

beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2024/01/11

Abstract

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

Contents

| | | |
|----------|--|-----------|
| 1 | Installation | 3 |
| 1.1 | Package manager | 3 |
| 1.2 | Manual installation | 3 |
| 1.3 | Usage | 3 |
| 2 | Minimal example | 3 |
| 3 | Named overlay sets | 4 |
| 3.1 | Presentation | 4 |
| 3.2 | Named overlay reference | 4 |
| 3.3 | Defining named overlay sets | 5 |
| 3.3.1 | Value specifiers | 5 |
| 3.3.2 | Range specifiers | 6 |
| 3.3.3 | List specifiers | 6 |
| 4 | Resolution of <code>?(...)</code> query expressions | 6 |
| 4.1 | Range overlay queries | 7 |
| 4.2 | Value counter queries | 8 |
| 4.3 | The beamer counters | 9 |
| 4.4 | Multiple queries | 10 |
| 4.5 | Frame id | 10 |
| 4.6 | Resolution command | 10 |
| 5 | Support | 10 |

| | | |
|----------|--|-----------|
| 6 | Implementation | 10 |
| 6.1 | Package declarations | 11 |
| 6.2 | Overview | 11 |
| 6.3 | Facility layer: definitions and naming | 11 |
| 6.3.1 | Debug | 13 |
| 6.3.2 | Facility layer: Functions | 13 |
| 6.3.3 | logging | 13 |
| 6.3.4 | Facility layer: Variables | 14 |
| 6.3.5 | Regex | 20 |
| 6.3.6 | Token lists | 21 |
| 6.3.7 | Strings | 24 |
| 6.3.8 | Sequences | 26 |
| 6.3.9 | Integers | 27 |
| 6.4 | Debug facilities | 28 |
| 6.5 | Local variables | 28 |
| 6.6 | Infinite loop management | 30 |
| 6.7 | Overlay specification | 30 |
| 6.7.1 | Registration | 30 |
| 6.7.2 | Basic model functions | 35 |
| 6.7.3 | The provide mode | 43 |
| 6.7.4 | Initialize | 44 |
| 6.8 | Implicit value counter | 46 |
| 6.8.1 | Unresolvable | 46 |
| 6.9 | Regular expressions | 47 |
| 6.10 | End group setter | 52 |
| 6.11 | beamer.cls interface | 52 |
| 6.12 | Utilities | 53 |
| 6.12.1 | Split utilities | 53 |
| 6.12.2 | Match utilities | 54 |
| 6.12.3 | Utilities | 57 |
| 6.12.4 | Next index | 59 |
| 6.13 | Parsing | 59 |
| 6.13.1 | Square brackets | 60 |
| 6.13.2 | Range or value lists | 61 |
| 6.13.3 | List specifiers | 62 |
| 6.13.4 | Items between braces | 64 |
| 6.13.5 | High level and root parser | 66 |
| 6.14 | Scanning named overlay specifications | 69 |
| 6.14.1 | Top level | 70 |
| 6.15 | Resolution | 74 |
| 6.16 | Evaluation bricks | 76 |
| 6.16.1 | Helpers | 76 |
| 6.16.2 | Resolve from initial values | 76 |
| 6.16.3 | V for value | 79 |
| 6.16.4 | R for range | 81 |
| 6.16.5 | Already complete | 82 |
| 6.16.6 | Assignment | 82 |
| 6.16.7 | beamer counters | 83 |
| 6.17 | Value counter | 94 |
| 6.18 | Functions for the resolution | 95 |

| | |
|--|-----|
| 6.18.1 Resolve one query | 107 |
| 6.19 Resetting counters and values | 113 |

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 `TeX Live` and `mpm` for `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 `LaTeX` document that uses the `beamer` class. Should the package cause problems, its features can be temporarily deactivated with simple commands `\BeanovesOff` and `\BeanovesOn`.

2 Minimal example

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

```

1 \documentclass{beamer}
2 \RequirePackage{beanoves}
3 \begin{document}
4 \Beanoves {
5   A = 1:4,
6   B = A.last::3,
7   C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \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. frame $\langle id \rangle!$, 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 T_EX 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 letters¹.

The mapping from *named overlay references* to sets of integers is defined at the global T_EX 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.

`\Beanoves` `\Beanoves{\langle ref_1 \rangle=\langle spec_1 \rangle, \dots, \langle ref_j \rangle=\langle spec_j \rangle}`

`\Beanoves*` 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.

When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside L^AT_EX 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 time².

`beanoves` `beanoves = {\langle ref_1 \rangle=\langle spec_1 \rangle, \dots, \langle ref_j \rangle=\langle spec_j \rangle}`

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

3.3.1 Value specifiers

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

Standalone

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

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*.

¹This will avoid collisions with the `fp` module of `exp13`.

²Precision is needed about the exact time when the expansion occurs.

3.3.2 Range specifiers

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

Standalone

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

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

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

3.3.3 List specifiers

$\langle ref \rangle = [\langle def_1 \rangle, \dots, \langle def_j \rangle]$, where $\langle def_k \rangle$, $1 \leq k \leq j$, is one of

- $\langle index \rangle = \langle value \rangle$,
- $\langle value \rangle$, a shortcut for $\langle i \rangle = \langle value \rangle$, $\langle i \rangle$ being the smallest positive integer such that $\langle ref \rangle . \langle i \rangle$ is not already defined.

The first step is to remove previous $\langle ref \rangle$ related definitions, then execute the various $\langle ref \rangle . \langle i \rangle = \langle value \rangle$ definitions in the order given. Here is the [implementation](#).

$\langle ref \rangle = \{ \langle def_1 \rangle, \dots, \langle def_j \rangle \}$, where $\langle def_k \rangle$, $1 \leq k \leq j$, is one of

- $\langle name \rangle = \langle spec \rangle$,
- $\langle name \rangle$ for $\langle name \rangle = 1$,

The first step is to remove previous $\langle ref \rangle$ related definitions, then execute the various $\langle ref \rangle . \langle name \rangle = \langle spec \rangle$ definitions in the order given. In a final step, the $\langle name \rangle$'s are collected in a comma separated list to initialize $\langle ref \rangle$ with. $\langle spec \rangle$ is any specifier. Here is the [implementation](#).

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

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

The $\langle queries \rangle$ argument is a comma separated list of individual $\langle query \rangle$'s processed from left to right as explained below. Notice that nesting a $?(\langle \dots \rangle)$ query expression inside another query expression is 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 $\langle A.1 \rangle$ 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 $\langle \text{value} \rangle$, $\langle \text{first} \rangle$, $\langle \text{last} \rangle$, $\langle \text{length} \rangle$ and $\langle i \text{ expr} \rangle$ 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 $\langle \text{first} \rangle ::$ define 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 $\langle \text{first} \rangle + \langle \text{length} \rangle = \langle \text{last} \rangle + 1$.

Define a range limited from both above and below:

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

Notice that the resolution of the $\langle \text{ref} \rangle$ overlay query 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 \insertframenumber} {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}

```

$\langle \text{ref} \rangle = [\dots]$

$\langle \text{ref} \rangle = \{\dots\}$ See the list range specifiers in section 3.3.3.

4.2 Value counter queries

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

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

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 space³.

$\langle \text{ref} \rangle += \langle \text{integer expression} \rangle$, 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.

³The parser for algebraic expression is very rudimentary.


```

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](A.C) == \BeanovesResolve[show](A.20) == 20,
12 \BeanovesResolve[show](A.C+=10) == \BeanovesResolve[show](A.20) == 20+10,
13 \BeanovesResolve[show](A.C) == \BeanovesResolve[show](A.20) == 30,
14 \BeanovesResolve[show](B.C) == \BeanovesResolve[show](B.20) == 20,
15 \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 [*options*] {*<ref₁*[=*spec₁*], ..., <ref_{*j*}[=*spec_j*]]}

This command is very similar to `\Beanoves`, except that a standalone *<ref_{*i*}* resets the counter to the last value it was initialized with, 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.

4.3 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{<ref>=pauses}` or in a query `?(...<ref>=pauses)...`. Then later on, we can use `?(...<ref>...)` to refer to this saved value in the same frame⁴. 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{<ref>=slideinframe}`

⁴See [stackexchange](#) for an alternative that needs at least two passes.

or inside a query $\langle \langle \dots \rangle \langle \text{ref} \rangle = \text{slideinframe} \rangle \langle \dots \rangle$. It uses the numerical value of `\insertslideinframe`.

4.4 Multiple queries

It is possible to replace the comma separated list of queries $\langle \langle \text{query}_1 \rangle \rangle, \dots, \langle \langle \text{query}_j \rangle \rangle$ with the shorter single query $\langle \langle \text{query}_1 \rangle, \dots, \langle \text{query}_j \rangle \rangle$.

4.5 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 $\langle \text{last frame id} \rangle !A.1$, but when used just after `Y!C.2`, for example, it becomes a shortcut to `Y!A.1` because the *last frame id* is then `Y`.

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

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

We can use the same frame $\langle id \rangle$ for different frames to share named overlay sets. Another possibility offered by the `beanoves` package to share named overlay sets is a fall back mechanism, for example when `X!A` cannot be resolved, resolve `!A` instead.

4.6 Resolution command

```
\BeanovesResolve \BeanovesResolve [ $\langle \text{setup} \rangle$ ] { $\langle \text{queries} \rangle$ }
```

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

`show` the result is left into the input stream

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

5 Support

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

6 Implementation

Identify the internal prefix (`LATEX3` DocStrip convention).

1 `\@@=bnvs`

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 Overview

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

There are mainly two steps: parsing and resolution. Parsing occurs while executing the `\Beanoves` command to build a data model whereas the resolution translates queries into **beamer** specifications based on this data model.

The development is easier due to a facility layer over **expl**.

6.3 Facility layer: definitions and naming

In order to make the code shorter and easier to read during development, we add a layer over **L^AT_EX3**. The **c** and **v** argument specifiers take a slightly different meaning when used in a function which name contains **bnvs** or **BNVS**. Where **L^AT_EX3** would transform `l__bnvs_ref_tl` into `\l__bnvs_ref_tl`, **bnvs** will directly transform **ref** into `\l__bnvs_ref_tl`. The type of the local variable used depends on the context and may be **seq** or **int** for example. There are however a pair of exceptions mentionned 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.

```

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

```

These are naming functions internally used to focus on the discriminating part of variable or function names.

```

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

`\BNVS_use_raw:c` is a convenient wrapper over `\use:c`. possibly prepended with some code, for debugging and testing. 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 specific to the beanoves package.

```

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_args:NNc \BNVS_use_raw:NN #1 { #2 }
27 }

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

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

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

38 \tl_new:N \l__bnvs_last_unbraced_tl
39 \cs_new:Npn \BNVS_tl_last_unbraced:nv #1 {
40   \tl_set:Nn \l__bnvs_last_unbraced_tl { #1 }
41   \BNVS_tl_use:nc { \exp_last_unbraced:NV \l__bnvs_last_unbraced_tl }
42 }

```

```

43 \cs_new:Npn \BNVS_tl_use:nvv #1 #2 {
44   \BNVS_tl_use:nv { \BNVS_tl_use:nv { #1 } { #2 } }
45 }
46 \cs_new:Npn \BNVS_tl_use:nvvv #1 #2 {
47   \BNVS_tl_use:nvv { \BNVS_tl_use:nv { #1 } { #2 } }
48 }

49 \cs_new:Npn \BNVS_log:n #1 { }
50 \cs_generate_variant:Nn \BNVS_log:n { x }

```

6.3.1 Debug

| | |
|---------------------|--|
| \BNVS_DEBUG_on: | \BNVS_DEBUG_on: |
| \BNVS_DEBUG_off: | \BNVS_DEBUG_off: |
| \BNVS_DEBUG_push:nn | \BNVS_DEBUG_push:nn {<types on>} {<types off>} |
| \BNVS_DEBUG_pop: | \BNVS_DEBUG_pop: |

These functions activate or deactivate the debug mode. They are only active in the `beanoves-debug` package. Manage debug messaging for one given *<type>* or *<types>*. The implementation is not publicly exposed. The *<type>* is a single letter of ******.

6.3.2 Facility layer: Functions

| | |
|---------------|--|
| \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. |

```

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

54 \cs_new:Npn \BNVS_set:cpn #1 {
55   \cs_set:cpn { \BNVS:c { #1 } }
56 }

57 \cs_generate_variant:Nn \cs_generate_variant:Nn { c }
58 \cs_new:Npn \BNVS_generate_variant:cn #1 {
59   \cs_generate_variant:cn { \BNVS:c { #1 } }
60 }

```

6.3.3 logging

| | |
|-----------------|--|
| \BNVS_warning:n | \BNVS_warning:n {<message>} |
| \BNVS_warning:x | \BNVS_error:n {<message>} |
| \BNVS_error:n | \BNVS_fatal:n {<message>} |
| \BNVS_error:x | Very rudimentary. Non long message for error recovery. |
| \BNVS_fatal:n | |
| \BNVS_fatal:x | |

```

61 \msg_new:nnn { beanoves } { :n } { #1 }
62 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }

63 \cs_new:Npn \BNVS_warning:n {
64   \msg_warning:nnn { beanoves } { :n }
65 }
66 \cs_new:Npn \BNVS_warning:x {
67   \msg_warning:nnx { beanoves } { :n }
68 }

69 \cs_new:Npn \BNVS_error:n {
70   \msg_error:nnn { beanoves } { :n }
71 }
72 \cs_generate_variant:Nn \BNVS_error:n { x }

73 \cs_new:Npn \BNVS_fatal:n {
74   \msg_fatal:nnn { beanoves } { :n }
75 }
76 \cs_new:Npn \BNVS_fatal:x {
77   \msg_fatal:nnx { beanoves } { :n }
78 }
79 \cs_new:Npn \BNVS_fatal_unreachable: {
80   \BNVS_fatal:n { Unreachable }
81 }
82 \cs_new:Npn \BNVS_fatal_unreachable: { }

```

6.3.4 Facility layer: Variables

```

\BNVS_N_new:c \BNVS_N_new:c {<type>}
\BNVS_v_new:c \BNVS_v_new:c {<type>}

```

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

```

83 \cs_new:Npn \BNVS_N_new:c #1 {
84   \cs_new:cpn { BNVS_#1:c } ##1 {
85     1 \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
86   }
87   \cs_new:cpn { BNVS_#1_new:c } ##1 {
88     \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
89   }
90   \cs_new:cpn { BNVS_#1_use:c } ##1 {
91     \use:c { \cs:w BNVS_#1:c \cs_end: { ##1 } }
92   }
93   \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
94     \BNVS_use_raw:Nc
95     ##1 { \cs:w BNVS_#1:c \cs_end: { ##2 } }
96   }
97   \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
98     \BNVS_use_raw:nc
99     { ##1 } { \cs:w BNVS_#1:c \cs_end: { ##2 } }
100  }
101 }

```

```

102 \cs_new:Npn \BNVS_v_new:c #1 {
103   \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
104     \BNVS_use_raw:nc
105     { \exp_args:NV ##1 }
106     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
107   }
108   \cs_new:cpn { BNVS_#1_use:cv } ##1 ##2 {
109     \BNVS_use_raw:nc
110     { \exp_args:NnV \BNVS_use:c { ##1 } }
111     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
112   }
113   \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
114     \BNVS_use_raw:nc
115     { \exp_args:NnV \use:n { ##1 } }
116     { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
117   }
118 }

119 \BNVS_N_new:c { bool }
120 \BNVS_N_new:c { int }
121 \BNVS_v_new:c { int }
122 \BNVS_N_new:c { tl }
123 \BNVS_v_new:c { tl }
124 \cs_new:Npn \BNVS_tl_use:Nvv #1 #2 {
125   \BNVS_tl_use:nv { \BNVS_tl_use:Nv #1 { #2 } }
126 }

127 \cs_new:Npn \BNVS_tl_use:Nvvv #1 #2 {
128   \BNVS_tl_use:nvv { \BNVS_tl_use:Nv #1 { #2 } }
129 }

130 \BNVS_N_new:c { str }
131 \BNVS_v_new:c { str }
132 \BNVS_N_new:c { seq }
133 \BNVS_v_new:c { seq }
134 \cs_undefine:N \BNVS_N_new:c

```

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

```

135 \cs_new:Npn \BNVS_use:Ncn #1 #2 #3 {
136   \BNVS_use_raw:c { BNVS_#3_use:Nc } #1 { #2 }
137 }

138 \cs_new:Npn \BNVS_use:ncn #1 #2 #3 {
139   \BNVS_use_raw:c { BNVS_#3_use:nc } { #1 } { #2 }
140 }

141 \cs_new:Npn \BNVS_use:Nvn #1 #2 #3 {
142   \BNVS_use_raw:c { BNVS_#3_use:Nv } #1 { #2 }
143 }

144 \cs_new:Npn \BNVS_use:nvn #1 #2 #3 {
145   \BNVS_use_raw:c { BNVS_#3_use:nv } { #1 } { #2 }
146 }

```

```

147 \cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
148   \BNVS_use:ncn {
149     \BNVS_use:Ncn   #1   { #2 } { #3 }
150   }
151 }

152 \cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
153   \BNVS_use:ncn {
154     \BNVS_use:ncn { #1 } { #2 } { #3 }
155   }
156 }

157 \cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
158   \BNVS_use:ncn {
159     \BNVS_use:Nvn   #1   { #2 } { #3 }
160   }
161 }

162 \cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
163   \BNVS_use:ncn {
164     \BNVS_use:nvn { #1 } { #2 } { #3 }
165   }
166 }

167 \cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
168   \BNVS_use:ncn {
169     \BNVS_use:Ncncn   #1   { #2 } { #3 } { #4 } { #5 }
170   }
171 }

172 \cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
173   \BNVS_use:ncn {
174     \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
175   }
176 }

```

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

```

177 \cs_new:Npn \BNVS_new_c:nc #1 #2 {
178   \BNVS_new_cpn { #1_#2:c } {
179     \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
180   }
181 }

182 \cs_new:Npn \BNVS_new_cn:nc #1 #2 {
183   \BNVS_new_cpn { #1_#2:cn } ##1 {
184     \BNVS_use:ncn { \BNVS_use_raw:c { #1_#2:Nn } } { ##1 } { #1 }
185   }
186 }

187 \cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
188   \BNVS_new_cpn { #2:cnn } ##1 {
189     \BNVS_use:Ncn { #3 } { ##1 } { #1 }
190   }
191 }

```



```

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

197 \cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
198   \BNVS_new:cpn { #2:cnv } ##1 ##2 {
199     \BNVS_tl_use:nv {
200       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
201     }
202   }
203 }

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

209 \cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
210   \BNVS_new:cpn { #2:cnx } ##1 ##2 {
211     \exp_args:Nnx \use:n {
212       \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 }
213     }
214   }
215 }

216 \cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
217   \BNVS_use_raw:nc {
218     \BNVS_new_cnx:ncN { #1 } { #1_#2 }
219   } { #1_#2:Nnn }
220 }

221 \cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
222   \BNVS_new:cpn { #2:cc } ##1 ##2 {
223     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } { #4 }
224   }
225 }

226 \cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
227   \BNVS_use_raw:nc {
228     \BNVS_new_cc:ncNn { #1 } { #1_#2 }
229   } { #1_#2:NN }
230 }

231 \cs_new:Npn \BNVS_new_cc:nc #1 #2 {
232   \BNVS_new_cc:ncn { #1 } { #2 } { #1 }
233 }

234 \cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
235   \BNVS_new:cpn { #2:cn } ##1 {
236     \BNVS_use:Ncn #3 { ##1 } { #1 }
237   }
238 }

```

```

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

244 \cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
245   \BNVS_new_cpn { #2:cv } ##1 ##2 {
246     \BNVS_use:nvn {
247       \BNVS_use:Ncn #3 { ##1 } { #1 }
248     } { ##2 } { #4 }
249   }
250 }

251 \cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
252   \BNVS_use_raw:nc {
253     \BNVS_new_cv:ncNn { #1 } { #1_#2 }
254   } { #1_#2:Nn }
255 }

256 \cs_new:Npn \BNVS_new_cv:nc #1 #2 {
257   \BNVS_new_cv:ncn { #1 } { #2 } { #1 }
258 }

259 \cs_new:Npn \BNVS_l_use:Ncn #1 #2 #3 {
260   \BNVS_use_raw:Nc #1 { \BNVS_l:cn { #2 } { #3 } }
261 }

262 \cs_new:Npn \BNVS_l_use:ncn #1 #2 #3 {
263   \BNVS_use_raw:nc { #1 } { \BNVS_l:cn { #2 } { #3 } }
264 }

265 \cs_new:Npn \BNVS_g_use:Ncn #1 #2 #3 {
266   \BNVS_use_raw:Nc #1 { \BNVS_g:cn { #2 } { #3 } }
267 }

268 \cs_new:Npn \BNVS_g_use:ncn #1 #2 #3 {
269   \BNVS_use_raw:nc { #1 } { \BNVS_g:cn { #2 } { #3 } }
270 }

```

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

```

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

272 \cs_new:Npn \BNVS_new_conditional:cpnn #1 {
273   \prg_new_conditional:cpnn { \BNVS:c { #1 } }
274 }
275 \cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
276 \cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
277   \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
278 }

```

```

279 \cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
280   \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
281     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
282       \prg_return_true:
283     } {
284       \prg_return_false:
285     }
286   }
287 }

288 \cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
289   \BNVS_use:nc {
290     \BNVS_new_conditional_vn:cNnn { #1 }
291   } { #1:nn TF } { #2 }
292 }

293 \cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
294   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
295     \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
296       \prg_return_true:
297     } {
298       \prg_return_false:
299     }
300   }
301 }

302 \cs_new:Npn \BNVS_new_conditional_vc:cnn #1 {
303   \BNVS_use:nc {
304     \BNVS_new_conditional_vc:cNnn { #1 }
305   } { #1:ncTF }
306 }

307 \cs_new:Npn \BNVS_new_conditional_vvc:cNnnn #1 #2 #3 #4 #5 {
308   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #5 } {
309     \BNVS_use:nvn {
310       \BNVS_use:Nvn #2 { ##1 } { #3 }
311     } { ##2 } { #4 } { ##3 } {
312       \prg_return_true:
313     } {
314       \prg_return_false:
315     }
316   }
317 }

318 \cs_new:Npn \BNVS_new_conditional_vvc:cnnn #1 {
319   \BNVS_use:nc {
320     \BNVS_new_conditional_vvc:cNnnn { #1 }
321   } { #1:nncTF }
322 }

```

```

323 \cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
324   \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
325     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
326       \prg_return_true:
327     } {
328       \prg_return_false:
329     }
330   }
331 }

332 \cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
333   \BNVS_use:nc {
334     \BNVS_new_conditional_vc:cNn { #1 }
335   } { #1:ncTF }
336 }

337 \cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
338   \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
339     \BNVS_tl_use:nv {
340       \BNVS_tl_use:Nv #2 { ##1 }
341     } { ##2 } { ##3 } {
342       \prg_return_true:
343     } {
344       \prg_return_false:
345     }
346   }
347 }

348 \cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
349   \BNVS_use:nc {
350     \BNVS_new_conditional_vvc:cNn { #1 }
351   } { #1:nncTF }
352 }

```

6.3.5 Regex

```

353 \cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
354   \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
355 }

```

6.3.6 Token lists

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

These are shortcuts to

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

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

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

```

356 \cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
357   \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
358     \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
359       \prg_return_true:
360     } {
361       \prg_return_false:
362     }
363   }
364 }

365 \cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
366   \BNVS_use:nc {
367     \BNVS_new_conditional_vnc:cNn { #1 }
368   } { #1:nncTF }
369 }
```

```
\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.

```

370 \cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
371   \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
372     \BNVS_tl_use:nv {
373       \BNVS_tl_use:Nv #2 { ##1 }
374     } { ##2 } { ##3 } { ##4 } {
375       \prg_return_true:
376     } {
377       \prg_return_false:
378     }
379   }
380 }

381 \cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
382   \BNVS_use:nc {
383     \BNVS_new_conditional_vvnc:cNn { #1 }
384   } { #1:nncTF }
385 }

386 \cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
387   \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
388     \BNVS_tl_use:nvv {
389       \BNVS_tl_use:Nv #2 { ##1 }
390     } { ##2 } { ##3 } { ##4 } {
391       \prg_return_true:
392     } {
393       \prg_return_false:
394     }
395   }
396 }

397 \cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
398   \BNVS_use:nc {
399     \BNVS_new_conditional_vvvc:cNn { #1 }
400   } { #1:nncTF }
401 }

402 \cs_new:Npn \BNVS_new_tl_c:c {
403   \BNVS_new_c:nc { tl }
404 }
405 \BNVS_new_tl_c:c { clear }
406 \BNVS_new_tl_c:c { use }
407 \BNVS_new_tl_c:c { count }

408 \BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
409   \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
410 }

411 \cs_new:Npn \BNVS_new_tl_cn:c {
412   \BNVS_new_cn:nc { tl }
413 }
```

```

414 \cs_new:Npn \BNVS_new_tl_cv:c #1 {
415   \BNVS_new_cv:ncn { tl } { #1 } { tl }
416 }
417 \BNVS_new_tl_cn:c { set }
418 \BNVS_new_tl_cv:c { set }

419 \BNVS_new:cpn { tl_set:cx } {
420   \exp_args:Nnx \__bnvs_tl_set:cn
421 }
422 \BNVS_new_tl_cn:c { put_right }
423 \BNVS_new_tl_cv:c { put_right }
424 % \BNVS_generate_variant:cn { tl_put_right:cn } { cx }

425 \BNVS_new:cpn { tl_put_right:cx } {
426   \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
427 }
428 \BNVS_new_tl_cn:c { put_left }
429 \BNVS_new_tl_cv:c { put_left }
430 % \BNVS_generate_variant:cn { tl_put_left:cn } { cx }

431 \BNVS_new:cpn { tl_put_left:cx } {
432   \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
433 }

```

```

\__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>}

```

434 \cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
435   \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
436     \BNVS_use:Ncn #3 { ##1 } { #1 } {
437       \prg_return_true:
438     } {
439       \prg_return_false:
440     }
441   }
442 }

443 \cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
444   \BNVS_use_raw:nc {
445     \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
446   } { #1_#2:NTF }
447 }
448 \BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }

```

```

449 \BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
450   \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
451     \prg_return_true:
452   } {
453     \prg_return_false:
454   }
455 }

456 \cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
457   \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
458     \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
459       \prg_return_true:
460     } {
461       \prg_return_false:
462     }
463   }
464 }

465 \cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
466   \BNVS_use_raw:nc {
467     \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
468   } { #1_#2:NnTF }
469 }
470 \BNVS_new_conditional_cn:ncn { tl } { if_eq } { T, F, TF }

471 \cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
472   \BNVS_new_conditional:cpnn { #2:cv } ##1 ##2 { #4 } {
473     \BNVS_use:nvn {
474       \BNVS_use:Ncn #3 { ##1 } { #1 }
475     } { ##2 } { #1 } {
476       \prg_return_true:
477     } {
478       \prg_return_false:
479     }
480   }
481 }

482 \cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
483   \BNVS_use_raw:nc {
484     \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
485   } { #1_#2:NnTF }
486 }
487 \BNVS_new_conditional_cv:ncn { tl } { if_eq } { T, F, TF }

```

6.3.7 Strings

_bnvs_str_if_eq:vvTF _bnvs_str_if_eq:vvTF {<core>} {<tl>} {<yes code>} {<no code>}

These are shortcuts to

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


```

488 \cs_new:Npn \BNVS_new_conditional_vv:cNn #1 #2 #3 {
489   \BNVS_new_conditional:cpnn { #1:vv } ##1 ##2 { #3 } {
490     \BNVS_tl_use:nv {
491       \BNVS_tl_use:Nv #2 { ##1 }
492     } { ##2 } {
493       \prg_return_true:
494     } {
495       \prg_return_false:
496     }
497   }
498 }

499 \cs_new:Npn \BNVS_new_conditional_vv:cn #1 {
500   \BNVS_use:nc {
501     \BNVS_new_conditional_vvnc:cNn { #1 }
502   } { #1:nnTF }
503 }

504 \cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
505   \BNVS_new_conditional:cpnn { #2:vn } ##1 ##2 { #4 } {
506     \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
507       \prg_return_true:
508     } {
509       \prg_return_false:
510     }
511   }
512 }

513 \cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
514   \BNVS_use_raw:nc {
515     \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
516   } { #1_#2:nnTF }
517 }
518 \BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }

519 \cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
520   \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
521     \BNVS_use:nvn {
522       \BNVS_use:Nvn #3 { ##1 } { #1 }
523     } { ##2 } { #1 } {
524       \prg_return_true:
525     } {
526       \prg_return_false:
527     }
528   }
529 }

530 \cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
531   \BNVS_use_raw:nc {
532     \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
533   } { #1_#2:nnTF }
534 }
535 \BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }

```

6.3.8 Sequences

| | |
|--|--|
| <code>__bnvs_seq_count:c</code> | <code>__bnvs_seq_new:c {<core>}</code> |
| <code>__bnvs_seq_clear:c</code> | <code>__bnvs_seq_count:c {<core>}</code> |
| <code>__bnvs_seq_set_eq:cc</code> | <code>__bnvs_seq_clear:c {<core>}</code> |
| <code>__bnvs_seq_gset_eq:cc</code> | <code>__bnvs_seq_set_eq:cc {<core₁>} {<core₂>}</code> |
| <code>__bnvs_seq_use:cn</code> | <code>__bnvs_seq_use:cn {<core>} {<separator>}</code> |
| <code>__bnvs_seq_item:cn</code> | <code>__bnvs_seq_item:cn {<core>} {<integer expression>}</code> |
| <code>__bnvs_seq_remove_all:cn</code> | <code>__bnvs_seq_remove_all:cn {<core>} {<t1>}</code> |
| <code>__bnvs_seq_put_left:cv</code> | <code>__bnvs_seq_put_right:cn {<seq core>} {<t1>}</code> |
| <code>__bnvs_seq_put_right:cn</code> | <code>__bnvs_seq_put_right:cv {<seq core>} {<t1 core>}</code> |
| <code>__bnvs_seq_put_right:cv</code> | <code>__bnvs_seq_set_split:cnn {<seq core>} {<t1>} {<separator>}</code> |
| <code>__bnvs_seq_set_split:cnn</code> | <code>__bnvs_seq_pop_left:cc {<core₁>} {<core₂>}</code> |
| <code>__bnvs_seq_set_split:(cnv cnx)</code> | |
| <code>__bnvs_seq_pop_left:cc</code> | |

These are shortcuts to

- `\seq_set_eq:cc {l__bnvs_<core1>_seq} {l__bnvs_<core2>_seq}`
- `\seq_count:c {l__bnvs_<core>_seq}`
- `\seq_use:cn {l__bnvs_<core>_seq}{<separator>}`
- `\seq_item:cn {l__bnvs_<core>_seq}{<integer expression>}`
- `\seq_remove_all:cn {l__bnvs_<core>_seq}{<t1>}`
- `__bnvs_seq_clear:c {l__bnvs_<core>_seq}`
- `\seq_put_right:cv {l__bnvs_<seq core>_seq} {l__bnvs_<t1 core>_t1}`
- `\seq_set_split:cnn{l__bnvs_<seq core>_seq}{l__bnvs_<t1 core>_t1}{<t1>}`

```

536 \BNVS_new_c:nc { seq } { count }
537 \BNVS_new_c:nc { seq } { clear }
538 \BNVS_new_cn:nc { seq } { use }
539 \BNVS_new_cn:nc { seq } { item }
540 \BNVS_new_cn:nc { seq } { remove_all }
541 \BNVS_new_cn:nc { seq } { map_inline }
542 \BNVS_new_cc:nc { seq } { set_eq }
543 \BNVS_new_cc:nc { seq } { gset_eq }
544 \BNVS_new_cv:ncn { seq } { put_left } { t1 }
545 \BNVS_new_cn:ncn { seq } { put_right } { t1 }
546 \BNVS_new_cv:ncn { seq } { put_right } { t1 }
547 \BNVS_new_cnn:nc { seq } { set_split }
548 \BNVS_new_cnv:nc { seq } { set_split }
549 \BNVS_new_cnx:nc { seq } { set_split }
550 \BNVS_new_cc:ncn { seq } { pop_left } { t1 }
551 \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>} {<tl core>} {<yes code>} {<no code>}
\__bnvs_seq_pop_left:ccTF
\__bnvs_seq_pop_right:ccTF

```

```

552 \cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
553   \BNVS_new_conditional:cpnn { #1_#2:cc } ##1 ##2 { #4 } {
554     \BNVS_use:ncncn {
555       \BNVS_use_raw:c { #1_#2:NNTF }
556     } { ##1 } { #1 } { ##2 } { #3 } {
557       \prg_return_true:
558     } {
559       \prg_return_false:
560     }
561   }
562 }
563 \BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
564 \BNVS_new_conditional_cc:ncnn
565   { seq } { get_right } { tl } { T, F, TF }
566 \BNVS_new_conditional_cc:ncnn
567   { seq } { pop_left } { tl } { T, F, TF }
568 \BNVS_new_conditional_cc:ncnn
569   { seq } { pop_right } { tl } { T, F, TF }

```

6.3.9 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>}

```

570 \BNVS_new_c:nc { int } { new }
571 \BNVS_new_c:nc { int } { use }
572 \BNVS_new_c:nc { int } { zero }
573 \BNVS_new_c:nc { int } { incr }
574 \BNVS_new_c:nc { int } { decr }
575 \BNVS_new_cn:nc { int } { set }
576 \BNVS_new_cv:ncn { int } { set } { int }

```

6.4 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. For example, we have aliases for `\group_begin:` and `\group_end:` to allow the display of supplemental informations while debugging. We also have some techniques for coverage as well as inline testing.

6.5 Local variables

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

`\l__bnvs_I_tl`: A frame identifier, contains a regex caught group.

`\l__bnvs_J_tl`: The last frame identifier

`\l__bnvs_K_tl`: ! as regex caught group, if non empty.

`\l__bnvs_S_tl`: A short name, a regex caught group.

`\l__bnvs_P_tl`: A path, a regex caught group.

`\l__bnvs_G_tl`: - as regex caught group, if non empty.

`\l__bnvs_N_tl`: The representation of an unsigned decimal integer.

`\l__bnvs_U_tl`: The representation of an unsigned decimal integer.

`\l__bnvs_R_tl`: The rhs for assignment.

`\l__bnvs_O_tl`: The rhs for post assignment.

```
577 \tl_new:N \l__bnvs_J_tl
578 \tl_new:N \l__bnvs_I_tl
579 \tl_new:N \l__bnvs_K_tl
580 \tl_new:N \l__bnvs_S_tl
581 \tl_new:N \l__bnvs_P_tl
582 \tl_new:N \l__bnvs_G_tl
583 \tl_new:N \l__bnvs_N_tl
584 \tl_new:N \l__bnvs_U_tl
585 \tl_new:N \l__bnvs_R_tl
586 \tl_new:N \l__bnvs_O_tl
587 \tl_new:N \l__bnvs_T_tl
588 \tl_new:N \l__bnvs_V_tl
589 \tl_new:N \l__bnvs_A_tl
590 \tl_new:N \l__bnvs_F_tl
591 \tl_new:N \l__bnvs_L_tl
592 \tl_new:N \l__bnvs_Z_tl
593 \tl_new:N \l__bnvs_a_tl
594 \tl_new:N \l__bnvs_z_tl
```

```

595 \tl_new:N \l__bnvs_l_tl
596 \tl_new:N \l__bnvs_r_tl
597 \tl_new:N \l__bnvs_n_tl
598 \tl_new:N \l__bnvs_ref_tl
599 \tl_new:N \l__bnvs_v_tl
600 \tl_new:N \l__bnvs_b_tl
601 \tl_new:N \l__bnvs_c_tl
602 \tl_new:N \l__bnvs_ans_tl
603 \tl_new:N \l__bnvs_base_tl
604 \tl_new:N \l__bnvs_group_tl
605 \tl_new:N \l__bnvs_scan_tl
606 \tl_new:N \l__bnvs_query_tl
607 \tl_new:N \l__bnvs_n_incr_tl
608 \tl_new:N \l__bnvs_incr_tl
609 \tl_new:N \l__bnvs_plus_tl
610 \tl_new:N \l__bnvs_rhs_tl
611 \tl_new:N \l__bnvs_post_tl
612 \tl_new:N \l__bnvs_suffix_tl
613 \tl_new:N \l__bnvs_index_tl
614 \int_new:N \g__bnvs_call_int
615 \int_new:N \l__bnvs_i_int
616 \seq_new:N \g__bnvs_def_seq
617 \seq_new:N \l__bnvs_a_seq
618 \seq_new:N \l__bnvs_b_seq
619 \seq_new:N \l__bnvs_ans_seq
620 \seq_new:N \l__bnvs_match_seq
621 \seq_new:N \l__bnvs_split_seq
622 \seq_new:N \l__bnvs_P_seq
623 \seq_new:N \l__bnvs_P_head_seq
624 \seq_new:N \l__bnvs_P_tail_seq
625 \seq_new:N \l__bnvs_query_seq
626 \bool_new:N \l__bnvs_parse_bool
627 \bool_set_false:N \l__bnvs_parse_bool
628 \bool_new:N \l__bnvs_deep_bool
629 \bool_set_false:N \l__bnvs_deep_bool
630 \bool_new:N \l__bnvs_no_range_bool
631 \bool_set_false:N \l__bnvs_no_range_bool

632 \BNVS_new:cpn { tl_clear_ans: } {
633   \__bnvs_tl_clear:c { ans }
634 }

635 \cs_new:Npn \BNVS_error_ans:x {
636   \__bnvs_tl_put_right:cn { ans } 0
637   \BNVS_error:x
638 }

```

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

```

639 \BNVS_new:cpn { set_true:c } #1 {
640   \exp_args:Nc \bool_set_true:N { \BNVS_l:cn { #1 } { bool } }
641 }

642 \BNVS_new:cpn { set_false:c } #1 {
643   \exp_args:Nc \bool_set_false:N { \BNVS_l:cn { #1 } { bool } }
644 }

```

6.6 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.)

```
645 \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.

```
646 \BNVS_new:cpn { greset_call: } {
647   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
648 }
```

`__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.

```
649 \BNVS_new_conditional:cpnn { if_call: } { T, F, TF } {
650   \int_gdecr:N \g__bnvs_call_int
651   \int_compare:nNnTF \g__bnvs_call_int > 0 {
652     \prg_return_true:
653   } {
654     \prg_return_false:
655   }
656 }
```

6.7 Overlay specification

6.7.1 Registration

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

`__bnvs_name:nnn` `__bnvs_name:nnn` `{\id}` `{\tag}` `{\key}`
`__bnvs_name:nn` `__bnvs_name:nn` `{\id}` `{\tag}`
`__bnvs_id_seq:n` `__bnvs_id_seq:n` `{\id}`

Create a unique name from the arguments.

```
657 \BNVS_new:cpn { name:nnn } #1 #2 #3 { \BNVS_g:cn { #1!#2/#3 } { t1 } }
658 \BNVS_new:cpn { name:nn } #1 #2 { __bnvs_#1!#2 }
659 \BNVS_new:cpn { id_seq:n } #1 { \BNVS_g:cn { #1! } { seq } }
```

`\g__bnvs_I_seq` List of registered identifiers.

(End of definition for \g__bnvs_I_seq.)

```
660 \seq_new:N \g__bnvs_I_seq
```

| | |
|------------------------------------|--|
| <code>__bnvs_register:nn</code> | <code>__bnvs_register:nn {<id>}{<tag>}</code> |
| <code>__bnvs_unregister:nn</code> | <code>__bnvs_unregister:nn {<id>}{<tag>}</code> |
| <code>__bnvs_unregister:n</code> | <code>__bnvs_unregister:n {<id>}</code> |
| <code>__bnvs_unregister:</code> | <code>__bnvs_unregister:</code> |

Register and unregister according to the arguments. The `<id>!``<tag>` combination must be registered on definition and unregistered on disposal.

| | |
|------------------------------------|---|
| <code>__bnvs_register:NNnn</code> | <code>__bnvs_register:NNnn <cs> <seq> {<id>}{<tag>}</code> |
|------------------------------------|---|

Registering a `<id>!``<tag>` combination is not straightforward. `<cs>` and `<seq>` are respectively the command and the sequence uniquely associated to this combination, through `__bnvs_name:nn` and `__bnvs_name:nnn`.

```

661 \seq_new:N \l__bnvs_register_seq
662 \BNVS_new:cpn { register:NNnn } #1 #2 #3 #4 {
663   \cs_if_exist:NF #1 {
664     \cs_gset:Npn #1 { }
665     \seq_if_exist:NTF #2 {
666       \__bnvs_seq_clear:c { register }
667       \cs_set:Npn \BNVS_register: {
668         \__bnvs_seq_put_right:cn { register } { #4 }
669         \cs_set:Npn \BNVS_register: { }
670       }
671       \cs_set:Npn \BNVS_register:w ##1 ##2 {
672         \str_compare:nNnTF { ##2 } < { #4 } {
673           \__bnvs_seq_put_right:cn { register } { ##2 }
674         } {
675           \BNVS_register:
676           \__bnvs_seq_put_right:cn { register } { ##2 }
677           \cs_set:Npn \BNVS_register:w ####1 ####2 {
678             \__bnvs_seq_put_right:cn { register } { ####2 }
679           }
680         }
681       }
682       \__bnvs_foreach_T:nNTF { #3 } \BNVS_register:w {
683         \BNVS_register:
684         \seq_gset_eq:NN #2 \l__bnvs_register_seq
685       } {
686         \BNVS_error:n { Unreachable/register:NNnn-id-#3 }
687       }
688     } {
689       \seq_new:N #2
690       \seq_gput_right:Nn #2 { #4 }
691       \__bnvs_seq_clear:c { register }
692       \cs_set:Npn \BNVS_register: {
693         \__bnvs_seq_put_right:cn { register } { #3 }
694         \cs_set:Npn \BNVS_register: { }
695       }
696       \cs_set:Npn \BNVS_register:w ##1 {
697         \str_compare:nNnTF { ##1 } < { #3 } {
698           \__bnvs_seq_put_right:cn { register } { ##1 }
699         } {
700           \BNVS_register:
701           \__bnvs_seq_put_right:cn { register } { ##1 }

```

```

702         \cs_set:Npn \BNVS_register:w #####1 {
703             \__bnvs_seq_put_right:cn { register } { #####1 }
704         }
705     }
706 }
707 \__bnvs_foreach_I:N \BNVS_register:w
708 \BNVS_register:
709 \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_register_seq
710 }
711 }
712 }

713 \BNVS_new:cpn { register:nn } #1 #2 {
714     \exp_args:Ncc \__bnvs_register:NNnn
715     { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
716     { #1 } { #2 }
717 }

```

__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, through [__bnvs_name:nn](#) and [__bnvs_name:nnn](#).

```

718 \seq_new:N \l__bnvs_unregister_seq
719 \BNVS_new:cpn { unregister:NNnn } #1 #2 #3 #4 {
720     \cs_if_exist:NT #1 {
721         \cs_undefine:N #1
722         \__bnvs_seq_clear:c { unregister }
723         \cs_set:Npn \BNVS_unregister_NNnn:n ##1 { ##1 }
724         \cs_set:Npn \BNVS_unregister_NNnn:w ##1 ##2 {
725             \str_compare:nNnTF { ##2 } < { #4 } {
726                 \__bnvs_seq_put_right:cn { unregister } { ##2 }
727                 \cs_set:Npn \BNVS_unregister_NNnn:n #####1 { }
728             } {
729                 \cs_set:Npn \BNVS_unregister_NNnn:w #####1 #####2 {
730                     \__bnvs_seq_put_right:cn { unregister } { #####2 }
731                     \cs_set:Npn \BNVS_unregister_NNnn:n #####1 { }
732                 }
733             }
734         }
735         \__bnvs_foreach_T:nNTF { #3 } \BNVS_unregister_NNnn:w {
736             \seq_gset_eq:NN #2 \l__bnvs_unregister_seq
737         } {
738             \BNVS_error:n { Unreachable / unregister:NNnn~#3!#4 }
739         }
740         \BNVS_unregister_NNnn:n {
741             \__bnvs_seq_clear:c { unregister }
742             \cs_set:Npn \BNVS_unregister_NNnn:w ##1 {
743                 \str_compare:nNnTF { ##1 } < { #3 } {
744                     \__bnvs_seq_put_right:cn { unregister } { ##1 }
745                 } {
746                     \cs_set:Npn \BNVS_unregister_NNnn:n #####1 {
747                         \__bnvs_seq_put_right:cn { unregister } { #####1 }
748                     }

```



```

749     }
750   }
751   \__bnvs_foreach_I:N \BNVS_unregister_NNnn:w
752   \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_unregister_seq
753   \cs_undefine:N #2
754 }
755 }
756 }

757 \BNVS_new:cpn { unregister:nn } #1 #2 {
758   \exp_args:Ncc \__bnvs_unregister:NNnn
759   { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
760   { #1 } { #2 }
761 }

```

__bnvs_if_registered:nnTF __bnvs_if_registered:nnTF {<id>} {<tag>} {<yes code>} {<no code>}

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

```

762 \BNVS_new_conditional:cpnn { if_registered:nn } #1 #2 { T, F, TF } {
763   \cs_if_exist:cTF { \__bnvs_name:nn { #1 } { #2 } } {
764     \prg_return_true:
765   } {
766     \prg_return_false:
767   }
768 }

```

| | |
|-------------------------|------------------------------------|
| __bnvs_foreach_I:N | __bnvs_foreach_I:N {<function:n>} |
| __bnvs_foreach_I:n | __bnvs_foreach_I:n {<code>} |
| __bnvs_foreach_break: | __bnvs_foreach:, |
| __bnvs_foreach_break:n | __bnvs_foreach_break:n {<code>} |

Execute the <function:n> or the <code> for each declared identifier. The break commands work like \seq_map_break: and \seq_map_break:n.

```

769 \BNVS_new:cpn { foreach_I:N } {
770   \seq_map_function:NN \g__bnvs_I_seq
771 }

772 \BNVS_new:cpn { foreach_I:n } {
773   \seq_map_inline:Nn \g__bnvs_I_seq
774 }

775 \cs_set_eq:NN \__bnvs_foreach_break: \seq_map_break:
776 \cs_set_eq:NN \__bnvs_foreach_break:n \seq_map_break:n

```

| | |
|------------------------|--|
| __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. Both are the arguments passed to <function:nn> or <code> (through ##1 and ##2).

```

777 \BNVS_new_conditional:cpnn { foreach_T:nN } #1 #2 { T, F, TF } {
778   \if_cs_exist:w \_bnvs_id_seq:n { #1 } \cs_end:
779   \seq_map_inline:cn { \_bnvs_id_seq:n { #1 } } { #2 { #1 } { ##1 } }
780   \prg_return_true:
781   \else:
782   \prg_return_false:
783   \fi
784 }

785 \BNVS_new_conditional:cpnn { foreach_T:nn } #1 #2 { T, F, TF } {
786   \if_cs_exist:w \_bnvs_id_seq:n { #1 } \cs_end:
787   \cs_set:Npn \BNVS_foreach_T_nn:nn ##1 ##2 { #2 }
788   \seq_map_inline:cn { \_bnvs_id_seq:n { #1 } }
789   { \BNVS_foreach_T_nn:nn { #1 } { ##1 } }
790   \prg_return_true:
791   \else:
792   \prg_return_false:
793   \fi
794 }

```

| | |
|--|---|
| $\backslash_bnvs_foreach_IT:N$ $\backslash_bnvs_foreach_IT:n$ | $\backslash_bnvs_foreach_IT:nNn$ $\langle before \rangle$ $\langle function:nn \rangle$ $\langle after \rangle$ $\backslash_bnvs_foreach_IT:N$ $\langle function:nn \rangle$ $\backslash_bnvs_foreach_IT:n$ $\{ \langle code \rangle \}$ |
|--|---|

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

```

795 \BNVS_new:cpn { foreach_IT:N } #1 {
796   \_bnvs_foreach_I:n {
797     \_bnvs_foreach_T:nNT { ##1 } #1 { }
798   }
799 }

800 \BNVS_new:cpn { foreach_IT:NT } #1 #2 {
801   \_bnvs_foreach_I:n {
802     \_bnvs_foreach_T:nNT { ##1 } #1 { #2 }
803   }
804 }

805 \BNVS_new:cpn { foreach_IT:n } #1 {
806   \cs_set:Npn \BNVS_foreach_IT_n:nn ##1 ##2 { #1 }
807   \_bnvs_foreach_I:n {
808     \_bnvs_foreach_T:nNT { ##1 } \BNVS_foreach_IT_n:nn { }
809   }
810 }

```

| | |
|--|--|
| $\backslash_bnvs_foreach_key:N$ $\backslash_bnvs_foreach_key:n$ $\backslash_bnvs_foreach_key_init:N$ $\backslash_bnvs_foreach_key_init:n$ $\backslash_bnvs_foreach_key_main:N$ $\backslash_bnvs_foreach_key_main:n$ $\backslash_bnvs_foreach_key_sub:N$ $\backslash_bnvs_foreach_key_sub:n$ | $\backslash_bnvs_foreach_key:N$ $\langle function:n \rangle$ $\backslash_bnvs_foreach_key:n$ $\{ \langle code \rangle \}$ $\backslash_bnvs_foreach_key_init:N$ $\langle function:n \rangle$ $\backslash_bnvs_foreach_key_init:n$ $\{ \langle code \rangle \}$ $\backslash_bnvs_foreach_key_main:N$ $\langle function:n \rangle$ $\backslash_bnvs_foreach_key_main:n$ $\{ \langle code \rangle \}$ $\backslash_bnvs_foreach_key_sub:N$ $\langle function:n \rangle$ $\backslash_bnvs_foreach_key_sub:n$ $\{ \langle code \rangle \}$ |
|--|--|

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

```

811 \BNVS_new:cpn { foreach_key_main:N } {
812   \tl_map_function:nN { VRAZL }
813 }

814 \BNVS_new:cpn { foreach_key_main:n } {
815   \tl_map_inline:nn { VRAZL }
816 }

817 \BNVS_new:cpn { foreach_key_init:N } {
818   \tl_map_function:nN { 0 }
819 }

820 \BNVS_new:cpn { foreach_key_init:n } {
821   \tl_map_inline:nn { 0 }
822 }

823 \BNVS_new:cpn { foreach_key_sub:N } {
824   \tl_map_function:nN { PNvnr }
825 }

826 \BNVS_new:cpn { foreach_key_sub:n } {
827   \tl_map_inline:nn { PN }
828 }

829 \BNVS_new:cpn { foreach_key:n } #1 {
830   \__bnvs_foreach_key_init:n { #1 }
831   \__bnvs_foreach_key_main:n { #1 }
832   \__bnvs_foreach_key_sub:n { #1 }
833 }

834 \BNVS_new:cpn { foreach_key:N } #1 {
835   \__bnvs_foreach_key_init:N #1
836   \__bnvs_foreach_key_main:N #1
837   \__bnvs_foreach_key_sub:N #1
838 }

```

6.7.2 Basic model functions

| | |
|---|--|
| <code>__bnvs_gset:nnnn</code> | <code>__bnvs_gset:nnnn {<id>} {<tag>} {<key>} {<spec>}</code> |
| <code>__bnvs_gset:(nnnv nnnx vnnn nvnn vvnn vvnv)</code> | |
| <code>__bnvs_gset:nnv</code> | |

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

```

839 \BNVS_new:cpn { gset:nnnn } #1 #2 #3 {
840   \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { #2 } {
841     \use_none:n
842   } {
843     \__bnvs_register:nn { #1 } { #2 }
844     \tl_gclear_new:c { g__bnvs_#1!#2/#3_tl }
845     \tl_gset:cn { g__bnvs_#1!#2/#3_tl }
846   }
847 }

```

```

848 \BNVS_new:cpn { gset:vnnn } {
849   \BNVS_t1_use:cv { gset:nnnn }
850 }

851 \BNVS_new:cpn { gset:nvnn } #1 {
852   \BNVS_t1_use:nv { \_bnvs_gset:nnnn { #1 } }
853 }

854 \BNVS_new:cpn { gset:vvnn } {
855   \BNVS_t1_use:Nvv \_bnvs_gset:nnnn
856 }

857 \BNVS_new:cpn { gset:nnnv } #1 #2 #3 {
858   \BNVS_t1_use:nv {
859     \_bnvs_gset:nnnn { #1 } { #2 } { #3 }
860   }
861 }

862 \BNVS_new:cpn { gset:nnv } #1 #2 #3 {
863   \BNVS_t1_use:nv {
864     \_bnvs_gset:nnnn { #1 } { #2 } { #3 }
865   } { #3 }
866 }

867 \BNVS_new:cpn { gset:vvnv } #1 #2 #3 {
868   \BNVS_t1_use:nv {
869     \_bnvs_gset:vvnn { #1 } { #2 } { #3 }
870   }
871 }

872 \cs_generate_variant:Nn \_bnvs_gset:nnnn { nnnx }

```

```

\_bnvs_gunset:nnn \_bnvs_gunset:nnn {<id>} {<tag>} {<key>}

```

```

\_bnvs_gunset:vvnn \_bnvs_gunset:nn {<id>} {<tag>}

```

```

\_bnvs_gunset:nn
\_bnvs_gunset:nv

```

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

```

873 \BNVS_new:cpn { gunset:nnn } #1 #2 #3 {
874   \cs_undefine:c { g\_bnvs_#1!#2/#3_t1 }
875 }

876 \BNVS_new:cpn { gunset:vvnn } {
877   \BNVS_t1_use:Nvv \_bnvs_gunset:nnn
878 }

879 \BNVS_new:cpn { gunset:nn } #1 #2 {

```

```

880 \__bnvs_if_registered:nnT { #1 } { #2 } {
881   \tl_map_inline:nn {
882     \__bnvs_foreach_key_main:n
883     \__bnvs_foreach_key_sub:n
884   } {
885     ##1 {
886       \__bnvs_gunset:nnn { #1 } { #2 } { ####1 }
887     }
888   }
889 }
890 }

891 \BNVS_new:cpn { gunset:nv } #1 {
892   \BNVS_tl_use:nv { \__bnvs_gunset:nn { #1 } }
893 }

894 \BNVS_new:cpn { gunset_deep:nn } #1 #2 {
895   \__bnvs_foreach_T:nnT { #1 } {
896     \tl_if_in:nnT { .. ##2 } { .. #2 . } {
897       \__bnvs_gunset:nn { #1 } { ##2 }
898     }
899   } { }
900   \__bnvs_gunset:nn { #1 } { #2 }
901 }

902 \BNVS_new:cpn { gunset_deep:nv } #1 {
903   \BNVS_tl_use:nv { \__bnvs_gunset:nn { #1 } }
904 }

905 \BNVS_new:cpn { gunset:vn } {
906   \BNVS_tl_use:cv { gunset:nn }
907 }

908 \BNVS_new:cpn { gunset:vv } {
909   \BNVS_tl_use:Nvv \__bnvs_gunset:nn
910 }

911 \BNVS_new:cpn { gunset_deep:vv } {
912   \BNVS_tl_use:Nvv \__bnvs_gunset_deep:nn
913 }

914 \seq_new:N \l__bnvs_gunset_n_seq
915 \BNVS_new:cpn { gunset:n } #1 {
916   \__bnvs_seq_clear:c { gunset_n }
917   \__bnvs_foreach_I:n {
918     \tl_if_eq:nnTF { ##1 } { #1 } {
919       \__bnvs_foreach_T:nnT { #1 } {
920         \__bnvs_gunset:nn { #1 } { ####1 }
921       } {}
922     } {
923       \__bnvs_seq_put_right:cn { gunset_n } { ##1 }
924     }
925   }
926   \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_gunset_n_seq
927 }

```

```

928 \BNVS_new:cpn { gunset: } {
929   \_bnvs_foreach_IT:N \_bnvs_gunset:nn
930 }

```

```

\_bnvs_gclear_init:nnTF \_bnvs_gclear:nn {<id>} {<tag>}
\_bnvs_gclear:nnTF \_bnvs_gclear:n {<id>}
\_bnvs_gclear:nvTF \_bnvs_gclear:
\_bnvs_gclear:TF \_bnvs_greset:nn {<id>} {<tag>}
\_bnvs_greset:nnTF \_bnvs_greset:n {<id>}
\_bnvs_greset:nvTF \_bnvs_greset:
\_bnvs_greset:TF

```

The first ones clear out every data stored for $\langle id \rangle! \langle tag \rangle$, including deeper and registration data. When $\langle tag \rangle$ is omitted, any known tag is considered. When $\langle id \rangle$ is omitted, any known id is considered. The second one only clears out the data created on the fly after the initialization step. The initial data (for key 0) is left untouched.

```

931 \BNVS_new:cpn { greset:nn } #1 #2 {
932   \_bnvs_foreach_T:nnT { #1 } {
933     \tl_if_in:nnT { .. ##2 } { .. #2 . } {
934       \tl_map_inline:nn {
935         \_bnvs_foreach_key_main:n
936         \_bnvs_foreach_key_sub:n
937       } {
938         #####1 {
939           \_bnvs_gunset:nnn { #1 } { ##2 } { #####1 }
940         }
941       }
942     }
943   } { }
944   \_bnvs_if_registered:nnT { #1 } { #2 } {
945     \tl_map_inline:nn {
946       \_bnvs_foreach_key_main:n
947       \_bnvs_foreach_key_sub:n
948     } {
949       ##1 {
950         \_bnvs_gunset:nnn { #1 } { #2 } { #####1 }
951       }
952     }
953   }
954 }

```

Clears out every data associated to $\langle id \rangle! \langle tag \rangle$, and $\langle id \rangle! \langle tag \rangle. \langle subtag \rangle$.

```

955 \seq_new:N \l__bnvs_gclear_nn_seq
956 \BNVS_new:cpn { gclear:nn } #1 #2 {
957   \_bnvs_seq_clear:c { gclear_nn }
958   \_bnvs_foreach_T:nnT { #1 } {
959     \tl_if_in:nnT { .. ##2 } { .. #2 . } {
960       \tl_map_inline:nn {
961         \_bnvs_foreach_key_init:n
962         \_bnvs_foreach_key_main:n
963         \_bnvs_foreach_key_sub:n
964       } {
965         #####1 {

```

```

966     \_bnvs_gunset:nnn { #1 } { ##2 } { #####1 }
967   }
968 }
969 \_bnvs_seq_put_right:cn { gclear_nn } {
970   \_bnvs_unregister:nn { #1 } { ##2 }
971 }
972 }
973 } { \_bnvs_seq_use:cn { gclear_nn } {} }
974 \_bnvs_if_registered:nnT { #1 } { #2 } {
975   \tl_map_inline:nn {
976     \_bnvs_foreach_key_init:n
977     \_bnvs_foreach_key_main:n
978     \_bnvs_foreach_key_sub:n
979   } {
980     ##1 {
981       \_bnvs_gunset:nnn { #1 } { #2 } { #####1 }
982     }
983   }
984   \_bnvs_unregister:nn { #1 } { #2 }
985 }
986 }

987 \BNVS_new:cpn { gclear:n } #1 {
988   \_bnvs_foreach_T:nNT { #1 } \_bnvs_gclear:nn {}
989 }

990 \BNVS_new:cpn { gclear: } {
991   \_bnvs_foreach_IT:N \_bnvs_gclear:nn
992 }

993 \BNVS_new:cpn { greset:n } #1 {
994   \_bnvs_foreach_T:nNT { #1 } \_bnvs_greset:nn {}
995 }

996 \BNVS_new:cpn { greset: } {
997   \_bnvs_foreach_IT:N \_bnvs_greset:nn
998 }

```

| | |
|--|--|
| _bnvs_is_gset:nnn <i>TF</i> | _bnvs_is_gset:nnnTF {<id>} {<tag>} {<key>} {<yes code>} {<no code>} |
| _bnvs_is_gset:(nvn vvv vxn) <i>TF</i> | _bnvs_is_gset:nnTF {<id>} {<tag>} {<yes code>} {<no code>} |
| _bnvs_is_gset:nn <i>TF</i> | _bnvs_if_spec:nnnTF {<id>} {<tag>} {<key>} {<yes code>} {<no code>} |
| _bnvs_if_spec:nnn <i>TF</i> | _bnvs_if_spec:nnTF {<id>} {<tag>} {<yes code>} {<no code>} |
| _bnvs_if_spec:nn <i>TF</i> | |

Test for the existence of a *<spec>* for that *<id>!**<tag>/<key>* combination. The version with no *<key>* is the or combination for keys V and R.

The *_spec:* variant is similar except that it uses *<id>!**<tag>/<key>* or *!**<tag>/<key>* combinations.

```

999 \BNVS_new_conditional:cpnn { is_gset:nnn } #1 #2 #3 { T, F, TF } {
1000   \if_cs_exist:w \_bnvs_name:nnn { #1 } { #2 } { #3 } \cs_end:
1001   \prg_return_true:
1002   \else:
1003     \prg_return_false:
1004   \fi:
1005 }

```

```

1006 \BNVS_new_conditional:cpnn { is_gset:vxv } #1 #2 #3 { T, F, TF } {
1007   \exp_args:NNnx \BNVS_tl_use:Nv \_bnvs_is_gset:nnnTF { #1 }
1008   { #2 } { #3 } {
1009     \prg_return_true:
1010   } {
1011     \prg_return_false:
1012   }
1013 }

1014 \BNVS_new_conditional:cpnn { is_gset:nvn } #1 #2 #3 { T, F, TF } {
1015   \BNVS_tl_use:nv { \_bnvs_is_gset:nnnTF { #1 } } { #2 } { #3 } {
1016     \prg_return_true:
1017   } {
1018     \prg_return_false:
1019   }
1020 }

1021 \BNVS_new_conditional:cpnn { is_gset:vvv } #1 #2 #3 { T, F, TF } {
1022   \BNVS_tl_use:Nvv \_bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
1023     \prg_return_true:
1024   } {
1025     \prg_return_false:
1026   }
1027 }

1028 \BNVS_new_conditional:cpnn { is_gset:nn } #1 #2 { T, F, TF } {
1029   \_bnvs_is_gset:nnnTF { #1 } { #2 } V {
1030     \prg_return_true:
1031   } {
1032     \_bnvs_is_gset:nnnTF { #1 } { #2 } A {
1033       \prg_return_true:
1034     } {
1035       \_bnvs_is_gset:nnnTF { #1 } { #2 } Z {
1036         \prg_return_true:
1037       } {
1038         \prg_return_false:
1039       }
1040     }
1041   }
1042 }

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

```



```

1056 }
1057 }

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

```

| | |
|---|--|
| <code>__bnvs_if_get:nnncTF</code> <code>__bnvs_if_get:nncTF</code> <code>__bnvs_if_get:vvcTF</code> <code>__bnvs_if_spec:nnncTF</code> | <code>__bnvs_if_get:nnncTF {<id>} {<tag>} {<key>} {<ans>} {<yes code>} {<no code>}</code> <code>__bnvs_if_spec:nnncTF {<id>} {<tag>} {<key>} {<ans>} {<yes code>} {<no code>}</code> <code>__bnvs_if_get:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no code>}</code> |
|---|--|

The `__bnvs_if_get:nnnc...` variant puts what was stored for `<id>!``<tag>/<key>` 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 `...spec...` variant is similar except that it uses what was stored for `<id>!``<tag>/<key>` or `!``<tag>/<key>` (with an empty `<id>`).

`__bnvs_if_get:nncTF` is a shortcut for `__bnvs_if_get:nnncTF` when `<key>` and `<ans>` are the same.

```

1073 \BNVS_new_conditional:cpnn { if_get:nnnc } #1 #2 #3 #4 { T, F, TF } {
1074   \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
1075     \tl_set_eq:cc { \BNVS_l:cn { #4 } { tl } } {
1076       \__bnvs_name:nnn { #1 } { #2 } { #3 }
1077     }
1078     \prg_return_true:
1079   } {
1080     \prg_return_false:
1081   }
1082 }

1083 \BNVS_new_conditional:cpnn { if_get:nnc } #1 #2 #3 { T, F, TF } {
1084   \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #3 } {
1085     \prg_return_true:
1086   } {
1087     \prg_return_false:
1088   }
1089 }

1090 \BNVS_new_conditional:cpnn { if_get:vvc } #1 #2 #3 { T, F, TF } {
1091   \BNVS_tl_use:Nvv \__bnvs_if_get:nncTF { #1 } { #2 } { #3 } {
1092     \prg_return_true:
1093   } {
1094     \prg_return_false:

```

```

1095 }
1096 }

```

```

__bnvs_if_resolved:nnncTF __bnvs_if_resolved:nnncTF {<id>} {<tag>} {<key>} {<ans>} {<yes code>} {<no code>}

```

Execute <yes code> if resolution succeeded for either <id>!<tag>/<key> or !<tag>/<key>.

```

1097 \BNVS_new_conditional:cpnn { if_resolved:nnnc } #1 #2 #3 #4 { T, F, TF } {
1098   __bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #4 } {
1099     __bnvs_if_unresolvable:cTF { #3 } {
1100       __bnvs_if_resolved:nnnc { #1 } { #2 } { #3 } { #4 } {
1101         \prg_return_true:
1102       } {
1103         \prg_return_false:
1104       }
1105     } {
1106       __bnvs_is_will_gset:cTF { #3 } {
1107         \prg_return_false:
1108       } {
1109         \prg_return_true:
1110       }
1111     }
1112   } {
1113     \tl_if_empty:nTF { #1 } {
1114       \prg_return_false:
1115     } {
1116       __bnvs_if_resolved:nnnc { #1 } { #2 } { #3 } { #4 } {
1117         \prg_return_true:
1118       } {
1119         \prg_return_false:
1120       }
1121     }
1122   }
1123 }

```

```

__bnvs_require:nnnc __bnvs_require:nnnc {<id>} {<tag>} {<key>} {<ans>}

```

```

__bnvs_require:nnnc __bnvs_require:nnnc {<id>} {<tag>} {<ans>}

```

__bnvs_require:vvc Calls `__bnvs_if_get:nnncTF`, does nothing on success and on failure, does nothing or raise when in debug mode. `__bnvs_require:nnnc` is a shortcut for the more qualified function `__bnvs_require:nnnc` when <key> and <ans> are the same.

```

1124 \BNVS_new:cpn { require:nnnc } #1 #2 #3 #4 {
1125   __bnvs_if_get:nnncF { #1 } { #2 } { #3 } { #4 } {
1126     \BNVS_fatal_unreachable:
1127   }
1128 }

1129 \BNVS_new_conditional:cpnn { if_spec:nnnc } #1 #2 #3 #4 { T, F, TF } {
1130   __bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #4 } {
1131     \prg_return_true:
1132   } {
1133     \tl_if_empty:nTF { #1 } {
1134       \prg_return_false:
1135     } {

```

```

1136     \__bnvs_if_get:nnncTF { } { #2 } { #3 } { #4 } {
1137         \prg_return_true:
1138     } {
1139         \prg_return_false:
1140     }
1141 }
1142 }
1143 }

```

6.7.3 The provide mode

```

1144 \bool_new:N \l__bnvs_provide_bool

1145 \BNVS_new:cpn { provide_on: } {
1146     \__bnvs_set_true:c { provide }
1147 }

1148 \BNVS_new:cpn { provide_off: } {
1149     \__bnvs_set_false:c { provide }
1150 }
1151 \__bnvs_provide_off:

```

| | |
|---|---|
| <code>__bnvs_is_provide_and_gset:nnTF</code> <code>__bnvs_is_provide_and_gset:nnnTF</code> <code>__bnvs_is_provide_and_gset:(nvn vvn)TF</code> | <code>__bnvs_is_provide_and_gset:nnnTF {<id>} {<tag>} {<key>} {<yes</code> <code>code>} {<no code>}</code> <code></code> |
|---|---|

Execute *<yes code>* when in provide mode and *<id>!**<tag>/<key>* is known, *<no code>* otherwise.

```

1152 \BNVS_new_conditional:cpnn { if_should_provide:nn } #1 #2 { T, F, TF } {
1153     \__bnvs_if:cTF { provide } {
1154         \if_cs_exist:w \__bnvs_name:nnn { #1 } { #2 } 0 \cs_end:
1155         \prg_return_false:
1156     } else:
1157         \prg_return_true:
1158     } \fi:
1159 } {
1160     \prg_return_true:
1161 }
1162 }

1163 \BNVS_new_conditional:cpnn { is_provide_and_gset:nnn }
1164     #1 #2 #3 { T, F, TF } {
1165     \__bnvs_if:cTF { provide } {
1166         \if_cs_exist:w \__bnvs_name:nnn { #1 } { #2 } { #3 } \cs_end:
1167         \prg_return_true:
1168     } else:
1169         \prg_return_false:
1170     } \fi:
1171 } {
1172     \prg_return_false:
1173 }
1174 }

```

```

1175 \BNVS_new_conditional:cpnn { is_provide_and_gset:nvn }
1176         #1 #2 #3 { T, F, TF } {
1177     \BNVS_t1_use:nv {
1178         \__bnvs_is_provide_and_gset:nnnTF { #1 }
1179     } { #2 } { #3 } {
1180         \prg_return_true:
1181     } {
1182         \prg_return_false:
1183     }
1184 }

1185 \BNVS_new_conditional:cpnn { is_provide_and_gset:vvv }
1186         #1 #2 #3 { T, F, TF } {
1187     \BNVS_t1_use:Nvv \__bnvs_is_provide_and_gset:nnnTF { #1 } { #2 } { #3 } {
1188         \prg_return_true:
1189     } {
1190         \prg_return_false:
1191     }
1192 }

```

| | |
|---|---|
| <pre> __bnvs_gprovide:TnnnnF __bnvs_gprovide:TvnnnF __bnvs_gprovide:TvvnnF </pre> | <pre> __bnvs_gprovide:TnnnnF {<pre code>} {<id>} {<tag>} {<key>} {<value>} {<no code>} </pre> <p>Execute <i><no code></i> exclusively when not in provide mode. Does nothing when something was set for the <i><id>!</i><i><tag>/<key></i> combination. Execute <i><pre code></i> before setting.</p> |
|---|---|

```

1193 \BNVS_new:cpn { gprovide:TnnnnF } #1 #2 #3 #4 #5 {
1194     \__bnvs_if:cTF { provide } {
1195         \__bnvs_is_provided:nnnF { #2 } { #3 } { #4 } {
1196             #1
1197         \__bnvs_gset:nnnn { #2 } { #3 } { #4 } { #5 }
1198     }
1199 }
1200 }

1201 \BNVS_new:cpn { gprovide:TvnnnF } #1 {
1202     \BNVS_t1_use:nv { \__bnvs_gprovide:TnnnnF { #1 } }
1203 }
1204 \BNVS_new:cpn { gprovide:TvvnnF } #1 {
1205     \BNVS_t1_use:nvv { \__bnvs_gprovide:TnnnnF { #1 } }
1206 }

```

6.7.4 Initialize

| | |
|--|--|
| <pre> __bnvs_ginit:nnn __bnvs_ginit:nnv __bnvs_ginit:nnnn </pre> | <pre> __bnvs_ginit:nnn {<id>} {<tag>} {<definition>} __bnvs_ginit:nnv {<id>} {<tag>} {<index>} {<definition>} __bnvs_ginit:nnnn {<id>} {<tag>} {<definition>} </pre> |
|--|--|

These functions support provide mode. Somehow a shortcut to `__bnvs_gset:nnnn` with key 0. `__bnvs_ginit:nnnn` calls `__bnvs_ginit:nnn` with the same id, the tag *<id>!**<tag>.<index>* and *<spec>*. When *<index>* is provided, if *<id>!**<tag>/0* is not set, it is initialized from *<spec>* shifted appropriately.

```

1207 \BNVS_new:cpn { ginit:nnn } #1 #2 #3 {

```

```

1208 \__bnvs_is_provide_and_gset:nnnF { #1 } { #2 } 0 {
1209   \__bnvs_gclear:nn { #1 } { #2 }
1210   \__bnvs_gset:nnnn { #1 } { #2 } 0 { #3 }
1211 }
1212 }

1213 \BNVS_new:cpn { ginit:nnv } #1 #2 {
1214   \BNVS_tl_use:nv { \__bnvs_ginit:nnn { #1 } { #2 } }
1215 }

1216 \BNVS_new:cpn { ginit:nnnn } #1 #2 #3 #4 {
1217   \__bnvs_is_gset:nnnT { #1 } { #2 } 0 {
1218     \__bnvs_gclear:nn { #1 } { #2 }
1219   }
1220   \__bnvs_is_provide_and_gset:nnnF { #1 } { #2.#3 } 0 {
1221     \__bnvs_gclear:nn { #1 } { #2 }
1222     \__bnvs_gset:nnnn { #1 } { #2.#3 } 0 { #4 }
1223   }
1224   \__bnvs_is_gset:nnnF { #1 } { #2 } 0 {
1225     \__bnvs_gset:nnnx { #1 } { #2 } 0 {
1226       \exp_not:n { #4 - } \int_eval:n { #3 - 1 }
1227     }
1228   }
1229 }

```

__bnvs_will_gset:nnn __bnvs_will_gset:nnn {<id>} {<tag>} {<key>}

To manage circular dependencies. After this command, __bnvs_is_will_gset:nnnTF is false for the same arguments.

```

1230 \BNVS_new:cpn { will_gset:nnn } #1 #2 #3 {
1231   \__bnvs_gset:nnnn { #1 } { #2 } { #3 } { \q_no_value }
1232 }

```

__bnvs_is_will_gset:cTF __bnvs_is_will_gset:cTF {<in>} {<yes code>} {<no code>}

__bnvs_if_will_get:nnncTF

To manage circular dependencies. See __bnvs_will_gset:nnn above.

```

1 \__bnvs_will_gset:nnn {<id>} {<tag>} {<key>}
2 ...
3 \__bnvs_if_get:nnncT {<id>} {<tag>} {<key>} {<in>} {
4   ...
5   \__bnvs_is_will_gset:cTF {<in>} {
6     {<yes code>} } { {<no code>} }
7   ...
8 }

```

```

1233 \BNVS_new_conditional:cpnn { is_will_gset:c } #1 { T, F, TF } {
1234   \BNVS_tl_use:Nv \quark_if_no_value:nTF { #1 } {
1235     \prg_return_true:
1236   } {
1237     \prg_return_false:
1238   }
1239 }

```

6.8 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:nn {\langle id \rangle} {\langle tag \rangle}
\__bnvs_V_gunset:n \__bnvs_V_gunset:n {\langle id \rangle}
\__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 \LaTeX 3 eponym functions.

```

1240 \BNVS_new:cpn { V_gunset: } {
1241   \__bnvs_foreach_IT:n {
1242     \__bnvs_gunset:nnn { ##1 } { ##2 } V
1243   }
1244 }

```

6.8.1 Unresolvable

If resolution has failed for $\langle id \rangle! \langle tag \rangle / \langle key \rangle$, it is marked unresolvable in order to save some computation time.

```

\__bnvs_gset_unresolvable:nnn

```

See below for the usage.

```

1245 \BNVS_new:cpn { gset_unresolvable:nnn } #1 #2 #3 {
1246   \__bnvs_gset:nnnn { #1 } { #2 } { #3 } { \q_nil }
1247 }

```

```

\__bnvs_if_unresolvable:cTF

```

```

1 \__bnvs_gset_unresolvable:nnn {\langle id \rangle} {\langle tag \rangle} {\langle key \rangle}
2 ...
3 \__bnvs_if_get:nnncT {\langle id \rangle} {\langle tag \rangle} {\langle key \rangle} {\langle in \rangle} {
4   ...
5   \__bnvs_if_unresolvable:cTF {\langle in \rangle} {
6     {\langle yes code \rangle} } {\langle no code \rangle} }
7   ...
8 }

```

```

1248 \BNVS_new_conditional:cpnn { if_unresolvable:c } #1 { T, F, TF } {
1249   \BNVS_tl_use:nc { \exp_args:NV \quark_if_nil:nTF } { #1 } {
1250     \prg_return_true:
1251   } {
1252     \prg_return_false:
1253   }
1254 }

```

`__bnvs_is_provided:nnnTF` `__bnvs_is_provided:nnnTF {<id>}{<tag>}{<key>}{<yes code>}{<no code>}`

If `<id>!``<tag>/``<key>` has been successfully resolved and not unset since then, execute `<yes code>`. Otherwise execute `<no code>`.

```

1255 \tl_new:N \l__bnvs_is_provided_tl
1256 \BNVS_new_conditional:cpnn { is_provided:nnn } #1 #2 #3 { T, F, TF } {
1257   \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { is_provided } {
1258     \__bnvs_if_unresolvable:cTF { is_provided } {
1259       \prg_return_false:
1260     } {
1261       \__bnvs_is_will_gset:cTF { is_provided } {
1262         \prg_return_false:
1263       } {
1264         \prg_return_true:
1265       }
1266     }
1267   } {
1268     \prg_return_false:
1269   }
1270 }

```

6.9 Regular expressions

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

```

1271 \regex_const:Nn \c__bnvs_S_regex {
1272   [[[:alpha:]]_][[:alnum:]]_*
1273 }

```

(End of definition for `\c__bnvs_S_regex`.)

`\c__bnvs_P_regex` A possibly empty list of `.<short name>` items representing a path.

```

1274 \regex_const:Nn \c__bnvs_P_regex {
1275   (?: \. \ur{c__bnvs_S_regex} ) *
1276 }

```

(End of definition for `\c__bnvs_P_regex`.)

`\c__bnvs_A_GN_Z_regex`

(End of definition for `\c__bnvs_A_GN_Z_regex`.)

```

1277 \regex_const:Nn \c__bnvs_A_GN_Z_regex {
1278   \A (?: \. (?: [+ ] | ( [- ] ) ) ? 0*? ( [1-9] \d* | 0 ) ) \Z
1279 }
1280

```

`\c__bnvs_A_index_Z_regex`

(End of definition for `\c__bnvs_A_index_Z_regex`.)

```

1281 \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.)

```
1282 \regex_const:Nn \c__bnvs_A_reserved_Z_regex {  
1283   \A_*[a-z][_a-z0-9]*\Z  
1284 }
```

`\c__bnvs_A_XP_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_XP_Z_regex.)

```
1285 \regex_const:Nn \c__bnvs_A_XP_Z_regex {  
  
  1: the frame  $\langle id \rangle$   
  2: the exclamation mark  
  
1286   \A (?: ( \ur{c__bnvs_S_regex} )? ( ! ) )?  
  
  3: The short name.  
  
1287   ( \ur{c__bnvs_S_regex} )  
  
  4: the path, if any.  
  
1288   ( \ur{c__bnvs_P_regex} ) \Z  
1289 }
```

`\c__bnvs_A_XP_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_XP_Z_regex.)

```
1290 \regex_const:Nn \c__bnvs_A_XPGN_Z_regex {  
  
  1: the frame  $\langle id \rangle$   
  2: the exclamation mark  
  
1291   \A (?: ( \ur{c__bnvs_S_regex} )? ( ! ) )?  
  
  3: The short name.  
  
1292   ( \ur{c__bnvs_S_regex} )  
  
  4: the path, if any.  
  
1293   ( \ur{c__bnvs_P_regex} )  
  
  5: The optional - sign  
  6: and unsigned value of the last optional  $\langle index \rangle$  component, no leading 0 except  
for 0 itself.  
  
1294   (?: \. (?: [+ ] | ( [- ] ) )? 0*? ( [1-9] \d* | 0 ) )? \Z  
1295 }
```

`\c__bnvs_A_XP_Z_regex`

Matches the whole string.

(End of definition for `\c__bnvs_A_XP_Z_regex`.)

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

(End of definition for `\c__bnvs_A_IKTGN_Z_regex`.)

1: The full match,

```
1296      \regex_const:Nn \c__bnvs_A_IKTGN_Z_regex {
```

2,3: the optional frame $\langle id \rangle$ and !

```
1297      \A {?: ( \ur{c__bnvs_S_regex} )? (!) }?
```

4: The tag (short name and dotted path)

```
1298      ( \ur{c__bnvs_S_regex} \ur{c__bnvs_P_regex} ? )
```

5: The sign

6: and unsigned value of the last optional $\langle index \rangle$ component, no leading 0 except for 0 itself.

```
1299      (?: \. (?: [+ ] | ( [- ] ) )? 0*? ( [1-9] \d* | 0 ) )? \Z
1300    }
```

`\c__bnvs_A_SP_Z_regex` Matches the whole string.

(End of definition for `\c__bnvs_A_SP_Z_regex`.)

```
1301 \regex_const:Nn \c__bnvs_A_SP_Z_regex {
```

1: The full match,

2: the frame $\langle id \rangle$

```
1302      \A ( \ur{c__bnvs_S_regex} | [-+]? \d+ )
```

3: The dotted path.

```
1303      ( (?: \. \ur{c__bnvs_S_regex} | \. [-+]? \d+ )* ) \Z
1304    }
```

`\c__bnvs_A_PGN_Z_regex` Matches the whole string.

(End of definition for `\c__bnvs_A_PGN_Z_regex`.)

```
1305 \regex_const:Nn \c__bnvs_A_PGN_Z_regex {
```

1: The full match,

2: the path

1306 \A (\ur{c__bnvs_S_regex} \ur{c__bnvs_P_regex})

3: The sign

3: abd the value of the trailing digits component.

1307 (?: \. (?: [+] | ([-]))? 0*? ([1-9] \d* | 0))? \Z
1308 }

\c__bnvs_A_P_Z_regex Matches the whole string.

(End of definition for \c__bnvs_A_P_Z_regex.)

1309 \regex_const:Nn \c__bnvs_A_P_Z_regex {
1310 \A \ur{c__bnvs_S_regex} \ur{c__bnvs_P_regex} \Z
1311 }

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

1312 \regex_const:Nn \c__bnvs_colons_regex { :(:+)? }

(End of definition for \c__bnvs_colons_regex.)

\c__bnvs_XPXPGNURO_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.

1313 \regex_const:Nn \c__bnvs_XPXPGNURO_regex {
1314 \s* (? :

We start with ‘++’ instrussions⁵.

1 incrementation prefix

1315 \+\+

1.1: optional identifier: optional frame <id>

1.2: followed by required !

1316 (?: (\ur{c__bnvs_S_regex})? (!))?

1.3: <short name>

1317 (\ur{c__bnvs_S_regex})

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

1318 (\ur{c__bnvs_P_regex})

2: without incement prefix

2.1: optional frame <id> followed by

2.2: required !

1319 | (?: (\ur{c__bnvs_S_regex})? (!))?

2.3: <short name>

1320 (\ur{c__bnvs_S_regex})

2.4: optionally followed by a dotted path

```
1321      ( \ur{c__bnvs_P_regex} )
```

2.5: The sign

2.6: and unsigned value of the last optional $\langle index \rangle$ component, no leading 0 except for 0 itself.

```
1322      (?: \. (?: [+ ] | ( [- ] ) )? 0*? ( [1-9] \d* | 0 ) )?
```

We continue with other expressions

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

2.8: the poor man integer expression after ‘+=’, which is the longest sequence of black characters, which ends just before a space or at the very last character. This tricky definition allows quite any algebraic expression, even those involving parenthesis.

```
1323      (?: \s* (\+?)= \s* ( \S+ )
```

2.9: the post increment

```
1324      | (\+)\+
1325      )?
1326      ) \s*
1327      }
```

(End of definition for `\c__bnvs_XPXPGNURO_regex`.)

`\c__bnvs_A_VAZLLZZL_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_VAZLLZZL_Z_regex`.)

```
1328 \regex_const:Nn \c__bnvs_A_VAZLLZZL_Z_regex {
1329   \A \s* (?:
```

- 2 \rightarrow V

```
1330      ( [^:]+\+? )
```

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

```
1331      | (?: ( [^:]+\+? ) \s* : (?: \s* ( [^:]*? ) | : \s* ( [^:]*? ) ) ) )
```

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

```
1332      | (?: :: \s* (?: ( [^:]+\+? ) \s* : \s* ( [^:]+\+? ) )? )
```

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

```
1333      | (?: : \s* (?: ( [^:]+\+? ) \s* :: \s* ( [^:]*? ) )? )
```

```
1334      )
```

```
1335      \s* \Z
```

```
1336      }
```

⁵At the same time an instruction and an expression... this is a synonym of expression

6.10 End group setter

```
1337 \cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
1338   \BNVS_tl_use:nv {
1339     \BNVS_end:
1340     \__bnvs_tl_put_right:cn { #1 }
1341   } { #2 }
1342 }
1343 \cs_new:Npn \BNVS_end_tl_gset:nnnv #1 #2 #3 {
1344   \BNVS_tl_use:nv {
1345     \BNVS_end:
1346     \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
1347   }
1348 }
1349 \cs_new:Npn \BNVS_end_tl_set:cv #1 {
1350   \BNVS_tl_use:nv {
1351     \BNVS_end: \__bnvs_tl_set:cn { #1 }
1352   }
1353 }
1354 \cs_new:Npn \BNVS_end_int_set:cv #1 {
1355   \BNVS_int_use:nv {
1356     \BNVS_end: \__bnvs_int_set:cn { #1 }
1357   }
1358 }
```

6.11 beamer.cls interface

Work in progress.

```
1359 \RequirePackage{keyval}
1360 \define@key{beamerframe}{beanoves-id}[]{}
1361 \tl_set:Nx \l__bnvs_J_tl { #1 }
1362 }
1363 \bool_new:N \l__bnvs_in_frame_bool
1364 \bool_set_false:N \l__bnvs_in_frame_bool
1365 \AddToHook{env/beamer@frameslide/before}{
1366   \__bnvs_greset_call:
1367   \__bnvs_V_gunset:
1368   \__bnvs_set_true:c { in_frame }
1369 }
1370 \AddToHook{env/beamer@frameslide/after}{
1371   \__bnvs_set_false:c { in_frame }
1372 }
```

6.12 Utilities

6.12.1 Split utilities

```

__bnvs_if_regex_split:cncTF {<yes code>} {<no code>}
__bnvs_if_regex_split:cnTF  \__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_split:NnNTF with the split sequence as last N argument

```

1373 \BNVS_new_conditional:cpnn { if_regex_split:cnc } #1 #2 #3 { T, F, TF } {
1374   \BNVS_seq_use:nc {
1375     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
1376   } { #3 } {
1377     \prg_return_true:
1378   } {
1379     \prg_return_false:
1380   }
1381 }

1382 \BNVS_new_conditional:cpnn { if_regex_split:cn } #1 #2 { T, F, TF } {
1383   \BNVS_seq_use:nc {
1384     \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
1385   } { split } {
1386     \prg_return_true:
1387   } {
1388     \prg_return_false:
1389   }
1390 }

1391 \BNVS_new_conditional:cpnn { split_if_pop_left:c } #1 { T, F, TF } {
1392   \__bnvs_seq_pop_left:ccTF { split } { #1 } {
1393     \prg_return_true:
1394   } {
1395     \prg_return_false:
1396   }
1397 }

1398 \cs_set_eq:NN \BNVS_split_F:n \use_none:n

1399 \BNVS_new:cpn { split_if_pop_left:cTn } #1 #2 #3 {
1400   \__bnvs_split_if_pop_left:cTF { #1 } { #2 } { \BNVS_split_F:n { #3 } }
1401 }

1402 \BNVS_new:cpn { split_pop_XPGNURO:T } #1 {

```

```

1403 \__bnvs_split_if_pop_left:cT a {
1404   \__bnvs_split_if_pop_left:cT a {
1405     \__bnvs_split_if_pop_left:cT a {
1406       \__bnvs_split_if_pop_left:cT a {
1407         \__bnvs_split_if_pop_left:cT G {
1408           \__bnvs_split_if_pop_left:cT N {
1409             \__bnvs_split_if_pop_left:cT U {
1410               \__bnvs_split_if_pop_left:cT R {
1411                 \__bnvs_split_if_pop_left:cT O {
1412                   #1
1413                 }
1414               }
1415             }
1416           }
1417         }
1418       }
1419     }
1420   }
1421 }
1422 }

1423 \BNVS_new:cpn { split_if_pop_GNURO:T } #1 {
1424   \__bnvs_split_if_pop_left:cT G {
1425     \__bnvs_split_if_pop_left:cT N {
1426       \__bnvs_split_if_pop_left:cT U {
1427         \__bnvs_split_if_pop_left:cT R {
1428           \__bnvs_split_if_pop_left:cT O {
1429             #1
1430           }
1431         }
1432       }
1433     }
1434   }
1435 }

```

6.12.2 Match utilities

```

\__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)}

```

```

{(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

```

1436 \BNVS_new_conditional:cpnn { if_extract_once:Ncn } #1 #2 #3 { T, F, TF } {
1437   \BNVS_use:ncn {
1438     \regex_extract_once:NnNTF #1 { #3 }
1439   } { #2 } { seq } {
1440     \prg_return_true:
1441   } {
1442     \prg_return_false:
1443   }
1444 }

```

\l__bnvs_match_seq Local storage for the match result.

(End of definition for \l__bnvs_match_seq.)

```

1445 \BNVS_new_conditional:cpnn { match_if_once:Nn } #1 #2 { T, F, TF } {
1446   \BNVS_use:ncn {
1447     \regex_extract_once:NnNTF #1 { #2 }
1448   } { match } { seq } {
1449     \prg_return_true:
1450   } {
1451     \prg_return_false:
1452   }
1453 }

```

```

1454 \BNVS_new_conditional:cpnn { if_extract_once:Ncv } #1 #2 #3 { T, F, TF } {
1455   \BNVS_seq_use:nc {
1456     \BNVS_tl_use:nv {
1457       \regex_extract_once:NnNTF #1
1458     } { #3 }
1459   } { #2 } {
1460     \prg_return_true:
1461   } {
1462     \prg_return_false:
1463   }
1464 }

```

```

1465 \BNVS_new_conditional:cpnn { match_if_once:Nv } #1 #2 { T, F, TF } {
1466   \BNVS_seq_use:nc {
1467     \BNVS_tl_use:nv {
1468       \regex_extract_once:NnNTF #1
1469     } { #2 }
1470   } { match } {
1471     \prg_return_true:
1472   } {
1473     \prg_return_false:
1474   }
1475 }

```

```

1476 \BNVS_new_conditional:cpnn { match_if_once:nn } #1 #2 { T, F, TF } {
1477   \BNVS_seq_use:nc {
1478     \regex_extract_once:nnNTF { #1 } { #2 }
1479   } { match } {
1480     \prg_return_true:
1481   } {
1482     \prg_return_false:

```

```

1483 }
1484 }

1485 \BNVS_new_conditional:cpnn { match_if_pop_left:c } #1 { T, F, TF } {
1486   \BNVS_tl_use:nc {
1487     \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
1488   } { #1 } {
1489     \prg_return_true:
1490   } {
1491     \prg_return_false:
1492   }
1493 }

1494 \cs_set_eq:NN \BNVS_match_F:n \use_none:n

1495 \BNVS_new:cpn { match_if_pop_left:cTn } #1 #2 #3 {
1496   \__bnvs_match_if_pop_left:cTF { #1 } { #2 } { \BNVS_match_F:n { #3 } }
1497 }

```

```

\__bnvs_if_ISP:nTF \__bnvs_if_ISP:nTF {<name>} {<yes code>} {<no code>}
\__bnvs_if_ISP:nnTF {<root>} {<relative>} {<yes code>} {<no code>}

```

If $\langle name \rangle$ is a reference, put the frame id it defines, or the current frame id, into I the short name into S, the dotted path into P, then execute $\langle yes\ code \rangle$. Otherwise execute $\{ \langle no\ code \rangle \}$.

The second version calls the first one with $\langle name \rangle$ equals $\langle relative \rangle$ prepended with $\langle root \rangle$.

The third version accepts integers as $\langle relative \rangle$ argument. It assumes that $\langle id \rangle$, $\langle short \rangle$ and $\langle path \rangle$ are already set. The $\langle path \rangle$ and $\langle tag \rangle$ are updated accordingly

```

1498 \BNVS_new_conditional:cpnn { if_ISP:n } #1 { T, F, TF } {
1499   \BNVS_begin:
1500   \__bnvs_match_if_once:NnTF \c__bnvs_A_XP_Z_regex { #1 } {
1501     \__bnvs_match_if_pop_left:cT I {
1502       \__bnvs_match_if_pop_left:cT I {
1503         \__bnvs_match_if_pop_left:cT K {
1504           \__bnvs_match_if_pop_left:cT S {
1505             \__bnvs_match_if_pop_left:cT P {
1506               \cs_set:Npn \BNVS_if_ISP:nnn ##1 ##2 ##3 {
1507                 \BNVS_end:
1508                 \__bnvs_tl_set:cn I { ##1 }
1509                 \__bnvs_tl_set:cn S { ##2 }
1510                 \__bnvs_tl_set:cn P { ##3 }
1511               }
1512               \__bnvs_tl_if_empty:cTF K {
1513                 \BNVS_tl_use:Nvvv \BNVS_if_ISP:nnn J
1514               } {
1515                 \BNVS_tl_use:Nvvv \BNVS_if_ISP:nnn I
1516               } S P
1517               \__bnvs_tl_set:cv T P
1518               \__bnvs_tl_put_left:cv T S
1519               \__bnvs_tl_set:cv J I

```



```

1520         \prg_return_true:
1521     }
1522 }
1523 }
1524 }
1525 }
1526 } {
1527     \BNVS_end:
1528     \prg_return_false:
1529 }
1530 }

```

6.12.3 Utilities

```

1531 \BNVS_new:cpn { warn_until_s_stop:w } #1 \s_stop {
1532     \tl_if_empty:nF { #1 } {
1533         \BNVS_warning:n { Ignored:~#1 }
1534     }
1535 }
1536 \BNVS_new:cpn { scan_until_s_stop:w } {
1537     \peek_meaning:NTF \s_stop {
1538         \use_none:n
1539     } {
1540         \__bnvs_warn_until_s_stop:w
1541     }
1542 }

```

```

\__bnvs_peek_branch_until_s_stop:nnnnw \__bnvs_peek_branch_until_s_stop:nnnnw
{<square:n>} {<curly:n>} {<non empty:n>} {<empty:>}
<...> \s_stop

```

Branch according to the following token. Each argument, except `<empty:>`, is the body of a function that takes one argument. It catches errors like `foo=[bar]baz`, where `baz` is not expected.

```

1543 \BNVS_new:cpn { peek_branch_until_s_stop:nnnnw } #1 #2 #3 #4 {
1544     \peek_meaning:NTF \BNVS_square:n {
1545         \cs_set:Npn \BNVS_peek_branch:w ##1 ##2 {
1546             \cs_set:Npn \BNVS_peek_branch:w #####1 {
1547                 #1
1548             }
1549             \BNVS_peek_branch:w { ##2 }
1550             \__bnvs_warn_until_s_stop:w
1551         }
1552         \BNVS_peek_branch:w
1553     } {
1554         \peek_meaning:NTF \BNVS_curly:n {
1555             \cs_set:Npn \BNVS_peek_branch:w ##1 ##2 {
1556                 \cs_set:Npn \BNVS_peek_branch:w #####1 {
1557                     #2
1558                 }
1559                 \BNVS_peek_branch:w { ##2 }
1560                 \__bnvs_warn_until_s_stop:w
1561             }

```

```

1562     \BNVS_peek_branch:w
1563   } {
1564     \peek_meaning:NTF \s_stop {
1565       #4
1566       \use_none:n
1567     } {
1568       \cs_set:Npn \BNVS_peek_branch:w ##1 \s_stop {
1569         #3
1570       }
1571       \BNVS_peek_branch:w
1572     }
1573   }
1574 }
1575 }

```

__bnvs_keyval:ncc __bnvs_keyval:ncc {<definitions>} {<V>} {<a>}

Parse definitions into the named `tl` variables (which should be different). Outer braces in definitions are kept. It is meant to be called within a group. On return `<V>` contains the leading standalone value, if any, `<a>` is a list of `\BNVS:nn{<key>}{<value>}` and `\BNVS:n{<key or value>}`.

```

1576 \BNVS_new:cpn { keyval:ncc } #1 #2 #3 {
1577   \cs_set:Npn \BNVS: {
1578     \cs_set:Npn \BNVS:n #####1 {
1579       \__bnvs_tl_put_right:cn { #3 } { \BNVS:n { #####1 } }
1580     }
1581     \cs_set:Npn \BNVS:nn #####1 #####2 {
1582       \__bnvs_tl_put_right:cn { #3 } { \BNVS:nn { #####1 } { #####2 } }
1583     }
1584   }
1585   \cs_set:Npn \BNVS:n ##1 {
1586     \BNVS:
1587     \__bnvs_tl_set:cn { #2 } { ##1 }
1588   }
1589   \cs_set:Npn \BNVS:nn {
1590     \BNVS:
1591     \BNVS:nn
1592   }
1593   \__bnvs_tl_set:cn { #2 } { #1 }
1594   \BNVS_tl_use:nc {
1595     \regex_replace_all:nnN { \cB. } { \c{BNVS:} \0 }
1596   } { #2 }
1597   \__bnvs_tl_clear:c { #3 }
1598   \BNVS_tl_use:nv {
1599     \__bnvs_tl_clear:c { #2 }
1600     \keyval_parse:NNn \BNVS:n \BNVS:nn
1601   } { #2 }
1602   \BNVS_tl_use:nc {
1603     \regex_replace_all:nnN { \c{BNVS:} } { }
1604   } { #2 }
1605   \BNVS_tl_use:nc {
1606     \regex_replace_all:nnN { \c{BNVS:} } { }
1607   } { #3 }

```

```

1608 }

1609 \BNVS_new:cpn { normalize_GN: } {
1610   \__bnvs_tl_if_eq:cnF N 0 {
1611     \__bnvs_tl_if_empty:cF G {
1612       \__bnvs_tl_put_left:cn N { - }
1613     }
1614   }
1615 }

```

6.12.4 Next index

```

1616 \BNVS_int_new:c { next_index }
1617 \BNVS_new:cpn { next_index:nn } #1 #2 {
1618   \__bnvs_int_incr:c { next_index }
1619   \__bnvs_tl_set:cn a { #2 . }
1620   \BNVS_int_use:nv { \__bnvs_tl_put_right:cn a } { next_index }
1621   \__bnvs_is_gset:nvNT { #1 } a 0 {
1622     \__bnvs_next_index:nn { #1 } { #2 }
1623   }
1624 }

```

```

\__bnvs_next_index:nnn \__bnvs_next_index:nnn {<id>} {<tag>} {<code:n>}

```

`<code:n>` takes one argument: the next available tag.

```

1625 \BNVS_new:cpn { next_index:nnn } #1 #2 #3 {
1626   \BNVS_begin:
1627     \__bnvs_int_zero:c { next_index }
1628     \__bnvs_next_index:nn { #1 } { #2 }
1629     \cs_set:Npn \BNVS:n ##1 {
1630       \BNVS_end:
1631       \__bnvs_int_set:cn { next_index } { ##1 }
1632       #3
1633     }
1634     \BNVS_tl_use:Nv \BNVS:n { next_index }
1635 }

```

6.13 Parsing

Workflows:

```

▶ \Beanoves or \BeanovesReset
▶ \__bnvs_root_keyval:NNn
▶ \__bnvs_root_keyval:nc or \__bnvs_root_parsed:nn
▶ \__bnvs_keyval:nnn or \__bnvs_indexed_keyval:nnnn

```

6.13.1 Square brackets

Here is the [documentation](#).

```

__bnvs_square_parse:nnn  \__bnvs_square_keyval:nnn {\<id>} {\<tag>} {\<definitions>}
__bnvs_square_parse:nnnn \__bnvs_square_parse:nnn {\<id>} {\<tag>} {\<definition>}
__bnvs_square_keyval:nnn \__bnvs_square_parse:nnnn {\<id>} {\<tag>} {\<index>} {\<definition>}

```

To parse what is inside square brackets.

```

1636 \BNVS_new:cpn { square_parse:nnn } #1 #2 #3 {
1637   \__bnvs_match_if_once:NnTF \c__bnvs_colons_regex { #3 } {
1638     \BNVS_error:n { No~colon~allowed:~[...#1...] }
1639   } {

```

Find the first available index.

```

1640   \__bnvs_next_index:nnn { #1 } { #2 } {
1641     \__bnvs_ginit:nnn { #1 } { #2.##1 } { \BNVS_value:n { #3 } }
1642   }
1643 }
1644 }

```

Why no colon allowed?

```

1645 \BNVS_new:cpn { square_parse:nnnn } #1 #2 #3 #4 {
1646   \__bnvs_match_if_once:NnTF \c__bnvs_colons_regex { #4 } {
1647     \BNVS_error:n { No~colon~allowed:~...#3=#4 }
1648   } {
1649     \__bnvs_match_if_once:NnTF \c__bnvs_A_GN_Z_regex { #3 } {
1650       \__bnvs_match_if_pop_left:cT G {
1651         \__bnvs_match_if_pop_left:cT G {
1652           \__bnvs_match_if_pop_left:cT N {
1653             \__bnvs_normalize_GN:
1654             \exp_args:Nnx \__bnvs_ginit:nnn { #1 } {
1655               #2.\l__bnvs_N_tl
1656             } { #4 }
1657           }
1658         }
1659       }
1660     } {
1661       \BNVS_error:n { Not~an~integer:~#3~(=#4) }
1662     }
1663   }
1664 }

```

For X=[...].

```

1665 \BNVS_new:cpn { square_keyval:nnn } #1 #2 #3 {

```

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

```

1666   \tl_if_empty:nTF { #2 } {
1667     \BNVS_error:n { Unexpected~list~at~top~level. }
1668   } {
1669     \tl_if_empty:nF { #3 } {
1670       \__bnvs_is_provide_and_gset:nnnF { #1 } { #2 } V {
1671         \BNVS_begin:

```

```

1672     \_bnvs_gunset_deep:nn { #1 } { #2 }
1673     \keyval_parse:nnn
1674     { \_bnvs_square_parse:nnn { #1 } { #2 } }
1675     { \_bnvs_square_parse:nnnn { #1 } { #2 } }
1676     { #3 }
1677     \BNVS_end:
1678   }
1679 }
1680 }
1681 }

1682 \BNVS_new:cpn { square_keyval:nvn } #1 {
1683   \BNVS_tl_use:nv { \_bnvs_square_keyval:nnn { #1 } }
1684 }

1685 \BNVS_new:cpn { square_keyval:vvv } {
1686   \BNVS_tl_use:Nvv \_bnvs_square_keyval:nnn
1687 }

1688 \BNVS_new:cpn { square_keyval_I:nn } {
1689   \BNVS_tl_use:cv { square_keyval:nnn } I
1690 }

1691 \BNVS_new:cpn { square_keyval_I:vn } {
1692   \BNVS_tl_use:cv { square_keyval_I:nn }
1693 }

1694 \BNVS_new:cpn { square_keyval_IT:n } {
1695   \BNVS_tl_use:Nvv \_bnvs_square_keyval:nnn I T
1696 }

```

6.13.2 Range or value lists

_bnvs_indexed_keyval:nnnn _bnvs_indexed_keyval:nnnn {<id> } {<tag> } {<index> } {<definitions> }

Parses the definitions as a standalone value. Other values are errors. It does not mean that standalone values are free from errors by themselves. Called by [_bnvs_root_parsed:nn](#).

```

1697 \BNVS_new:cpn { indexed_keyval:nnnn } #1 #2 #3 #4 {
1698   \tl_if_empty:nTF { #4 } {
1699     \_bnvs_gclear:nn { #1 } { #2.#3 }
1700   } {
1701     \BNVS_begin:
1702     \_bnvs_keyval:ncc { #4 } V a
1703     \BNVS_tl_last_unbraced:nv {
1704       \_bnvs_peek_branch_until_s_stop:nnnnw {
1705         \BNVS_error:n { No~^[... ]~allowed~with~index. }
1706       } {

```

For ID!TAG.1={FOO}}, go recursive.

```

1707     \_bnvs_indexed_keyval:nnnn { #1 } { #2 } { #3 } { ##1 }
1708   } {
1709     \_bnvs_ginit:nnnn { #1 } { #2 } { #3 } { \BNVS:n { #4 } }

```

Support \BNVS_value:n

```
1710 \__bnvs_ginit:nnn { #1 } { #2.#3 } { #4 }
1711 \__bnvs_is_gset:nnnF { #1 } { #2 } 0 {
1712 \__bnvs_gset:nnnx { #1 } { #2 } 0 {
1713 \exp_not:n { \BNVS_value:n { #4 } - } \int_eval:n { #3 - 1 }
1714 }
1715 }
1716 } {
1717 }
1718 } V \s_stop
1719 \__bnvs_tl_if_empty:cF a {
1720 \cs_set:Npn \BNVS:n ##1 { \exp_not:n { ##1, } }
1721 \cs_set:Npn \BNVS:nn ##1 ##2 { \exp_not:n { ##1 = ##2, } }
1722 \BNVS_error:x { Ignored:~\__bnvs_tl_use:c a }
1723 }
1724 \BNVS_end:
1725 }
1726 }
```

6.13.3 List specifiers

`__bnvs_list_keyval:nnn` `__bnvs_list_keyval:nnn {<id>} {<tag>} {<definitions>}`

Calls `\keyval_parse:nnn`. `<definition>` is the corresponding definition. For the list variant, `<definitions>` is a comma separated list of `<definition>`'s.

We parse all at once, then manage what is parsed. We could avoid a grouping level. At the top level, `id` is default and `tag` is not yet set. We do not remove outer braces from values. We parse key-value lists with the help of `\keyval_parse:nnn` or `\keyval_parse:NNn` except that we do not always remove one pair of outer braces for keys or values. The `list` mode is for ordered lists of integers or ranges.

The `dict` mode is for key-value lists between braces. We parse all at once, then manage what is parsed. We could avoid a grouping level.

```
1727 \BNVS_new:cpn { list_keyval:nnn } #1 #2 #3 {
1728 \BNVS_begin:
1729 \__bnvs_keyval:ncc { #3 } V a
1730 \__bnvs_tl_if_empty:cTF a {
```

Standalone key stored in V.

```
1731 \BNVS_tl_last_unbraced:nv {
1732 \__bnvs_peek_branch_until_s_stop:nnnnw {
1733 \BNVS_error:n { No~^[...]~allowed. }
1734 } {
```

For `<id>{<tag>{<...>}}`.

```
1735 \__bnvs_list_keyval:nnn { #1 } { #2 } { ##1 }
1736 } {
1737 \__bnvs_gunset_deep:nn { #1 } { #2 }
1738 \__bnvs_gset:nnnv { #1 } { #2 } 0 V
1739 } {
1740 }
1741 } V \s_stop
1742 } {
```

A single value or range specification.

```

1743     \__bnvs_gunset_deep:nn { #1 } { #2 }
1744     \__bnvs_tl_clear:c b
1745     \cs_set:Npn \BNVS:nn ##1 ##2 {
1746         \__bnvs_list_keyval:nnn { #1 } { #2.##1 } { ##2 }
1747         \__bnvs_tl_put_right:cn b { \BNVS:n { ##1 } }
1748     }
1749     \cs_set:Npn \BNVS:n ##1 {
1750         \__bnvs_next_index:nnn { #1 } { #2 } {
1751             \BNVS:nn { #####1 } { ##1 }
1752         }
1753     }
1754     \__bnvs_tl_if_empty:cF V {
1755         \BNVS_tl_use:Nv \BNVS:n V
1756     }
1757     \__bnvs_tl_use:c a
1758     \__bnvs_gset:nnnv { #1 } { #2 } 0 b
1759 }
1760 \BNVS_end:
1761 }

```

```

\__bnvs_keyval:nnn \__bnvs_keyval:nnn {<id>} {<tag>} {<definition>}
\__bnvs_keyval:vvv

```

Calls `\keyval_parse:nnn`. `<definition>` is the corresponding definition. Workflow:

```

└─ \__bnvs_root_parsed:nn
└─ \__bnvs_keyval:nnn

```

```

\__bnvs_keyval:nnn \__bnvs_keyval:nnn {<id>} {<key>} {<definition>}

```

```

1762 \BNVS_new:cpn { keyval:nnn } #1 #2 #3 {
1763     \tl_if_empty:nTF { #3 } {
1764         \__bnvs_gclear:nn { #1 } { #2 }
1765     } {
1766         \__bnvs_if_should_provide:nnT { #1 } { #2 } {
1767             \BNVS_begin:
1768             \__bnvs_keyval:ncc { #3 } V a
1769             \__bnvs_tl_if_empty:cTF a {

```

Standalone key stored in V.

```

1770         \BNVS_tl_last_unbraced:nv {
1771             \__bnvs_peek_branch_until_s_stop:nnnnw {
1772                 \tl_if_empty:nTF { ##1 }
1773                 \__bnvs_keyval:nnn
1774                 \__bnvs_square_keyval:nnn
1775                 { #1 } { #2 } { ##1 }
1776             } {

```

For ID!TAG={F00}.

```

1777         \tl_if_empty:nTF { ##1 }
1778             \__bnvs_keyval:nnn
1779             \__bnvs_curly_keyval:nnn
1780             { #1 } { #2 } { ##1 }
1781     } {
1782         \__bnvs_ginit:nnv { #1 } { #2 } V
1783     } {
1784     }
1785 } V \s_stop
1786 } {

```

A single value or range specification.

```

1787         \__bnvs_gunset_deep:nn { #1 } { #2 }
1788         \__bnvs_tl_clear:c b
1789         \cs_set:Npn \BNVS:nn ##1 ##2 {
1790             \__bnvs_keyval:nnn { #1 } { #2.##1 } { ##2 }
1791             \__bnvs_tl_put_right:cn b { \BNVS:n { ##1 } }
1792         }

```

Tiny subtlety with great impact:

```

1793         \__bnvs_tl_if_empty:cF V {
1794             \BNVS_tl_use:Nv \BNVS:nn V 1
1795         }
1796         \__bnvs_tl_use:c a
1797         \__bnvs_gset:nnnv { #1 } { #2 } 2 b
1798     }
1799     \BNVS_end:
1800 }
1801 }
1802 }

1803 \BNVS_new:cpn { keyval:vvv } {
1804     \BNVS_tl_use:Nvv \__bnvs_keyval:nnn
1805 }

```

6.13.4 Items between braces

Workflow:

```

▶ \__bnvs_keyval:nnn or \__bnvs_curly_parse:nnnn
▶ \__bnvs_curly_keyval:nnn
▶ \__bnvs_curly_parse:nnn or \__bnvs_curly_parse:nnnn
▶ \__bnvs_curly_keyval:nnn, \__bnvs_square_keyval:nnn or \__bnvs_ginit:nnn.

```

| | |
|---|---|
| <code>__bnvs_curly_keyval:nnn</code> | <code>__bnvs_curly_keyval:nnn {<id>} {<tag>} {<definitions>}</code> |
| <code>__bnvs_curly_keyval:(vvv nnv)</code> | <code>__bnvs_curly_parse:nnn {<id>} {<tag>} {<definition>}</code> |
| <code>__bnvs_curly_parse:nnn</code> | <code>__bnvs_curly_parse:nnnn {<id>} {<tag>} {<key>} {<definition>}</code> |
| <code>__bnvs_curly_parse:nnnn</code> | |

For $\langle ref \rangle = \{ \langle def_1 \rangle, \dots, \langle def_j \rangle \}$. Deep first traversal.

```

1806 \BNVS_new:cpn { curly_parse:nnn } #1 #2 #3 {
1807     \__bnvs_curly_parse:nnnn { #1 } { #2 } { #3 } 1

```



```

1808 }

1809 \BNVS_new:cpn { curly_parse:nnnn } #1 #2 #3 #4 {
1810   \BNVS_begin:
1811   \__bnvs_match_if_once:NnTF \c__bnvs_A_PGN_Z_regex { #3 } {
1812     \__bnvs_match_if_pop_left:cT P {
1813       \__bnvs_match_if_pop_left:cT P {
1814         \__bnvs_tl_put_left:cn P { #2. }
1815         \__bnvs_match_if_pop_left:cT G {
1816           \__bnvs_match_if_pop_left:cT N {
1817             \__bnvs_tl_if_empty:cTF { N } {
This is not a A...⟨number⟩ reference.
1818             \__bnvs_peek_branch_until_s_stop:nnnw {
1819               \__bnvs_square_keyval:nvn { #1 } P { ##1 }
1820             } {
1821               \__bnvs_curly_keyval:nvn { #1 } P { ##1 }
1822             } {
1823               \BNVS_tl_use:nv { \__bnvs_ginit:nnn { #1 } } P { ##1 }
1824             } {
No specification: A= nothing, unset everything.
1825             \__bnvs_gunset_deep:nv { #1 } P
1826           }
1827         } {
1828         \__bnvs_normalize_GN:
1829         \__bnvs_peek_branch_until_s_stop:nnnw {
1830           \BNVS_error:n { Unexpected-[]:~##1 }
1831         } {
1832           \BNVS_error:n { Unexpected-{:~##1 }
1833         } {
1834           \__bnvs_if_set:ncccTF A Z L { ##1 } {
1835             \BNVS_error:n { Unexpected~colon:~##1 }
1836           } {
1837             \__bnvs_is_gset:nvnTF { #1 } P 0 {
1838               \__bnvs_tl_put_right:cn P .
1839               \__bnvs_tl_put_right:cv P N
1840               \__bnvs_is_provide_and_gset:nvnF { #1 } P 0 {
1841                 \__bnvs_ginit:nvn { #1 } P { ##1 }
1842               }
1843             } {
1844               \__bnvs_tl_if_eq:cnTF N 1 {
1845                 \__bnvs_ginit:nvn { #1 } P { ##1 }
1846               } {
1847                 \exp_args:Nnnx \__bnvs_ginit:nvn { #1 } P {
1848                   \exp_not:n { ##1 -} \int_eval:n { \l__bnvs_N - 1 }
1849                 }
1850               }
1851               \__bnvs_tl_put_right:cn P .
1852               \__bnvs_tl_put_right:cv P N
1853               \__bnvs_ginit:nvn { #1 } P { ##1 }
1854             }
1855           }
1856         } {
1857           \__bnvs_tl_put_right:cn P .

```

```

1858         \_bnvs_tl_put_right:cv P N
1859         \_bnvs_gunset_deep:nv { #1 } P
1860         \_bnvs_gunset:nv { #1 } P
1861     }
1862 }
1863 #4 \s_stop
1864 }
1865 }
1866 }
1867 }
1868 } {
1869     \BNVS_error:n { Unsupported~#3 }
1870 }
1871 \BNVS_end:
1872 }

```

Here is the [documentation](#).

```

1873 \BNVS_new:cpn { curly_keyval:nnn } #1 #2 #3 {
1874     \tl_if_empty:nTF { #3 } {
1875     } {
1876         \BNVS_begin:
1877         %     \_bnvs_curly_keyval:nc { #3 } a
1878         \keyval_parse:nnn {
1879             \_bnvs_curly_parse:nnn { #1 } { #2 }
1880         } {
1881             \_bnvs_curly_parse:nnnn { #1 } { #2 }
1882         } { #3 }
1883         \BNVS_end:
1884     }
1885 }

1886 \BNVS_new:cpn { curly_keyval:nvn } #1 {
1887     \BNVS_tl_use:nv { \_bnvs_curly_keyval:nnn { #1 } }
1888 }

1889 \BNVS_new:cpn { curly_keyval:vvv } {
1890     \BNVS_tl_use:Nvv \_bnvs_curly_keyval:nnn
1891 }

```

6.13.5 High level and root parser

These are the first functions called. Workflow:

```

▶ \Beanoves or \BeanovesReset
▶ \_bnvs_root_keyval:NNn
▶ \_bnvs_root_keyval:nc
▶ \_bnvs_root_parsed:nn
▶ \_bnvs_keyval:nnn or \_bnvs_indexed_keyval:nnnn

```

```

\_bnvs_root_parsed:nn \_bnvs_root_parsed:nn {<name>} {<definition>}

```

A top level *<name>*–*<definition>* has been parsed. Ensure that *<name>* is correct and then deal with the definition.

```

1892 \BNVS_new:cpn { root_parsed:nn } #1 #2 {

```

```

1893 \__bnvs_match_if_once:NnTF \c__bnvs_A_IKTGN_Z_regex { #1 } {
1894   \__bnvs_match_if_pop_left:cT a {
1895     \__bnvs_match_if_pop_left:cT I {
1896       \__bnvs_match_if_pop_left:cT K {
1897         \__bnvs_tl_if_empty:cTF K {
1898           \__bnvs_tl_set_eq:cc I J
1899         } {
1900           \__bnvs_tl_set_eq:cc J I
1901         }
1902         \__bnvs_match_if_pop_left:cT T {
1903           \__bnvs_match_if_pop_left:cT G {
1904             \__bnvs_match_if_pop_left:cT N {
1905               \__bnvs_tl_if_empty:cTF N {
1906                 \__bnvs_keyval:vvv ITN { #2 }
1907               } {
1908                 \__bnvs_normalize_GN:
1909                 \BNVS_tl_use:Nvvv \__bnvs_indexed_keyval:nnnn ITN { #2 }
1910               }
1911             }
1912           }
1913         }
1914       }
1915     }
1916   }
1917 } {
1918   \BNVS_error:n { Unexpected~ref:~#1 }
1919 }
1920 }

```

```

\__bnvs_root_keyval:nc \__bnvs_root_keyval:nc {<definitions>} {<aux>}

```

{<aux>} is an auxiliary variable name unused on return. Calls `__bnvs_root_parsed:nn` and is called by `\BeanovesReset` and `__bnvs_root_keyval:NNn` Auxiliary function called by `__bnvs_root_keyval:NNn`.

1. `x`-expands `<definitions>` into the `<aux>` variable.
2. prepend any \TeX group with `\BNVS_curly:n` such that outer braces are not removed from values.
3. replace any `[<...>]` with `\BNVS_bracket:n{<...>}`, testing for unbalanced delimiter,
4. Parse with keyval based on `\keyval_parsed:nn`.

Does not remove .

```

1921 \BNVS_new:cpn { root_keyval:nc } #1 #2 {
1922   \__bnvs_tl_set:cx { #2 } { #1 }
1923   \BNVS_tl_use:nc {
1924     \regex_replace_all:nnN { \cB. } { \c { BNVS_curly:n } \0 }
1925   } { #2 }
1926   \cs_set:Npn \BNVS:N ##1 {
1927     \cs_set:Npn \BNVS: {
1928       \regex_replace_all:nnNT { \[ (.*?) \] } {

```

```

1929         \c { BNVS_square:n } \cB\{ \1 \cE \}
1930     } ##1 {
1931         \BNVS:
1932     }
1933 }
1934 \BNVS:
1935 \regex_match:nNT { \[|\] } ##1 {
1936     \BNVS_error:n { Unbalanced~[|] }
1937 }
1938 }
1939 \BNVS_tl_use:Nc \BNVS:N { #2 }
1940 \cs_set:Npn \BNVS:n ##1 {
1941     \__bnvs_root_parsed:nn { ##1 } 1
1942 }
1943 \BNVS_tl_use:nv {
1944     \keyval_parse:nnn {
1945         \exp_after:wN \__bnvs_root_parsed:nn \use_ii_i:nn 1
1946     } {
1947         \__bnvs_root_parsed:nn
1948     }
1949 } { #2 }
1950 }

```

`__bnvs_root_keyval:NNn` `__bnvs_root_keyval:NNn` *<is at top>* *<on provide mode>* *{<definitions>}*

Calls `__bnvs_root_keyval:nc` after some setup. Called by `\Beanoves`. The a tl auxiliary variable is used.

```

1951 \BNVS_new:cpn { root_keyval:NNn } #1 #2 #3 {
1952     \bool_if:NT #1 {

```

At the document level, clear everything.

```

1953     \__bnvs_gclear:
1954 }
1955 \BNVS_begin:
1956 \bool_if:NTF #2 {
1957     \__bnvs_provide_on:
1958 } {
1959     \__bnvs_provide_off:
1960 }
1961 \__bnvs_int_zero:c { i }
1962 \__bnvs_tl_set:cn a { #3 }
1963 \bool_if:NT #1 {

```

At the document level, use the global definitions.

```

1964     \seq_if_empty:NF \g__bnvs_def_seq {
1965         \__bnvs_tl_put_left:cx a {
1966             \seq_use:Nn \g__bnvs_def_seq , ,
1967         }
1968     }
1969 }
1970 \BNVS_tl_use:Nv \__bnvs_root_keyval:nc a a

```

This is a list, possibly at the top

```

1971 \BNVS_end_tl_set:cv J J
1972 }

```

`\Beanoves` `\Beanoves {⟨key-value list⟩}`

The keys are the slide overlay references. When no value is provided, it defaults to 1. On the contrary, `⟨key-value⟩` items are parsed by `__bnvs_parse:nn`.

```

1973 \NewDocumentCommand \Beanoves { sm } {
1974   \__bnvs_set_false:c { reset }
1975   \__bnvs_set_false:c { reset_all }
1976   \__bnvs_set_false:c { only }
1977   \tl_if_empty:NTF \@currenvir {

```

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

```

1978   \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
1979 } {
1980   \tl_if_eq:NnTF \@currenvir { document } {
1981     \IfBooleanTF {#1} {
1982       \__bnvs_root_keyval:NNn \c_true_bool \c_true_bool
1983     } {
1984       \__bnvs_root_keyval:NNn \c_true_bool \c_false_bool
1985     }
1986   } {
1987     \IfBooleanTF {#1} {
1988       \__bnvs_root_keyval:NNn \c_false_bool \c_true_bool
1989     } {
1990       \__bnvs_root_keyval:NNn \c_false_bool \c_false_bool
1991     }
1992   } { #2 }
1993   \ignorespaces
1994 }
1995 }

```

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

```

1996 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}

```

6.14 Scanning named overlay specifications

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

| | |
|---|---|
| <code>__bnvs_@frame</code> <code>__bnvs_@masterdecode</code> | <code>__bnvs_@frame {⟨overlay specification⟩}</code> <code>__bnvs_@masterdecode {⟨overlay specification⟩}</code> |
|---|---|

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

`\l__bnvs_ans_tl` Storage for the translated overlay specification, `⟨...⟩` 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.

```

1997 \makeatletter
1998 \cs_set:Npn \__bnvs_@frame < #1 > {
1999   \BNVS_begin:
2000   \__bnvs_tl_clear:c { ans }
2001   \__bnvs_resolve_queries:nc { #1 } { ans }

```

```

2002 \BNVS_set:cpn { :n } ##1 { \BNVS_end: \BNVS_saved@frame < ##1 > }
2003 \BNVS_tl_use:cv { :n } { ans }
2004 }

2005 \cs_set:Npn \__bnvs_@masterdecode #1 {
2006 \BNVS_begin:
2007 \__bnvs_tl_clear:c { ans }
2008 \__bnvs_resolve_queries:nc { #1 } { ans }
2009 \BNVS_tl_use:nv {
2010 \BNVS_end:
2011 \BNVS_saved@masterdecode
2012 } { ans }
2013 }

2014 \cs_new:Npn \BeanovesOff {
2015 \cs_set_eq:NN \beamer@frame \BNVS_saved@frame
2016 \cs_set_eq:NN \beamer@masterdecode \BNVS_saved@masterdecode
2017 }

2018 \cs_new:Npn \BeanovesOn {
2019 \cs_set_eq:NN \beamer@frame \__bnvs_@frame
2020 \cs_set_eq:NN \beamer@masterdecode \__bnvs_@masterdecode
2021 }

2022 \AddToHook{begindocument/before}{
2023 \cs_if_exist:NTF \beamer@frame {
2024 \cs_set_eq:NN \BNVS_saved@frame \beamer@frame
2025 \cs_set_eq:NN \BNVS_saved@masterdecode \beamer@masterdecode
2026 } {
2027 \cs_set:Npn \BNVS_saved@frame < #1 > {
2028 \BNVS_error:n {Missing-package-beamer}
2029 }
2030 \cs_set:Npn \BNVS_saved@masterdecode < #1 > {
2031 \BNVS_error:n {Missing-package-beamer}
2032 }
2033 }
2034 \BeanovesOn
2035 }
2036 \makeatother

```

6.14.1 Top level

| | |
|---|---|
| <code>__bnvs_resolve_queries:nc</code> | <code>__bnvs_resolve_queries:nc {<queries>} {<ans>}</code> |
| <code>__bnvs_prepare_queries:c</code> | <code>__bnvs_prepare_queries:c {<ans>}</code> |

Replace in the `<queries>` all the `?(<...>)` query instructions in `<queries>` with their static counterpart. The function `__bnvs_prepare_queries:c` is called first. The implementation is not very efficient, but it does not cost that much.

```

2037 \BNVS_new:cpn { resolve_queries:nc } #1 #2 {

```

```

2038 \__bnvs_tl_set:cn { #2 } { #1 }
2039 \__bnvs_prepare_queries:c { #1 }
2040 \__bnvs_if_resolve_queries:cF {
2041   \BNVS_error:n { Failure~#1 }
2042 }
2043 }

2044 \BNVS_new:cpn { prepare_queries:c } #1 {
2045   \BNVS_begin:
2046   \__bnvs_tl_set:cn L { #1 }
2047   \cs_set:Npn \BNVS: {
2048     \regex_replace_once:nnNT { \ ( (?:(\?)|(!)/)? ([%(
2049       ^)*)\ )
2050   } {
2051     \c{BNVS_round\1\2:n } \cB\{ \3 \cE\}
2052   } \l__bnvs_L_tl \BNVS:
2053 }
2054 \BNVS:
2055 \regex_replace_all:nnN
2056 { \c{ BNVS_round(?:\?|!):n } } { \c{BNVS_query:n} } \l__bnvs_L_tl
2057
2058 \cs_set:cpn \BNVS_query:n { \exp_not:N \BNVS_query:n }
2059 \cs_set:Npn \BNVS_round:n ##1 { ( ##1 ) }
2060 \__bnvs_tl_set:cx L { \l__bnvs_L_tl }
2061 \BNVS_end_tl_set:cv { #1 } L
2062 }

```

__bnvs_if_resolve_flat:ncTF __bnvs_if_resolve_flat:ncTF {<query>} {<ans>} {<yes code>} {<no code>}

Called by __bnvs_if_resolve_queries:ncTF

```

2063 \BNVS_new_conditional:cpnn { if_resolve_flat:nc } #1 #2 { T, F, TF } {
2064   \__bnvs_if_call:TF {
2065     \BNVS_begin:

```

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

```

2066   \__bnvs_if_regex_split:cnTF { XPXPGNURO } { #1 } {

```

The leftmost item is not a special item: we start feeding the `ans` tl variable with it. We first define the function that concludes the resolution and the function that rounds the result.

```

2067   \BNVS_set:cpn { if_resolve_flat_end_return_true: } {

```

Normal and unique end of the loop.

```

2068   \__bnvs_resolution_round_ans:
2069   \BNVS_end_tl_set:cv { #2 } { ans }
2070   \prg_return_true:
2071 }

```

The leftmost item is not a special item: we start feeding the `ans` tl variable with it. We first define the function that concludes the resolution and the function that rounds the result.

```

2072   \BNVS_set:cpn { if_resolve_flat_end_return_false: } {

```

Normal and unique end of the loop.

```

2073     \BNVS_error:n { Unsupported~query: #1}
2074     \BNVS_end:
2075     \prg_return_false:
2076 }

```

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

```

2077     \BNVS_set:cpn { resolution_round_ans: } { \_bnvs_round:c { ans } }
2078     \_bnvs_tl_clear_ans:
2079     \_bnvs_resolve_init_end_return:
2080 } {

```

There is not reference.

```

2081     \_bnvs_tl_set:cn { ans } { #1 }
2082     \_bnvs_round:c { ans }
2083     \BNVS_end_tl_set:cv { #2 } { ans }
2084     \prg_return_true:
2085 }
2086 } {
2087     \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
2088     \BNVS_end:
2089     \prg_return_false:
2090 }
2091 }

```

```

\_bnvs_if_resolve_query:nc $\overline{TF}$  \_bnvs_if_resolve_query:ncTF {<query>} {<tl core>} {<yes code>} {<no
\_bnvs_if_resolve_query:vc $\overline{TF}$  code>}

```

Evaluate only one query.

```

2092 \BNVS_new_conditional:cpnn { if_resolve_query:nc } #1 #2 { T, F, TF } {
2093   \_bnvs_greset_call:
2094   \_bnvs_match_if_once:NnTF \c__bnvs_A_VAZLLZZL_Z_regex { #1 } {
2095     \BNVS_begin:
2096     \_bnvs_if_resolve_query_branch:TF {
2097       \BNVS_end_tl_set:cv { #2 } { ans }
2098       \prg_return_true:
2099     } {
2100       \BNVS_end:
2101       \prg_return_false:
2102     }
2103   } {
2104     \BNVS_error:n { Syntax~error:~#1 }
2105     \BNVS_end:
2106     \prg_return_false:
2107   }
2108 }

2109 \BNVS_new_conditional_vc:cn { if_resolve_query } { T, F, TF }

```

```

\_bnvs_if_resolve_queries:c $\overline{TF}$  \_bnvs_if_resolve_queries:c {<queries>} {<yes code>} {<no code>}

```

Resolve the queries in place and branch to *<yes code>* otherwise leave the variable untouched and branch to *<no code>*. The queries were prepared beforehand.


```

2110 \tl_new:N \l__bnvs_if_resolve_queries_tl
2111 \BNVS_new:cpn { if_resolve_queries:c } #1 {
2112   \BNVS_begin:
2113   \__bnvs_tl_clear:c { if_resolve_queries }
2114   \BNVS_tl_use:Nv \clist_map_inline:nn { #1 } {
2115     \regex_match:nnTF { \c{ BNVS_query:n } } { ##1 } {
2116       \cs_set:Npn \BNVS:w #####1 \BNVS_query:n #####2 #####3 \s_stop {
2117         \__bnvs_tl_put_right:cn { if_resolve_queries } { ##1 }
2118         \clist_map_inline:nn { #####2 } {
2119           \__bnvs_if_append:ncF { #####1 } { if_resolve_queries } {
2120             \BNVS_error:x { Bad~query:~\tl_to_str:n { #####1 } }
2121           }
2122         }
2123         \regex_if_match:nnNTF { \c{ BNVS_query:n } } { #####3 } {
2124           \BNVS:w #####3 \s_stop
2125         } {
2126           \__bnvs_tl_put_right:cn { if_resolve_queries } { #####3 }
2127         }
2128       }
2129       \regex_if_match:nnTF { \c{ BNVS_query:n } } { #####3 } {
2130         \BNVS:w #####3 \s_stop
2131       } {
2132         \__bnvs_tl_put_right:cn { if_resolve_queries } { #####3 }
2133       }
2134       \cs_set:Npn \BNVS:w #####1 \BNVS_query:n #####2 #####3 \s_stop {
2135         \__bnvs_tl_put_right:cn A { #####1 }
2136         \clist_map_inline:nn { #####2 } {
2137           \__bnvs_if_append:ncF { #####1 } A {
2138             \BNVS_error:x { Bad~query:~\tl_to_str:n { #####1 } }
2139           }
2140         }
2141         \regex_if_match:nnNTF { \c{ BNVS_query:n } } { #####3 } {
2142           \BNVS:w #####3 \s_stop
2143         } {
2144           \__bnvs_tl_put_right:cn A { #####3 }
2145         }
2146       }
2147       \BNVS:w ##1 \s_stop
2148     } {
2149       \__bnvs_tl_put_right:cn { if_resolve_queries } { ##1 }
2150     }
2151   }
2152   \BNVS_end_tl_set:cv { #1 } { if_resolve_queries }
2153 }

```

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_query_tl` Storage for the overlay query expression to be evaluated.

(End of definition for `\l__bnvs_query_tl`.)

6.15 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_ISP:cccTF \\_bnvs_if_ISP:cccTF {<id>} {<name>} {<path>} {<yes code>} {<no code>}
```

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

```
2154 \\BNVS_new_conditional:cpnn { if_ISP:ccc } #1 #2 #3 { T, F, TF } {
2155   \\_bnvs_match_if_once:NvTF \\c__bnvs_A_XP_Z_regex { #1 } {
2156     \\BNVS_begin:
```

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

```
2157   \\_bnvs_match_if_pop_ISP:cccTF { #1 } { #2 } { #3 } {
2158     \\_bnvs_export_ISP:cccN { #1 } { #2 } { #3 }
2159     \\BNVS_end:
2160     \\prg_return_true:
2161   } {
2162     \\BNVS_end:
2163     \\prg_return_false:
2164   }
2165 } {
2166   \\prg_return_false:
2167 }
2168 }
2169 \\quark_new:N \\q__bnvs_X

2170 \\tl_new:N \\l__bnvs_export_ISP_cccN_tl
2171 \\BNVS_new:cpn { export_ISP:cccN } #1 #2 #3 #4 {
2172   \\cs_set:Npn \\BNVS_export_ISP_cccN:w ##1 ##2 ##3 {
2173     #4
2174     \\_bnvs_tl_set:cn { #1 } { ##1 }
2175     \\_bnvs_tl_set:cn { #2 } { ##2 }
2176     \\_bnvs_tl_set:cn { export_ISP_cccN } { ##3 }
2177   }
2178   \\_bnvs_tl_set:cx { export_ISP_cccN }
2179   { \\_bnvs_seq_use:cn { #1 } { \\q__bnvs_X } }
2180   \\BNVS_tl_use:nvv {
2181     \\BNVS_tl_use:Nv \\BNVS_export_ISP_cccN:w { #1 }
2182   } { #2 } { export_ISP_cccN }
2183   \\BNVS_tl_use:nv {
2184     \\_bnvs_seq_set_split:cnn { #3 } { \\q__bnvs_X }
2185   } { export_ISP_cccN }
2186   \\_bnvs_seq_remove_all:cn { #3 } { }
2187 }
```

```

2188 \tl_new:N \l__bnvs_match_if_export_XP_cccc_tl
2189 \BNVS_new:cpn { match_if_export_XP:ccccN } #1 #2 #3 #4 #5 {
2190   \cs_set:Npn \BNVS_match_if_export_XP_ccccN:w ##1 ##2 ##3 ##4 {
2191     #5
2192     \__bnvs_tl_set:cn { #1 } { ##1 }
2193     \__bnvs_tl_set:cn { #2 } { ##2 }
2194     \__bnvs_tl_set:cn { #3 } { ##3 }
2195     \__bnvs_tl_set:cn { #4 } { ##4 }
2196   }
2197   \__bnvs_tl_set:cx { match_if_export_XP_cccc }
2198   { \__bnvs_seq_use:cn { #1 } { \q__bnvs_X } }
2199   \BNVS_tl_use:nvvv {
2200     \BNVS_tl_use:Nv \BNVS_match_if_export_XP_ccccN:w { #1 }
2201   } { #2 } { match_if_export_XP_cccc } { #4 }
2202   \BNVS_tl_use:nv {
2203     \__bnvs_seq_set_split:cn { #3 } { \q__bnvs_X }
2204   } { match_if_export_XP_cccc }
2205   \__bnvs_seq_remove_all:cn { #3 } { }
2206 }

2207 \BNVS_new_conditional:cpnn { match_if_pop_XP:cccc } #1 #2 #3 #4 { TF } {
2208   \BNVS_begin:
2209   \__bnvs_match_if_pop_left:cTF { #1 } {
2210     \__bnvs_match_if_pop_left:cTF { #1 } {
2211       \__bnvs_match_if_pop_left:cTF { #2 } {
2212         \__bnvs_match_if_pop_left:cTF { #3 } {
2213           \__bnvs_seq_set_split:cn { #3 } { . } { #3 }
2214           \__bnvs_seq_remove_all:cn { #3 } { }
2215           \__bnvs_match_if_pop_left:cTF { #4 } {
2216             \__bnvs_match_if_export_XP:ccccN { #1 } { #2 } { #3 } { #4 }
2217             \BNVS_end:
2218             \prg_return_true:
2219           } {
2220             \BNVS_end_return_false:
2221           }
2222         } {
2223           \BNVS_end_return_false:
2224         }
2225       } {
2226         \BNVS_end_return_false:
2227       }
2228     } {
2229       \BNVS_end_return_false:
2230     }
2231   } {
2232     \BNVS_end_return_false:
2233   }
2234 }

```

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.

```

2235 \BNVS_new:cpn { seq_merge:cc } #1 #2 {
2236   \__bnvs_seq_if_empty:cF { #2 } {
2237     \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs_X } {
2238       \__bnvs_seq_use:cn { #1 } { \q__bnvs_X }
2239       \exp_not:n { \q__bnvs_X }
2240       \__bnvs_seq_use:cn { #2 } { \q__bnvs_X }
2241     }
2242     \__bnvs_seq_remove_all:cn { #1 } { }
2243   }
2244 }

```

6.16 Evaluation bricks

6.16.1 Helpers

| | |
|--|---|
| <code>__bnvs_round:N</code> <code>__bnvs_round:c</code> | <code>__bnvs_round:N <tl variable></code> <code>__bnvs_round:c {(tl core name)}</code> |
|--|---|

Replaces the variable content with its rounded floating point evaluation.

```

2245 \BNVS_new:cpn { round:N } #1 {
2246   \tl_if_empty:Ntf #1 {
2247     \tl_set:Nn #1 { 0 }
2248   } {
2249     \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
2250   }
2251 }

2252 \BNVS_new:cpn { round:c } {
2253   \BNVS_tl_use:Nc \__bnvs_round:N
2254 }

```

6.16.2 Resolve from initial values

Lower level resolution functions.

| | |
|--|---|
| <code>__bnvs_resolve_V_or_R:nncc</code> | <code>__bnvs_resolve_V_or_R:nnn {(id)} {(tag)} {(spec)} {(V)} {(R)}</code> |
|--|---|

Auxiliary function used by `__bnvs_resolve:nn`. It does not change the `<id>!``<tag>` data model. It resolves `<spec>` into either `{(V)}` or `{(R)}` in the `<id>!``<tag>` context.

```

2255 \regex_const:Nn \c__bnvs_A_R_Z_regex { \A ([^~]*)[-]([~]*)\Z}
2256 \tl_new:N \l__bnvs_vr_tl
2257 \tl_new:N \l__bnvs_VR_tl
2258 \BNVS_new:cpn { resolve_V_or_R:nncc } #1 #2 #3 #4 #5 {
2259   \__bnvs_tl_clear:c { VR }
2260   \cs_set:Npn \BNVS:n ##1 {
2261     \__bnvs_tl_clear:c { vr }
2262     \__bnvs_if_append:nnccF { #1 } { #2 } { ##1 } { vr } {

```

```

2263 \__bnvs_if_resolve:ncTF { ##1 } { vr } {
2264 \BNVS_tl_use:Nv \clist_map_inline:nn { vr } {
2265 \tl_if_empty:nF { #####1 } {
2266 \__bnvs_tl_if_empty:cF { VR } {
2267 \__bnvs_tl_put_right:cn { VR } { , }
2268 }
2269 \__bnvs_match_if_once:NnTF \c__bnvs_A_R_Z_regex { #####1 } {
2270 \__bnvs_match_if_pop_left:cT a {
2271 \__bnvs_tl_if_empty:cT a {
2272 \__bnvs_tl_set:cn a 1
2273 }
2274 \__bnvs_match_if_pop_left:cT z { }
2275 }
2276 \__bnvs_gset_azl:nn { #1 } { #2 }
2277 \__bnvs_tl_put_right:cx { VR } {
2278 \l__bnvs_a_tl - \l__bnvs_z_tl
2279 }
2280 } {
2281 \__bnvs_tl_put_right:cn { VR } { #####1 }
2282 }
2283 }
2284 }
2285 } {
2286 \__bnvs_tl_if_empty:cF { VR } {
2287 \__bnvs_tl_put_right:cn { VR } { , }
2288 }
2289 \__bnvs_tl_put_right:cn { VR } { 1 }
2290 }
2291 }
2292 }
2293 \tl_if_head_eq_meaning:nNTF { #3 } \BNVS:n {
2294 #3
2295 } {
2296 \BNVS:n { #3 }
2297 }
2298 \__bnvs_tl_if_empty:cTF { VR } {
2299 \tl_if_empty:nF { #4 } { \__bnvs_tl_clear:c { #4 } }
2300 \tl_if_empty:nF { #5 } { \__bnvs_tl_clear:c { #5 } }
2301 } {
2302 \__bnvs_gunset:nn { #1 } { #2 }
2303 \BNVS_tl_use:nv { \regex_match:nnTF { [-,] } } { VR } {
2304 \__bnvs_gset:nnnv { #1 } { #2 } R { VR }
2305 \tl_if_empty:nF { #4 } { \__bnvs_tl_clear:c { #4 } }
2306 \tl_if_empty:nF { #5 } { \__bnvs_tl_set:cv { #5 } { VR } }
2307 } {
2308 \__bnvs_gset:nnnv { #1 } { #2 } V { VR }
2309 \tl_if_empty:nF { #4 } { \__bnvs_tl_set:cv { #4 } { VR } }
2310 \tl_if_empty:nF { #5 } { \__bnvs_tl_clear:c { #5 } }
2311 }
2312 }
2313 }

```

```

2314 \__bnvs_if_resolve_init:nnTF \__bnvs_if_resolve_init:nnTF {<id>} {<tag>} {<yes code>} {<no code>}

```

Auxiliary function used by `__bnvs_if_resolve_?:`. If it can resolve `<spec>`, setup the `<id>!``<tag>` data model accordingly and execute `<yes code>`. Otherwise mark unsolvability and execute `<no code>`.

On entering, this function assumes that there is absolutely no data for `<id>!``<tag>`/`V` nor `<id>!``<tag>`/`R` (it has been unset for example).

This function is executed within a `TEX` group in order not to alter the outer world, because in must be reentrant.

```

2314 \BNVS_new_conditional:cpnn { if_resolve_init:nn } #1 #2 { T, F, TF } {
2315   \BNVS_begin:
2316   \__bnvs_if_get:nnncTF { #1 } { #2 } 0 { VR } {
2317     \__bnvs_tl_clear:c { VR }
2318     \cs_set:Npn \BNVS:n ##1 {
2319       \__bnvs_tl_clear:c { vr }
2320       \__bnvs_if_append:nnncF { #1 } { #2 } { ##1 } { vr } {
2321         \__bnvs_if_resolve:ncTF { ##1 } { vr } {
2322           \BNVS_tl_use:Nv \clist_map_inline:nn { vr } {
2323             \tl_if_empty:nF { #####1 } {
2324               \__bnvs_tl_if_empty:cF { VR } {
2325                 \__bnvs_tl_put_right:cn { VR } { , }
2326               }
2327               \__bnvs_match_if_once:NnTF \c__bnvs_A_R_Z_regex { #####1 } {
2328                 \__bnvs_match_if_pop_left:cT a {
2329                   \__bnvs_tl_if_empty:cT a {
2330                     \__bnvs_tl_set:cn a 1
2331                   }
2332                   \__bnvs_match_if_pop_left:cT z { }
2333                 }
2334                 \__bnvs_gset_azl:nn { #1 } { #2 }
2335                 \__bnvs_tl_put_right:cx { VR } {
2336                   \l__bnvs_a_tl - \l__bnvs_z_tl
2337                 }
2338               } {
2339                 \__bnvs_tl_put_right:cn { VR } { #####1 }
2340               }
2341             }
2342           }
2343         } {
2344           \__bnvs_tl_if_empty:cF { VR } {
2345             \__bnvs_tl_put_right:cn { VR } { , }
2346           }
2347           \__bnvs_tl_put_right:cn { VR } { 1 }
2348         }
2349       }
2350     }
2351     \__bnvs_will_gset:nnn { #1 } { #2 } V
2352     \__bnvs_will_gset:nnn { #1 } { #2 } R
2353     \cs_set:Npn \BNVS_if_resolve_init:n ##1 {
2354       \__bnvs_tl_clear:c { VR }
2355       \tl_if_head_eq_meaning:nNTF { ##1 } \BNVS:n {
2356         ##1
2357       } {

```

```

2358     \BNVS:n { ##1 }
2359   }
2360 }
2361 \BNVS_tl_use:Nv \BNVS_if_resolve_init:n { VR }
2362 \BNVS_tl_use:nv { \regex_match:nnTF { [-,] } } { VR } {
2363   \__bnvs_gunset:nn { #1 } { #2 }
2364   \__bnvs_gset_unsolvable:nnn { #1 } { #2 } V
2365   \__bnvs_gset:nnnv { #1 } { #2 } R { VR }
2366   \BNVS_end:
2367   \prg_return_true:
2368 } {
2369   \__bnvs_tl_if_empty:cTF { VR } {
2370     \__bnvs_gunset:nnn { #1 } { #2 } V
2371     \__bnvs_gunset:nnn { #1 } { #2 } R
2372     \BNVS_end:
2373     \prg_return_false:
2374   } {
2375     \__bnvs_gunset:nn { #1 } { #2 }
2376     \__bnvs_gset:nnnv { #1 } { #2 } V { VR }
2377     \__bnvs_gset_unsolvable:nnn { #1 } { #2 } R
2378     \BNVS_end:
2379     \prg_return_true:
2380   }
2381 } {
2382 } {
2383   \BNVS_end:
2384   \prg_return_false:
2385 }
2386 }

```

6.16.3 V for value

| | |
|--|---|
| <code>__bnvs_if_resolve_V:nncTF</code> | <code>__bnvs_if_resolve_V:nncTF {<id>}{<tag>}{<ans>}{<yes code>}{<no code>}</code> |
| <code>__bnvs_if_resolve_V:nvcTF</code> | <code>__bnvs_if_append_V:nncTF {<id>}{<tag>}{<ans>}{<yes code>}{<no code>}</code> |
| <code>__bnvs_if_append_V:nncTF</code> | |
| <code>__bnvs_if_append_V:(nxc nvc)TF</code> | |

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

If a range is associated to $\langle id \rangle!$ $\langle tag \rangle$, $\langle no\ code \rangle$ is executed.

```

2387 \BNVS_new_conditional:cpnn { if_resolve_V:nnc } #1 #2 #3 { T, F, TF } {
2388   \__bnvs_if_get:nnncTF { #1 } { #2 } V { #3 } {
2389     \__bnvs_if_unresolvable:cTF { #3 } {

```

We already tried to compute but failed.

```

2390     \prg_return_false:
2391   } {
2392     \__bnvs_is_will_gset:cTF { #3 } {
2393       \BNVS_error:n { Circular-definition:~#1!#2/V (Error~recovery~1) }
2394       \__bnvs_gunset:nn { #1 } { #2 }
2395       \__bnvs_gset:nnnn { #1 } { #2 } 1 1
2396       \__bnvs_gset:nnnn { #1 } { #2 } V 1
2397       \__bnvs_tl_set:cn { #3 } 1
2398     \prg_return_true:

```

Circular definition call during resolution.

```

2399   } {
2400     \prg_return_true:
2401   }
2402 }
2403 } {
2404   \__bnvs_is_provided:nnnTF { #1 } { #2 } R {
2405     \__bnvs_set_unresolvable:nnn { #1 } { #2 } V
2406     \prg_return_false:
2407   } {
2408     \__bnvs_if_resolve_init:nnTF { #1 } { #2 } {
2409       \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
2410         \prg_return_true:
2411       } {
2412         \prg_return_false:
2413       }
2414     } {
2415       \prg_return_false:
2416     }
2417   }
2418 }
2419 }
2420 \BNVS_new_conditional_vvc:cn { if_resolve_V } { T, F, TF }
2421 \BNVS_new_conditional_cpn { if_append_V:nnc } #1 #2 #3 { T, F, TF } {
2422   \BNVS_begin:
2423     \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
2424       \BNVS_end_tl_put_right:cv { #3 } { #3 }
2425       \prg_return_true:
2426     } {
2427       \BNVS_end:
2428       \prg_return_false:
2429     }
2430   }
2431   \BNVS_new_conditional_vvc:cn { if_append_V } { T, F, TF }

```


6.16.4 R for range

```

2432 \__bnvs_if_resolve_R:nncTF \__bnvs_if_resolve_R:nncTF {<id>} {<tag>} {<ans>} {<yes code>} {<no code>}

```

Auxiliary function only called by __bnvs_if_resolve_R:nncTF.

```

2432 \tl_new:N \l__bnvs_if_resolve_R_nnc_tl
2433 \BNVS_new_conditional:cpnn { if_resolve_R:nnc } #1 #2 #3 { T, F, TF } {
2434   \__bnvs_if_get:nnncTF { #1 } { #2 } R { if_resolve_R_nnc } {
2435     \__bnvs_if_unresolvable:cTF { if_resolve_R_nnc } {

```

We already tried to compute but failed.

```

2436   \prg_return_false:
2437 } {
2438   \__bnvs_is_will_gset:cTF { if_resolve_R_nnc } {
2439     \BNVS_error:n { Circular-definition:~#1!#2/r (Error~recovery~1) }
2440     \__bnvs_gset:nnnn { #1 } { #2 } R 1,
2441     \__bnvs_gset:nnnn { #1 } { #2 } A 1
2442     \__bnvs_gset:nnnn { #1 } { #2 } Z 1
2443     \__bnvs_gset:nnnn { #1 } { #2 } L 1
2444     \__bnvs_tl_set:cn { #3 } { 1, }
2445   \prg_return_true:

```

Circular definition call during resolution.

```

2446   } {
2447     \__bnvs_tl_set:cv { #3 } { if_resolve_R_nnc }
2448     \prg_return_true:
2449   }
2450 } {
2451 } {
2452   \__bnvs_is_provided:nnnTF { #1 } { #2 } V {
2453     \__bnvs_set_unresolvable:nnn { #1 } { #2 } R
2454     \prg_return_false:
2455   } {
2456     \__bnvs_if_resolve_init:nnTF { #1 } { #2 } {
2457       \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
2458         \prg_return_true:
2459       } {
2460         \prg_return_false:
2461       }
2462     } {
2463       \prg_return_false:
2464     }
2465   }
2466 } {
2467 }

2468 \BNVS_new_conditional_vvc:cn { if_resolve_R } { T, F, TF }

```

6.16.5 Already complete

| | |
|--------------------------------|---|
| <u>_bnvs_if_resolve:nncTF</u> | _bnvs_if_resolve:nncTF {<id> } {<tag> } <ans> {<yes code> } {<no code> } |
| <u>_bnvs_if_append:nncTF</u> | _bnvs_if_append:nncTF {<id> } {<tag> } <ans> {<yes code> } {<no code> } |

If resolution has already complete as value or range for <id> !<tag>, put the result into or to the right of the ans tl variable and executes <yes code>. Otherwise execute <no code>.

Next fonction is only called by _bnvs_if_resolve:nncTF when

```

2469 \tl_new:N \l__bnvs_if_resolve_nnc_tl
2470 \BNVS_new_conditional:cpnn { if_resolve:nnc } #1 #2 #3 { T, F, TF } {
2471   \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { if_resolve_nnc } {
2472     \__bnvs_tl_set:cv { #3 } { if_resolve_nnc }
2473     \prg_return_true:
2474   } {
2475     \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { if_resolve_nnc } {
2476       \BNVS_set:cpn { resolution_round_ans: } { }
2477       \__bnvs_tl_set:cv { #3 } { if_resolve_nnc }
2478       \prg_return_true:
2479     } {
2480       \prg_return_false:
2481     }
2482   }
2483 }

2484 \BNVS_new_conditional:cpnn { if_append:nnc } #1 #2 #3 { T, F, TF } {
2485   \BNVS_group_begin:
2486   \__bnvs_if_resolve:nncTF { #1 } { #2 } { #3 } {
2487     \BNVS_end_tl_put_right:cv { #3 } { #3 }
2488     \prg_return_true:
2489   } {
2490     \prg_return_false:
2491   }
2492 }
```

6.16.6 Assignment

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

```

2493 \BNVS_new_conditional:cpnn { if_assign_value:nnn } #1 #2 #3 { T, F, TF } {
2494   \BNVS_begin:
2495   \__bnvs_set_true:c { no_range }
2496   \__bnvs_if_resolve:ncTF { #3 } a {
2497     \__bnvs_gclear:nn { #1 } { #2 }
2498     \__bnvs_gset:nnnv { #1 } { #2 } V a
2499   \BNVS_end:
2500   \prg_return_true:
2501 } {
```

```

2502     \BNVS_end:
2503     \prg_return_false:
2504   }
2505 }

2506 \BNVS_new_conditional:cpnn { if_assign_value:nnv } #1 #2 #3 { T, F, TF } {
2507   \BNVS_tl_use:nv {
2508     \__bnvs_if_assign_value:nnnTF { #1 } { #2 }
2509   } { #3 } {
2510     \prg_return_true:
2511   } {
2512     \prg_return_false:
2513   }
2514 }

2515 \BNVS_new_conditional:cpnn { if_assign_value:vvv } #1 #2 #3 { T, F, TF } {
2516   \BNVS_tl_use:nnv {
2517     \BNVS_tl_use:Nv \__bnvs_if_assign_value:nnnTF { #1 }
2518   } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
2519 }

```

6.16.7 beamer counters

```

\__bnvs_if_resolve_counter:ncTF \__bnvs_if_resolve_counter:ncTF {<tag>} <ans> {<yes code>} {<no code>}
\__bnvs_if_resolve_counter:vcTF \__bnvs_if_append_special:ncTF {<tag>} <ans> {<yes code>} {<no code>}
\__bnvs_if_append_special:ncTF
\__bnvs_if_append_special:vcTF

```

When `<tag>` is `pauses` or `slideinframe`, resolves the value into the `ans` `tl` variable and executes `<yes code>`, otherwise executes `<no code>`.

```

2520 \makeatletter
2521 \BNVS_new_conditional:cpnn { if_resolve_counter:nc } #1 #2 { T, F, TF } {
2522   \tl_if_eq:nnTF { #1 } { pauses } {
2523     \cs_if_exist:NTF \c@beamerpauses {
2524       \exp_args:Nnx \__bnvs_tl_set:cn { #2 } { \the\c@beamerpauses }
2525       \prg_return_true:
2526     } {
2527       \prg_return_false:
2528     }
2529   } {
2530     \tl_if_eq:nnTF { #1 } { slideinframe } {
2531       \cs_if_exist:NTF \beamer@slideinframe {
2532         \exp_args:Nnx \__bnvs_tl_set:cn { #2 } { \beamer@slideinframe }
2533         \prg_return_true:
2534       } {
2535         \prg_return_false:
2536       }
2537     } {
2538       \prg_return_false:
2539     }
2540   }
2541 }
2542 \makeatother

```

```

__bnvs_if_set:ncccTF \__bnvs_if_set:ncccTF {<A>} {<Z>} {<L>} {<query>} {<yes code>} {<no code>}

```

If $\langle query \rangle$ is a range (with colons), put its components into tl variables named $\langle A \rangle$, $\langle Z \rangle$ and $\langle L \rangle$ (names are evident) then execute $\langle yes\ code \rangle$. Otherwise execute $\langle no\ code \rangle$.

```

2543 \BNVS_new_conditional:cpnn { if_set:nccc } #1 #2 #3 #4 { T, F, TF } {
2544   \__bnvs_if_regex_split:cnTF { colons } { #1 } {
2545     \BNVS_begin:
2546     \tl_map_inline:nn { AZL } {
2547       \__bnvs_tl_clear:c ##1
2548     }
2549     \__bnvs_split_if_pop_left:cT A {

```

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

```

2550   \__bnvs_split_if_pop_left:cT Z {
2551     \__bnvs_tl_if_empty:cTF Z {

```

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

```

2552     \__bnvs_split_if_pop_left:cT Z {
2553       \__bnvs_seq_pop_left:ccT { split } L {

```

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

```

2554       \__bnvs_tl_if_empty:cTF L {
2555         \BNVS_error:n { Invalid-range-expression(1)/#1 }

```

A :: was expected:

```

2556       } {
2557         \int_compare:nNnT { \__bnvs_tl_count:c L } > { 1 } {
2558           \BNVS_error:n { Too-many-colons(1)/#1 }

```

A :: was expected:

```

2559       }
2560       \__bnvs_split_if_pop_left:cT L {

```

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

```

2561         \__bnvs_seq_if_empty:cF { split } {
2562           \BNVS_error:n { Invalid-range-expression(2)/#1 }

```

No more material expected.

```

2563       }
2564     }
2565   }
2566 }
2567 }
2568 } {

```

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

```

2569   \int_compare:nNnT { \__bnvs_tl_count:c Z } > { 1 } {
2570     \BNVS_warning:n { Too-many-colons(2)/#1 }
2571   }
2572   \__bnvs_split_if_pop_left:cT L {

```

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

```

2573     \__bnvs_split_if_pop_left:cTF Z {
2574       \__bnvs_tl_if_empty:cF Z {
2575         \BNVS_error:n { Too-many-colons(3)/#1 }

```

```

2576         }
2577         \__bnvs_split_if_pop_left:cT Z {
Z may contain the  $\langle last \rangle$ .
2578         \__bnvs_seq_if_empty:cF { split } {
2579         \BNVS_error:n { Invalid~range~expression(3)/#1 }
2580         }
2581     }
2582 } {
2583     \__bnvs_tl_clear:c Z
2584 }
2585 }
2586 }
2587 }
2588 }

```

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 $\langle first \rangle$.

```

2589 \__bnvs_tl_if_empty:cT A {
2590 \__bnvs_tl_if_empty:cTF Z {
2591 \__bnvs_tl_set:cn A 1
2592 } {
2593 \__bnvs_tl_if_empty:cT L {
2594 \__bnvs_tl_set:cn A 1
2595 }
2596 }
2597 }
2598 \cs_set:Npn \BNVS_if_set_ncccTF:w ##1 ##2 ##3 {
2599 \BNVS_end:
2600 \__bnvs_tl_set:cn { #2 } { ##1 }
2601 \__bnvs_tl_set:cn { #3 } { ##2 }
2602 \__bnvs_tl_set:cn { #4 } { ##3 }
2603 }
2604 \BNVS_tl_use:Nvvv \BNVS_if_set_ncccTF:w A Z L
2605 \prg_return_true:
2606 } {
2607 \prg_return_false:
2608 }
2609 }

```

__bnvs_if_resolve:nnnnTF __bnvs_if_resolve:nnnnTF { $\langle query \rangle$ } { $\langle A \rangle$ } { $\langle Z \rangle$ } { $\langle L \rangle$ } { $\langle yes code \rangle$ } { $\langle no code \rangle$ }

Calls __bnvs_if_set:ncccTF and resolves each variable.

```

2610 \BNVS_new:cpn { will_gset_azl: } {
2611 \__bnvs_tl_if_empty:cT a {
2612 \__bnvs_tl_if_empty:cTF z {
2613 \__bnvs_tl_set:cn a 1
2614 } {
2615 \__bnvs_tl_if_empty:cTF l {
2616 \__bnvs_tl_set:cn a 1
2617 } {
2618 \__bnvs_tl_set:cn a { max ( 0,
2619 \l__bnvs_z_tl - \l__bnvs_l_tl
2620 ) + 1 }

```

```

2621         \__bnvs_round:c a
2622     }
2623 }
2624 }
2625 }
2626 \__bnvs_tl_if_empty:cT z {
2627     \__bnvs_tl_if_empty:cF l {
2628         \__bnvs_tl_set:cn z {
2629             \l__bnvs_a_tl + \l__bnvs_l_tl - 1
2630         }
2631         \__bnvs_round:c z
2632     }
2633 }

```

a and z are properly set, l is unused afterwards.

```

2634 }

2635 \BNVS_new:cpn { gset_azl:nn } #1 #2 {
2636     \__bnvs_will_gset_azl:
2637     \tl_map_inline:nn { AZL } {
2638         \__bnvs_tl_clear:c ##1
2639     }
2640     \__bnvs_tl_if_empty:cTF z {
2641         \__bnvs_if_get:nncTF { #1 } { #2 } A {
2642             \int_compare:nNnTF \l__bnvs_a_tl < \l__bnvs_A_tl {
2643                 \__bnvs_tl_set:cv A a
2644             } {
2645             }
2646         } {
2647             \__bnvs_tl_set:cv A a
2648         }
2649     } {
2650         \int_compare:nNnF \l__bnvs_z_tl < \l__bnvs_a_tl {

```

When the a:z::l range is empty, nothing is done for the AZL storage.

```

2651         \__bnvs_if_get:nncTF { #1 } { #2 } A {
2652             \int_compare:nNnT \l__bnvs_a_tl < \l__bnvs_A_tl {

```

A will diminish.

```

2653         \__bnvs_tl_set:cv A a
2654     }
2655     \__bnvs_require:nnc { #1 } { #2 } Z
2656     \__bnvs_tl_if_empty:cTF Z {
2657     } {
2658         \int_compare:nNnTF \l__bnvs_z_tl > \l__bnvs_Z_tl {
2659             \__bnvs_tl_set:cv Z z
2660         } {
2661         }
2662     }
2663 } {

```

```

2664     \__bnvs_tl_set:cv A a
2665     \__bnvs_tl_set:cv Z z
2666   }
2667 }
2668 }
2669 \tl_if_empty:cF A {
2670   \tl_map_inline:nn { AZ } {
2671     \__bnvs_gset:nnv { #1 } { #2 } ##1
2672   }
2673   \__bnvs_tl_if_empty:cTF Z {
2674     \__bnvs_gset:nnnn { #1 } { #2 } L {}
2675   } {
2676     \__bnvs_tl_set:cn L {
2677       \l__bnvs_Z_tl - \l__bnvs_A_tl + 1
2678     }
2679     \__bnvs_round:c L
2680     \__bnvs_gset:nnv { #1 } { #2 } L
2681   }
2682 }
2683 }

2684 \BNVS_new_conditional:cpnn { if_append:nnnc } #1 #2 #3 #4 { T, F, TF } {
2685   \BNVS_begin:
2686   \__bnvs_if_set:ncccTF { #3 } azl {
2687     \tl_map_inline:nn { azl } {
2688       \__bnvs_tl_if_empty:cF ##1 {
2689         \__bnvs_if_resolve:vcTF ##1 ##1 {
2690           \BNVS_tl_use:Nv \regex_match:nnT { [-,] } ##1 {
2691             \BNVS_warning:n { Unexpected~range~. }
2692             \__bnvs_tl_set:cn ##1 0
2693           }
2694         } {
2695           \BNVS_warning:n { Something~got~wrong. }
2696           \__bnvs_tl_set:cn ##1 0
2697         }
2698       }
2699     }
2700     \__bnvs_gset_azl:nn { #1 } { #2 }
2701     \__bnvs_tl_set:cv r a
2702     \__bnvs_tl_if_empty:cTF z {
2703       \__bnvs_tl_put_right:cn r { - }
2704     } {
2705       \__bnvs_tl_if_eq:cvF a z {
2706         \__bnvs_tl_put_right:cn r { - }
2707         \__bnvs_tl_put_right:cv r z
2708       }
2709     }
2710
2711     \__bnvs_tl_if_empty:cF { #4 } {
2712       \__bnvs_tl_put_right:cn { #4 } { , }
2713     }
2714     \BNVS_end_tl_put_right:cv { #4 } r
2715     \prg_return_true:
2716   } {

```

```

2717     \BNVS_end:
2718     \prg_return_false:
2719   }
2720 }

```

```

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

```

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

```

2721 \BNVS_new:cpn { if_resolve_return:cnnc } #1 #2 #3 #4 {
2722   \t1_if_empty:nTF { #2 } {
2723     \prg_return_false:
2724   } {
2725     \__bnvs_use:c { if_resolve_#1:nncTF } {} { #3 } { #4 } {
2726       \prg_return_true:
2727     } {
2728       \prg_return_false:
2729     }
2730   }
2731 }

2732 \BNVS_new_conditional:cpnn { if_resolve_first:nnc } #1 #2 #3 { T, F, TF } {
2733   \__bnvs_if_resolve_V:nncTF { #1 } { #2.first } { #3 } {
2734     \prg_return_true:
2735   } {
2736     \__bnvs_if_resolved:nncTF { #1 } { #2 } A { #3 } {
2737       \prg_return_true:
2738     } {
2739       \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
2740         \__bnvs_require:nnc { #1 } { #2 } A { #3 }
2741         \prg_return_true:
2742       } {
2743         \prg_return_false:
2744       }
2745     }
2746   }
2747 }
2748 \BNVS_new_conditional_vvc:cn { if_resolve_first } { T, F, TF }

2749 \BNVS_new_conditional:cpnn { if_append_first:nnc } #1 #2 #3 { T, F, TF } {
2750   \BNVS_begin:
2751   \__bnvs_if_resolve_first:nncTF { #1 } { #2 } { #3 } {
2752     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2753     \prg_return_true:
2754   } {
2755     \BNVS_end:
2756     \prg_return_false:
2757   }
2758 }

```



```
2759 \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 $\langle yes\ code \rangle$ when there is a $\langle last \rangle$ index, $\langle no\ code \rangle$ otherwise. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only.

```
2760 \BNVS_new_conditional:cpnn { if_resolve_last:nnc } #1 #2 #3 { T, F, TF } {  
2761 \_bnvs_if_resolve_V:nncTF { #1 } { #2.last } { #3 } {  
2762 \prg_return_true:  
2763 } {  
2764 \_bnvs_if_resolved:nnncTF { #1 } { #2 } Z { #3 } {  
2765 \prg_return_true:  
2766 } {  
2767 \_bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {  
2768 \_bnvs_require:nnnc { #1 } { #2 } Z { #3 }  
2769 \prg_return_true:  
2770 } {  
2771 \prg_return_false:  
2772 }  
2773 }  
2774 }  
2775 }  
  
2776 \BNVS_new_conditional_vvc:cn { if_resolve_last } { T, F, TF }  
  
2777 \BNVS_new_conditional:cpnn { if_append_last:nnc } #1 #2 #3 { T, F, TF } {  
2778 \BNVS_begin:  
2779 \_bnvs_if_resolve_last:nncTF { #1 } { #2 } { #3 } {  
2780 \BNVS_end_t1_put_right:cv { #3 } { #3 }  
2781 \prg_return_true:  
2782 } {  
2783 \BNVS_end:  
2784 \prg_return_false:  
2785 }  
2786 }  
  
2787 \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 $\langle yes\ code \rangle$ when there is a $\langle last \rangle$ index, $\langle no\ code \rangle$ otherwise. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only.

```

2788 \BNVS_new_conditional:cpnn { if_resolve_length:nnc } #1 #2 #3 { T, F, TF } {
2789   \_bnvs_if_resolve_V:nncTF { #1 } { #2.length } { #3 } {
2790     \prg_return_true:
2791   } {
2792     \_bnvs_if_resolved:nnncTF { #1 } { #2 } L { #3 } {
2793       \prg_return_true:
2794     } {
2795       \_bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
2796         \_bnvs_require:nnnc { #1 } { #2 } L { #3 }
2797         \prg_return_true:
2798       } {
2799         \prg_return_false:
2800       }
2801     }
2802   }
2803 }
2804 \BNVS_new_conditional_vvc:cn { if_resolve_length } { T, F, TF }

2805 \BNVS_new_conditional:cpnn { if_append_length:nnc } #1 #2 #3 { T, F, TF } {
2806   \BNVS_begin:
2807   \_bnvs_if_resolve_length:nncTF { #1 } { #2 } { #3 } {
2808     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2809     \prg_return_true:
2810   } {
2811     \BNVS_end:
2812     \prg_return_false:
2813   }
2814 }
2815 \BNVS_new_conditional_vvc:cn { if_append_length } { 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>* t1 variable, or append this index to that variable. Execute *<yes code>* when there is a *<next>* index, *<no code>* otherwise. In the latter case, the *<t1 variable>* is undefined on resolution only.

```

2816 \BNVS_new_conditional:cpnn
2817   { if_resolve_previous:nnc } #1 #2 #3 { T, F, TF } {
2818   \_bnvs_if_resolve_V:nncTF { #1 } { #2.previous } { #3 } {
2819     \prg_return_true:
2820   } {
2821     \_bnvs_if_resolve_first:nncTF { #1 } { #2 } { #3 } {
2822       \_bnvs_t1_put_right:cn { #3 } { -1 }
2823       \_bnvs_round:c { #3 }
2824       \_bnvs_gset:nnnv { #1 } { #2 } P { #3 }
2825       \prg_return_true:
2826     } {
2827       \prg_return_false:
2828     }
2829   }
2830 }

```

```

2831 \BNVS_new_conditional_vvc:cn { if_resolve_previous } { T, F, TF }

2832 \BNVS_new_conditional:cpnn { if_append_previous:nnc } #1 #2 #3 { T, F, TF } {
2833   \BNVS_begin:
2834   \__bnvs_if_resolve_previous:nncTF { #1 } { #2 } { #3 } {
2835     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2836     \prg_return_true:
2837   } {
2838     \BNVS_end:
2839     \prg_return_false:
2840   }
2841 }

2842 \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 $\langle yes\ code \rangle$ when there is a $\langle next \rangle$ index, $\langle no\ code \rangle$ otherwise. In the latter case, the content of the $\langle ans \rangle$ t1 variable is undefined, on resolution only.

```

2843 \BNVS_new_conditional:cpnn { if_resolve_next:nnc } #1 #2 #3 { T, F, TF } {
2844   \__bnvs_if_resolve_V:nncTF { #1 } { #2.next } { #3 } {
2845     \prg_return_true:
2846   } {
2847     \__bnvs_if_resolve_last:nncTF { #1 } { #2 } { #3 } {
2848       \__bnvs_t1_put_right:cn { #3 } { +1 }
2849       \__bnvs_round:c { #3 }
2850       \prg_return_true:
2851     } {
2852       \prg_return_false:
2853     }
2854   }
2855 }

2856 \BNVS_new_conditional_vvc:cn { if_resolve_next } { T, F, TF }

2857 \BNVS_new_conditional:cpnn { if_append_next:nnc } #1 #2 #3 { T, F, TF } {
2858   \BNVS_begin:
2859   \__bnvs_if_resolve_next:nncTF { #1 } { #2 } { #3 } {
2860     \BNVS_end_t1_put_right:cv { #3 } { #3 }
2861     \prg_return_true:
2862   } {
2863     \BNVS_end:
2864     \prg_return_true:
2865   }
2866 }

2867 \BNVS_new_conditional_vvc:cn { if_append_next } { T, F, TF }

```

```

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

```

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

```

2868 \BNVS_new_conditional:cpnn { index_can:nn } #1 #2 { T, F, TF } {
2869   \__bnvs_is_gset:nnnTF { #1 } { #2 } V {
2870     \prg_return_true:
2871   } {
2872     \__bnvs_is_gset:nnnTF { #1 } { #2 } A {
2873       \prg_return_true:
2874     } {
2875       \__bnvs_is_gset:nnnTF { #1 } { #2 } Z {
2876         \prg_return_true:
2877       } {
2878         \prg_return_false:
2879       }
2880     }
2881   }
2882 }

2883 \BNVS_new_conditional:cpnn { index_can:vv } #1 #2 { T, F, TF } {
2884   \BNVS_t1_use:Nvv \__bnvs_index_can:nnTF { #1 } { #2 }
2885   { \prg_return_true: } { \prg_return_false: }
2886 }

2887 \BNVS_new_conditional:cpnn { if_resolve_N:nnnc } #1 #2 #3 #4 { T, F, TF } {
2888   \__bnvs_if_resolve_V:nnncTF { #1 } { #2.#3 } { #4 } {
2889     \prg_return_true:
2890   } {
2891     \__bnvs_if_resolve_first:nnncTF { #1 } { #2 } { #4 } {
2892       \__bnvs_t1_put_right:cn { #4 } { + #3 - 1 }
2893       \__bnvs_round:c { #4 }
2894     } \prg_return_true:
2895   } {
2896     \__bnvs_if_resolve_V:nnncTF { #1 } { #2 } { #4 } {
2897       \__bnvs_t1_put_right:cn { #4 } { + #3 - 1 }
2898       \__bnvs_round:c { #4 }

```

Limited overlay set.

```

2899         \prg_return_true:
2900     } {
2901         \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
2902             \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
2903             \__bnvs_round:c { #4 }
2904             \prg_return_true:
2905         } {
2906             \prg_return_false:
2907         }
2908     }
2909 }
2910 }
2911 }

2912 \BNVS_new_conditional:cpnn { if_resolve_N:nnvc } #1 #2 #3 #4 { T, F, TF } {
2913     \BNVS_tl_use:nv {
2914         \__bnvs_if_resolve_N:nnncTF { #1 } { #2 }
2915     } { #3 } { #4 } {
2916         \prg_return_true:
2917     } {
2918         \prg_return_false:
2919     }
2920 }

2921 \BNVS_new_conditional:cpnn { if_resolve_N:vvvc } #1 #2 #3 #4 { T, F, TF } {
2922     \BNVS_tl_use:nvv {
2923         \BNVS_tl_use:Nv \__bnvs_if_resolve_N:nnncTF { #1 }
2924     } { #2 } { #3 } { #4 } {
2925         \prg_return_true:
2926     } {
2927         \prg_return_false:
2928     }
2929 }

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

2940 \BNVS_new_conditional:cpnn { if_append_N:vvvc } #1 #2 #3 #4 { T, F, TF } {
2941     \BNVS_tl_use:nvv {
2942         \BNVS_tl_use:Nv \__bnvs_if_append_N:nnncTF { #1 }
2943     } { #2 } { #3 } { #4 } {
2944         \prg_return_true:
2945     } {
2946         \prg_return_false:
2947     }
2948 }

```

6.17 Value counter

```

__bnvs_if_resolve_incr:nncTF __bnvs_if_resolve_incr:nnnTF {<id>} {<tag>} {<offset>} <ans> {<yes code>}
__bnvs_if_append_incr:nncTF {<no code>}
__bnvs_if_append_incr:vnncTF __bnvs_if_append_incr:nncTF {<id>} {<tag>} {<offset>} <ans> {<yes
code>} {<no code>}

```

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

```

2949 \BNVS_new_conditional:cpnn
2950 { if_resolve_incr:nnc } #1 #2 #3 #4 { T, F, TF } {
2951   __bnvs_if_resolve_query:ncTF { #3 } { #4 } {
2952     \BNVS_t1_use:Nv \int_compare:nNnTF { #4 } = 0 {
2953       __bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
2954         \prg_return_true:
2955       } {
2956         \prg_return_false:
2957       }
2958     } {
2959       __bnvs_t1_put_right:cn { #4 } { + }
2960       __bnvs_if_append_V:nncTF { #1 } { #2 } { #4 } {
2961         __bnvs_round:c { #4 }
2962         __bnvs_gset:nnv { #1 } { #2 } V { #4 }
2963         \prg_return_true:
2964       } {
2965         \prg_return_false:
2966       }
2967     }
2968   } {
2969     \prg_return_false:
2970   }
2971 }

2972 \BNVS_new_conditional:cpnn { if_append_incr:nnc } #1 #2 #3 #4 { T, F, TF } {
2973   \BNVS_begin:
2974   __bnvs_if_resolve_incr:nncTF { #1 } { #2 } { #3 } { #4 } {
2975     \BNVS_end_t1_put_right:cv { #4 } { #4 }
2976     \prg_return_true:
2977   } {
2978     \BNVS_end:
2979     \prg_return_false:
2980   }
2981 }
2982 \BNVS_new_conditional_vnnc:cn { if_append_incr } { T, F, TF }

2983 \BNVS_new_conditional:cpnn
2984 { if_resolve_post:nnc } #1 #2 #3 #4 { T, F, TF } {
2985   __bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
2986     \BNVS_begin:
2987     __bnvs_if_resolve_query:ncTF { #3 } a {
2988       \BNVS_t1_use:Nv \int_compare:nNnTF a = 0 {
2989         \BNVS_end:

```

```

2990         \prg_return_true:
2991     } {
2992         \__bnvs_tl_put_right:cn a +
2993         \__bnvs_tl_put_right:cv a { #4 }
2994         \__bnvs_round:c a
2995         \BNVS_end_tl_gset:nnv { #1 } { #2 } V a
2996         \prg_return_true:
2997     }
2998 } {
2999     \BNVS_end:
3000     \prg_return_false:
3001 }
3002 } {
3003     \prg_return_false:
3004 }
3005 }
3006 \BNVS_new_conditional_vvvc:cn { if_resolve_post } { T, F, TF }

3007 \BNVS_new_conditional:cpnn { if_append_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
3008     \BNVS_begin:
3009     \__bnvs_if_resolve_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
3010         \BNVS_end_tl_put_right:cv { #4 } { #4 }
3011         \prg_return_true:
3012     } {
3013         \prg_return_false:
3014     }
3015 }
3016 \BNVS_new_conditional_vvnc:cn { if_append_post } { T, F, TF }
3017 \BNVS_new_conditional_vvvc:cn { if_append_post } { T, F, TF }

```

6.18 Functions for the resolution

They manily start with `__bnvs_if_resolve_` or `__bnvs_split_`

| | |
|---|--|
| <code>__bnvs_split_pop_XP:TFF</code> | <code>__bnvs_split_pop_XP:TFF {<black code>} {<blank code>} {<end</code> |
| <code>__bnvs_split_if_pop_GNURO_or_end_return:T</code> | <code>code>}</code> |
| <code>__bnvs_split_pop_XPGNURO:T</code> | <code>__bnvs_split_if_pop_GNURO_or_end_return:T {<blank code>}</code> |
| | <code>__bnvs_split_pop_XPGNURO:T {<black code>}</code> |

For `__bnvs_split_pop_XP: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_if_pop_GNURO_or_end_return:T`: pops the four heading items of the `split` sequence into the five variables `G`, `N`, `U`, `R` and `O`. Then execute `<black code>`.

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

Next is called after a query has been splitted.

```

3018 \BNVS_new:cpn { split_pop_XP:TFF } #1 #2 #3 {

```

```

3019 \__bnvs_split_if_pop_left:cTF I {
3020   \cs_set:Npn \BNVS_split_F:n ##1 {
3021     \BNVS_fatal:n { split_pop_XP:TFF/##1 }
3022   }
3023   \__bnvs_split_if_pop_left:cTn K {
3024     \__bnvs_split_if_pop_left:cTn S {
3025       \__bnvs_split_if_pop_left:cTn P {
3026         \__bnvs_tl_if_blank:vTF S {

```

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

```

3027         #2
3028       } {
3029         \BNVS_tl_use:nv {
3030           \regex_match:NnT \c__bnvs_A_reserved_Z_regex
3031         } S {
3032           \__bnvs_tl_if_eq:cnF S { pauses } {
3033             \__bnvs_tl_if_eq:cnF S { slideinframe } {
3034 \BNVS_error:x { Use-of-reserved-``\__bnvs_tl_use:c T' ' }
3035           }
3036         }
3037       }
3038       \__bnvs_tl_if_blank:vTF K {
3039         \__bnvs_tl_set:cv I J
3040       } {
3041         \__bnvs_tl_set:cv J I
3042       }

```

Build the path sequence and lowercase components conditionals.

```

3043       \__bnvs_tl_if_empty:cTF P {
3044         \__bnvs_seq_clear:c P
3045       } {
3046         \__bnvs_seq_set_split:cnv P { . } P
3047       }
3048       #1
3049     }
3050   } P
3051 } S
3052 } K
3053 } {
3054   #3
3055 }
3056 }

```

```

3057 \BNVS_new:cpn { split_if_pop_GNURO_or_end_return:T } #1 {
3058   \__bnvs_split_if_pop_left:cT G {
3059     \__bnvs_split_if_pop_left:cT N {
3060       \__bnvs_split_if_pop_left:cT U {
3061         \__bnvs_split_if_pop_left:cT R {
3062           \__bnvs_split_if_pop_left:cT O {

```



```

3063         #1
3064     }
3065 }
3066 }
3067 }
3068 }
3069 }

```

```

\__bnvs_if_resolve_query:ncTF \__bnvs_if_resolve_query:ncTF {<query>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_resolve_query:vcTF \__bnvs_if_append_query:ncTF {<query>} {<ans>} {<yes code>} {<no code>}
\__bnvs_if_append:ncTF
\__bnvs_if_append:vcTF

```

Resolves the $\langle query \rangle$, 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 the function $\backslash_bnvs_if_resolve_query:ncTF$, 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 $\backslash l_bnvs_split_int$.)

```

3070 \BNVS_int_new:c { split }

```

$\backslash l_bnvs_T_tl$ Storage for `split` sequence items that represent names.

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

$\backslash l_bnvs_P_tl$ Storage for `split` sequence items that represent integer paths.

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

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 $\backslash \dots end_return \dots$ functions.

```

3071 \BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
3072   \BNVS_begin:
3073   \__bnvs_if_resolve_query:ncTF { #1 } { #2 } {
3074     \BNVS_end_tl_put_right:cv { #2 } { #2 }
3075     \prg_return_true:
3076   } {
3077     \BNVS_end:
3078     \prg_return_false:
3079   }
3080 }

```

```
3081 \BNVS_new_conditional_vc:cn { if_append } { T, F, TF }
```

Heavily used.

```
3082 \cs_new:Npn \BNVS_end_unreachable_return_false:n #1 {
3083   \BNVS_error:n { UNREACHABLE/#1 }
3084   \BNVS_end:
3085   \prg_return_false:
3086 }
3087 \cs_new:Npn \BNVS_end_unreachable_return_false:x #1 {
3088   \BNVS_error:x { UNREACHABLE/#1 }
3089   \BNVS_end:
3090   \prg_return_false:
3091 }
```

Naming convention: in the sequel, functions with a `or_end_return` in the name are called in the body of a conditional. They use `\prg_return_true:` or `\prg_return_false:`. They also end a \TeX group.

```
3092 \BNVS_new:cpn { split_pop_spec_or_end_return: } {
3093   \__bnvs_split_pop_XP:TFF {
3094     \__bnvs_split_pop_XPGNURO:T {
3095       \__bnvs_prepare_context:N \c_true_bool
3096       \__bnvs_build_T:
3097       \__bnvs_resolve_loop_or_end_return_iadd:n { 1 }
3098     }
3099   } {
3100     \__bnvs_split_pop_XP:TFF {
3101       \__bnvs_split_if_pop_GNURO_or_end_return:T {
3102         \__bnvs_tl_if_eq:cnTF N 0 {
3103           \__bnvs_build_T:
3104           \__bnvs_resolve_loop_N_or_end_return:
3105         } {
3106           \__bnvs_tl_if_empty:cTF N {
3107             \__bnvs_tl_if_blank:vTF U {
3108               \__bnvs_tl_if_blank:vTF R {
3109                 \__bnvs_tl_if_blank:vTF O {
3110                   \__bnvs_prepare_context:N \c_false_bool
3111                   \__bnvs_build_T:
3112                   \BNVS_tl_use:Nv
3113                   \__bnvs_if_resolve_counter:ncTF T V {
3114                     \__bnvs_tl_put_right:cv { ans } V
3115                     \__bnvs_resolve_loop_or_end_return:
3116                   } {
3117                     \__bnvs_if_resolve_V:vvTF IT V {
3118                       \__bnvs_tl_put_right:cv { ans } V
3119                       \__bnvs_resolve_loop_or_end_return:
3120                     } {

```

Not a beamer counter. Only the dotted path, branch according to the last component, if any.

```
3121   \__bnvs_if_resolve_R:vvTF IT r {
3122     \__bnvs_tl_put_right:cv { ans } r

```

```

3123         \__bnvs_resolve_loop_or_end_return:
3124     } {
3125         \__bnvs_tl_if_empty:cTF { suffix } {
3126             \__bnvs_resolve_loop_or_end_return_V:
3127         } {
3128             \__bnvs_resolve_loop_or_end_return_suffix:
3129         }
3130     }
3131 }
3132 }
3133 } {
3134     \__bnvs_prepare_context:N \c_true_bool
3135     \__bnvs_build_T:
3136     \BNVS_use:c { resolve_loop_or_end_return[...++]: }
3137 }
3138 } {
3139     \__bnvs_prepare_context:N \c_true_bool
3140     \__bnvs_build_T:
3141     \__bnvs_resolve_loop_R_or_end_return:
3142 }
3143 } {
3144     \__bnvs_if_resolve_query:vcTF R R {
3145         \__bnvs_prepare_context:N \c_true_bool
3146         \__bnvs_build_T:
3147         \BNVS_tl_use:Nv
3148         \__bnvs_resolve_loop_or_end_return_iadd:n R
3149     } {
3150         \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c R }
3151         \__bnvs_resolve_loop_or_end_return:
3152     }
3153 }
3154 } {
3155     \__bnvs_normalize_GN:
3156     \__bnvs_build_T:
3157     \__bnvs_resolve_loop_N_or_end_return:
3158 }
3159 }
3160 }
3161 } {
3162 \BNVS_end_unreachable_return_false:n { resolve_loop_or_end_return:/3 }
3163 } {
3164 \BNVS_end_unreachable_return_false:n { resolve_loop_or_end_return:/2 }
3165 }
3166 } {

```

The `split` sequence is empty.

```

3167     \__bnvs_if_resolve_end_return_true:
3168 }
3169 }

```

First function called after the query is splitted. The items of the split sequence between two overlay specifications must not be empty. Equivalently, after an empty item which is not an overlay specifications and which is not first, there must be no overlay specification.

```

3170 \BNVS_new:cpn { resolve_init_end_return: } {

```

```

3171 \__bnvs_split_if_pop_left:cTF a {
3172   \__bnvs_tl_put_right:cv { ans } a
3173   \__bnvs_split_pop_spec_or_end_return:
3174 } {
3175 \BNVS_end_unreachable_return_false:n { resolve_init_end_return:/1 }
3176 }
3177 }

```

Next is a main entry point.

```

3178 \tl_new:N \l__bnvs_if_resolved_A_tl
3179 \tl_new:N \l__bnvs_if_resolved_B_tl
3180 \BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { T, F, TF } {
3181   \__bnvs_if_call:TF {
3182     \BNVS_begin:

```

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

```

3183   \regex_match:nnTF { \c { BNVS_query:n } } { #1 } {
3184     \__bnvs_tl_clear:c { if_resolved_A }
3185     \cs_set:Npn \BNVS_if_resolve_return: {
3186       \BNVS_end_tl_put_right:cv { #2 } { if_resolved_A }
3187       \prg_return_true:
3188     }
3189     \cs_set:Npn \BNVS:w ##1 \BNVS_query:n ##2 ##3 \s_stop {
3190       \__bnvs_tl_put_right:cn { if_resolved_A } { ##1 }
3191       \__bnvs_if_append:ncF { ##2 } { if_resolved_A } {
3192         \cs_set:Npn \BNVS_if_resolve_return: {
3193           \prg_return_false:
3194         }
3195       }
3196       \regex_match:nnTF { \c { BNVS_query:n } } { ##3 } {
3197         \BNVS:w #1 \s_stop
3198       } {
3199         \__bnvs_if_resolve_flat:ncTF { ##3 } { if_resolved_B } {
3200           \__bnvs_tl_put_right:cv { if_resolved_A } { if_resolved_B }
3201         } {
3202           \cs_set:Npn \BNVS_if_resolve_return: {
3203             \prg_return_false:
3204           }
3205         }
3206       }
3207     }
3208     \BNVS:w #1 \s_stop
3209     \BNVS_if_resolve_return:
3210   } {
3211     \__bnvs_if_resolve_flat:ncTF { #1 } { if_resolved_A } {
3212       \BNVS_end_tl_put_right:cv { #2 } { if_resolved_A }
3213       \prg_return_true:
3214     } {
3215       \prg_return_false:
3216     }
3217   }
3218 } {
3219   \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
3220   \BNVS_end:

```

```

3221     \prg_return_false:
3222   }
3223 }

3224 \BNVS_new_conditional_vc:cn { if_resolve } { T, F, TF }

3225 \BNVS_new:cpn { build_T: } {
3226   \__bnvs_tl_set_eq:cc T S
3227   \__bnvs_seq_map_inline:cn P {
3228     \__bnvs_tl_put_right:cn T { . ##1 }
3229   }
3230 }

Build the tag from the S tl variable and the P_head sequence.

3231 \BNVS_new:cpn { build_tag_head: } {
3232   \__bnvs_tl_set_eq:cc T S
3233   \__bnvs_seq_map_inline:cn { P_head } {
3234     \__bnvs_tl_put_right:cn T { . ##1 }
3235   }
3236 }

```

__bnvs_resolve_loop_or_end_return: __bnvs_resolve_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.

```

3237 \tl_map_inline:nn {
3238   {n}{reset}{reset_all}{v}{first}{last}{length}
3239   {previous}{next}{range}{assign}{only}
3240 } {
3241   \bool_new:c { \BNVS_1:cn { #1 } { bool } }
3242 }

3243 \BNVS_new_conditional:cpnn { if:c } #1 { p, T, F, TF } {
3244   \bool_if:cTF { \BNVS_1:cn { #1 } { bool } } {
3245     \prg_return_true:
3246   } {
3247     \prg_return_false:
3248   }
3249 }

3250 \BNVS_new_conditional:cpnn { bool_if_exist:c } #1 { p, T, F, TF } {
3251   \bool_if_exist:cTF { \BNVS_1:cn { #1 } { bool } } {
3252     \prg_return_true:
3253   } {
3254     \prg_return_false:
3255   }
3256 }

```

```

3257 \BNVS_new:cpn { prepare_context:N } #1 {
3258   \clist_map_inline:nn {
3259     reset, reset_all, first, last, length,
3260     previous, next, range, assign, only
3261   } {
3262     \__bnvs_set_false:c { ##1 }
3263   }
3264   \__bnvs_seq_clear:c { P_head }
3265   \__bnvs_seq_clear:c { P_tail }
3266   \__bnvs_tl_clear:c N
3267   \__bnvs_tl_clear:c { suffix }
3268   \BNVS_set:cpn { prepare_context_N:n } ##1 {
3269     \tl_if_blank:nF { ##1 } {
3270       \__bnvs_tl_if_empty:cF N {
3271         \__bnvs_seq_put_right:cv { P_head } N
3272         \__bnvs_tl_clear:c N
3273       }
3274       \__bnvs_seq_put_right:cn { P_head } { ##1 }
3275     }
3276   }
3277   \__bnvs_seq_map_inline:cn P {
3278     \__bnvs_bool_if_exist:cTF { ##1 } {
3279       \__bnvs_set_true:c { ##1 }
3280       \clist_if_in:nnF { reset, reset_all } { ##1 } {
3281         \bool_if:NT #1 {
3282           \BNVS_error:n {Unexpected~##1~in~assignment }
3283         }
3284         \__bnvs_tl_set:cn { suffix } { ##1 }
3285       }
3286       \BNVS_set:cpn { prepare_context_N:n } ####1 {
3287         \tl_if_blank:nF { ####1 } {
3288           \BNVS_error:n {Unexpected~####1 }
3289         }
3290       }
3291     } {
3292       \regex_match:NnTF \c__bnvs_A_index_Z_regex { ##1 } {
3293         \__bnvs_tl_if_empty:cF N {
3294           \__bnvs_seq_put_right:cv { P_head } N
3295         }
3296         \__bnvs_tl_set:cn N { ##1 }
3297       } {
3298         \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { ##1 } {
3299           \BNVS_error:n { Unsupported~##1 }
3300         } {
3301           \__bnvs_prepare_context_N:n { ##1 }
3302         }
3303       }
3304     }
3305   }
3306   \__bnvs_seq_set_eq:cc P { P_head }
3307 }

3308 \BNVS_new:cpn { resolve_loop_or_end_return: } {

```

```

3309 \__bnvs_split_if_pop_left:cTF a {
3310 \__bnvs_tl_put_right:cv { ans } a
3311 \__bnvs_tl_if_empty:cTF a {

```

This is where the difference with \resolve_loop_or_end_return: lives.

```

3312 \__bnvs_seq_if_empty:cTF { split } {
3313 \__bnvs_if_resolve_end_return_true:
3314 } {
3315 \__bnvs_if_resolve_end_return_false:
3316 }
3317 } {
3318 \__bnvs_split_pop_spec_or_end_return:
3319 }
3320 } {
3321 \BNVS_end_unreachable_return_false:n { resolve_loop_or_end_return:/1 }
3322 }
3323 }

3324 \BNVS_new_conditional:cpnn { if_suffix: } { T, F, TF } {
3325 \__bnvs_tl_if_empty:cTF { suffix } {
3326 \__bnvs_seq_pop_right:ccTF P { suffix } {
3327 \prg_return_true:
3328 } {
3329 \prg_return_false:
3330 }
3331 } {
3332 \prg_return_true:
3333 }
3334 }

```

I and T tl variables are set beforehand. Implementation detail: tl variable a is used.

```

3335 \BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {
3336 \__bnvs_build_T:
3337 \__bnvs_tl_set:cx a {
3338 \__bnvs_tl_use:c T . \__bnvs_tl_use:c { suffix }
3339 }
3340 \__bnvs_if_resolve_v:vvTF I a a {
3341 \__bnvs_tl_put_right:cv { ans } a
3342 \__bnvs_resolve_loop_or_end_return:
3343 } {
3344 \__bnvs_if_resolve_V:vvTF I a a {
3345 \__bnvs_tl_put_right:cv { ans } a
3346 \__bnvs_resolve_loop_or_end_return:
3347 } {
3348 #1
3349 }
3350 }
3351 }

3352 \BNVS_new:cpn { path_branch_loop_or_end_return: } {
3353 \__bnvs_if_call:TF {
3354 \__bnvs_if_path_branch:TF {
3355 \__bnvs_path_branch_end_return:
3356 } {
3357 \__bnvs_if_get:vvTF I T V {

```

```

3358     \__bnvs_if_ISP:cccTF I V P {
3359         \__bnvs_tl_set_eq:cc T V
3360         \__bnvs_seq_merge:cc P { P_tail }
3361         \__bnvs_seq_clear:c { P_tail }
3362         \__bnvs_seq_set_eq:cc { P_head } P
3363         \__bnvs_path_branch_IT_loop_or_end_return:
3364     } {
3365         \__bnvs_path_branch_head_to_tail_end_return:
3366     }
3367 } {
3368     \__bnvs_path_branch_head_to_tail_end_return:
3369 }
3370 }
3371 } {
3372     \__bnvs_path_branch_end_return_false:n {
3373         Too-many-calls.
3374     }
3375 }
3376 }

3377 \BNVS_new:cpn { path_branch_end_return: } {
3378     \__bnvs_resolve_loop_or_end_return:
3379 }

3380 \BNVS_new:cpn { set_if_path_branch:n } {
3381     \prg_set_conditional:Npnn \__bnvs_if_path_branch: { TF }
3382 }

3383 \BNVS_new:cpn { path_branch_head_to_tail_end_return: } {
3384     \__bnvs_seq_pop_right:ccTF { P_head } a {
3385         \__bnvs_seq_put_left:cv { P_tail } a
3386         \__bnvs_build_tag_head:
3387         \__bnvs_path_branch_IT_loop_or_end_return:
3388     } {
3389         \__bnvs_build_T:
3390         \__bnvs_seq_set_eq:cc { P_head } { P_tail }
3391         \__bnvs_seq_clear:c { P_tail }
3392         \STOP
3393         \__bnvs_is_gset:vxnTF I { \__bnvs_tl_use:c T.1 } V {
3394             \__bnvs_tl_set:cn N 1
3395             \__bnvs_resolve_loop_N_or_end_return:
3396         } {
3397             \__bnvs_gset:vvnn I T V 1
3398             \__bnvs_path_branch_IT_loop_or_end_return:
3399         }
3400     }
3401 }

```

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

```

3402 \BNVS_new:cpn { path_branch_IT_loop_or_end_return: } {
3403     \__bnvs_build_tag_head:
3404     \__bnvs_if_resolve_V:vvctf I T V {

```



```

3405     \__bnvs_tl_put_right:cv { ans } V
3406     \__bnvs_resolve_loop_or_end_return:
3407   } {
3408     \__bnvs_path_branch_loop_or_end_return:
3409   }
3410 }

```

- Case*<index>*.

```

3411 \BNVS_new:cpn { resolve_loop_N_or_end_return: } {
3412   % known, id, tag, path, suffix
3413   \__bnvs_set_if_path_branch:n {
3414     \__bnvs_if_append_N:vvvcTF I T N { ans } {
3415       \prg_return_true:
3416     } {
3417       \prg_return_false:
3418     }
3419   }
3420   \__bnvs_path_branch_loop_or_end_return:
3421 }

```

```

3422 \BNVS_new:cpn { resolve_loop_reset: } {
3423   \__bnvs_if:cTF { reset_all } {
3424     \__bnvs_set_false:c { reset_all }
3425     \__bnvs_set_false:c { reset }
3426     \__bnvs_gunset:vvn I T v
3427   } {
3428     \__bnvs_if:cT { reset } {
3429       \__bnvs_set_false:c { reset }
3430       \__bnvs_gunset:vvn I T v
3431     }
3432   }
3433 }

```

- Case

```

3434 \BNVS_new:cpn { resolve_loop_or_end_return_V: } {
3435   \__bnvs_resolve_loop_reset:
3436   \__bnvs_set_if_path_branch:n {
3437     \__bnvs_if_append_V:vvvcTF I T { ans } {
3438       \prg_return_true:
3439     } {
3440       \__bnvs_if_append_V:vvvcTF I T { ans } {
3441         \prg_return_true:
3442       } {
3443         \prg_return_false:
3444       }
3445     }
3446   }
3447   \__bnvs_path_branch_loop_or_end_return:
3448 }

```

- Case*<suffix>*.

```

3449 \BNVS_new:cpn { resolve_loop_or_end_return_suffix: } {

```

```

3450  \__bnvs_if_resolve_V_loop_or_end_return_true:F {
3451    \__bnvs_set_if_path_branch:n {
3452      \BNVS_use:c {
3453        if_append_ \__bnvs_tl_use:c { suffix } :vvcTF
3454      } I T { ans } {
3455        \__bnvs_if:cT { range } {
3456          \BNVS_set:cpn { resolution_round_ans: } { }
3457        }
3458        \prg_return_true:
3459      } {
3460        \prg_return_false:
3461      }
3462    }
3463    \__bnvs_path_branch_loop_or_end_return:
3464  }
3465 }

• Case ...++.

3466 \BNVS_new:cpn { resolve_loop_or_end_return[...++]: } {
3467   \__bnvs_if:cTF { reset } {

• Case ....reset++.

3468   \cs_set:Npn \BNVS_resolve_loop: {
3469     NO~....reset++~for
3470     ~\__bnvs_tl_use:c I!\__bnvs_tl_use:c T
3471   }
3472 } {

• Case ...(.reset_all)++.

3473   \cs_set:Npn \BNVS_resolve_loop: {
3474     \BNVS_error_ans:x {
3475       NO~...(.reset_all)++~for
3476       ~\__bnvs_tl_use:c I!\__bnvs_tl_use:c T
3477     }
3478   }
3479 }
3480 \__bnvs_build_T:
3481 \__bnvs_resolve_loop_reset:
3482 \__bnvs_if_append_post:vvncTF I T { 1 } { ans } {
3483 } {
3484   \BNVS_error_ans:x {
3485     Problem~with~\__bnvs_tl_use:c I!\__bnvs_tl_use:c T~use.
3486   }
3487 }
3488 \__bnvs_resolve_loop_or_end_return:
3489 }

3490 \BNVS_new:cpn { resolve_loop_R_or_end_return: } {

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

3491   \__bnvs_if_resolve_query:vcTF R R {

```

```

3492 \__bnvs_is_gset:vvvF I T V {
3493 \__bnvs_gset:vvvv I T V R
3494 }
3495 \__bnvs_gset:vvvv I T V R
3496 \__bnvs_if_append_V:vvvTF I T { ans } {
3497 } {
3498 \BNVS_error_ans:n { No~...=... }
3499 }
3500 } {
3501 \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c R. }
3502 }
3503 \__bnvs_resolve_loop_or_end_return:
3504 }

• Case ...+=....

3505 \BNVS_new:cpn { resolve_loop_or_end_return_iadd:n } #1 {
3506 \__bnvs_if_resolve_query:ncTF { #1 } { rhs } {
3507 \__bnvs_resolve_loop_reset:
3508 \__bnvs_if_append_incr:vvvncTF I T { #1 } { ans } {
3509 } {
3510 \BNVS_error_ans:n { No~...+=... }
3511 }
3512 } {
3513 \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c { rhs } }
3514 }
3515 \__bnvs_resolve_loop_or_end_return:
3516 }

```

6.18.1 Resolve one query

`__bnvs_if_resolve_query:ncTF` `__bnvs_if_resolve_query:ncTF {<query>} {<ans>} {<yes code>} {<no code>}`

Evaluates the single `<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 the `<ans>` `tl` variable. Ranges are supported with the colon syntax. This is executed within a local `TeX` group managed by the caller. Below are local variables and constants.

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

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

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

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

```

3517 \BNVS_new:cpn { resolve_query_end_return_true: } {
3518     \BNVS_end:
3519     \prg_return_true:
3520 }

3521 \BNVS_new:cpn { resolve_query_end_return_false: } {
3522     \BNVS_end:
3523     \prg_return_false:
3524 }

3525 \BNVS_new:cpn { resolve_query_end_return_false:n } #1 {
3526     \BNVS_end:
3527     \prg_return_false:
3528 }

3529 \BNVS_new:cpn { if_resolve_query_return_false:n } #1 {
3530     \prg_return_false:
3531 }

3532 \BNVS_new:cpn { resolve_query_error_return_false:n } #1 {
3533     \BNVS_error:n { #1 }
3534     \__bnvs_if_resolve_query_return_false:
3535 }
3536 \BNVS_generate_variant:cn { resolve_query_error_return_false:n } { x }

3537 \BNVS_new:cpn { if_resolve_query_return_unreachable: } {
3538     \__bnvs_resolve_query_error_return_false:n { UNREACHABLE }
3539 }

3540 \BNVS_new:cpn { if_blank:cTF } #1 {
3541     \BNVS_tl_use:Nc \tl_if_blank:VTF { #1 }
3542 }

```

__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_VAZLLZZL_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.

```

3543 \BNVS_new_conditional:cpnn { if_resolve_query_branch: } { T, F, TF } {

```

At start, we ignore the whole match.

```

3544     \__bnvs_match_if_pop_left:cT V {

```

```

3545 \__bnvs_match_if_pop_left:cT V {
3546 \__bnvs_if_blank:cTF V {
3547 \__bnvs_match_if_pop_left:cT A {
3548 \__bnvs_match_if_pop_left:cT Z {
3549 \__bnvs_match_if_pop_left:cT L {
3550 \__bnvs_if_blank:cTF A {
3551 \__bnvs_match_if_pop_left:cT L {
3552 \__bnvs_match_if_pop_left:cT Z {
3553 \__bnvs_if_blank:cTF L {
3554 \__bnvs_match_if_pop_left:cT Z {
3555 \__bnvs_match_if_pop_left:cT L {
3556 \__bnvs_if_blank:cTF L {
3557 \BNVS_use:c { if_resolve_query_return[:Z]: }
3558 } {
3559 \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3560 }
3561 }
3562 }
3563 } {
3564 \__bnvs_if_blank:cTF Z {
3565 \__bnvs_resolve_query_error_return_false:n { Missing-first~or-last }
3566 } {
3567 \BNVS_use:c { if_resolve_query_return[:Z::L]: }
3568 }
3569 }
3570 }
3571 }
3572 } {
3573 \__bnvs_if_blank:cTF Z {
3574 \__bnvs_if_blank:cTF L {
3575 \BNVS_use:c { if_resolve_query_return[A:]: }
3576 } {
3577 \BNVS_use:c { if_resolve_query_return[A::L]: }
3578 }
3579 } {
3580 \__bnvs_if_blank:cTF L {
3581 \BNVS_use:c { if_resolve_query_return[A:Z]: }
3582 } {
3583 \__bnvs_if_resolve_query_return_unreachable:
3584 }
3585 }
3586 }
3587 }
3588 }
3589 }
3590 } {
3591 \BNVS_use:c { if_resolve_query_return[V]: }
3592 }
3593 }
3594 }
3595 }

```

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

Single value

```

3596 \BNVS_new:cpn { if_resolve_query_return[V]: } {
3597   \__bnvs_if_resolve:vcTF V { ans } {
3598     \prg_return_true:
3599   } {
3600     \prg_return_false:
3601   }
3602 }

🔹 <first>:<last> range
3603 \BNVS_new:cpn { if_resolve_query_return[A:Z]: } {
3604   \__bnvs_if_resolve_query:vcTF { A } { ans } {
3605     \__bnvs_tl_put_right:cn { ans } { - }
3606     \__bnvs_if_append:vcTF { Z } { ans } {
3607       \prg_return_true:
3608     } {
3609       \prg_return_false:
3610     }
3611   } {
3612     \prg_return_false:
3613   }
3614 }

🔹 <first>::<length> range
3615 \BNVS_new:cpn { if_resolve_query_return[A::L]: } {
3616   \__bnvs_if_resolve_query:vcTF A A {
3617     \__bnvs_if_resolve_query:vcTF L { ans } {
3618       \__bnvs_tl_put_right:cn { ans } { + \l__bnvs_A_tl - 1 }
3619       \__bnvs_round:c { ans }
3620       \__bnvs_tl_put_left:cn { ans } -
3621       \__bnvs_tl_put_left:cv { ans } A
3622     \prg_return_true:
3623   } {
3624     \prg_return_false:
3625   }
3626 } {
3627   \prg_return_false:
3628 }
3629 }

🔹 <first>: and <first>:: range
3630 \BNVS_new:cpn { if_resolve_query_return[A:]: } {
3631   \__bnvs_if_resolve_query:vcTF A { ans } {
3632     \__bnvs_tl_put_right:cn { ans } -
3633     \prg_return_true:
3634   } {
3635     \prg_return_false:
3636   }
3637 }

🔹 :<last>::<length> or ::<length>:<last> range
3638 \BNVS_new:cpn { if_resolve_query_return[:Z::L]: } {

```

```

3639 \__bnvs_if_resolve_query:vcTF Z Z {
3640   \__bnvs_if_resolve_query:vcTF L { ans } {
3641     \__bnvs_tl_put_left:cn { ans } { 1-}
3642     \__bnvs_tl_put_right:cn { ans } +
3643     \__bnvs_tl_put_right:cv { ans } Z
3644     \__bnvs_round:c { ans }
3645     \__bnvs_tl_put_right:cn { ans } -
3646     \__bnvs_tl_put_right:cv { ans } Z
3647     \prg_return_true:
3648   } {
3649     \prg_return_false:
3650   }
3651 } {
3652   \prg_return_false:
3653 }
3654 }

```

☛ : or :: range

```

3655 \BNVS_new:cpn { if_resolve_query_return[:]: } {
3656   \__bnvs_tl_set:cn { ans } { - }
3657   \prg_return_true:
3658 }

```

☛ : <last> range

```

3659 \BNVS_new:cpn { if_resolve_query_return[:Z]: } {
3660   \__bnvs_tl_set:cn { ans } -
3661   \__bnvs_if_append:vcTF Z { ans } {
3662     \prg_return_true:
3663   } {
3664     \prg_return_false:
3665   }
3666 }

```

__bnvs_if_resolve_queries:ncTF __bnvs_if_resolve_queries:ncTF {<queries>} {<ans>} {<yes code>} {<no code>}

This is called by the *named overlay specifications* scanner. Evaluates the comma separated <queries>, 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.

```

3667 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End of definition for \c__bnvs_comma_regex.)

No other variable is used.

```

3668 \BNVS_new_conditional:cpnn { if_resolve_queries:nc } #1 #2 { TF } {
3669   \BNVS_begin:

```

Local variables cleared

```

3670   \__bnvs_seq_clear:c { ans }

```

In this main evaluation step, we evaluate the integer expression and put the result in a variable which content will be copied after the group is closed. We authorize comma separated expressions and $\langle first \rangle :: \langle last \rangle$ range expressions as well. We first split the expression around commas into the `query` sequence.

```

3671   \regex_split:NnN \c__bnvs_comma_regex { #1 } \l__bnvs_query_seq

```

Then each component is evaluated and the result is stored in `\l__bnvs_ans_seq` that we just cleared above.

```

3672   \BNVS_set:cpn { if_resolve_queries_end_return: } {
3673     \__bnvs_seq_if_empty:cTF { ans } {
3674       \BNVS_end:
3675       \__bnvs_tl_clear_ans:
3676     } {

```

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

```

3677       \exp_args:NnNx
3678       \BNVS_end:
3679       \__bnvs_tl_put_right:cn { #2 }
3680       { \__bnvs_seq_use:cn { ans } { , } }
3681     }
3682     \prg_return_true:
3683   }
3684   \__bnvs_seq_map_inline:cn { query } {
3685     \__bnvs_tl_clear_ans:
3686     \__bnvs_if_resolve_query:ncTF { ##1 } { ans } {
3687       \__bnvs_tl_if_empty:cF { ans } {
3688         \__bnvs_seq_put_right:cv { ans } { ans }
3689       }
3690     } {
3691       \seq_map_break:n {
3692         \BNVS_set:cpn { if_resolve_queries_end_return: } {
3693           \BNVS_error:n { Circular/Undefined~dependency~in~#1}
3694           \exp_args:NnNx
3695           \use:n {
3696             \BNVS_end:
3697             \__bnvs_tl_put_right:cn { #2 }
3698           } { \__bnvs_seq_use:cn { ans } { , } }
3699           \prg_return_false:
3700         }
3701       }
3702     }
3703   }
3704   \__bnvs_if_resolve_queries_end_return:
3705 }

```

```

3706 \NewDocumentCommand \BeanovesResolve { O{} m } {

```



```

3707 \BNVS_begin:
3708 \keys_define:nn { BeanovesResolve } {
3709   in:N .tl_set:N = \l__bnvs_resolve_in_tl,
3710   in:N .initial:n = { },
3711   show .bool_set:N = \l__bnvs_resolve_show_bool,
3712   show .default:n = true,
3713   show .initial:n = false,
3714 }
3715 \keys_set:nn { BeanovesResolve } { #1 }
3716 \__bnvs_tl_clear_ans:
3717 \__bnvs_if_resolve_queries:ncTF { #2 } { ans } {
3718   \__bnvs_tl_if_empty:cTF { resolve_in } {
3719     \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3720       \BNVS_tl_use:Nv \BNVS_end: { ans }
3721     } {
3722       \BNVS_end:
3723     }
3724   } {
3725     \bool_if:nTF { \l__bnvs_resolve_show_bool } {
3726       \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3727         \BNVS_end:
3728         \tl_set:Nn ##1 { ##2 }
3729         ##2
3730       }
3731       \BNVS_tl_use:nv {
3732         \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
3733       } { ans }
3734     } {
3735       \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
3736         \BNVS_end:
3737         \tl_set:Nn ##1 { ##2 }
3738       }
3739       \BNVS_tl_use:nv {
3740         \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
3741       } { ans }
3742     }
3743   }
3744 } {}
3745 }

```

6.19 Resetting counters and values

```

3746 \makeatletter
3747 \NewDocumentCommand \BeanovesReset { 0{} m } {
3748   \tl_if_empty:NTF \@currentenvir {

```

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

```

3749   \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}}~in~the~preamble.}
3750 } {
3751   \tl_if_eq:NnT \@currentenvir { document } {

```

At the top level, clear everything.

```

3752   \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}}~at~the~top~level.}
3753 }

```

```

3754 \BNVS_begin:
3755 \__bnvs_set_true:c { reset }
3756 \__bnvs_set_false:c { provide }
3757 \keys_define:nn { BeanovesReset } {
3758   all .bool_set:N = \l__bnvs_reset_all_bool,
3759   all .default:n = true,
3760   all .initial:n = false,
3761   only .bool_set:N = \l__bnvs_only_bool,
3762   only .default:n = true,
3763   only .initial:n = false,
3764 }
3765 \keys_set:nn { BeanovesReset } { #1 }
3766 \__bnvs_int_zero:c { i }
3767 \__bnvs_provide_off:
3768 \__bnvs_root_keyval:nc { #2 } a
3769 \BNVS_end_tl_set:cv J J
3770 \ignorespaces
3771 }
3772 }
3773 \makeatother
3774 \ExplSyntaxOff

```