccnTF A Z L } V {
1706     } {
1707         \_bnvs_is_provide_gset:nvvF V { id } { tag } {
1708             \_bnvs_gunset_deep:vv { id } { tag }
1709             \_bnvs_gunset:vv { id } { tag }
1710             \_bnvs_gset:nvvv V { id } { tag } V
1711         }
1712     }
1713 }

```

A single value or range specification.

```

1714     } {
1715     }
1716     \BNVS_end:
1717 }

```

---

<code>\_bnvs_brace_parse:n</code> <code>\_bnvs_brace_parse:nn</code>	<code>\_bnvs_brace_parse:nn {&lt;name&gt;} {&lt;spec&gt;}</code> <code>\_bnvs_brace_parse:nn {&lt;named&gt;}</code>
---	--

---

Auxiliary functions called within a group by `\keyval:nnn`. `<name>` is the overlay set name, including eventually a dotted path or a frame identifier, `<definition>` is the corresponding definition.

```

1718 \exp_args_generate:n { nne }
1719 \exp_args_generate:n { nnne }
1720 \BNVS_new:cpn { brace_parse:n } #1 {

```

For  $X=\{\{\dots,Y,\dots\}\}$ .

```

1721     \_bnvs_range_if_set:cccnTF A Z L { #1 } {
1722         \_bnvs_int_zero:c { next_IT }
1723         \_bnvs_next_IT:n {
1724             \_bnvs_set_true:c { deep }
1725             \BNVS_log_tl:c A
1726             \BNVS_log_tl:c Z
1727             \BNVS_log_tl:c L
1728             \typein { F00000000-#1 }
1729             \BNVS_int_use:Nv \_bnvs_parsed:nn { next_IT } { #1 }
1730         }
1731     } {
1732         \typein { F000000-#1 }
1733         \_bnvs_set_true:c { deep }
1734         \_bnvs_parsed:nn { #1 } { 1 }
1735         \typein { F000000-#1 }
1736     }
1737 }

1738 \BNVS_new:cpn { warn_up_to_q_stop:w } #1 \q_stop {
1739     \BNVS_warning:n { Ignored:~#1 }
1740 }
1741 \BNVS_new:cpn { scan_up_to_q_stop:w } {
1742     \peek_meaning:NTF \q_stop {
1743         \use_none:n
1744     } {
1745         \_bnvs_warn_up_to_q_stop:w

```

```

1746 }
1747 }

```

Key only item.

```

1748 \BNVS_new:cpn { parse_P_IT:n } #1 {
1749   \__bnvs_match_if_once:NnTF \c__bnvs_A_P_Z_regex { #1 } {
1750     \exp_args:Nnnx \BNVS_tl_use:nv { \__bnvs_gset:nnnn V } { id } {
1751       \BNVS_tl_use:c { root } #1
1752     } 1
1753   } {
1754     \BNVS_error:n { Unsupported~#1 }
1755   }
1756 }

```

```

1757 \BNVS_new:cpn { parse_Pd_IT:nn } #1 #2 {
1758   \__bnvs_match_if_once:NnTF \c__bnvs_A_Pd_Z_regex { #1 } {
1759     \exp_args:Nnnx \BNVS_tl_use:nv { \__bnvs_gset:nnnn V } { id } {
1760       \BNVS_tl_use:c { root } #1
1761     } 1
1762   } {
1763     \BNVS_error:n { Unsupported~#1 }
1764   }
1765 }

```

```

1766 \BNVS_new:cpn { brace_parse_IT:nw } #1 {
1767   \BNVS_begin:

```

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

```

1768   \__bnvs_tl_put_right:cv { root } { tag }
1769   \__bnvs_tl_put_right:cn { root } { . }
1770
1771   \keyval_parse:nnn \BNVS_parse_P_IT:n \BNVS:nn { #1 }
1772
1773
1774
1775
1776
1777   \BNVS_end:
1778   \__bnvs_scan_up_to_q_stop:w
1779 }

```

```

1780 \BNVS_new:cpn { brace_parse_IT:n } #1 {
1781   \BNVS_begin:

```

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

```

1782   \exp_args:Nc
1783   \peek_meaning:NnTF { BNVS[]:w } {

```

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

```

1784   \BNVS_begin:

```

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

```

1785   \cs_set:cpn { BNVS[]:w } ##1 ##2 \q_stop {
1786     \regex_match:nnT { \S } { ##2 } {
1787       \BNVS_warning:n { Ignoring~##2 }
1788     }

```

```

1789     \__bnvs_bracket_keyval_IT:n { ##1 }
1790     \BNVS_end:
1791   }
1792 } {
1793   \peek_meaning:NTF \c_group_begin_token {
1794     \BNVS_use:c { {...}:w }
1795   } {
1796     \peek_meaning:NTF \q_stop: {
1797       \use_none:n
1798     } {
1799       \BNVS_use:c { {...}:w }
1800     }
1801   }
1802 }
1803 #1 \q_stop

```

We export \l\_\_bnvs\_id\_last\_tl:

```

1804   \BNVS_end_tl_set:cv { id_last } { id_last }
1805 }

1806 \BNVS_new:cpn { brace_parse:nn } #1 #2 {
1807   \BNrcode}
1808 % \begin{BNVS.gobble}
1809 </!debug>
1810 % \end{BNVS.gobble}
1811 % \end{BNVS.macrocode}
1812 % We prepend the argument with |root|, in case we are recursive.
1813 % \begin{BNVS.macrocode}
1814 %   \begin{macrocode}
1815   \__bnvs_if_ref:vnTF { root } { #1 } {
1816     \exp_args:Nc
1817     \peek_meaning:NTF { BNVS[]:w } {

```

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

```

1818   \BNVS_begin:

```

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

```

1819   \cs_set:cpn { BNVS[]:w } ##1 ##2 \s_stop {
1820     \regex_match:nnT { \S } { ##2 } {
1821       \BNVS_warning:n { Ignoring~##2 }
1822     }
1823     \__bnvs_bracket_keyval_IT:n { ##1 }
1824     \BNVS_end:
1825   }
1826 } {
1827   \peek_meaning:NTF \s_stop: {
1828     \use_none:n
1829   } {
1830     \BNVS_use:c { X={...}:w }
1831   }
1832 }
1833 #2 \s_stop
1834 } {
1835   \BNVS_error:n { Invalid-name:~#2 }
1836 }

```

We export `\l__bnvs_id_last_tl`:

```

1837 \BNVS_end_tl_set:cv { id_last } { id_last }
1838 }

1839 \cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
1840 \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
1841 { #4 }
1842 }
1843 \cs_new:Npn \BNVS_end_tl_set:cv #1 {
1844 \BNVS_tl_use:nv {
1845 \BNVS_end: \__bnvs_tl_set:cn { #1 }
1846 }
1847 }

1848 \cs_new:Npn \BNVS_end_int_set:cv #1 {
1849 \BNVS_int_use:nv {
1850 \BNVS_end: \__bnvs_int_set:cn { #1 }
1851 }
1852 }

```

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

- raw beamer lists  $X=A$  or  $X=\{A,\dots\}$ ,
- integer-value lists  $X=[A,B]$ .
- key-value lists  $X=\{\{A,B\}\}$ ,

```

1853 \BNVS_new:cpn { parsed:nn } #1 #2 {
1854 \BNVS_begin:

```

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

```

1855 \__bnvs_if_ref:vnTF { root } { #1 } {
1856 \exp_args:Nc
1857 \peek_meaning:NTF { BNVS[]:w } {

```

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

```

1858 \BNVS_begin:

```

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

```

1859 \cs_set:cpn { BNVS[]:w } ##1 ##2 \s_stop {
1860 \regex_match:nnT { \S } { ##2 } {
1861 \BNVS_warning:n { Ignoring~##2 }
1862 }
1863 \__bnvs_bracket_keyval_IT:n { ##1 }
1864 \BNVS_end:
1865 }
1866 } {
1867 \peek_meaning:NTF \s_stop: {
1868 \use_none:n
1869 } {
1870 \BNVS_use:c { X={\dots}:w }
1871 }
1872 }

```

```

1873     #2 \s_stop
1874 } {
1875     \BNVS_error:n { Invalid-name:~#2 }
1876 }

```

We export \l\_\_bnvs\_id\_last\_tl:

```

1877 \__bnvs_match_if_once:NvT \c__bnvs_one_suffix_regex { tag } {
1878     \__bnvs_if_match_pop_left:cTF { a } {
1879         \__bnvs_if_match_pop_left:cTF { a } {
1880             \cs_set:Npn \BNVS_aux_parsed_nn: {
1881                 \__bnvs_gset:nvvn V { id } { a } { #2 }
1882             }
1883             \__bnvs_if_get:nvvcT V { id } { a } { b } {
1884                 \__bnvs_quark_if_nil:cF { b } {
1885                     \cs_set:Npn \BNVS_aux_parsed_nn: { }
1886                 }
1887             }
1888             \BNVS_aux_parsed_nn:
1889         } {
1890             \BNVS_error:n { Unreachable~2 }
1891         }
1892     } {
1893         \BNVS_error:n { Unreachable~1 }
1894     }
1895 }

```

We export \l\_\_bnvs\_id\_last\_tl:

```

1896 \BNVS_end_tl_set:cv { id_last } { id_last }
1897 }

```

```

1898 \BNVS_new:cpn { X={...}:w } #1 \s_stop {
1899     \__bnvs_gunset_deep:vv { id } { tag }

```

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

```

1900     \BNVS_begin:

```

Remove the elements that contain a =.

```

1901     \__bnvs_tl_put_right:cv { root } { tag }
1902     \__bnvs_tl_put_right:cn { root } { . }
1903     \__bnvs_brace_keyval:n { #1 }
1904     \BNVS_end:
1905 }

```

```

1906 \BNVS_new:cpn { X=...:n } #1 {

```

This is a  $X=...$

```

1907     \BNVS_begin:
1908     \__bnvs_tl_clear:c { a }
1909     \__bnvs_seq_clear:c { a }
1910     \cs_set:Npn \BNVS_aux_parsed_nn:n ##1 {
1911         \__bnvs_tl_set:cn { a } { ##1 }
1912         \cs_set:Npn \BNVS_aux_parsed_nn:n ####1 {
1913             \__bnvs_seq_put_right:cn { a } { ####1 }
1914         }
1915     }
1916     \cs_set:Npn \BNVS_aux_parsed_nn:nn ##1 ##2 {

```

```

1917     \BNVS_warning:n { Ignored:~##1=##2 }
1918 }
1919 \keyval_parse:nnn \BNVS_aux_parsed_nn:n \BNVS_aux_parsed_nn:nn { #1 }
1920 \__bnvs_tl_if_empty:cTF { a } {

```

Clean everything, whether in provide mode or not.

```

1921     \__bnvs_gunset_deep:vv { id } { tag }
1922     \__bnvs_gunset:vv { id } { tag }
1923 } {

```

The first definition.

```

1924     \BNVS_tl_use:Nv \__bnvs_parsed_IT:n { a }
1925 }
1926 \BNVS_end:
1927 }

```

---

```

\__bnvs_parsed_IT[=...]:n \__bnvs_parsed_IT[=...]:n {<definitions>}

```

---

Used by \\_\_bnvs\_parse:nn . <id> and <tag> are set. Store the associate values. <definitions> is a comma separated list of <definition>'s, either for ranges or values. The first <definition> is for the V and AZL keys.

```

1928 \BNVS_new:cpn { parsed_IT[=...]:n } #1 {
1929   \BNVS_begin:
1930   \__bnvs_tl_clear:c { a }
1931   \__bnvs_seq_clear:c { a }
1932   \cs_set:Npn \BNVS_aux_parsed_nn:n ##1 {
1933     \__bnvs_tl_set:cn { a } { ##1 }
1934     \cs_set:Npn \BNVS_aux_parsed_nn:n ##1 {
1935       \__bnvs_seq_put_right:cn { a } { ##1 }
1936     }
1937   }
1938   \cs_set:Npn \BNVS_aux_parsed_nn:nn ##1 ##2 {
1939     \BNVS_warning:n { Ignored:~##1=##2 }
1940   }
1941   \keyval_parse:nnn \BNVS_aux_parsed_nn:n \BNVS_aux_parsed_nn:nn { #1 }
1942   \__bnvs_tl_if_empty:cTF { a } {

```

Clean everything, whether in provide mode or not.

```

1943     \__bnvs_gunset:vv { id } { tag }
1944 } {

```

The first definition.

```

1945     \__bnvs_seq_if_empty:cTF { a } {
1946     \BNVS_tl_use:Nv \__bnvs_parsed_IT:n { a }
1947   } {
1948     \__bnvs_int_zero:c { i }
1949     \__bnvs_seq_map_inline:cn { a } {
1950       \__bnvs_int_incr:c { i }
1951     }
1952     \__bnvs_tl_put_right:cn { tag } { . }
1953     \exp_args:Nnx \__bnvs_tl_put_right:cn { tag } {
1954       \__bnvs_int_use:c { i }
1955     }
1956     \__bnvs_parsed_IT:n { i }

```



```

1957     }
1958   }
1959 }
1960 }
1961 \BNVS_end:
1962 }

1963 \BNVS_new:cpn { parse_prepare:N } #1 {
1964   \tl_set:Nx #1 #1
1965   \__bnvs_set_false:c { parse }
1966   \bool_do_until:Nn \l__bnvs_parse_bool {
1967     \tl_if_in:NnTF #1 {%---[
1968       ]} {
1969       \regex_replace_all:nnNF { \[ ([^\]]%---)
1970       ]*%---[(
1971       ) \] } { \c{BNVS[]:w} { \1 } } #1 {
1972       \__bnvs_set_true:c { parse }
1973     }
1974   } {
1975     \__bnvs_set_true:c { parse }
1976   }
1977 }
1978 \tl_if_in:NnTF #1 {%---[
1979 ]} {
1980   \BNVS_error:n { Unbalanced~%---[
1981   ]}
1982 } {
1983   \tl_if_in:NnT #1 { [%---]
1984   } {
1985     \BNVS_error:n { Unbalanced~[ %---]
1986     }
1987   }
1988 }
1989 }

```

---

<code>\__bnvs_bracket_keyval:IT:n</code>	<code>\__bnvs_bracket_keyval:IT:n {⟨definitions⟩}</code>
<code>\__bnvs_brace_keyval:n</code>	<code>\__bnvs_brace_keyval:n {⟨definitions⟩}</code>

---

At the top level, the Y in X=123, Y, Z=456 or X={X=123, Y, Z=456}.

```

1990 \BNVS_new:cpn { brace_keyval:n } {
1991   \keyval_parse:nnn \__bnvs_brace_parse:n \__bnvs_brace_parse:nn
1992 }

```

---

<code>\Beanoves</code>	<code>\Beanoves {⟨key-value list⟩}</code>
------------------------	---

---

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`.

```

1993 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1994   \BNVS_tl_use:nv {
1995     \BNVS_end:
1996     \__bnvs_tl_put_right:cn { #1 }
1997   } { #2 }
1998 }

```

```

1999 \cs_new:Npn \BNVS_end_gset:nnnv #1 #2 #3 {
2000   \BNVS_tl_use:nv {
2001     \BNVS_end:
2002     \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
2003   }
2004 }

```

```

2005 \NewDocumentCommand \Beanoves { sm } {
2006   \__bnvs_set_false:c { reset }
2007   \__bnvs_set_false:c { reset_all }
2008   \__bnvs_set_false:c { only }
2009   \tl_if_empty:NTF \@currentenvir {

```

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

```

2010   \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
2011 } {
2012   \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, clear everything.

```

2013   \__bnvs_gunset:
2014 }
2015 \BNVS_begin:
2016 \__bnvs_tl_clear:c { root }
2017 \__bnvs_int_zero:c { i }
2018 \__bnvs_tl_set:cn { a } { #2 }
2019 \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, use the global definitions.

```

2020   \seq_if_empty:NF \g__bnvs_def_seq {
2021     \__bnvs_tl_put_left:cx { a } {
2022       \seq_use:Nn \g__bnvs_def_seq , ,
2023     }
2024   }
2025 }
2026 \__bnvs_parse_prepare:N \l__bnvs_a_tl
2027 \IfBooleanTF {#1} {
2028   \__bnvs_provide_on:
2029 } {
2030   \__bnvs_provide_off:
2031 }
2032 \BNVS_tl_use:Nv \__bnvs_brace_keyval:n { a }
2033 \BNVS_end_tl_set:cv { id_last } { id_last }
2034 \ignorespaces
2035 }
2036 }

```

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

```

2037 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}

```

## 6.15 Scanning named overlay specifications

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

---

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

---

Preprocess `⟨overlay specification⟩` before beamer reads it.

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

(End of definition for `\l__bnvs_ans_tl`.)

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

```

2038 \makeatletter
2039 \cs_set:Npn \_bnvs@frame < #1 > {
2040   \BNVS_begin:
2041   \_bnvs_tl_clear:c { ans }
2042   \_bnvs_scan:nNc { #1 } \_bnvs_if_resolve:ncTF { ans }
2043   \BNVS_set:cpn { :n } ##1 { \BNVS_end: \BNVS_saved@frame < ##1 > }
2044   \BNVS_tl_use:cv { :n } { ans }
2045 }

2046 \cs_set:Npn \_bnvs@masterdecode #1 {
2047   \BNVS_begin:
2048   \_bnvs_tl_clear:c { ans }
2049   \_bnvs_scan:nNc { #1 } \_bnvs_if_resolve_queries:ncTF { ans }
2050   \BNVS_tl_use:nv {
2051     \BNVS_end:
2052     \BNVS_saved@masterdecode
2053   } { ans }
2054 }

2055 \cs_new:Npn \BeanovesOff {
2056   \cs_set_eq:NN \beamer@frame \BNVS_saved@frame
2057   \cs_set_eq:NN \beamer@masterdecode \BNVS_saved@masterdecode
2058 }

2059 \cs_new:Npn \BeanovesOn {
2060   \cs_set_eq:NN \beamer@frame \_bnvs@frame
2061   \cs_set_eq:NN \beamer@masterdecode \_bnvs@masterdecode
2062 }

2063 \AddToHook{begindocument/before}{
2064   \cs_if_exist:NTF \beamer@frame {
2065     \cs_set_eq:NN \BNVS_saved@frame \beamer@frame
2066     \cs_set_eq:NN \BNVS_saved@masterdecode \beamer@masterdecode
2067   } {
2068     \cs_set:Npn \BNVS_saved@frame < #1 > {
2069       \BNVS_error:n {Missing~package~beamer}
2070     }
2071     \cs_set:Npn \BNVS_saved@masterdecode < #1 > {
2072       \BNVS_error:n {Missing~package~beamer}
2073     }
2074   }

```

```

2075 \BeanovesOn
2076 }
2077 \makeatother

```

---

\\_bnvs\_scan:nNc \\_bnvs\_scan:nNc {<overlay query>} <resolve> {<ans>}

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

\l\_\_bnvs\_ans\_tl The token list that will be appended to <tl variable> on return.  
(End of definition for \l\_\_bnvs\_ans\_tl.)

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

\l\_\_bnvs\_query\_tl Storage for the overlay query expression to be evaluated.  
(End of definition for \l\_\_bnvs\_query\_tl.)

\l\_\_bnvs\_token\_seq The <overlay expression> is split into the sequence of its tokens.  
(End of definition for \l\_\_bnvs\_token\_seq.)

\l\_\_bnvs\_token\_tl Storage for just one token.  
(End of definition for \l\_\_bnvs\_token\_tl.)

---

\\_bnvs\_scan:nNcTF \\_bnvs\_scan:nNcTF {<overlay query>} <resolve> {<ans>} {<yes code>} {<no code>}

Next are helpers.

---

\\_bnvs\_scan\_for\_query\_then\_end\_return: \\_bnvs\_scan\_for\_query\_then\_end\_return:

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

```

2078 \BNVS_new:cpn { scan_for_query_then_end_return: } {
2079   \_bnvs_seq_pop_left:ccTF { token } { token } {
2080     \_bnvs_tl_if_eq:cnTF { token } { ? } {
2081       \_bnvs_scan_require_open_end_return:
2082     } {
2083       \_bnvs_tl_put_right:cv { ans } { token }
2084       \_bnvs_scan_for_query_then_end_return:
2085     }
2086   } {
2087     \_bnvs_scan_end_return_true:
2088   }
2089 }

```

---

\\_\_bnvs\_scan\_require\_open\_end\_return: \\_\_bnvs\_scan\_require\_open\_end\_return:

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

```
2090 \BNVS_new:cpn { scan_require_open_end_return: } {
```

Get next token.

```
2091 \__bnvs_seq_pop_left:ccTF { token } { token } {
2092   \str_if_eq:VnTF \l__bnvs_token_tl { ( % )
2093   } {
```

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

```
2094   \__bnvs_int_set:cn { } { 1 }
```

Record the forthcoming content in the \l\_\_bnvs\_query\_tl variable, up to the next balancing ')':

```
2095   \__bnvs_tl_clear:c { query }
2096   \__bnvs_scan_require_close_and_return:
2097   }
```

Ignore this token and loop.

```
2098   \__bnvs_scan_require_open_end_return:
2099   }
2100 }
```

Get next token.

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

```
2101 \BNVS_error:x {Missing~'('%---)
2102   ~after~a~? }
2103 \__bnvs_scan_end_return_true:
2104 }
2105 }
```

---

\\_\_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 ')':

```
2106 \BNVS_new:cpn { scan_require_close_and_return: } {
```

Get next token.

```
2107 \__bnvs_seq_pop_left:ccTF { token } { token } {
2108   \str_case:VnF \l__bnvs_token_tl {
2109     { ( %---)
2110   } {
```

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

```
2111   \__bnvs_int_incr:c { }
2112   \__bnvs_tl_put_right:cv { query } { token }
2113   \__bnvs_scan_require_close_and_return:
2114   }
2115   { %(---
2116   )
2117   }
```

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

```

2118         \__bnvs_int_decr:c { }
2119         \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 ‘?’.

```

2120         \__bnvs_scan_handle_query_then_end_return:
2121     } {

```

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

```

2122         \__bnvs_tl_put_right:cv { query } { token }
2123         \__bnvs_scan_require_close_and_return:
2124     }
2125 }
2126 } {

```

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

```

2127         \__bnvs_tl_put_right:cv { query } { token }
2128         \__bnvs_scan_require_close_and_return:
2129     }
2130 } {

```

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 ‘)’.

```

2131     \BNVS_error:x { Missing~%(---
2132     `)' }
2133     \__bnvs_tl_put_right:cx { query } {
2134         \prg_replicate:nn { \l__bnvs_int } {%(---
2135         )}
2136     }
2137     \__bnvs_scan_end_return_true:
2138 }
2139 }

2140 \BNVS_new_conditional:cpnn { scan:nNc } #1 #2 #3 { T, F, TF } {
2141     \BNVS_begin:
2142     \BNVS_set:cpn { error:x } ##1 {
2143         \msg_error:nnx { beanoves } { :n }
2144         { \tl_to_str:n { #1 } :~##1}
2145     }
2146     \__bnvs_tl_set:cn { scan } { #1 }
2147     \__bnvs_tl_clear:c { ans }
2148     \__bnvs_seq_clear:c { token }

```

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

```

2149     \regex_split:nnN { } { #1 } \l__bnvs_token_seq

```

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

```

2150     \BNVS_set:cpn { scan_handle_query_then_end_return: } {
2151         \BNVS_tl_use:Nv #2 { query } { ans } {

```

```

2152     \__bnvs_scan_for_query_then_end_return:
2153 } {
2154     \BNVS_end_tl_put_right:cv { #3 } { ans }

```

Stop on the first error.

```

2155     \prg_return_false:
2156 }
2157 }
2158 \BNVS_set:cpn { scan_end_return_true: } {
2159     \BNVS_end_tl_put_right:cv { #3 } { ans }
2160     \prg_return_true:
2161 }
2162 \BNVS_set:cpn { scan_end_return_false: } {
2163     \BNVS_end_tl_put_right:cv { #3 } { ans }
2164     \prg_return_false:
2165 }
2166 \__bnvs_scan_for_query_then_end_return:
2167 }
2168 \BNVS_new:cpn { scan:nNc } #1 #2 #3 {
2169     \BNVS_use:c { scan:nNcTF } { #1 } #2 { #3 } {} {}
2170 }

```

## 6.16 Resolution

Given a name, a frame id and a dotted path, we resolve any intermediate standalone reference. For example, with A=B and B=C, A is resolved in C. But with A=B+1 and B=C, A is not resolved in C+1. With A=B:D and B=C, A is not resolved in C:D neither.

---

```

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

```

---

Auxiliary function. On input, the *<name>* tl variable contains a set name whereas the *<id>* tl variable contains a frame id. If *<name>* tl variable contents is a recorded set, on return, *<tag>* 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.

```

2171 \BNVS_new_conditional:cpnn { if_TIP:ccc } #1 #2 #3 { T, F, TF } {
2172     \BNVS_begin:
2173     \__bnvs_match_if_once:NvTF \c__bnvs_A_ref_Z_regex { #1 } {

```

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

```

2174     \__bnvs_if_match_pop_TIP:cccTF { #1 } { #2 } { #3 } {
2175         \__bnvs_export_TIP:cccN { #1 } { #2 } { #3 }
2176         \BNVS_end:
2177         \prg_return_true:
2178     } {
2179         \BNVS_end:
2180         \prg_return_false:
2181     }
2182 } {
2183     \BNVS_end:

```

```

2184     \prg_return_false:
2185 }
2186 }
2187 \quark_new:N \q__bnvs

2188 \tl_new:N \l__bnvs_export_TIP_cccN_tl
2189 \BNVS_new:cpn { export_TIP:cccN } #1 #2 #3 #4 {
2190   \cs_set:Npn \BNVS_export_TIP_cccN:w ##1 ##2 ##3 {
2191     #4
2192     \__bnvs_tl_set:cn { #1 } { ##1 }
2193     \__bnvs_tl_set:cn { #2 } { ##2 }
2194     \__bnvs_tl_set:cn { export_TIP_cccN } { ##3 }
2195   }
2196   \__bnvs_tl_set:cx { export_TIP_cccN }
2197   { \__bnvs_seq_use:cn { #1 } { \q__bnvs } }
2198   \BNVS_tl_use:nvv {
2199     \BNVS_tl_use:Nv \BNVS_export_TIP_cccN:w { #1 }
2200   } { #2 } { export_TIP_cccN }
2201   \BNVS_tl_use:nv {
2202     \__bnvs_seq_set_split:cn { #3 } { \q__bnvs }
2203   } { export_TIP_cccN }
2204   \__bnvs_seq_remove_all:cn { #3 } { }
2205 }

2206 \tl_new:N \l__bnvs_if_match_export_ISPn_cccc_tl
2207 \BNVS_new:cpn { if_match_export_ISPn:cccc } #1 #2 #3 #4 #5 {
2208   \cs_set:Npn \BNVS_if_match_export_ISPn_ccccN:w ##1 ##2 ##3 ##4 {
2209     #5
2210     \__bnvs_tl_set:cn { #1 } { ##1 }
2211     \__bnvs_tl_set:cn { #2 } { ##2 }
2212     \__bnvs_tl_set:cn { #3 } { ##3 }
2213     \__bnvs_tl_set:cn { #4 } { ##4 }
2214   }
2215   \__bnvs_tl_set:cx { if_match_export_ISPn_cccc }
2216   { \__bnvs_seq_use:cn { #1 } { \q__bnvs } }
2217   \BNVS_tl_use:nvvv {
2218     \BNVS_tl_use:Nv \BNVS_if_match_export_ISPn_ccccN:w { #1 }
2219   } { #2 } { if_match_export_ISPn_cccc } { #4 }
2220   \BNVS_tl_use:nv {
2221     \__bnvs_seq_set_split:cn { #3 } { \q__bnvs }
2222   } { if_match_export_ISPn_cccc }
2223   \__bnvs_seq_remove_all:cn { #3 } { }
2224 }

2225 \BNVS_new_conditional:cpnn { if_match_pop_ISPn:cccc } #1 #2 #3 #4 { TF } {
2226   \BNVS_begin:
2227   \__bnvs_if_match_pop_left:cTF { #1 } {
2228     \__bnvs_if_match_pop_left:cTF { #1 } {
2229       \__bnvs_if_match_pop_left:cTF { #2 } {
2230         \__bnvs_if_match_pop_left:cTF { #3 } {
2231           \__bnvs_seq_set_split:cn { #3 } { . } { #3 }
2232           \__bnvs_seq_remove_all:cn { #3 } { }
2233           \__bnvs_if_match_pop_left:cTF { #4 } {
2234             \__bnvs_if_match_export_ISPn:ccccN { #1 } { #2 } { #3 } { #4 }
2235             \BNVS_end:

```



```

2236         \prg_return_true:
2237     } {
2238         \BNVS_end_return_false:
2239     }
2240 } {
2241     \BNVS_end_return_false:
2242 }
2243 } {
2244     \BNVS_end_return_false:
2245 }
2246 } {
2247     \BNVS_end_return_false:
2248 }
2249 } {
2250     \BNVS_end_return_false:
2251 }
2252 }

```

Local variables:

- \l\_\_bnvs\_a\_tl contains the name with a partial index path currently resolved.
- \l\_\_bnvs\_path\_head\_seq contains the index path components currently resolved.
- \l\_\_bnvs\_b\_tl contains the resolution.
- \l\_\_bnvs\_path\_tail\_seq contains the index path components to be resolved.

```

2253 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
2254     \__bnvs_seq_if_empty:cF { #2 } {
2255         \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
2256             \__bnvs_seq_use:cn { #1 } { \q__bnvs }
2257             \exp_not:n { \q__bnvs }
2258             \__bnvs_seq_use:cn { #2 } { \q__bnvs }
2259         }
2260         \__bnvs_seq_remove_all:cn { #1 } { }
2261     }
2262 }

```

## 6.17 Evaluation bricks

We start by helpers.

---

```

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

```

---

Replaces the variable content with its rounded floating point evaluation.

```

2263 \BNVS_new:cpn { round:N } #1 {
2264     \tl_if_empty:NTF #1 {
2265         \tl_set:Nn #1 { 0 }
2266     } {
2267         \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
2268     }
2269 }

```

```

2270 \BNVS_new:cpn { round:c } {
2271   \BNVS_t1_use:Nc \_bnvs_round:N
2272 }

```

---

```

\_bnvs_if_assign_value:nnnTF      \_bnvs_if_assign_value:nnnTF {<id>} {<tag>} <value> {<yes code>}
\_bnvs_if_assign_value:(nnv|vvv)TF {<no code>}

```

---

```

2273 \BNVS_new_conditional:cpnn { if_assign_value:nnn } #1 #2 #3 { T, F, TF } {
2274   \BNVS_begin:
2275   \_bnvs_if_resolve:ncTF { #3 } { a } {
2276     \_bnvs_gunset:nn { #1 } { #2 }
2277     \tl_map_inline:nn { V { V * } v } {
2278       \_bnvs_gset:nnv { ##1 } { #1 } { #2 } { a }
2279     }
2280     \BNVS_end:
2281     \prg_return_true:
2282   } {
2283     \BNVS_end:
2284     \prg_return_false:
2285   }
2286 }

2287 \BNVS_new_conditional:cpnn { if_assign_value:nnv } #1 #2 #3 { T, F, TF } {
2288   \BNVS_t1_use:nv {
2289     \_bnvs_if_assign_value:nnnTF { #1 } { #2 }
2290   } { #3 } {
2291     \prg_return_true:
2292   } {
2293     \prg_return_false:
2294   }
2295 }

2296 \BNVS_new_conditional:cpnn { if_assign_value:vvv } #1 #2 #3 { T, F, TF } {
2297   \BNVS_t1_use:nnv {
2298     \BNVS_t1_use:Nv \_bnvs_if_assign_value:nnnTF { #1 }
2299   } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
2300 }

```

---

```

\_bnvs_if_resolve_V:nncTF      \_bnvs_if_resolve_V:nncTF {<id>} {<tag>} <ans> {<yes code>} {<no code>}
\_bnvs_if_resolve_V:nvcTF      \_bnvs_if_append_V:nncTF {<id>} {<tag>} <ans> {<yes code>} {<no code>}
\_bnvs_if_append_V:nncTF
\_bnvs_if_append_V:(nxc|nvc)TF

```

---

Resolve the content of the  $\langle id \rangle$ ,  $\langle tag \rangle$  value counter into the  $\langle ans \rangle$  t1 variable or append this value to the right of this variable. Execute  $\langle yes\ code \rangle$  when there is a  $\langle value \rangle$ ,  $\{ \langle no\ code \rangle \}$  otherwise. 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.

```

2301 \BNVS_new:cpn { if_resolve_V:return:nnncT } #1 #2 #3 #4 #5 {
2302   \__bnvs_tl_if_empty:cTF { #4 } {
2303     \prg_return_false:
2304   } {
2305     \__bnvs_gset_cache:nnnv V { #2 } { #3 } { #4 }
2306     #5
2307     \prg_return_true:
2308   }
2309 }

2310 \makeatletter
2311 \BNVS_new_conditional:cpnn { if_resolve_V:nnc } #1 #2 #3 { T, F, TF } {
2312   \__bnvs_if_get_cache:nnncTF V { #1 } { #2 } { #3 } {
2313     \prg_return_true:
2314   } {
2315     \__bnvs_if_get:nnncTF V { #1 } { #2 } { #3 } {
2316       \__bnvs_quark_if_nil:cTF { #3 } {

```

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

```

2317   \__bnvs_gset:nnnn V { #1 } { #2 } { \q_no_value }
2318   \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2319     \__bnvs_if_resolve_V:return:nnncT A { #1 } { #2 } { #3 } {
2320       \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2321     }
2322   } {
2323     \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2324       \__bnvs_if_resolve_V:return:nnncT Z { #1 } { #2 } { #3 } {
2325         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2326       }
2327     } {
2328       \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2329       \prg_return_false:
2330     }
2331   }
2332 } {

```

Possible recursive call.

```

2333   \__bnvs_quark_if_no_value:cTF { #3 } {
2334     \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2335     \__bnvs_gset:nnnn V { #1 } { #2 } { 1 }
2336     \__bnvs_tl_set:cn { #3 } { 1 }
2337     \prg_return_true:
2338   } {
2339     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2340       \__bnvs_if_resolve_V:return:nnncT V { #1 } { #2 } { #3 } {
2341         \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2342       }
2343     } {
2344       \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
2345       \prg_return_false:
2346     }
2347   }
2348 }
2349 } {

```

```

2350 \tl_if_eq:nnTF { #2 } { pauses } {
2351   \cs_if_exist:NTF \c@beamerpauses {
2352     \exp_args:Nnx \__bnvs_tl_set:cn { #3 } { \the\c@beamerpauses }
2353     \__bnvs_gunset:nn { #1 } { #2 }
2354     \prg_return_true:
2355   } {
2356     \prg_return_false:
2357   }
2358 } {
2359   \tl_if_eq:nnTF { #2 } { slideinframe } {
2360     \cs_if_exist:NTF \beamer@slideinframe {
2361       \exp_args:Nnx \__bnvs_tl_set:cn { #3 } { \beamer@slideinframe }
2362       \__bnvs_gunset:nn { #1 } { #2 }
2363       \prg_return_true:
2364     } {
2365       \prg_return_false:
2366     }
2367   } {
2368     \prg_return_false:
2369   }
2370 }
2371 }
2372 }
2373 }
2374 \makeatother
2375 \BNVS_new_conditional_vvc:cn { if_resolve_V } { T, F, TF }

2376 \BNVS_new:cpn { end_put_right:vc } #1 #2 {
2377   \BNVS_tl_use:nv {
2378     \BNVS_end:
2379     \__bnvs_tl_put_right:cn { #2 }
2380   } { #1 }
2381 }

2382 \BNVS_new_conditional:cpnn { if_append_V:nnc } #1 #2 #3 { T, F, TF } {
2383   \BNVS_begin:
2384   \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
2385     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2386     \prg_return_true:
2387   } {
2388     \BNVS_end:
2389     \prg_return_false:
2390   }
2391 }
2392 \BNVS_new_conditional_vvc:cn { if_append_V } { T, F, TF }

```

---

```

\__bnvs_if_resolve_A:nncTF \__bnvs_if_resolve_A:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append_A:nncTF \__bnvs_if_append_A:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no code>}

```

---

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

```

2393 \BNVS_new_conditional:cpnn { if_resolve_A:nnc } #1 #2 #3 { T, F, TF } {
2394   \__bnvs_if_get_cache:nnncTF A { #1 } { #2 } { #3 } {
2395     \prg_return_true:
2396   } {
2397     \__bnvs_if_get:nnncTF A { #1 } { #2 } { #3 } {
2398       \__bnvs_quark_if_nil:cTF { #3 } {
2399         \__bnvs_gset:nnnn A { #1 } { #2 } { \q_no_value }

```

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

```

2400   \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2401     \__bnvs_tl_put_right:cn { #3 } { - }
2402     \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 } {
2403       \__bnvs_tl_put_right:cn { #3 } { + 1 }
2404       \__bnvs_round:c { #3 }
2405       \__bnvs_tl_if_empty:cTF { #3 } {
2406         \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2407         \prg_return_false:
2408       } {
2409         \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2410         \__bnvs_gset_cache:nnnv A { #1 } { #2 } { #3 }
2411         \prg_return_true:
2412       }
2413     } {
2414       \BNVS_error:n {
2415         Unavailable~length~for~#1~(\token_to_str:N\__bnvs_if_resolve_A:nncTF/2) }
2416       \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2417       \prg_return_false:
2418     }
2419   } {
2420     \BNVS_error:n {
2421       Unavailable~last~for~#1~(\token_to_str:N\__bnvs_if_resolve_A:nncTF/1) }
2422     \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
2423     \prg_return_false:
2424   }
2425 } {
2426   \__bnvs_quark_if_no_value:cTF { a } {
2427     \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2428     \__bnvs_gset:nnnn A { #1 } { #2 } { 1 }
2429     \__bnvs_tl_set:cn { #3 } { 1 }
2430     \prg_return_true:
2431   } {
2432     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2433       \__bnvs_gset:nnnv A { #1 } { #2 } { #3 }
2434       \prg_return_true:
2435     } {
2436       \prg_return_false:
2437     }
2438   }
2439 }
2440 } {
2441   \prg_return_false:
2442 }
2443 }

```

```

2444 }

2445 \BNVS_new_conditional:cpnn { if_append_A:nnc } #1 #2 #3 { T, F, TF } {
2446   \BNVS_begin:
2447   \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2448     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2449     \prg_return_true:
2450   } {
2451     \BNVS_end:
2452     \prg_return_false:
2453   }
2454 }

```

---

```

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

```

---

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

```

2455 \BNVS_new_conditional:cpnn { if_resolve_Z:nnc } #1 #2 #3 { T, F, TF } {
2456   \__bnvs_if_get_cache:nncTF Z { #1 } { #2 } { #3 } {
2457     \prg_return_true:
2458   } {
2459     \__bnvs_if_get:nncTF Z { #1 } { #2 } { #3 } {
2460       \__bnvs_quark_if_nil:cTF { #3 } {
2461         \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_no_value }

```

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

```

2462   \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2463     \__bnvs_tl_put_right:cn { #3 } { + }
2464     \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 } {
2465       \__bnvs_tl_put_right:cn { #3 } { - 1 }
2466       \__bnvs_round:c { #3 }
2467       \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
2468       \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
2469       \prg_return_true:
2470     } {
2471       \BNVS_error:x {
2472 Unavailable~last~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
2473       \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
2474       \prg_return_false:
2475     }
2476   } {
2477     \BNVS_error:x {
2478 Unavailable~first~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
2479     \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
2480     \prg_return_false:
2481   }
2482 } {
2483   \__bnvs_quark_if_no_value:cTF { #3 } {
2484     \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2485     \__bnvs_tl_set:cn { #3 } { 1 }
2486     \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
2487     \prg_return_true:

```

```

2488     } {
2489         \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2490             \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
2491             \prg_return_true:
2492         } {
2493             \prg_return_false:
2494         }
2495     }
2496 }
2497 } {
2498     \prg_return_false:
2499 }
2500 }
2501 }
2502 \BNVS_new_conditional_vvc:cn { if_resolve_Z } { T, F, TF }

2503 \BNVS_new_conditional:cpnn { if_append_Z:nnc } #1 #2 #3 { T, F, TF } {
2504     \BNVS_begin:
2505     \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2506         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2507         \prg_return_true:
2508     } {
2509         \BNVS_end:
2510         \prg_return_false:
2511     }
2512 }
2513 \BNVS_new_conditional_vvc:cn { if_append_Z } { T, F, TF }

```

---

<u>\__bnvs_if_resolve_L:nncTF</u> <u>\__bnvs_if_append_L:nncTF</u>	\__bnvs_if_resolve_L:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no code>} \__bnvs_if_append_L:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no code>}
---	---

---

Resolve the length of the <id>{<tag>} 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.

```

2514 \BNVS_new_conditional:cpnn { if_resolve_L:nnc } #1 #2 #3 { T, F, TF } {
2515     \__bnvs_if_get_cache:nnncTF L { #1 } { #2 } { #3 } {
2516         \prg_return_true:
2517     } {
2518         \__bnvs_if_get:nnncTF L { #1 } { #2 } { #3 } {
2519             \__bnvs_quark_if_nil:cTF { #3 } {
2520                 \__bnvs_gset:nnnn L { #1 } { #2 } { \q_no_value }

```

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

```

2521     \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2522         \__bnvs_tl_put_right:cn { #3 } { - }
2523         \__bnvs_if_append_A:nncTF { #1 } { #2 } { #3 } {
2524             \__bnvs_tl_put_right:cn { #3 } { + 1 }
2525             \__bnvs_round:c { #3 }
2526             \__bnvs_gset:nnnn L { #1 } { #2 } { \q_nil }
2527             \__bnvs_gset_cache:nnnv L { #1 } { #2 } { #3 }

```

```

2528         \prg_return_true:
2529     } {
2530         \BNVS_error:n {
2531 Unavailable~first~for~#1~(\__bnvs_if_resolve_L:nncTF/2) }
2532         \prg_return_false:
2533     }
2534 } {
2535     \BNVS_error:n {
2536 Unavailable~last~for~#1~(\__bnvs_if_resolve_L:nncTF/1) }
2537     \prg_return_false:
2538 }
2539 } {
2540     \__bnvs_quark_if_no_value:cTF { #3 } {
2541         \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2542         \__bnvs_gset_cache:nnnn L { #1 } { #2 } { 1 }
2543         \__bnvs_tl_set:cn { #3 } { 1 }
2544         \prg_return_true:
2545     } {
2546         \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2547             \__bnvs_gset_cache:nnnv L { #1 } { #2 } { #3 }
2548             \prg_return_true:
2549         } {
2550             \prg_return_false:
2551         }
2552     }
2553 }
2554 } {
2555     \prg_return_false:
2556 }
2557 }
2558 }
2559 \BNVS_new_conditional_vvc:cn { if_resolve_L } { T, F, TF }

2560 \BNVS_new_conditional_cpnn { if_append_L:nnc } #1 #2 #3 { T, F, TF } {
2561     \BNVS_begin:
2562     \__bnvs_if_resolve_L:nncTF { #1 } { #2 } { #3 } {
2563         \BNVS_end_tl_put_right:cv { #3 } { #3 }
2564         \prg_return_true:
2565     } {
2566         \BNVS_end:
2567         \prg_return_false:
2568     }
2569 }
2570 \BNVS_new_conditional_vvc:cn { if_append_L } { T, F, TF }

```

---

```

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

```

---

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



---

```

__bnvs_if_resolve_first:nncTF __bnvs_if_resolve_first:nncTF {<id>}{<tag>}{<ans>}{<yes code>}{<no
__bnvs_if_resolve_first:vvcTF code>}
__bnvs_if_append_first:nncTF __bnvs_if_append_first:nncTF {<id>}{<tag>}{<ans>}{<yes code>}{<no
__bnvs_if_append_first:vvcTF code>}

```

---

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

```

2571 \BNVS_new_conditional:cpnn { if_resolve_first:nnc } #1 #2 #3 { T, F, TF } {
2572   __bnvs_if_resolve_V:nncTF { #1 } { #2.first } { #3 }
2573   { \prg_return_true: }
2574   { __bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 }
2575     { \prg_return_true: }
2576     { __bnvs_if_resolve_v:nncTF { #1 } { #2.1 } { #3 }
2577       { \prg_return_true: } { \prg_return_false: }
2578     }
2579   }
2580 }
2581 \BNVS_new_conditional_vvc:cn { if_resolve_first } { T, F, TF }

2582 \BNVS_new_conditional:cpnn { if_append_first:nnc } #1 #2 #3 { T, F, TF } {
2583   __bnvs_if_append_index:nncTF { #1 } { #2 } { 1 } { #3 } { \prg_return_true: } {
2584     __bnvs_if_append_A:nncTF { #1 } { #2 } { #3 }
2585     { \prg_return_true: } { \prg_return_false: }
2586   }
2587 }
2588 \BNVS_new_conditional_vvc:cn { if_append_first } { T, F, TF }

```

---

```

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

```

---

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

```

2589 \BNVS_new_conditional:cpnn { if_resolve_last:nnc } #1 #2 #3 { T, F, TF } {
2590   __bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 }
2591   { \prg_return_true: } { \prg_return_false: }
2592 }
2593 \BNVS_new_conditional_vvc:cn { if_resolve_last } { T, F, TF }

2594 \BNVS_new_conditional:cpnn { if_append_last:nnc } #1 #2 #3 { T, F, TF } {
2595   __bnvs_if_append_Z:nncTF { #1 } { #2 } { #3 }
2596   { \prg_return_true: } { \prg_return_false: }
2597 }
2598 \BNVS_new_conditional_vvc:cn { if_append_last } { T, F, TF }

```

---

```

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

```

---

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

```

2599 \BNVS_new_conditional:cpnn { if_resolve_length:nnc } #1 #2 #3 { T, F, TF } {
2600   __bnvs_if_resolve_L:nncTF { #1 } { #2 } { #3 }
2601   { \prg_return_true: } { \prg_return_false: }
2602 }
2603 \BNVS_new_conditional_vvc:cn { if_resolve_length } { T, F, TF }

2604 \BNVS_new_conditional:cpnn { if_append_length:nnc } #1 #2 #3 { T, F, TF } {
2605   __bnvs_if_append_L:nncTF { #1 } { #2 } { #3 }
2606   { \prg_return_true: } { \prg_return_false: }
2607 }
2608 \BNVS_new_conditional_vvc:cn { if_append_length } { T, F, TF }

```

---

```

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

```

---

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

```

2609 \BNVS_new_conditional:cpnn { if_append_range:nnc } #1 #2 #3 { T, F, TF } {
2610   \BNVS_begin:
2611     __bnvs_if_resolve_A:nncTF { #1 } { #2 } { a } {
2612       \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
2613         __bnvs_tl_set:cn { a } { 0 }
2614       }
2615       __bnvs_if_resolve_Z:nncTF { #1 } { #2 } { b } {

```

Limited from above and below.

```

2616       \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2617         __bnvs_tl_set:cn { b } { 0 }
2618       }
2619       __bnvs_tl_put_right:cn { a } { - }
2620       __bnvs_tl_put_right:cv { a } { b }
2621       \BNVS_end_tl_put_right:cv { #3 } { a }
2622       \prg_return_true:
2623     } {

```

Limited from below.

```

2624       \BNVS_end_tl_put_right:cv { #3 } { a }
2625       __bnvs_tl_put_right:cn { #3 } { - }

```

```

2626     \prg_return_true:
2627   }
2628 } {
2629   \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { b } {

```

Limited from above.

```

2630     \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2631       \__bnvs_tl_set:cn { b } { 0 }
2632     }
2633     \__bnvs_tl_put_left:cn { b } { - }
2634     \BNVS_end_tl_put_right:cv { #3 } { b }
2635     \prg_return_true:
2636   } {
2637     \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { b } {
2638       \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
2639         \__bnvs_tl_set:cn { b } { 0 }
2640       }

```

Unlimited range.

```

2641     \BNVS_end_tl_put_right:cv { #3 } { b }
2642     \__bnvs_tl_put_right:cn { #3 } { - }
2643     \prg_return_true:
2644   } {
2645     \BNVS_end:
2646     \prg_return_false:
2647   }
2648 }
2649 }
2650 }
2651 \BNVS_new_conditional_vvc:cn { if_append_range } { T, F, TF }

2652 \BNVS_new_conditional:cpnn { if_resolve_range:nnc } #1 #2 #3 { T, F, TF } {
2653   \__bnvs_tl_clear:c { #3 }
2654   \__bnvs_if_append_range:ncTF { #1 } { #2 } { #3 } {
2655     \prg_return_true:
2656   } {
2657     \prg_return_false:
2658   }
2659 }
2660 \BNVS_new_conditional_vvc:cn { if_resolve_range } { T, F, TF }

```

---

```

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

```

---

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

```

2661 \BNVS_new_conditional:cpnn { if_resolve_previous:nnc } #1 #2 #3 { T, F, TF } {
2662   \__bnvs_if_get_cache:nnncTF P { #1 } { #2 } { #3 } {

```

```

2663 \prg_return_true:
2664 } {
2665 \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2666 \__bnvs_t1_put_right:cn { #3 } { -1 }
2667 \__bnvs_round:c { #3 }
2668 \__bnvs_gset_cache:nnv P { #1 } { #2 } { #3 }
2669 \prg_return_true:
2670 } {
2671 \prg_return_false:
2672 }
2673 }
2674 }
2675 \BNVS_new_conditional_vvc:cn { if_resolve_previous } { T, F, TF }

2676 \BNVS_new_conditional:cpnn { if_append_previous:nnc } #1 #2 #3 { T, F, TF } {
2677 \BNVS_begin:
2678 \__bnvs_if_resolve_previous:nncTF { #1 } { #2 } { #3 } {
2679 \BNVS_end_t1_put_right:cv { #3 } { #3 }
2680 \prg_return_true:
2681 } {
2682 \BNVS_end:
2683 \prg_return_false:
2684 }
2685 }
2686 \BNVS_new_conditional_vvc:cn { if_append_previous } { T, F, TF }

```

---

```

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

```

---

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

```

2687 \BNVS_new_conditional:cpnn { if_resolve_next:nnc } #1 #2 #3 { T, F, TF } {
2688 \__bnvs_if_get_cache:nnncTF N { #1 } { #2 } { #3 } {
2689 \prg_return_true:
2690 } {
2691 \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2692 \__bnvs_t1_put_right:cn { #3 } { +1 }
2693 \__bnvs_round:c { #3 }
2694 \__bnvs_gset_cache:nnv N { #1 } { #2 } { #3 }
2695 \prg_return_true:
2696 } {
2697 \prg_return_false:
2698 }
2699 }
2700 }
2701 \BNVS_new_conditional_vvc:cn { if_resolve_next } { T, F, TF }

2702 \BNVS_new_conditional:cpnn { if_append_next:nnc } #1 #2 #3 { T, F, TF } {
2703 \BNVS_begin:

```

```

2704 \__bnvs_if_resolve_next:nncTF { #1 } { #2 } { #3 } {
2705   \BNVS_end_t1_put_right:cv { #3 } { #3 }
2706   \prg_return_true:
2707 } {
2708   \BNVS_end:
2709   \prg_return_true:
2710 }
2711 }
2712 \BNVS_new_conditional_vvc:cn { if_append_next } { T, F, TF }

```

---

<pre> \__bnvs_if_resolve_v:nncTF \__bnvs_if_resolve_v:vvcTF \__bnvs_if_append_v:nnc </pre>	<pre> \__bnvs_if_resolve_v:nncTF {&lt;id&gt;} {&lt;tag&gt;} {&lt;ans&gt; {&lt;yes code&gt;} {&lt;no code&gt;} \__bnvs_if_append_v:vvcTF \__bnvs_if_append_v:nncTF {&lt;id&gt;} {&lt;tag&gt;} {&lt;ans&gt; {&lt;yes code&gt;} {&lt;no code&gt;} </pre>
--	---

---

Resolve the value of the  $\langle id \rangle! \langle tag \rangle$  overlay set into the  $\langle ans \rangle$  t1 variable or append this value to the right of this variable. Execute  $\langle yes \text{ code} \rangle$  when there is a  $\langle value \rangle$ ,  $\langle no \text{ code} \rangle$  otherwise. In the latter case, the content of the  $\langle ans \rangle$  t1 variable is undefined, on resolution only. Calls  $\backslash\_bnvs\_if\_resolve\_V:nncTF$ .

```

2713 \BNVS_new_conditional:cpnn { if_resolve_v:nnc } #1 #2 #3 { T, F, TF } {
2714   \__bnvs_if_get:nnncTF v { #1 } { #2 } { #3 } {
2715     \__bnvs_quark_if_no_value:cTF { #3 } {
2716       \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
2717       \__bnvs_gset:nnnn V { #1 } { #2 } { 1 }
2718       \__bnvs_t1_set:cn { #3 } { 1 }
2719       \prg_return_true:
2720     } {
2721       \prg_return_true:
2722     }
2723   } {
2724     \__bnvs_gset:nnnn v { #1 } { #2 } { \q_no_value }
2725     \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
2726       \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
2727       \prg_return_true:
2728     } {
2729       \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
2730         \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
2731         \prg_return_true:
2732       } {
2733         \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
2734           \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
2735           \prg_return_true:
2736         } {
2737           \__bnvs_gunset:nnn v { #1 } { #2 }
2738           \prg_return_false:
2739         }
2740       }
2741     }
2742   }
2743 }
2744 \BNVS_new_conditional_vvc:cn { if_resolve_v } { T, F, TF }
2745 \BNVS_new_conditional:cpnn { if_append_v:nnc } #1 #2 #3 { T, F, TF } {

```

```

2746 \BNVS_begin:
2747 \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #3 } {
2748   \BNVS_end_tl_put_right:cv { #3 } { #3 }
2749   \prg_return_true:
2750 } {
2751   \BNVS_end:
2752   \prg_return_false:
2753 }
2754 }

2755 \BNVS_new_conditional_vvc:cn { if_append_v } { T, F, TF }

```

---

```

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

```

---

Resolve the index associated to the  $\langle id \rangle$ !  $\langle tag \rangle$  set 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,  $\langle yes \text{ code} \rangle$  is executed, otherwise  $\{ \langle no \text{ code} \rangle \}$  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.

```

2756 \BNVS_new_conditional:cpnn { index_can:nn } #1 #2 { T, F, TF } {
2757   \__bnvs_is_gset:nnnTF V { #1 } { #2 } {
2758     \prg_return_true:
2759   } {
2760     \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
2761       \prg_return_true:
2762     } {
2763       \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {
2764         \prg_return_true:
2765       } {
2766         \prg_return_false:
2767       }
2768     }
2769   }
2770 }

2771 \BNVS_new_conditional:cpnn { index_can:vv } #1 #2 { T, F, TF } {
2772   \BNVS_tl_use:nv {
2773     \BNVS_tl_use:Nv \__bnvs_index_can:nTF { #1 }
2774   } { #2 } { \prg_return_true: } { \prg_return_false: }
2775 }

2776 \BNVS_new_conditional:cpnn { if_resolve_index:nnnc } #1 #2 #3 #4 { T, F, TF } {
2777   \exp_args:Ne \__bnvs_if_resolve_V:nncTF { #1 } { #2.#3 } { #4 } {

```

```

2778     \prg_return_true:
2779 } {
2780     \__bnvs_if_resolve_first:nncTF { #1 } { #2 } { #4 } {
2781         \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
2782         \__bnvs_round:c { #4 }
2783     \prg_return_true:

```

Limited overlay set.

```

2784     } {
2785         \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #4 } {
2786             \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
2787             \__bnvs_round:c { #4 }
2788             \prg_return_true:
2789         } {
2790             \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
2791                 \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
2792                 \__bnvs_round:c { #4 }
2793                 \prg_return_true:
2794             } {
2795                 \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
2796                     \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
2797                     \__bnvs_round:c { #4 }
2798                     \prg_return_true:
2799                 } {
2800                     \prg_return_false:
2801                 }
2802             }
2803         }
2804     }
2805 }
2806 }

2807 \BNVS_new_conditional:cpnn { if_resolve_index:nnvc } #1 #2 #3 #4 { T, F, TF } {
2808     \BNVS_tl_use:nv {
2809         \__bnvs_if_resolve_index:nnncTF { #1 } { #2 }
2810     } { #3 } { #4 } {
2811         \prg_return_true:
2812     } {
2813         \prg_return_false:
2814     }
2815 }

2816 \BNVS_new_conditional:cpnn { if_resolve_index:vvvc } #1 #2 #3 #4 { T, F, TF } {
2817     \BNVS_tl_use:nvv {
2818         \BNVS_tl_use:Nv \__bnvs_if_resolve_index:nnncTF { #1 }
2819     } { #2 } { #3 } { #4 } {
2820         \prg_return_true:
2821     } {
2822         \prg_return_false:
2823     }
2824 }

2825 \BNVS_new_conditional:cpnn { if_append_index:nnnc } #1 #2 #3 #4 { T, F, TF } {
2826     \BNVS_begin:

```

```

2827 \__bnvs_if_resolve_index:nnncTF { #1 } { #2 } { #3 } { #4 } {
2828   \BNVS_end_tl_put_right:cv { #4 } { #4 }
2829   \prg_return_true:
2830 } {
2831   \BNVS_end:
2832   \prg_return_false:
2833 }
2834 }

2835 \BNVS_new_conditional:cpnn { if_append_index:vvvc } #1 #2 #3 #4 { T, F, TF } {
2836   \BNVS_tl_use:nvv {
2837     \BNVS_tl_use:Nv \__bnvs_if_append_index:nnncTF { #1 }
2838   } { #2 } { #3 } { #4 } {
2839     \prg_return_true:
2840   } {
2841     \prg_return_false:
2842   }
2843 }

```

## 6.18 Index counter

---

\\_\_bnvs\_n\_assign:nnn \\_\_bnvs\_n\_assign:nnn {<id> } {<tag> } {<value> }

\\_\_bnvs\_n\_assign:vvv Assigns the resolved <value> to n counter <id>{<tag>}. Execute <yes code> when resolution succeeds, <no code> otherwise.

```

2844 \BNVS_new:cpn { n_assign:nnn } #1 #2 #3 {
2845   \__bnvs_if_get:nnncF V { #1 } { #2 } { a } {
2846     \BNVS_warning:n { Unknwown~ #1!#2,~defaults-to~0 }
2847     \__bnvs_gset:nnnn V { #1 } { #2 } { 0 }
2848   }
2849   \__bnvs_if_resolve:ncTF { #3 } { a } {
2850     \__bnvs_gset:nnnv v { #1 } { #2 } { a }
2851   } {
2852     \BNVS_error:n { NO~resolution~of~#3,~defaults-to~0 }
2853     \__bnvs_gset:nnnn v { #1 } { #2 } { 0 }
2854   }
2855 }

2856 \BNVS_new:cpn { n_assign:vvv } #1 {
2857   \BNVS_tl_use:nvv {
2858     \BNVS_tl_use:cv { n_assign:nn } { #1 }
2859   }
2860 }

```

---

\\_\_bnvs\_if\_resolve\_n:ncTF \\_\_bnvs\_if\_resolve\_n:nnncTF {<id> } {<tag> } {<ans> } {<yes code> } {<no code> }

\\_\_bnvs\_if\_append\_n:ncTF Evaluate the n counter associated to the <id>{<tag>} 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.

```

2861 \BNVS_new_conditional:cpnn { if_resolve_n:nnnc } #1 #2 #3 { T, F, TF } {
2862   \__bnvs_if_get:nnncTF n { #1 } { #2 } { #3 } {
2863     \__bnvs_if_resolve:vcTF { #3 } { #3 } {

```



```

2864     \prg_return_true:
2865   } {
2866     \prg_return_false:
2867   }
2868 } {
2869   \__bnvs_tl_set:cn { #3 } { 1 }
2870   \__bnvs_gset:nnnn n { #1 } { #2 } { 1 }
2871   \prg_return_true:
2872 }
2873 }
2874 \BNVS_new_conditional_vvc:cn { if_resolve_n } { T, F, TF }

2875 \BNVS_new_conditional:cpnn { if_append_n:nnc } #1 #2 #3 { T, F, TF } {
2876   \BNVS_begin:
2877   \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #3 } {
2878     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2879     \prg_return_true:
2880   } {
2881     \BNVS_end:
2882     \prg_return_false:
2883   }
2884 }

2885 \BNVS_new_conditional_vvc:cn { if_append_n } { T, F, TF }

```

---

```

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

```

---

Resolve the index for the value of the n counter associated to the {<tag>} 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.

```

2886 \BNVS_new_conditional:cpnn { if_resolve_n_index:nnc } #1 #2 #3 { T, F, TF } {
2887   \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #3 } {
2888     \__bnvs_tl_put_left:cn { #3 } { #1!#2. }
2889     \__bnvs_if_resolve:vcTF { #3 } { #3 } {
2890       \prg_return_true:
2891     } {
2892       \prg_return_false:
2893     }
2894   } {
2895     \prg_return_false:
2896   }
2897 }

2898 \BNVS_new_conditional:cpnn { if_append_n_index:nnc } #1 #2 #3 { T, F, TF } {
2899   \BNVS_begin:
2900   \__bnvs_if_resolve_n_index:nncTF { #1 } { #2 } { #3 } {
2901     \BNVS_end_tl_put_right:cv { #3 } { #3 }

```

```

2902     \prg_return_true:
2903   } {
2904     \BNVS_end:
2905     \prg_return_false:
2906   }
2907 }
2908 \BNVS_new_conditional_vvc:cn { if_append_n_index } { T, F, TF }

```

## 6.19 Value counter

---

```

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

```

---

Increment the value counter position accordingly. Put the result in the  $\langle ans \rangle$  tl variable.

```

2909 \BNVS_new_conditional:cpnn { if_resolve_v_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
2910   \__bnvs_if_resolve:ncTF { #3 } { #4 } {
2911     \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
2912       \__bnvs_if_resolve_v:nnncTF { #1 } { #2 } { #4 } {
2913         \prg_return_true:
2914       } {
2915         \prg_return_false:
2916       }
2917     } {
2918       \__bnvs_tl_put_right:cn { #4 } { + }
2919       \__bnvs_if_append_v:nnncTF { #1 } { #2 } { #4 } {
2920         \__bnvs_round:c { #4 }
2921         \__bnvs_gset:nnnv v { #1 } { #2 } { #4 }
2922         \prg_return_true:
2923       } {
2924         \prg_return_false:
2925       }
2926     }
2927   } {
2928     \prg_return_false:
2929   }
2930 }

2931 \BNVS_new_conditional:cpnn { if_append_v_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
2932   \BNVS_begin:
2933   \__bnvs_if_resolve_v_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
2934     \BNVS_end_tl_put_right:cv { #4 } { #4 }
2935     \prg_return_true:
2936   } {
2937     \BNVS_end:
2938     \prg_return_false:
2939   }
2940 }
2941 \BNVS_new_conditional_vvnc:cn { if_append_v_incr } { T, F, TF }

2942 \BNVS_new_conditional:cpnn { if_resolve_v_post:nnnc } #1 #2 #3 #4 { T, F, TF } {

```

```

2943 \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
2944 \BNVS_begin:
2945 \__bnvs_if_resolve:ncTF { #3 } { a } {
2946 \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
2947 \BNVS_end:
2948 \prg_return_true:
2949 } {
2950 \__bnvs_tl_put_right:cn { a } { + }
2951 \__bnvs_tl_put_right:cv { a } { #4 }
2952 \__bnvs_round:c { a }
2953 \BNVS_end_gset:nnnv v { #1 } { #2 } { a }
2954 \prg_return_true:
2955 }
2956 } {
2957 \BNVS_end:
2958 \prg_return_false:
2959 }
2960 } {
2961 \prg_return_false:
2962 }
2963 }
2964 \BNVS_new_conditional_vvvc:cn { if_resolve_v_post } { T, F, TF }

2965 \BNVS_new_conditional:cpnn { if_append_v_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
2966 \BNVS_begin:
2967 \__bnvs_if_resolve_v_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
2968 \BNVS_end_tl_put_right:cv { #4 } { #4 }
2969 \prg_return_true:
2970 } {
2971 \prg_return_false:
2972 }
2973 }
2974 \BNVS_new_conditional_vvnc:cn { if_append_v_post } { T, F, TF }
2975 \BNVS_new_conditional_vvvc:cn { if_append_v_post } { T, F, TF }

```

---

```

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

```

---

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

```

2976 \BNVS_new_conditional:cpnn { if_resolve_n_incr:nnnc } #1 #2 #3 #4 { T, TF } {
Resolve the <offset> into the <ans> variable.

```

```

2977 \__bnvs_if_resolve:ncTF { #3 } { #4 } {
2978 \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {

```

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

```

2979 \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #4 } {
2980 \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {

```

```

2981         \prg_return_true:
2982     } {
2983         \prg_return_false:
2984     }
2985 } {
2986     \prg_return_false:
2987 }
2988 } {
The <offset> does not resolve to 0.
2989     \__bnvs_t1_put_right:cn { #4 } { + }
2990     \__bnvs_if_append_n:nncTF { #1 } { #2 } { #4 } {
2991         \__bnvs_round:c { #4 }
2992         \__bnvs_gset:nnnv n { #1 } { #2 } { #4 }
2993         \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
2994             \prg_return_true:
2995         } {
2996             \prg_return_false:
2997         }
2998     } {
2999         \prg_return_false:
3000     }
3001 }
3002 } {
3003     \prg_return_false:
3004 }
3005 }

3006 \BNVS_new_conditional:cpnn
3007 { if_append_n_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
3008     \BNVS_begin:
3009     \__bnvs_if_resolve_n_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
3010         \BNVS_end_t1_put_right:cv { #4 } { #4 }
3011         \prg_return_true:
3012     } {
3013         \BNVS_end:
3014         \prg_return_false:
3015     }
3016 }
3017 \BNVS_new_conditional_vvnc:cn { if_append_n_incr } { T, F, TF }

```

---

```

\__bnvs_if_resolve_v_post:nnncTF \__bnvs_if_resolve_v_post:nnncTF {<id>} {<tag>} {<offset>} <ans> {<yes
\__bnvs_if_append_v_post:nnncTF code>} {<no code>}
\__bnvs_if_append_v_post:vvncTF \__bnvs_if_append_v_post:nnncTF {<id>} {<tag>} {<offset>} <ans> {<yes
code>} {<no code>}

```

---

Resolve the value of the free counter for the given <tag> 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.

```

3018 \BNVS_new_conditional:cpnn { if_resolve_n_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
3019     \__bnvs_if_resolve_n:nnncTF { #1 } { #2 } { #4 } {
3020         \BNVS_begin:

```

```

3021 \__bnvs_if_resolve:ncTF { #3 } { #4 } {
3022 \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
3023 \BNVS_end:
3024 \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
3025 \prg_return_true:
3026 } {
3027 \prg_return_false:
3028 }
3029 } {
3030 \__bnvs_tl_put_right:cn { #4 } { + }
3031 \__bnvs_if_append_n:nncTF { #1 } { #2 } { #4 } {
3032
3033 \__bnvs_round:c { #4 }
3034 \__bnvs_gset:nnnv n { #1 } { #2 } { #4 }
3035 \BNVS_end:
3036 \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
3037 \prg_return_true:
3038 } {
3039 \prg_return_false:
3040 }
3041 } {
3042 \BNVS_end:
3043 \prg_return_false:
3044 }
3045 }
3046 } {
3047 \BNVS_end:
3048 \prg_return_false:
3049 }
3050 } {
3051 \prg_return_false:
3052 }
3053 }

3054 \BNVS_new_conditional:cpnn { if_append_n_post:nnnc } #1 #2 #3 #4{ T, F, TF } {
3055 \BNVS_begin:
3056 \__bnvs_if_resolve_n_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
3057 \BNVS_end_tl_put_right:cv { #4 } { #4 }
3058 \prg_return_true:
3059 } {
3060 \BNVS_end:
3061 \prg_return_false:
3062 }
3063 }
3064 \BNVS_new_conditional_vvnc:cn { if_append_n_post } { T, F, TF }

```

## 6.20 Functions for the resolution

They manily start with \\_\_bnvs\_if\_resolve\_ or \\_\_bnvs\_split\_

---

```

\\_bnvs_split_pop_iksp:TFF          \\_bnvs_split_pop_iksp:TFF {<black code>} {<blank code>}
\\_bnvs_split_end_return_or_pop_complete:T {<end code>}
\\_bnvs_split_end_return_or_pop_void:T          \\_bnvs_split_end_return_or_pop_complete:T {<blank code>}
\\_bnvs_split_end_return_or_pop_void:T          \\_bnvs_split_end_return_or_pop_void:T {<black code>}

```

---

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

For \\\_bnvs\_split\_end\_return\_or\_pop\_complete:T: pops the four heading items of the split sequence into the four variables n\_incr, plus, rhs, post. Then execute <black code>.

For \\\_bnvs\_split\_end\_return\_or\_pop\_void:T: pops the eight heading items of the split sequence then execute <blank code>.

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

```

3065 \\BNVS_new:cpn { split_pop_iksp:TFF } #1 #2 #3 {
3066   \\_bnvs_split_if_pop_left:cTF { id } {
3067     \\_bnvs_split_if_pop_left:cTF { kri } {
3068       \\_bnvs_split_if_pop_left:cTF { short } {
3069         \\_bnvs_split_if_pop_left:cTF { path } {
3070           \\_bnvs_tl_if_blank:vTF { short } {

```

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

```

3071         #2
3072       } {
3073         \\BNVS_tl_use:nv {
3074           \\regex_match:NnT \\c\\_bnvs_A_reserved_Z_regex
3075         } { short } {
3076           \\_bnvs_tl_if_eq:cnF { short } { pauses } {
3077             \\_bnvs_tl_if_eq:cnF { short } { slideinframe } {
3078       \\BNVS_error:x { Use-of-reserved-``\\BNVS_tl_use:c { tag }'' }
3079         }
3080       }
3081     }
3082     \\_bnvs_tl_if_blank:vTF { kri } {
3083       \\_bnvs_tl_set:cv { id } { id_last }
3084     } {
3085       \\_bnvs_tl_set:cv { id_last } { id }
3086     }

```

Build the path sequence and lowercase components conditionals.

```

3087       \\_bnvs_seq_set_split:cnv { path } { . } { path }
3088       #1
3089     }
3090   } {
3091     \\BNVS_fatal:n { split_pop_iksp:TFF/path }
3092   }
3093 } {
3094   \\BNVS_fatal:n { split_pop_iksp:TFF/short }
3095 }
3096 } {

```

```

3097     \BNVS_fatal:n { split_pop_iksp:TFF/kri }
3098   }
3099 } {
3100   #3
3101 }
3102 }

```

conditional variants.

```

3103 \BNVS_new:cpn { split_end_return_or_pop_complete:T } #1 {
3104   \cs_set:Npn \BNVS_split_F:n ##1 {
3105     \BNVS_end_unreachable_return_false:n {
3106       split_end_return_or_pop_complete: ##1
3107     }
3108   }
3109   \__bnvs_split_if_pop_left_or:cT { n_incr } {
3110     \__bnvs_split_if_pop_left_or:cT { plus } {
3111       \__bnvs_split_if_pop_left_or:cT { rhs } {
3112         \__bnvs_split_if_pop_left_or:cT { post } {
3113           #1
3114         }
3115       }
3116     }
3117   }
3118 }

3119 \BNVS_new:cpn { split_end_return_or_pop_void:T } #1 {
3120   \cs_set:Npn \BNVS_split_F:n ##1 {
3121     \BNVS_end_unreachable_return_false:n {
3122       split_end_return_or_pop_void: ##1
3123     }
3124   }
3125   \__bnvs_split_if_pop_left:cTn { a } {
3126     \__bnvs_split_if_pop_left:cTn { a } {
3127       \__bnvs_split_if_pop_left:cTn { a } {
3128         \__bnvs_split_if_pop_left:cTn { a } {
3129           \__bnvs_split_if_pop_left:cTn { a } {
3130             \__bnvs_split_if_pop_left:cTn { a } {
3131               \__bnvs_split_if_pop_left:cTn { a } {
3132                 \__bnvs_split_if_pop_left:cTn { a } {
3133                   #1
3134                 } { T/8 }
3135               } { T/7 }
3136             } { T/6 }
3137           } { T/5 }
3138         } { T/4 }
3139       } { T/3 }
3140     } { T/2 }
3141   } { T/1 }
3142 }

```

---

```

\__bnvs_if_resolve:ncTF \__bnvs_if_resolve:ncTF {<expression>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_resolve:vcTF \__bnvs_if_append:ncTF {<expression>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append:ncTF
\__bnvs_if_append:vcTF

```

Resolves the  $\langle expression \rangle$ , replacing all the named overlay specifications by their static counterpart then put the rounded result in  $\langle ans \rangle$  tl variable when resolving or to the right of this variable when appending.

Implementation details. Executed within a group. Heavily used by  $\backslash\ldots\_if\_resolve\_query:ncTF$ , where  $\langle expression \rangle$  was initially enclosed inside ‘ $?(...)$ ’. Local variables:

$\backslash l\_bnvs\_ans\_tl$  To feed  $\langle tl\ variable \rangle$  with.

(End of definition for  $\backslash l\_bnvs\_ans\_tl$ .)

$\backslash l\_bnvs\_split\_seq$  The sequence of caught query groups and non queries.

(End of definition for  $\backslash l\_bnvs\_split\_seq$ .)

$\backslash l\_bnvs\_split\_int$  Is the index of the non queries, before all the caught groups.

(End of definition for  $\backslash l\_bnvs\_split\_int$ .)

```

3143 \BNVS_int_new:c { split }

```

$\backslash l\_bnvs\_tag\_tl$  Storage for split sequence items that represent names.

(End of definition for  $\backslash l\_bnvs\_tag\_tl$ .)

$\backslash l\_bnvs\_path\_tl$  Storage for split sequence items that represent integer paths.

(End of definition for  $\backslash l\_bnvs\_path\_tl$ .)

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  $\backslash\ldots end\_return\ldots$  functions.

```

3144 \BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
3145   \BNVS_begin:
3146   \__bnvs_if_resolve:ncTF { #1 } { #2 } {
3147     \BNVS_end_tl_put_right:cv { #2 } { #2 }
3148     \prg_return_true:
3149   } {
3150     \BNVS_end:
3151     \prg_return_false:
3152   }
3153 }
3154 \BNVS_new_conditional_vc:cn { if_append } { T, F, TF }

```

Heavily used.

```

3155 \cs_new:Npn \BNVS_end_unreachable_return_false:n #1 {
3156   \BNVS_error:n { UNREACHABLE/#1 }
3157   \BNVS_end:
3158   \prg_return_false:
3159 }
3160 \cs_new:Npn \BNVS_end_unreachable_return_false:x #1 {
3161   \BNVS_error:x { UNREACHABLE/#1 }
3162   \BNVS_end:

```



```

3163 \prg_return_false:
3164 }

```

```

3165 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { TF } {
3166   \__bnvs_if_call:TF {
3167     \BNVS_begin:

```

This T<sub>E</sub>X group will be closed just before returning. Implementation:

```

3168   \__bnvs_if_regex_split:cnTF { split } { #1 } {

```

The leftmost item is not a special item: we start feeding \l\_\_bnvs\_ans\_tl with it.

```

3169     \BNVS_set:cpn { if_resolve_end_return_true: } {

```

Normal and unique end of the loop.

```

3170       \__bnvs_if_resolve_round_ans:
3171       \BNVS_end_tl_set:cv { #2 } { ans }
3172       \prg_return_true:
3173     }

```

Ranges are not rounded: for them \...if\_resolve\_round\_ans: is a noop.

```

3174     \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round:c { ans } }
3175     \__bnvs_tl_clear:c { ans }
3176     \__bnvs_split_loop_or_end_return:
3177   } {

```

There is not reference.

```

3178     \__bnvs_tl_set:cn { ans } { #1 }
3179     \__bnvs_round:c { ans }
3180     \BNVS_end_tl_set:cv { #2 } { ans }
3181     \prg_return_true:
3182   }
3183 } {
3184   \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
3185   \BNVS_end:
3186   \prg_return_false:
3187 }
3188 }
3189 \BNVS_new_conditional_vc:cn { if_resolve } { T, F, TF }

```

```

3190 \BNVS_new:cpn { build_tag: } {
3191   \__bnvs_tl_set_eq:cc { tag } { short }
3192   \__bnvs_seq_map_inline:cn { path } {
3193     \__bnvs_tl_put_right:cn { tag } { . ##1 }
3194   }
3195 }
3196 \BNVS_new:cpn { build_tag_head: } {
3197   \__bnvs_tl_set_eq:cc { tag } { short }
3198   \__bnvs_seq_map_inline:cn { path_head } {
3199     \__bnvs_tl_put_right:cn { tag } { . ##1 }
3200   }
3201 }

```

---

\\_\_bnvs\_split\_loop\_or\_end\_return: \\_\_bnvs\_split\_loop\_or\_end\_return:

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

```
3202 \clist_map_inline:nn {
3203   n, reset, reset_all, v, first, last, length,
3204   previous, next, range, assign, only
3205 } {
3206   \bool_new:c { l__bnvs_#1_bool }
3207 }

3208 \BNVS_new_conditional:cpnn { if:c } #1 { p, T, F, TF } {
3209   \bool_if:cTF { l__bnvs_#1_bool } {
3210     \prg_return_true:
3211   } {
3212     \prg_return_false:
3213   }
3214 }

3215 \BNVS_new_conditional:cpnn { bool_if_exist:c } #1 { p, T, F, TF } {
3216   \bool_if_exist:cTF { l__bnvs_#1_bool } {
3217     \prg_return_true:
3218   } {
3219     \prg_return_false:
3220   }
3221 }

3222 \BNVS_new:cpn { prepare_context:N } #1 {
3223   \clist_map_inline:nn {
3224     n, v, reset, reset_all, first, last, length,
3225     previous, next, range, assign, only
3226   } {
3227     \__bnvs_set_false:c { ##1 }
3228   }
3229   \__bnvs_seq_clear:c { path_head }
3230   \__bnvs_seq_clear:c { path_tail }
3231   \__bnvs_tl_clear:c { index }
3232   \__bnvs_tl_clear:c { suffix }
3233   \BNVS_set:cpn { :n } ##1 {
3234     \tl_if_blank:nF { ##1 } {
3235       \__bnvs_tl_if_empty:cF { index } {
3236         \__bnvs_seq_put_right:cv { path_head } { index }
3237         \__bnvs_tl_clear:c { index }
3238       }
3239       \__bnvs_seq_put_right:cn { path_head } { ##1 }
3240     }
3241   }
3242   \__bnvs_seq_map_inline:cn { path } {
3243     \__bnvs_bool_if_exist:cTF { ##1 } {
3244       \__bnvs_set_true:c { ##1 }
3245     }
3246     \clist_if_in:nnF { n, v, reset, reset_all } { ##1 } {
```

```

3246     \bool_if:NT #1 {
3247         \BNVS_error:n {Unexpected~##1~in~assignment }
3248     }
3249     \__bnvs_tl_set:cn { suffix } { ##1 }
3250 }
3251 \BNVS_set:cpn { :n } #####1 {
3252     \tl_if_blank:nF { #####1 } {
3253         \BNVS_error:n {Unexpected~#####1 }
3254     }
3255 }
3256 } {
3257     \regex_match:NnTF \c__bnvs_A_index_Z_regex { ##1 } {
3258         \__bnvs_tl_if_empty:cF { index } {
3259             \__bnvs_seq_put_right:cv { path_head } { index }
3260         }
3261         \__bnvs_tl_set:cn { index } { ##1 }
3262     } {
3263         \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { ##1 } {
3264             \BNVS_error:n { Unsupported~##1 }
3265         } {
3266             \__bnvs_:n { ##1 }
3267         }
3268     }
3269 }
3270 }
3271 \__bnvs_seq_set_eq:cc { path } { path_head }
3272 }

3273 \BNVS_new:cpn { split_loop_or_end_return: } {
3274     \__bnvs_split_if_pop_left:cTF { a } {
3275         \__bnvs_tl_put_right:cv { ans } { a }
3276         \__bnvs_split_pop_iksp:TFF {
3277             \__bnvs_split_end_return_or_pop_void:T {
3278                 \__bnvs_prepare_context:N \c_true_bool
3279                 \__bnvs_build_tag:
3280                 \__bnvs_split_loop_or_end_return_iadd:n { 1 }
3281             }
3282         } {
3283             \__bnvs_split_pop_iksp:TFF {
3284                 \__bnvs_split_end_return_or_pop_complete:T {
3285                     \__bnvs_tl_if_blank:vTF { n_incr } {
3286                         \__bnvs_tl_if_blank:vTF { plus } {
3287                             \__bnvs_tl_if_blank:vTF { rhs } {
3288                                 \__bnvs_tl_if_blank:vTF { post } {
3289                                     \__bnvs_prepare_context:N \c_false_bool
3290                                     \__bnvs_build_tag:

```

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

```

3291         \__bnvs_tl_if_empty:cTF { index } {
3292             \__bnvs_tl_if_empty:cTF { suffix } {
3293                 \__bnvs_split_loop_or_end_return_v:
3294             } {
3295                 \__bnvs_split_loop_or_end_return_suffix:
3296             }
3297         } {

```

```

3298         \__bnvs_split_loop_or_end_return_index:
3299     }
3300     } {
3301         \__bnvs_prepare_context:N \c_true_bool
3302         \__bnvs_build_tag:
3303         \BNVS_use:c { split_loop_or_end_return[...++]: }
3304     }
3305     } {
3306         \__bnvs_prepare_context:N \c_true_bool
3307         \__bnvs_build_tag:
3308         \__bnvs_split_loop_or_end_return_assign:
3309     }
3310     } {
3311         \__bnvs_if_resolve:vcTF { rhs } { rhs } {
3312             \__bnvs_prepare_context:N \c_true_bool
3313             \__bnvs_build_tag:
3314             \BNVS_tl_use:Nv
3315             \__bnvs_split_loop_or_end_return_iadd:n { rhs }
3316         } {
3317             \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs }}
3318             \__bnvs_split_loop_or_end_return:
3319         }
3320     }
3321     } {
3322         \__bnvs_prepare_context:N \c_true_bool
3323         \__bnvs_build_tag:
3324         \__bnvs_set_true:c { n }
3325         \__bnvs_split_loop_or_end_return_iadd:n { 1 }
3326     }
3327 }
3328 } {
3329 \BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/3 }
3330 } {
3331 \BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/2 }
3332 }
3333 } {

```

The split sequence is empty.

```

3334     \__bnvs_if_resolve_end_return_true:
3335 }
3336 } {
3337 \BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/1 }
3338 }
3339 }

3340 \BNVS_new_conditional:cpnn { if_suffix: } { T, F, TF } {
3341     \__bnvs_tl_if_empty:cTF { suffix } {
3342         \__bnvs_seq_pop_right:ccTF { path } { suffix } {
3343             \prg_return_true:
3344         } {
3345             \prg_return_false:
3346         }
3347     } {

```

```

3348     \prg_return_true:
3349   }
3350 }

Implementation detail: tl variable a is used.
3351 \BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {
3352
3353 }

3354 \BNVS_new:cpn { error_end_return_false:n } #1 {
3355   \__bnvs_build_tag:
3356   \__bnvs_tl_set:cx { a } {
3357     \BNVS_tl_use:c { tag } . \BNVS_tl_use:c { suffix }
3358   }
3359   \__bnvs_if_resolve_v:vvcTF { id } { a } { a } {
3360     \__bnvs_tl_put_right:cv { ans } { a }
3361     \__bnvs_split_loop_or_end_return:
3362   } {
3363     \__bnvs_if_resolve_V:vvcTF { id } { a } { a } {
3364       \__bnvs_tl_put_right:cv { ans } { a }
3365       \__bnvs_split_loop_or_end_return:
3366     } {
3367       #1
3368     }
3369   }
3370 }

3371 \BNVS_new:cpn { path_branch_loop_or_end_return: } {
3372   \__bnvs_if_call:TF {
3373     \__bnvs_if_path_branch:TF {
3374       \__bnvs_path_branch_end_return:
3375     } {
3376       \__bnvs_if_get:nvvcTF V { id } { tag } { a } {
3377         \__bnvs_if_TIP:cccTF { id } { a } { path } {
3378           \__bnvs_tl_set_eq:cc { tag } { a }
3379           \__bnvs_seq_merge:cc { path } { path_tail }
3380           \__bnvs_seq_clear:c { path_tail }
3381           \__bnvs_seq_set_eq:cc { path_head } { path }
3382           \__bnvs_path_branch_TIPn_loop_or_end_return:
3383         } {
3384           \__bnvs_path_branch_head_to_tail_end_return:
3385         }
3386       } {
3387         \__bnvs_path_branch_head_to_tail_end_return:
3388       }
3389     }
3390   } {
3391     \__bnvs_path_branch_end_return_false:n {
3392       Too-many-calls.
3393     }
3394   }
3395 }

3396 \BNVS_new:cpn { path_branch_end_return: } {
3397   \__bnvs_split_loop_or_end_return:
3398 }

```

```

3399 \BNVS_new:cpn { set_if_path_branch:n } {
3400   \prg_set_conditional:Npnn \__bnvs_if_path_branch: { TF }
3401 }

3402 \BNVS_new:cpn { path_branch_head_to_tail_end_return: } {
3403   \__bnvs_seq_pop_right:ccTF { path_head } { a } {
3404     \__bnvs_seq_put_left:cv { path_tail } { a }
3405     \__bnvs_build_tag_head:
3406     \__bnvs_path_branch_TIPn_loop_or_end_return:
3407   } {
3408     \__bnvs_build_tag:
3409     \__bnvs_seq_set_eq:cc { path_head } { path_tail }
3410     \__bnvs_seq_clear:c { path_tail }
3411     \__bnvs_is_gset:nvxTF V { id } {
3412       \__bnvs_tl_use:c { tag }.1
3413     } {
3414       \__bnvs_tl_set:cn { index } { 1 }
3415       \__bnvs_split_loop_or_end_return_index:
3416     } {
3417       \__bnvs_gset:nvvnn V { id } { tag } { 0 }
3418       \__bnvs_gset_cache:nvvnn V { id } { tag } { 0 }
3419       \__bnvs_path_branch_TIPn_loop_or_end_return:
3420     }
3421   }
3422 }

```

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

```

3423 \BNVS_new:cpn { path_branch_TIPn_loop_or_end_return: } {
3424   \__bnvs_build_tag_head:
3425   \__bnvs_if_resolve_v:vvvTF { id } { tag } { a } {
3426     \__bnvs_tl_put_right:cv { ans } { a }
3427     \__bnvs_split_loop_or_end_return:
3428   } {
3429     \__bnvs_if_resolve_V:vvvTF { id } { tag } { a } {
3430       \__bnvs_tl_put_right:cv { ans } { a }
3431       \__bnvs_split_loop_or_end_return:
3432     } {
3433       \__bnvs_path_branch_loop_or_end_return:
3434     }
3435   }
3436 }

```

- Case .... $\langle index \rangle$ .

```

3437 \BNVS_new:cpn { split_loop_or_end_return_index: } {
3438   % known, id, tag, path, suffix
3439   \__bnvs_set_if_path_branch:n {
3440     \__bnvs_if_append_index:vvvTF { id } { tag } { index } { ans } {
3441       \prg_return_true:
3442     } {
3443       \prg_return_false:
3444     }
3445   }

```

```

3446 \__bnvs_path_branch_loop_or_end_return:
3447 }

```

```

3448 \BNVS_new:cpn { split_loop_reset: } {
3449 \__bnvs_if:cT { reset_all } {
3450 \__bnvs_set_false:c { reset_all }
3451 \__bnvs_greset_cache:
3452 }
3453 \__bnvs_if:cT { reset } {
3454 \__bnvs_set_false:c { reset }
3455 \__bnvs_gunset:nv v { id } { tag }
3456 }
3457 }

```

- Case ....

```

3458 \BNVS_new:cpn { split_loop_or_end_return_v: } {
3459 \__bnvs_split_loop_reset:
3460 \__bnvs_set_if_path_branch:n {
3461 \__bnvs_if_append_v:vvcTF { id } { tag } { ans } {
3462 \prg_return_true:
3463 } {
3464 \__bnvs_if_append_V:vvcTF { id } { tag } { ans } {
3465 \prg_return_true:
3466 } {
3467 \prg_return_false:
3468 }
3469 }
3470 }
3471 \__bnvs_path_branch_loop_or_end_return:
3472 }

```

- Case ....<suffix>.

```

3473 \BNVS_new:cpn { split_loop_or_end_return_suffix: } {
3474 \__bnvs_if_resolve_V_loop_or_end_return_true:F {
3475 \__bnvs_set_if_path_branch:n {
3476 \BNVS_use:c {
3477 if_append_ \__bnvs_tl_use:c { suffix } :vvcTF
3478 } { id } { tag } { ans } {
3479 \__bnvs_if:cT { range } {
3480 \BNVS_set:cpn { if_resolve_round_ans: } { }
3481 }
3482 \prg_return_true:
3483 } {
3484 \prg_return_false:
3485 }
3486 }
3487 }
3488 }

```

- Case ...++.

```

3489 \BNVS_new:cpn { split_loop_or_end_return[...++]: } {
3490 \__bnvs_if:cTF { reset } {

```

- Case ....reset++.

```

3491     \cs_set:Npn \BNVS_split_loop: {
3492         NO~....reset++~for
3493         ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3494     }
3495 } {

```

- Case ...(.reset\_all)++.

```

3496     \cs_set:Npn \BNVS_split_loop: {
3497         \BNVS_error_ans:x {
3498             NO~...(.reset_all)++~for
3499             ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
3500         }
3501     }
3502 }
3503 \__bnvs_build_tag:
3504 \__bnvs_split_loop_reset:
3505 \__bnvs_if_append_v:vvncTF { id } { tag } { 1 } { ans } {
3506 } {
3507     \BNVS_error_ans:x {
3508         Problem~with~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }~use.
3509     }
3510 }
3511 \__bnvs_split_loop_or_end_return:
3512 }

```

```

3513 \BNVS_new:cpn { split_loop_or_end_return_assign: } {

```

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

```

3514 \__bnvs_if_resolve:vcTF { rhs } { rhs } {
3515     \__bnvs_gset:nvvv v { id } { tag } { rhs }
3516     \__bnvs_if_append_v:vvncTF { id } { tag } { ans } {
3517     } {
3518         \BNVS_error_ans:n { No~...=. ... }
3519     }
3520 } {
3521     \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c { rhs }. }
3522 }
3523 \__bnvs_split_loop_or_end_return:
3524 }

```

- Case ...+= ....

```

3525 \BNVS_new:cpn { split_loop_or_end_return_iadd:n } #1 {
3526     \__bnvs_if_resolve:ncTF { #1 } { rhs } {
3527         \__bnvs_split_loop_reset:
3528         \__bnvs_if_append_v_incr:vvncTF { id } { tag } { #1 } { ans } {

```



```

3529     } {
3530       \BNVS_error_ans:n { No~...+=... }
3531     }
3532   } {
3533     \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs } }
3534   }
3535   \__bnvs_split_loop_or_end_return:
3536 }

```

---

\\_\_bnvs\_if\_resolve\_query:ncTF \\_\_bnvs\_if\_resolve\_query:ncTF {<overlay query>} {<ans>} {<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 te <ans> tl variable. Ranges are supported with the colon syntax. This is executed within a local T<sub>E</sub>X 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.)

```

3537 \BNVS_new:cpn { resolve_query_end_return_true: } {
3538   \BNVS_end:
3539   \prg_return_true:
3540 }

3541 \BNVS_new:cpn { resolve_query_end_return_false: } {
3542   \BNVS_end:
3543   \prg_return_false:
3544 }

3545 \BNVS_new:cpn { resolve_query_end_return_false:n } #1 {
3546   \BNVS_end:
3547   \prg_return_false:
3548 }

3549 \BNVS_new:cpn { if_resolve_query_return_false:n } #1 {
3550   \prg_return_false:
3551 }

```

```

3552 \BNVS_new:cpn { resolve_query_error_return_false:n } #1 {
3553   \BNVS_error:n { #1 }
3554   \__bnvs_if_resolve_query_return_false:
3555 }
3556 \BNVS_generate_variant:cn { resolve_query_error_return_false:n } { x }

3557 \BNVS_new:cpn { if_resolve_query_return_unreachable: } {
3558   \__bnvs_resolve_query_error_return_false:n { UNREACHABLE }
3559 }

3560 \BNVS_new:cpn { if_blank:cTF } #1 {
3561   \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3562 }

3563 \BNVS_new_conditional:cpnn { if_match_pop_left:c } #1 { T, F, TF } {
3564   \BNVS_tl_use:nc {
3565     \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
3566   } { #1 } {
3567     \prg_return_true:
3568   } {
3569     \prg_return_false:
3570   }
3571 }

```

---

\\_\_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.

```

3572 \BNVS_new_conditional:cpnn { if_resolve_query_branch: } { T, F, TF } {
At start, we ignore the whole match.
3573   \__bnvs_if_match_pop_left:cT V {
3574     \__bnvs_if_match_pop_left:cT V {
3575       \__bnvs_if_blank:cTF V {
3576         \__bnvs_if_match_pop_left:cT A {
3577           \__bnvs_if_match_pop_left:cT Z {
3578             \__bnvs_if_match_pop_left:cT L {
3579               \__bnvs_if_blank:cTF A {
3580                 \__bnvs_if_match_pop_left:cT L {
3581                   \__bnvs_if_match_pop_left:cT Z {
3582                     \__bnvs_if_blank:cTF L {
3583                       \__bnvs_if_match_pop_left:cT Z {
3584                         \__bnvs_if_match_pop_left:cT L {
3585                           \__bnvs_if_blank:cTF L {
3586                             \BNVS_use:c { if_resolve_query_return[:Z]: }
3587                           } {
3588                             \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3589                           }

```

```

3590         }
3591     }
3592     } {
3593         \__bnvs_if_blank:cTF Z {
3594     \__bnvs_resolve_query_error_return_false:n { Missing~first~or~last }
3595         } {
3596             \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3597         }
3598     }
3599 }
3600 }
3601 } {
3602     \__bnvs_if_blank:cTF Z {
3603         \__bnvs_if_blank:cTF L {
3604             \BNVS_use:c { if_resolve_query_return[A:]: }
3605         } {
3606             \BNVS_use:c { if_resolve_query_return[A::L]: }
3607         }
3608     } {
3609         \__bnvs_if_blank:cTF L {
3610             \BNVS_use:c { if_resolve_query_return[A:Z]: }
3611         } {
3612             \__bnvs_if_resolve_query_return_unreachable:
3613         }
3614     }
3615 }
3616 }
3617 }
3618 }
3619 } {
3620     \BNVS_use:c { if_resolve_query_return[V]: }
3621 }
3622 }
3623 }
3624 }

```

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

#### Single value

```

3625 \BNVS_new:cpn { if_resolve_query_return[V]: } {
3626     \__bnvs_if_resolve:vcTF { V } { ans } {
3627         \prg_return_true:
3628     } {
3629         \prg_return_false:
3630     }
3631 }

```

#### <first>:<last> range

```

3632 \BNVS_new:cpn { if_resolve_query_return[A:Z]: } {
3633     \__bnvs_if_resolve:vcTF { A } { ans } {
3634         \__bnvs_tl_put_right:cn { ans } { - }
3635         \__bnvs_if_append:vcTF { Z } { ans } {
3636             \prg_return_true:
3637         } {

```

```

3638     \prg_return_false:
3639   }
3640 } {
3641   \prg_return_false:
3642 }
3643 }

☛ <first>::<length> range
3644 \BNVS_new:cpn { if_resolve_query_return[A::L]: } {
3645   \__bnvs_if_resolve:vcTF { A } { A } {
3646     \__bnvs_if_resolve:vcTF { L } { ans } {
3647       \__bnvs_tl_put_right:cn { ans } { + }
3648       \__bnvs_tl_put_right:cv { ans } { A }
3649       \__bnvs_tl_put_right:cn { ans } { -1 }
3650       \__bnvs_round:c { ans }
3651       \__bnvs_tl_put_left:cn { ans } { - }
3652       \__bnvs_tl_put_left:cv { ans } { A }
3653       \prg_return_true:
3654     } {
3655       \prg_return_false:
3656     }
3657   } {
3658     \prg_return_false:
3659   }
3660 }

☛ <first>: and <first>:: range
3661 \BNVS_new:cpn { if_resolve_query_return[A:]: } {
3662   \__bnvs_if_resolve:vcTF { A } { ans } {
3663     \__bnvs_tl_put_right:cn { ans } { - }
3664     \prg_return_true:
3665   } {
3666     \prg_return_false:
3667   }
3668 }

☛ :<last>::<length> or ::<length>:<last> range
3669 \BNVS_new:cpn { if_resolve_query_return[:Z::L]: } {
3670   \__bnvs_if_resolve:vcTF { Z } { Z } {
3671     \__bnvs_if_resolve:vcTF { L } { ans } {
3672       \__bnvs_tl_put_left:cn { ans } { 1- }
3673       \__bnvs_tl_put_right:cn { ans } { + }
3674       \__bnvs_tl_put_right:cv { ans } { Z }
3675       \__bnvs_round:c { ans }
3676       \__bnvs_tl_put_right:cn { ans } { - }
3677       \__bnvs_tl_put_right:cv { ans } { Z }
3678       \prg_return_true:
3679     } {
3680       \prg_return_false:
3681     }
3682   } {
3683     \prg_return_false:
3684   }
3685 }

```

```

3686 \BNVS_new:cpn { if_resolve_query_return[:]: } {
3687   \_bnvs_tl_set:cn { ans } { - }
3688   \prg_return_true:
3689 }
3690 :<last> range
3690 \BNVS_new:cpn { if_resolve_query_return[:Z]: } {
3691   \_bnvs_tl_set:cn { ans } { - }
3692   \_bnvs_if_append:vcTF { Z } { ans } {
3693     \prg_return_true:
3694   } {
3695     \prg_return_false:
3696   }
3697 }

```

---

```

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

```

---

Evaluate only one query.

```

3698 \BNVS_new_conditional:cpnn { if_resolve_query:nc } #1 #2 { T, F, TF } {
3699   \_bnvs_greset_call:
3700   \_bnvs_match_if_once:NnTF \c\_bnvs_A_cln_Z_regex { #1 } {
3701     \BNVS_begin:
3702     \_bnvs_if_resolve_query_branch:TF {
3703       \BNVS_end_tl_set:cv { #2 } { ans }
3704       \prg_return_true:
3705     } {
3706       \BNVS_end:
3707       \prg_return_false:
3708     }
3709   } {
3710     \BNVS_error:n { Syntax~error:~#1 }
3711     \BNVS_end:
3712     \prg_return_false:
3713   }
3714 }

```

---

```

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

```

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

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

(End of definition for `\l__bnvs_query_seq`.)

`\l__bnvs_ans_seq` Storage for the evaluated result.

(End of definition for `\l__bnvs_ans_seq`.)

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

```

3715 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End of definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

3716 \BNVS_new_conditional:cpnn { if_resolve_queries:nc } #1 #2 { TF } {
3717   \BNVS_begin:

```

Local variables cleared

```

3718   \__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`.

```

3719   \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 justed cleared above.

```

3720   \BNVS_set:cpn { end_return: } {
3721     \__bnvs_seq_if_empty:cTF { ans } {
3722       \BNVS_end:
3723     } {

```

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

```

3724     \exp_args:Nnx
3725     \use:n {
3726       \BNVS_end:
3727       \__bnvs_tl_put_right:cn { #2 }
3728     } { \__bnvs_seq_use:cn { ans } , }
3729   }
3730   \prg_return_true:
3731 }
3732 \__bnvs_seq_map_inline:cn { query } {
3733   \__bnvs_tl_clear:c { ans }

```

```

3734 \__bnvs_if_resolve_query:ncTF { ##1 } { ans } {
3735 \__bnvs_tl_if_empty:cF { ans } {
3736 \__bnvs_seq_put_right:cv { ans } { ans }
3737 }
3738 } {
3739 \seq_map_break:n {
3740 \BNVS_set:cpn { end_return: } {
3741 \BNVS_error:n { Circular/Undefined~dependency~in~#1}
3742 \exp_args:Nnx
3743 \use:n {
3744 \BNVS_end:
3745 \__bnvs_tl_put_right:cn { #2 }
3746 } { \__bnvs_seq_use:cn { ans } , }
3747 \prg_return_false:
3748 }
3749 }
3750 }
3751 }
3752 \__bnvs_end_return:
3753 }

3754 \NewDocumentCommand \BeanovesResolve { 0{} m } {
3755 \BNVS_begin:
3756 \keys_define:nn { BeanovesResolve } {
3757 in:N .tl_set:N = \l__bnvs_resolve_in_tl,
3758 in:N .initial:n = { },
3759 show .bool_set:N = \l__bnvs_resolve_show_bool,
3760 show .default:n = true,
3761 show .initial:n = false,
3762 }
3763 \keys_set:nn { BeanovesResolve } { #1 }
3764 \__bnvs_tl_clear:c { ans }
3765 \__bnvs_if_resolve_queries:ncTF { #2 } { ans } {
3766 \__bnvs_tl_if_empty:cTF { resolve_in } {
3767 \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3768 \BNVS_tl_use:Nv \BNVS_end: { ans }
3769 } {
3770 \BNVS_end:
3771 }
3772 } {
3773 \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3774 \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3775 \BNVS_end:
3776 \tl_set:Nn ##1 { ##2 }
3777 ##2
3778 }
3779 \BNVS_tl_use:nv {
3780 \exp_last_unbraced:Nv \BNVS_end:Nn \l__bnvs_resolve_in_tl
3781 } { ans }
3782 } {
3783 \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3784 \BNVS_end:
3785 \tl_set:Nn ##1 { ##2 }

```

```

3786     }
3787     \BNVS_tl_use:nv {
3788         \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
3789     } { ans }
3790 }
3791 }
3792 } {}
3793 }

```

## 6.21 Resetting counters and values

```

3794 \BNVS_new:cpn { reset:n } #1 {
3795     \BNVS_begin:
3796     \__bnvs_set_true:c { reset }
3797     \__bnvs_set_false:c { provide }
3798     \__bnvs_tl_clear:c { root }
3799     \__bnvs_int_zero:c { i }
3800     \__bnvs_tl_set:cn { a } { #1 }
3801     \__bnvs_provide_off:
3802     \BNVS_tl_use:Nv \__bnvs_brace_keyval:n { a }
3803     \BNVS_end_tl_set:cv { id_last } { id_last }
3804 }

3805 \BNVS_new:cpn { reset:v } {
3806     \BNVS_tl_use:Nv \__bnvs_reset:n
3807 }

3808 \makeatletter
3809 \NewDocumentCommand \BeanovesReset { 0{} m } {
3810     \tl_if_empty:NTF \@currentenvir {

```

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

```

3811     \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}}~in~the~preamble.}
3812 } {
3813     \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, clear everything.

```

3814     \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}}~at~the~top~level.}
3815 }
3816 \BNVS_begin:
3817 \__bnvs_set_true:c { reset }
3818 \__bnvs_set_false:c { provide }
3819 \keys_define:nn { BeanovesReset } {
3820     all .bool_set:N = \l__bnvs_reset_all_bool,
3821     all .default:n = true,
3822     all .initial:n = false,
3823     only .bool_set:N = \l__bnvs_only_bool,
3824     only .default:n = true,
3825     only .initial:n = false,
3826 }
3827 \keys_set:nn { BeanovesReset } { #1 }
3828 \__bnvs_tl_clear:c { root }
3829 \__bnvs_int_zero:c { i }
3830 \__bnvs_tl_set:cn { a } { #2 }

```



```
3831     \__bnvs_provide_off:
3832     \BNVS_tl_use:Nv \__bnvs_brace_keyval:n { a }
3833     \BNVS_end_tl_set:cv { id_last } { id_last }
3834     \ignorespaces
3835   }
3836 }
3837 \makeatother
3838 \ExplSyntaxOff
```