beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0 2024/01/11

Abstract

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

Contents

1 Installation

1.1 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 id*⟩`!`, like `X!`, optional
2. ⟨*short name*⟩ like `A`, required
3. `.`⟨c_1⟩`....`⟨c_j⟩ like `.B.C`, optional ($j = 0$), globally denoted as *dotted path*.

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

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

3.3 Defining named overlay sets

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

`\Beanoves` `\Beanoves*`

`\Beanoves{`⟨ref_1⟩`=`⟨$spec_1$⟩`,...,`⟨ref_j⟩`=`⟨$spec_j$⟩`}`

Each ⟨*ref*⟩ key is a *named overlay reference* whereas each ⟨*spec*⟩ is an *overlay set specifier*. When the same ⟨*ref*⟩ key is used multiple times, only the last one is taken into account. Notice that ⟨*ref*⟩`=1` can be shortened to ⟨*ref*⟩.

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

`beanoves`

`beanoves = {`⟨ref_1⟩`=`⟨$spec_1$⟩`,...,`⟨ref_j⟩`=`⟨$spec_j$⟩`}`

[1] Precision is needed about the exact time when the expansion occurs.

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

3.3.1 Value specifiers

Hereafter ⟨*value*⟩ denotes a numerical expression.

⟨*ref*⟩`=`⟨*value*⟩`,` a *value specifier* for a single number. When omitted it defaults to 1.

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

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

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

3.3.2 Range specifiers

Hereafter ⟨*first*⟩, ⟨*last*⟩ and ⟨*length*⟩ are *value specifiers*.

⟨*ref*⟩`=`⟨*first*⟩`:`,
⟨*ref*⟩`=`⟨*first*⟩`::`, for the infinite range of signed integers starting at and including ⟨*first*⟩.

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

⟨*ref*⟩`=:`⟨*last*⟩,
⟨*ref*⟩`=::`⟨*length*⟩, are syntactic sugar when ⟨*first*⟩ is 1.

3.3.3 List specifiers

Here is the implementation.

⟨*ref*⟩`={`⟨*value*⟩`},` synonym of ⟨*ref*⟩`=`⟨*value*⟩.

⟨*ref*⟩`={`⟨*def*$_1$⟩`,...,`⟨*def*$_j$⟩`},` where ⟨*def*$_k$⟩, $1 \leq k \leq j$, is one of

- ⟨*index*⟩`=`⟨*spec*⟩,
- ⟨*spec*⟩ for ⟨*i*⟩`=`⟨*spec*⟩, ⟨*i*⟩ being the smallest positive integer such that ⟨*ref*⟩`.`⟨*i*⟩ is not already defined.

for value or range specifiers. The first step is to remove previous ⟨*ref*⟩ related definitions, then execute the various ⟨*ref*⟩.⟨*index*⟩=⟨*spec*⟩ definitions in the order given.

⟨*ref*⟩=**{{**⟨*def*$_1$⟩**,...,**⟨*def*$_j$⟩**}},** where ⟨*def*$_k$⟩, $1 \leq k \leq j$, is one of

- ⟨*name*⟩=⟨*spec*⟩,
- ⟨*name*⟩ for ⟨*name*⟩=1,

The first step is to remove previous ⟨*ref*⟩ related definitions, then execute the various ⟨*ref*⟩.⟨*name*⟩=⟨*spec*⟩ definitions in the order given. ⟨*spec*⟩ is any specifier.

4 Resolution of `?(...)` query expressions

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

The ⟨*queries*⟩ argument is a comma separated list of individual ⟨*query*⟩'s processed from left to right as explained below. Notice that nesting a `?(...)` query expression inside another query expression is not supported.

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

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

4.1 Range overlay queries

⟨*ref*⟩ `=` ⟨*first*⟩`:` as well as ⟨*first*⟩`::` defines a range limited from below:

overlay query	resolution
⟨*ref*⟩	⟨*first*⟩`-`
⟨*ref*⟩`.1`	⟨*first*⟩
⟨*ref*⟩`.2`	⟨*first*⟩ + 1
⟨*ref*⟩.⟨*i*⟩	⟨*first*⟩ + ⟨*i*⟩ − 1
⟨*ref*⟩`.previous`	⟨*first*⟩ − 1
⟨*ref*⟩`.first`	⟨*first*⟩

Notice that ⟨*ref*⟩`.previous` and ⟨*ref*⟩`.0` are most of the time synonyms.

⟨*ref*⟩ `=` ⟨*first*⟩`:`⟨*last*⟩ as well as variants ⟨*first*⟩`::`⟨*length*⟩, `::`⟨*length*⟩`:`⟨*last*⟩ or `:`⟨*last*⟩`::`⟨*length*⟩, which are equivalent provided ⟨*first*⟩+⟨*length*⟩ = ⟨*last*⟩+1.

For a range limited from both above and below:

overlay query	resolution
⟨*ref*⟩	⟨*first*⟩-⟨*last*⟩
⟨*ref*⟩`.1`	⟨*first*⟩
⟨*ref*⟩`.2`	⟨*first*⟩ + 1
⟨*ref*⟩`.`⟨*i*⟩	⟨*first*⟩ + ⟨*i*⟩ − 1
⟨*ref*⟩`.previous`	⟨*first*⟩ − 1
⟨*ref*⟩`.first`	⟨*first*⟩
⟨*ref*⟩`.last`	⟨*last*⟩
⟨*ref*⟩`.next`	⟨*last*⟩ + 1
⟨*ref*⟩`.length`	⟨*length*⟩

Notice that the resolution of ⟨*ref*⟩ 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.

```
\Beanoves {
A = 3:8, % or similarly A = 3::6, A = ::6:8 and A = :8::6
}
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\ttfamily
\BeanovesResolve[show](A)          == 3-8,
\BeanovesResolve[show](A.1)        == 3,
\BeanovesResolve[show](A.-1)       == 1,
\BeanovesResolve[show](A.previous) == 2,
\BeanovesResolve[show](A.first)    == 3,
\BeanovesResolve[show](A.last)     == 8,
\BeanovesResolve[show](A.next)     == 9,
\BeanovesResolve[show](A.length)   == 6,
\end{frame}
```

⟨*ref*⟩ `={...}` for an indexed list of specifications. The resolution of the overlay query ⟨*ref*⟩ gives the resolution of each item of the list, separated by commas, for the largest consecutive indices starting a 1.

⟨*ref*⟩ `={{...}}` for a named list of specifications. The resolution of the overlay query ⟨*ref*⟩ gives the resolution of each item of the list, in the given order, separated by commas. Notice that each item may in turn be resolved into a comma separated list of beamer ranges.

4.2 Value counter queries

⟨*ref*⟩ `=` ⟨*value*⟩ defines a counter value.

overlay query	resolution
⟨*ref*⟩	⟨*value*⟩
⟨*ref*⟩.1	⟨*value*⟩
⟨*ref*⟩.2	⟨*value*⟩ + 1
⟨*ref*⟩.⟨*i*⟩	⟨*value*⟩ + ⟨*i*⟩ − 1
⟨*ref*⟩.previous	⟨*value*⟩ − 1
⟨*ref*⟩.first	⟨*value*⟩
⟨*ref*⟩.last	⟨*value*⟩
⟨*ref*⟩.next	⟨*value*⟩ + 1

Additionnaly, resolution rules are provided for dedicated *overlay queries*, here ⟨*ref*⟩ is considered a standard programming variable:

⟨*ref*⟩=⟨*integer expression*⟩, resolve ⟨*integer expression*⟩ into ⟨*integer*⟩, assign it to the ⟨*ref*⟩ and use it. It defines ⟨*ref*⟩ globally if not already done. Here ⟨*integer expression*⟩ is the longest character sequence with no space[2].

⟨*ref*⟩+=⟨*integer expression*⟩, resolve ⟨*integer expression*⟩ into ⟨*integer*⟩, advance ⟨*ref*⟩ by ⟨*integer*⟩ and use the result.

++⟨*ref*⟩, increment ⟨*ref*⟩ by 1 and use it.

⟨*ref*⟩++, use ⟨*ref*⟩ and then increment it by 1.

This can be used for an indirection.

```
\Beanoves {
A = 1,
B = { 10 = 100 },
C = 10,
}
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\ttfamily
\BeanovesResolve[show](A.C)     == \BeanovesResolve[show](A.10) == 10,
\BeanovesResolve[show](B.C)     == \BeanovesResolve[show](B.10) == 100,
\BeanovesResolve[show](A.C+=10) == \BeanovesResolve[show](A.20) == 20,
\BeanovesResolve[show](B.C)     == \BeanovesResolve[show](B.20) == 20,
\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 `\BeanovesReset [⟨options⟩] {⟨ref$_1$⟩[=⟨spec$_1$⟩],..., ⟨ref$_j$⟩[=⟨spec$_j$⟩]}`

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

[2]The parser for algebraic expression is very rudimentary.

4.3 Dotted paths

Previous overlay queries may fail simply because ⟨*ref*⟩ is not defined. If the *dotted path* is empty, an error is raised and the resolution returns `0`.

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

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

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

Care should be taken to avoid circular dependencies.

4.4 The `beamer` counters

While inside a `frame` environment, it is possible to save the current value of the `beamerpauses` counter that controls whether elements should appear on the current slide. For that, we can execute one of `\Beanoves{`⟨*ref*⟩`=pauses}` or in a query `?(...(`⟨*ref*⟩`=pauses)...)`. Then later on, we can use `?(...`⟨*ref*⟩`...)` to refer to this saved value in the same frame[3]. 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}` or in a query `?(...(`⟨*ref*⟩`=slideinframe)...)`. It uses the numerical value of `\insertslideinframe`.

4.5 Multiple queries

It is possible to replace the comma separated list `?(`⟨$query_1$⟩`),...,?(`⟨$query_j$⟩`)` with the shorter `?(`⟨$query_1$⟩`,...,`⟨$query_j$⟩`)`.

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

4.6 Frame id

Except for very special situations, the *frame ids* can be left unspecified. When no *frame id* was explicitly provided, beanoves uses the *last frame id* and if the resolution fails an empty *frame id*. At the beginning of each frame, the *last frame id* is set to the *frame id* of the current frame, which is denoted *current frame id* and is empty by default. Then it gets updated after each named reference resolution where a *frame id* is explicitly given. For example, the first time `A.1` reference is resolved within a given frame, it is first translated to ⟨*last frame id*⟩`!A.1`, but when used just after `Y!C.2`, for example, it becomes a shortcut to `Y!A.1` because the *last frame id* is then `Y`.

In order to set the *frame id* of the current frame to ⟨*frame id*⟩, use the new `beanoves id` option of the beamer frame environment.

`beanoves id` `beanoves id=`⟨*frame id*⟩`,`

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

4.7 Resolution command

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

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

`show` the result is left into the input stream

`in:N=`⟨*command*⟩ the result is stored into ⟨*command*⟩.

5 Support

See the source repository. One can report issues there.

6 Implementation

Identify the internal prefix (LaTeX3 DocStrip convention, unused).

```
1 ⟨MY=bnvs⟩
```

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

6.1 Package declarations

```
\NeedsTeXFormat{LaTeX2e}[2020/01/01]
\ProvidesExplPackage
  {beanoves}
  {2024/01/11}
  {1.0}
  {Named overlay specifications for beamer}
```

6.2 Facility layer: definitions and naming

In order to make the code shorter and easier to read during development, we add a layer over LaTeX3. The `c` and `v` argument specifiers take a slightly different meaning when used in a function which name contains with `bnvs` or `BNVS`. Where LaTeX3 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.

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

`\BNVS_use_raw:c` `\BNVS_use_raw:c {⟨cs name⟩}`
`\BNVS_use_raw:Nc` `\BNVS_use_raw:Nc ⟨function⟩ {⟨cs name⟩}`
`\BNVS_use_raw:nc` `\BNVS_use_raw:nc {⟨tokens⟩} {⟨cs name⟩}`
`\BNVS_use:c` `\BNVS_use:c {⟨cs core⟩}`
`\BNVS_use:Nc` `\BNVS_use:Nc ⟨function⟩ {⟨cs core⟩}`
`\BNVS_use:nc` `\BNVS_use:nc {⟨tokens⟩} {⟨cs core⟩}`

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

```
\cs_new:Npn \BNVS_use_raw:N #1 { #1 }
\cs_new:Npn \BNVS_use_raw:c #1 {
  \exp_last_unbraced:No
  \BNVS_use_raw:N { \cs:w #1 \cs_end: }
}
```

```
\cs_new:Npn \BNVS_use:c #1 {
  \BNVS_use_raw:c { \BNVS:c { #1 } }
}

\cs_new:Npn \BNVS_use_raw:NN #1 #2 {
  #1 #2
}

\cs_new:Npn \BNVS_use_raw:nN #1 #2 {
  #1 #2
}

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

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

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

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

\cs_new:Npn \BNVS_tl_use:nvv #1 #2 {
  \BNVS_tl_use:nv { \BNVS_tl_use:nv { #1 } { #2 } }
}
\cs_new:Npn \BNVS_tl_use:nvvv #1 #2 {
  \BNVS_tl_use:nvv { \BNVS_tl_use:nv { #1 } { #2 } }
}

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

`\BNVS_DEBUG_on:n`	`\BNVS_DEBUG_on:n {`⟨*type*⟩`}`
`\BNVS_DEBUG_off:n`	`\BNVS_DEBUG_off:n {`⟨*type*⟩`}`
`\BNVS_DEBUG_push:n`	`\BNVS_DEBUG_push:n {`⟨*types*⟩`}`
`\BNVS_DEBUG_pop:`	`\BNVS_DEBUG_pop:`

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

```
\cs_new:Npn \BNVS_DEBUG:c #1 {
  BNVS_DEBUG~#1~
}
\cs_new:Npn \BNVS_DEBUG_on:n #1 {
  \tl_if_empty:nT { #1 } {
    \typein { Empty~argument~not~allowed }
  }
  \cs_set:cpn { \BNVS_DEBUG:c { #1 } log:n } { \BNVS_log:n }
  \cs_generate_variant:cn { \BNVS_DEBUG:c { #1 } log:n } { x }
}
\cs_new:Npn \BNVS_DEBUG_off:n #1 {
  \cs_set:cpn { \BNVS_DEBUG:c { #1 } log:n } { \use_none:n }
}
\seq_new:N \l_BNVS_DEBUG_push_n_seq
\cs_new:Npn \BNVS_DEBUG_push:n #1 {
  \tl_if_empty:nT { #1 } {
    \typein { Empty~argument~not~allowed }
  }
  \tl_map_inline:nn { #1 } {
    \BNVS_DEBUG_on:n { ##1 }
  }
  \seq_put_left:Nn \l_BNVS_DEBUG_push_n_seq {
    \tl_map_inline:nn { #1 } {
      \BNVS_DEBUG_off:n { ##1 }
    }
  }
}
\tl_new:N \l_BNVS_DEBUG_push_n_tl
\cs_new:Npn \BNVS_DEBUG_pop: {
  \seq_pop_left:NNTF \l_BNVS_DEBUG_push_n_seq \l_BNVS_DEBUG_push_n_tl {
    \l_BNVS_DEBUG_push_n_tl
  } {
    \BNVS_error:n { Unbalanced~\BNVS_DEBUG_pop: }
  }
}
\AddToHookNext { env/BNVS.test/begin } {
  \BNVS_DEBUG_push:n {CDBGpfarsRomqi}
  \BNVS_DEBUG_pop:
}
\cs_new:Npn \BNVS_DEBUG_log:nn #1 {
  \cs_if_exist_use:cF { \BNVS_DEBUG:c { #1 } log:n } {
    \BNVS_warning:n { Undeclared~DEBUG~type:~#1}
    \cs_new:cpn { \BNVS_DEBUG:c { #1 } log:n } { \use_none:n }
    \use_none:n
  }
}
\cs_new:Npn \BNVS_DEBUG_on: {
  \BNVS_DEBUG_on:n {C}
}
\cs_new:Npn \BNVS_DEBUG_off: {
  \BNVS_DEBUG_off:n {C}
}
```

\BNVS_new:cpn
\BNVS_set:cpn

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

```
\cs_new:Npn \BNVS_new:cpn #1 {
  \cs_new:cpn { \BNVS:c { #1 } }
}

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

\cs_generate_variant:Nn \cs_generate_variant:Nn { c }
\cs_new:Npn \BNVS_generate_variant:cn #1 {
  \cs_generate_variant:cn { \BNVS:c { #1 } }
}
```

6.3 logging

Utility messaging.

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

\cs_new:Npn \BNVS_warning:n {
  \msg_warning:nnn { beanoves } { :n }
}
\cs_new:Npn \BNVS_warning:x {
  \msg_warning:nnx { beanoves } { :n }
}

\cs_new:Npn \BNVS_error:n {
  \msg_error:nnn { beanoves } { :n }
}
\cs_new:Npn \BNVS_error:x {
  \msg_error:nnx { beanoves } { :n }
}

\cs_new:Npn \BNVS_fatal:n {
  \msg_fatal:nnn { beanoves } { :n }
}
\cs_new:Npn \BNVS_fatal:x {
  \msg_fatal:nnx { beanoves } { :n }
}
```

6.4 Facility layer: Variables

\BNVS_N_new:c
\BNVS_v_new:c

`\BNVS_N_new:n {⟨type⟩}`

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

```
\cs_new:Npn \BNVS_N_new:c #1 {
  \cs_new:cpn { BNVS_#1:c } ##1 {
    l \BNVS:c{ ##1 } \tl_if_empty:nF { ##1 } { _ } #1
  }
  \cs_new:cpn { BNVS_#1_new:c } ##1 {
    \use:c { #1_new:c } { \use:c { BNVS_#1:c } { ##1 } }
  }
  \cs_new:cpn { BNVS_#1_use:c } ##1 {
    \use:c { \cs:w BNVS_#1:c \cs_end: { ##1 } }
  }
  \cs_new:cpn { BNVS_#1_use:Nc } ##1 ##2 {
    \BNVS_use_raw:Nc
      ##1 { \cs:w BNVS_#1:c \cs_end: { ##2 } }
  }
  \cs_new:cpn { BNVS_#1_use:nc } ##1 ##2 {
    \BNVS_use_raw:nc
      { ##1 } { \cs:w BNVS_#1:c \cs_end: { ##2 } }
  }
}

\cs_new:Npn \BNVS_v_new:c #1 {
  \cs_new:cpn { BNVS_#1_use:Nv } ##1 ##2 {
    \BNVS_use_raw:nc
      { \exp_args:NV ##1 }
      { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
  }
  \cs_new:cpn { BNVS_#1_use:cv } ##1 ##2 {
    \BNVS_use_raw:nc
      { \exp_args:NnV \BNVS_use:c { ##1 } }
      { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
  }
  \cs_new:cpn { BNVS_#1_use:nv } ##1 ##2 {
    \BNVS_use_raw:nc
      { \exp_args:NnV \use:n { ##1 } }
      { \BNVS_use_raw:c { BNVS_#1:c } { ##2 } }
  }
}

\BNVS_N_new:c { bool }
\BNVS_N_new:c { int }
\BNVS_v_new:c { int }
\BNVS_N_new:c { tl }
\BNVS_v_new:c { tl }
\cs_new:Npn \BNVS_tl_use:Nvv #1 {
  \BNVS_exp_args:Nvv #1
}
```

```
\BNVS_N_new:c { str }
\BNVS_v_new:c { str }
\BNVS_N_new:c { seq }
\BNVS_v_new:c { seq }
\cs_undefine:N \BNVS_N_new:c
```

\BNVS_use:Ncn `\BNVS_use:Ncn` ⟨*function*⟩ {⟨*core name*⟩} {⟨*type*⟩}

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

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

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

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

\cs_new:Npn \BNVS_use:Ncncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:Ncn   #1   { #2 } { #3 }
  }
}

\cs_new:Npn \BNVS_use:ncncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:ncn { #1 } { #2 } { #3 }
  }
}

\cs_new:Npn \BNVS_use:Nvncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:Nvn   #1   { #2 } { #3 }
  }
}

\cs_new:Npn \BNVS_use:nvncn #1 #2 #3 {
  \BNVS_use:ncn {
    \BNVS_use:nvn { #1 } { #2 } { #3 }
  }
}

\cs_new:Npn \BNVS_use:Ncncncn #1 #2 #3 #4 #5 {
  \BNVS_use:ncn {
    \BNVS_use:Ncncn   #1   { #2 } { #3 } { #4 } { #5 }
  }
}
```

```
\cs_new:Npn \BNVS_use:ncncncn #1 #2 #3 #4 #5 {
  \BNVS_use:ncn {
    \BNVS_use:ncncn { #1 } { #2 } { #3 } { #4 } { #5 }
  }
}
```

\BNVS_new_c:cn \BNVS_new_c:nc {⟨*type*⟩} {⟨*core name*⟩}

```
\cs_new:Npn \BNVS_new_c:nc #1 #2 {
  \BNVS_new:cpn { #1_#2:c } {
    \BNVS_use_raw:c { BNVS_#1_use:nc } { \BNVS_use_raw:c { #1_#2:N } }
  }
}

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

\cs_new:Npn \BNVS_new_cnn:ncN #1 #2 #3 {
  \BNVS_new:cpn { #2:cnn } ##1 {
    \BNVS_use:Ncn { #3 } { ##1 } { #1 }
  }
}

\cs_new:Npn \BNVS_new_cnn:nc #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cnn:ncN { #1 } { #1_#2 }
  } { #1_#2:Nnn }
}

\cs_new:Npn \BNVS_new_cnv:ncN #1 #2 #3 {
  \BNVS_new:cpn { #2:cnv } ##1 ##2 {
    \BNVS_tl_use:nv {
      \BNVS_use:Ncn #3 { ##1 } { #1 } {  ##2 }
    }
  }
}

\cs_new:Npn \BNVS_new_cnv:nc #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cnv:ncN { #1 } { #1_#2 }
  } { #1_#2:Nnn }
}

\cs_new:Npn \BNVS_new_cnx:ncN #1 #2 #3 {
  \BNVS_new:cpn { #2:cnx } ##1 ##2 {
    \exp_args:Nnx \use:n {
      \BNVS_use:Ncn #3 { ##1 } { #1 } {  ##2 }
    }
  }
}
```

```
\cs_new:Npn \BNVS_new_cnx:nc #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cnx:ncN { #1 } { #1_#2 }
  } { #1_#2:Nnn }
}

\cs_new:Npn \BNVS_new_cc:ncNn #1 #2 #3 #4 {
  \BNVS_new:cpn { #2:cc } ##1 ##2 {
    \BNVS_use:Ncncn #3 { ##1 } { #1 } { ##2 } { #4 }
  }
}

\cs_new:Npn \BNVS_new_cc:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cc:ncNn { #1 } { #1_#2 }
  } { #1_#2:NN }
}

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

\cs_new:Npn \BNVS_new_cn:ncNn #1 #2 #3 #4 {
  \BNVS_new:cpn { #2:cn } ##1 {
    \BNVS_use:Ncn #3 { ##1 } { #1 }
  }
}

\cs_new:Npn \BNVS_new_cn:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cn:ncNn { #1 } { #1_#2 }
  } { #1_#2:Nn }
}

\cs_new:Npn \BNVS_new_cv:ncNn #1 #2 #3 #4 {
  \BNVS_new:cpn { #2:cv } ##1 ##2 {
    \BNVS_use:nvn {
      \BNVS_use:Ncn #3 { ##1 } { #1 }
    } { ##2 } { #4 }
  }
}

\cs_new:Npn \BNVS_new_cv:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_cv:ncNn { #1 } { #1_#2 }
  } { #1_#2:Nn }
}

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

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

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

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

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

\cs_new:Npn \BNVS_exp_args:Nvv #1 #2 #3 {
  \BNVS_use:ncncn { \exp_args:NVV #1 }
    { #2 } { tl } { #3 } { tl }
}
\cs_new:Npn \BNVS_exp_args:Nvvv #1 #2 #3 #4 {
  \BNVS_use:ncncncn { \exp_args:NVVV #1 }
    { #2 } { tl } { #3 } { tl } { #4 } { tl }
}
\cs_new:Npn \BNVS_exp_args:Nvvvv #1 #2 #3 #4 #5 {
  \BNVS_tl_use:nc {
  \exp_args:NnV \use:n {
    \BNVS_exp_args:Nvvv #1 { #2 } { #3 } { #4 }
  }
  } { #5 }
}
```

\BNVS_new_conditional:cpnn `\BNVS_new_conditional:cpnn {`⟨*core name*⟩`} ⟨`*parameter*`⟩ {`⟨*conditions*⟩`} {`⟨*code*⟩`}`

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

\cs_new:Npn \BNVS_new_conditional:cpnn #1 {
  \prg_new_conditional:cpnn { \BNVS:c { #1 } }
}
\cs_generate_variant:Nn \prg_generate_conditional_variant:Nnn { c }
\cs_new:Npn \BNVS_generate_conditional_variant:cnn #1 {
  \prg_generate_conditional_variant:cnn { \BNVS:c { #1 } }
}

\cs_new:Npn \BNVS_new_conditional_vn:cNnn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #1:vn } ##1 ##2 { #4 } {
    \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vn:cnn #1 #2 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vn:cNnn { #1 }
  } { #1:nn TF } { #2 }
}
```

```
\cs_new:Npn \BNVS_new_conditional_vc:cNnn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #4 } {
    \BNVS_use:Nvn #2 { ##1 } { #3 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vc:cnn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vc:cNnn { #1 }
  } { #1:ncTF }
}

\cs_new:Npn \BNVS_new_conditional_vvc:cNnnn #1 #2 #3 #4 #5 {
  \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #5 } {
    \BNVS_use:nvn {
      \BNVS_use:Nvn #2 { ##1 } { #3 }
    } { ##2 } { #4 } { ##3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vvc:cnnn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvc:cNnnn { #1 }
  } { #1:nncTF }
}

\cs_new:Npn \BNVS_new_conditional_vc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vc } ##1 ##2 { #3 } {
    \BNVS_tl_use:Nv #2 { ##1 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vc:cNn { #1 }
  } { #1:ncTF }
}
```

```
\cs_new:Npn \BNVS_new_conditional_vvc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vvc } ##1 ##2 ##3 { #3 } {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:Nv #2 { ##1 }
    } { ##2 } { ##3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vvc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvc:cNn { #1 }
  } { #1:nncTF }
}
```

6.4.1 Regex

```
\cs_new:Npn \BNVS_regex_use:Nc #1 #2 {
  \BNVS_use_raw:Nc #1 { c \BNVS:c { #2 } _regex }
}
```

__bnvs_match_if_once:NnTF
__bnvs_match_if_once:NvTF
__bnvs_match_if_once:nnTF
__bnvs_if_regex_split:cnTF

__bnvs_match_if_once:NnTF ⟨*regex variable*⟩ {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_match_if_once:nnTF {⟨*regex*⟩} {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_regex_split:cncTF {⟨*regex core*⟩} {⟨*expression*⟩} ⟨*seq core*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_regex_split:cnTF {⟨*regex core*⟩} {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

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

```
\BNVS_new_conditional:cpnn { if_extract_once:Ncn } #1 #2 #3 { T, F, TF } {
  \BNVS_use:ncn {
    \regex_extract_once:NnNTF #1 { #3 }
  } { #2 } { seq } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

```
\BNVS_new_conditional:cpnn { match_if_once:Nn } #1 #2 { T, F, TF } {
  \BNVS_use:ncn {
    \regex_extract_once:NnNTF #1 { #2 }
  } { match } { seq } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_extract_once:Ncv } #1 #2 #3 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_tl_use:nv {
      \regex_extract_once:NnNTF #1
    } { #3 }
  } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { match_if_once:Nv } #1 #2 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_tl_use:nv {
      \regex_extract_once:NnNTF #1
    } { #2 }
  } { match } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { match_if_once:nn } #1 #2 { T, F, TF } {
  \BNVS_seq_use:nc {
    \regex_extract_once:nnNTF { #1 } { #2 }
  } { match } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_regex_split:cnc } #1 #2 #3 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
  } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

```
\BNVS_new_conditional:cpnn { if_regex_split:cn } #1 #2 { T, F, TF } {
  \BNVS_seq_use:nc {
    \BNVS_regex_use:Nc \regex_split:NnNTF { #1 } { #2 }
  } { split } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.4.2 Token lists

`\__bnvs_tl_clear:c`	`\__bnvs_tl_clear:c {`⟨*core key tl*⟩`}`
`\__bnvs_tl_use:c`	`\__bnvs_tl_use:c {`⟨*core*⟩`}`
`\__bnvs_tl_set_eq:cc`	`\__bnvs_tl_count:c {`⟨*core*⟩`}`
`\__bnvs_tl_set:cn`	`\__bnvs_tl_set_eq:cc {`⟨*lhs core name*⟩`} {`⟨*rhs core name*⟩`}`
`\__bnvs_tl_set:(cv|cx)`	`\__bnvs_tl_set:cn {`⟨*core*⟩`} {`⟨*tl*⟩`}`
`\__bnvs_tl_put_left:cn`	`\__bnvs_tl_set:cv {`⟨*core*⟩`} {`⟨*value core name*⟩`}`
`\__bnvs_tl_put_right:cn`	`\__bnvs_tl_put_left:cn {`⟨*core*⟩`} {`⟨*tl*⟩`}`
`\__bnvs_tl_put_right:(cx|cv)`	`\__bnvs_tl_put_right:cn {`⟨*core*⟩`} {`⟨*tl*⟩`}`
	`\__bnvs_tl_put_right:cv {`⟨*core*⟩`} {`⟨*value core name*⟩`}`

These are shortcuts to

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

`\BNVS_new_conditional_vnc:cn`	`\BNVS_new_conditional_vnc:cn {`⟨*core*⟩`} {`⟨*conditions*⟩`}`

⟨*function*⟩ is the test function with signature `...:nncTF`. ⟨*core*⟩`:nncTF` is used for testing.

```
\cs_new:Npn \BNVS_new_conditional_vnc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vnc } ##1 ##2 ##3 { #3 } {
    \BNVS_tl_use:Nv #2 { ##1 } { ##2 } { ##3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
```

```
  }
}

\cs_new:Npn \BNVS_new_conditional_vnc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vnc:cNn { #1 }
  } { #1:nncTF }
}
```

`\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.

```
\cs_new:Npn \BNVS_new_conditional_vvnc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vvnc } ##1 ##2 ##3 ##4 { #3 } {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:Nv #2 { ##1 }
    } { ##2 } { ##3 } { ##4 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vvnc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvnc:cNn { #1 }
  } { #1:nnncTF }
}

\cs_new:Npn \BNVS_new_conditional_vvvc:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vvvc } ##1 ##2 ##3 ##4 { #3 } {
    \BNVS_tl_use:nvv {
      \BNVS_tl_use:Nv #2 { ##1 }
    } { ##2 } { ##3 } { ##4 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vvvc:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvvc:cNn { #1 }
  } { #1:nnncTF }
}

\cs_new:Npn \BNVS_new_tl_c:c {
  \BNVS_new_c:nc { tl }
}
\BNVS_new_tl_c:c { clear }
\BNVS_new_tl_c:c { use }
\BNVS_new_tl_c:c { count }
```

```
\BNVS_new:cpn { tl_set_eq:cc } #1 #2 {
  \BNVS_use:ncncn { \tl_set_eq:NN } { #1 } { tl } { #2 } { tl }
}

\cs_new:Npn \BNVS_new_tl_cn:c {
  \BNVS_new_cn:nc { tl }
}

\cs_new:Npn \BNVS_new_tl_cv:c #1 {
  \BNVS_new_cv:ncn { tl } { #1 } { tl }
}
\BNVS_new_tl_cn:c { set }
\BNVS_new_tl_cv:c { set }

\BNVS_new:cpn { tl_set:cx } {
  \exp_args:Nnx \__bnvs_tl_set:cn
}
\BNVS_new_tl_cn:c { put_right }
\BNVS_new_tl_cv:c { put_right }
% \BNVS_generate_variant:cn { tl_put_right:cn } { cx }

\BNVS_new:cpn { tl_put_right:cx } {
  \exp_args:Nnnx \BNVS_use:c { tl_put_right:cn }
}
\BNVS_new_tl_cn:c { put_left }
\BNVS_new_tl_cv:c { put_left }
% \BNVS_generate_variant:cn { tl_put_left:cn } { cx }

\BNVS_new:cpn { tl_put_left:cx } {
  \exp_args:Nnnx \BNVS_use:c { tl_put_left:cn }
}
```

`\__bnvs_tl_if_empty:c`*TF*
`\__bnvs_tl_if_blank:v`*TF*
`\__bnvs_tl_if_eq:cn`*TF*

`\__bnvs_tl_if_empty:cTF` {⟨*core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_tl_if_blank:vTF` {⟨*core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__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*⟩}

```
\cs_new:Npn \BNVS_new_conditional_c:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2 } ##1 { #4 } {
    \BNVS_use:Ncn #3 { ##1 } { #1 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
```

```
\cs_new:Npn \BNVS_new_conditional_c:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_c:ncNn { #1 } { #1_#2:c }
  } { #1_#2:NTF }
}
\BNVS_new_conditional_c:ncn { tl } { if_empty } { p, T, F, TF }

\BNVS_new_conditional:cpnn { tl_if_blank:v } #1 { T, F, TF } {
  \BNVS_tl_use:Nv \tl_if_blank:nTF { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\cs_new:Npn \BNVS_new_conditional_cn:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:cn } ##1 ##2 { #4 } {
    \BNVS_use:Ncn #3 { ##1 } { #1 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_cn:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_cn:ncNn { #1 } { #1_#2 }
  } { #1_#2:NnTF }
}
\BNVS_new_conditional_cn:ncn { tl } { if_eq } { T, F, TF }

\cs_new:Npn \BNVS_new_conditional_cv:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:cv } ##1 ##2 { #4 } {
    \BNVS_use:nvn {
      \BNVS_use:Ncn #3 { ##1 } { #1 }
    } { ##2 } { #1 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_cv:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_cv:ncNn { #1 } { #1_#2 }
  } { #1_#2:NnTF }
}
\BNVS_new_conditional_cv:ncn { tl } { if_eq } { T, F, TF }
```

6.4.3 Strings

__bnvs_str_if_eq:vn*TF* __bnvs_str_if_eq:vnTF {⟨*core*⟩} {⟨*tl*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

- \str_if_eq:ccTF {l__bnvs_⟨*core*⟩_tl}{⟨*yes code*⟩} {⟨*no code*⟩}

```
\cs_new:Npn \BNVS_new_conditional_vv:cNn #1 #2 #3 {
  \BNVS_new_conditional:cpnn { #1:vv } ##1 ##2 { #3 } {
    \BNVS_tl_use:nv {
      \BNVS_tl_use:Nv #2 { ##1 }
    } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vv:cn #1 {
  \BNVS_use:nc {
    \BNVS_new_conditional_vvnc:cNn { #1 }
  } { #1:nnTF }
}

\cs_new:Npn \BNVS_new_conditional_vn:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:vn } ##1 ##2 { #4 } {
    \BNVS_use:Nvn #3 { ##1 } { #1 } { ##2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\cs_new:Npn \BNVS_new_conditional_vn:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_vn:ncNn { #1 } { #1_#2 }
  } { #1_#2:nnTF }
}
\BNVS_new_conditional_vn:ncn { str } { if_eq } { T, F, TF }

\cs_new:Npn \BNVS_new_conditional_vv:ncNn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #2:vv } ##1 ##2 { #4 } {
    \BNVS_use:nvn {
      \BNVS_use:Nvn #3 { ##1 } { #1 }
    }  { ##2 } { #1 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
```

```
\cs_new:Npn \BNVS_new_conditional_vv:ncn #1 #2 {
  \BNVS_use_raw:nc {
    \BNVS_new_conditional_vv:ncNn { #1 } { #1_#2 }
  } { #1_#2:nnTF }
}
\BNVS_new_conditional_vv:ncn { str } { if_eq } { T, F, TF }
```

6.4.4 Sequences

__bnvs_seq_count:c	__bnvs_seq_new:c {⟨*core*⟩}
__bnvs_seq_clear:c	__bnvs_seq_count:c {⟨*core*⟩}
__bnvs_seq_set_eq:cc	__bnvs_seq_clear:c {⟨*core*⟩}
__bnvs_seq_gset_eq:cc	__bnvs_seq_set_eq:cc {⟨$core_1$⟩} {⟨$core_2$⟩}
__bnvs_seq_use:cn	__bnvs_seq_use:cn {⟨*core*⟩} {⟨*separator*⟩}
__bnvs_seq_item:cn	__bnvs_seq_item:cn {⟨*core*⟩} {⟨*integer expression*⟩}
__bnvs_seq_remove_all:cn	__bnvs_seq_remove_all:cn {⟨*core*⟩} {⟨*tl*⟩}
__bnvs_seq_put_left:cv	__bnvs_seq_put_right:cn {⟨*seq core*⟩} {⟨*tl*⟩}
__bnvs_seq_put_right:cn	__bnvs_seq_put_right:cv {⟨*seq core*⟩} {⟨*tl core*⟩}
__bnvs_seq_put_right:cv	__bnvs_seq_set_split:cnn {⟨*seq core*⟩} {⟨*tl*⟩} {⟨*separator*⟩}
__bnvs_seq_set_split:cnn	__bnvs_seq_pop_left:cc {⟨$core_1$⟩} {⟨$core_2$⟩}
__bnvs_seq_set_split:(cnv|cnx)	
__bnvs_seq_pop_left:cc	

These are shortcuts to

- \seq_set_eq:cc {l__bnvs_⟨$core_1$⟩_seq} {l__bnvs_⟨$core_2$⟩_seq}
- \seq_count:c {l__bnvs_⟨*core*⟩_seq}
- \seq_use:cn {l__bnvs_⟨*core*⟩_seq}{⟨*separator*⟩}
- \seq_item:cn {l__bnvs_⟨*core*⟩_seq}{⟨*integer expression*⟩}
- \seq_remove_all:cn {l__bnvs_⟨*core*⟩_seq}{⟨*tl*⟩}
- __bnvs_seq_clear:c {l__bnvs_⟨*core*⟩_seq}
- \seq_put_right:cv {l__bnvs_⟨*seq core*⟩_seq} {l__bnvs_⟨*tl core*⟩_tl}
- \seq_set_split:cnn{l__bnvs_⟨*seq core*⟩_seq}{l__bnvs_⟨*tl core*⟩_tl}{⟨*tl*⟩}

```
\BNVS_new_c:nc   { seq } { count }
\BNVS_new_c:nc   { seq } { clear }
\BNVS_new_cn:nc  { seq } { use }
\BNVS_new_cn:nc  { seq } { item }
\BNVS_new_cn:nc  { seq } { remove_all }
\BNVS_new_cn:nc  { seq } { map_inline }
\BNVS_new_cc:nc  { seq } { set_eq }
\BNVS_new_cc:nc  { seq } { gset_eq }
\BNVS_new_cv:ncn { seq } { put_left  } { tl }
\BNVS_new_cn:ncn { seq } { put_right } { tl }
\BNVS_new_cv:ncn { seq } { put_right } { tl }
```

```
\BNVS_new_cnn:nc { seq } { set_split }
\BNVS_new_cnv:nc { seq } { set_split }
\BNVS_new_cnx:nc { seq } { set_split }
\BNVS_new_cc:ncn { seq } { pop_left  } { tl }
\BNVS_new_cc:ncn { seq } { pop_right } { tl }
```

__bnvs_seq_if_empty:c*TF*
__bnvs_seq_get_right:cc*TF*
__bnvs_seq_pop_left:cc*TF*
__bnvs_seq_pop_right:cc*TF*

\ __bnvs_seq_if_empty:cTF {⟨*seq core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
\ __bnvs_seq_get_right:ccTF {⟨*seq core*⟩} {⟨*tl core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

```
\cs_new:Npn \BNVS_new_conditional_cc:ncnn #1 #2 #3 #4 {
  \BNVS_new_conditional:cpnn { #1_#2:cc } ##1 ##2 { #4 } {
    \BNVS_use:ncncn {
      \BNVS_use_raw:c { #1_#2:NNTF }
    } { ##1 } { #1 } { ##2 } { #3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_c:ncn { seq } { if_empty } { T, F, TF }
\BNVS_new_conditional_cc:ncnn
  { seq } { get_right } { tl } { T, F, TF }
\BNVS_new_conditional_cc:ncnn
  { seq } { pop_left  } { tl } { T, F, TF }
\BNVS_new_conditional_cc:ncnn
  { seq } { pop_right } { tl } { T, F, TF }
```

6.4.5 Integers

__bnvs_int_new:c
__bnvs_int_use:c
__bnvs_int_zero:c
__bnvs_int_inc:c
__bnvs_int_decr:c
__bnvs_int_set:cn
__bnvs_int_set:cv

\ __bnvs_int_new:c {⟨*core*⟩}
\ __bnvs_int_use:c {⟨*core*⟩}
\ __bnvs_int_incr:c {⟨*core*⟩}
\ __bnvs_int_decr:c {⟨*core*⟩}
\ __bnvs_int_set:cn {⟨*core*⟩} {⟨*value*⟩}

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_idecr:c {l__bnvs_⟨*core*⟩_int}
- \int_set:cn {l__bnvs_⟨*core*⟩_int} ⟨*value*⟩

```
\BNVS_new_c:nc   { int } { new  }
\BNVS_new_c:nc   { int } { use  }
\BNVS_new_c:nc   { int } { zero }
\BNVS_new_c:nc   { int } { incr }
\BNVS_new_c:nc   { int } { decr }
\BNVS_new_cn:nc  { int } { set  }
\BNVS_new_cv:ncn { int } { set  } { int }
```

6.5 Debug facilities

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

6.6 Debug messages

6.7 Testing facilities

6.8 Local variables

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

```
\tl_new:N \l__bnvs_id_last_tl
\tl_new:N \l__bnvs_id_tl
\tl_new:N \l__bnvs_kri_tl
\tl_new:N \l__bnvs_short_tl
\tl_new:N \l__bnvs_path_tl
\tl_new:N \l__bnvs_n_tl
\tl_new:N \l__bnvs_ref_tl
\tl_new:N \l__bnvs_tag_tl
\tl_new:N \l__bnvs_a_tl
\tl_new:N \l__bnvs_b_tl
\tl_new:N \l__bnvs_c_tl
\tl_new:N \l__bnvs_V_tl
\tl_new:N \l__bnvs_A_tl
\tl_new:N \l__bnvs_L_tl
\tl_new:N \l__bnvs_Z_tl
\tl_new:N \l__bnvs_ans_tl
\tl_new:N \l__bnvs_base_tl
\tl_new:N \l__bnvs_group_tl
\tl_new:N \l__bnvs_scan_tl
\tl_new:N \l__bnvs_query_tl
\tl_new:N \l__bnvs_token_tl
\tl_new:N \l__bnvs_root_tl
\tl_new:N \l__bnvs_n_incr_tl
\tl_new:N \l__bnvs_incr_tl
\tl_new:N \l__bnvs_plus_tl
\tl_new:N \l__bnvs_rhs_tl
```

```
\tl_new:N \l__bnvs_post_tl
\tl_new:N \l__bnvs_suffix_tl
\tl_new:N \l__bnvs_index_tl
\int_new:N \g__bnvs_call_int
\int_new:N \l__bnvs_int
\int_new:N \l__bnvs_i_int
\seq_new:N \g__bnvs_def_seq
\seq_new:N \l__bnvs_a_seq
\seq_new:N \l__bnvs_b_seq
\seq_new:N \l__bnvs_ans_seq
\seq_new:N \l__bnvs_match_seq
\seq_new:N \l__bnvs_split_seq
\seq_new:N \l__bnvs_path_seq
\seq_new:N \l__bnvs_path_head_seq
\seq_new:N \l__bnvs_path_tail_seq
\seq_new:N \l__bnvs_query_seq
\seq_new:N \l__bnvs_token_seq
\bool_new:N \l__bnvs_in_frame_bool
\bool_set_false:N \l__bnvs_in_frame_bool
\bool_new:N \l__bnvs_parse_bool
\bool_set_false:N \l__bnvs_parse_bool
\bool_new:N \l__bnvs_deep_bool
\bool_set_false:N \l__bnvs_deep_bool

\cs_new:Npn \BNVS_error_ans:x {
  \__bnvs_tl_put_right:cn { ans } { 0 }
  \BNVS_error:x
}
```

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

```
\bool_new:N \l__bnvs_provide_bool

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

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

\BNVS_new:cpn { provide_on: } {
  \__bnvs_set_true:c { provide }
}

\BNVS_new:cpn { provide_off: } {
  \__bnvs_set_false:c { provide }
}
\__bnvs_provide_off:
```

6.9 Infinite loop management

Unending recursivity is managed here.

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

(*End of definition for* \g__bnvs_call_int.)

```
\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.

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

__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.

```
\BNVS_new_conditional:cpnn { if_call: } { T, F, TF } {
  \int_gdecr:N \g__bnvs_call_int
  \int_compare:nNnTF \g__bnvs_call_int > 0 {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.10 Overlay specification

6.10.1 Registration

We keep track of the ⟨*id*⟩ ⟨*tag*⟩ combinations and provide looping mechanisms.

__bnvs_name:nnn `\__bnvs_name:nnn {`⟨*subkey*⟩`} {`⟨*id*⟩`} {`⟨*tag*⟩`}`
__bnvs_name:nn `\__bnvs_name:nn {`⟨*id*⟩`} {`⟨*tag*⟩`}`
__bnvs_id_seq:n `\__bnvs_id_seq:nn {`⟨*id*⟩`}`

Create a unique name from the arguments.

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

\g__bnvs_I_seq List of registered identifiers.

(*End of definition for* \g__bnvs_I_seq.)

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

`\__bnvs_register:nn`	`\__bnvs_register:nn {`⟨*id*⟩`} {`⟨*tag*⟩`}`
`\__bnvs_unregister:nn`	`\__bnvs_unregister:nn {`⟨*id*⟩`} {`⟨*tag*⟩`}`
`\__bnvs_unregister:n`	`\__bnvs_unregister:n {`⟨*id*⟩`}`
`\__bnvs_unregister:`	`\__bnvs_unregister:`

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

```
\seq_new:N \l__bnvs_register_NNnn_seq
\BNVS_new:cpn { register:NNnn } #1 #2 #3 #4 {
  \cs_if_exist:NF #1 {
    \cs_gset:Npn #1 { }
    \seq_if_exist:NTF #2 {
      \__bnvs_seq_clear:c { register_NNnn }
      \cs_set:Npn \BNVS_register_NNnn: {
        \__bnvs_seq_put_right:cn { register_NNnn } { #4 }
        \cs_set:Npn \BNVS_register_NNnn: { }
      }
      \cs_set:Npn \BNVS_register_NNnn:w ##1 ##2 {
        \str_compare:nNnTF { ##2 } < { #4 } {
          \__bnvs_seq_put_right:cn { register_NNnn } { ##2 }
        } {
          \BNVS_register_NNnn:
          \__bnvs_seq_put_right:cn { register_NNnn } { ##2 }
          \cs_set:Npn \BNVS_register_NNnn:w ####1 ####2 {
            \__bnvs_seq_put_right:cn { register_NNnn } { ####2 }
          }
        }
      }
      \__bnvs_foreach_T:nNTF { #3 } \BNVS_register_NNnn:w {
        \BNVS_register_NNnn:
        \seq_gset_eq:NN #2 \l__bnvs_register_NNnn_seq
      } {
        \BNVS_error:n { Unreachable/register:NNnn~id~#3 }
      }
    } {
      \seq_new:N #2
      \seq_gput_right:Nn #2 { #4 }
      \__bnvs_seq_clear:c { register_NNnn }
      \cs_set:Npn \BNVS_register_NNnn: {
        \__bnvs_seq_put_right:cn { register_NNnn } { #3 }
        \cs_set:Npn \BNVS_register_NNnn: {}
      }
      \cs_set:Npn \BNVS_register_NNnn:w ##1 {
        \str_compare:nNnTF { ##1 } < { #3 } {
          \__bnvs_seq_put_right:cn { register_NNnn } { ##1 }
        } {
          \BNVS_register_NNnn:
          \__bnvs_seq_put_right:cn { register_NNnn } { ##1 }
          \cs_set:Npn \BNVS_register_NNnn:w ####1 {
            \__bnvs_seq_put_right:cn { register_NNnn } { ####1 }
          }
        }
      }
      \__bnvs_foreach_I:N \BNVS_register_NNnn:w
```

```
      \BNVS_register_NNnn:
      \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_register_NNnn_seq
    }
  }
}

\BNVS_new:cpn { register:nn } #1 #2 {
  \exp_args:Ncc \__bnvs_register:NNnn
    { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
    { #1 } { #2 }
}
```

`\__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.

```
\seq_new:N \l__bnvs_unregister_NNnn_seq
\BNVS_new:cpn { unregister:NNnn } #1 #2 #3 #4 {
  \cs_if_exist:NT #1 {
    \cs_undefine:N #1
    \__bnvs_seq_clear:c { unregister_NNnn }
    \cs_set:Npn \BNVS_unregister_NNnn:n ##1 { ##1 }
    \cs_set:Npn \BNVS_unregister_NNnn:w ##1 ##2 {
      \str_compare:nNnTF { ##2 } < { #4 } {
        \__bnvs_seq_put_right:cn { unregister_NNnn } { ##2 }
        \cs_set:Npn \BNVS_unregister_NNnn:n ####1 { }
      } {
        \cs_set:Npn \BNVS_unregister_NNnn:w ####1 ####2 {
          \__bnvs_seq_put_right:cn { unregister_NNnn } { ####2 }
          \cs_set:Npn \BNVS_unregister_NNnn:n ########1 { }
        }
      }
    }
    \__bnvs_foreach_T:nNTF { #3 } \BNVS_unregister_NNnn:w {
      \seq_gset_eq:NN #2 \l__bnvs_unregister_NNnn_seq
    } {
      \BNVS_error:n { Unreachable / unregister:NNnn~#3!#4 }
    }
    \BNVS_unregister_NNnn:n {
      \__bnvs_seq_clear:c { unregister_NNnn }
      \cs_set:Npn \BNVS_unregister_NNnn:w ##1 {
        \str_compare:nNnTF { ##1 } < { #3 } {
          \__bnvs_seq_put_right:cn { unregister_NNnn } { ##1 }
        } {
          \cs_set:Npn \BNVS_unregister_NNnn:n ####1 {
            \__bnvs_seq_put_right:cn { unregister_NNnn } { ####1 }
          }
        }
      }
      \__bnvs_foreach_I:N \BNVS_unregister_NNnn:w
      \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_unregister_NNnn_seq
      \cs_undefine:N #2
    }
```

```
  }
}

\BNVS_new:cpn { unregister:nn } #1 #2 {
  \exp_args:Ncc \__bnvs_unregister:NNnn
    { \__bnvs_name:nn { #1 } { #2 } } { \__bnvs_id_seq:n { #1 } }
    { #1 } { #2 }
}
```

`\__bnvs_if_registered:nn`*TF* `\__bnvs_if_register:nnTF {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Execute ⟨*yes code*⟩ or ⟨*no code*⟩ depending on the ⟨*id*⟩`!`⟨*tag*⟩ combination being registered.

```
\BNVS_new_conditional:cpnn { if_registered:nn } #1 #2 { T, F, TF } {
  \cs_if_exist:cTF { \__bnvs_name:nn { #1 } { #2 } } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

`\__bnvs_foreach_I:N` `\__bnvs_foreach_I:N` ⟨*function:n*⟩
`\__bnvs_foreach_I:n` `\__bnvs_foreach_I:n {`⟨*code*⟩`}`

Execute the ⟨*function:n*⟩ or the ⟨*code*⟩ for each declared identifier.

```
\BNVS_new:cpn { foreach_I:N } {
  \seq_map_function:NN \g__bnvs_I_seq
}

\BNVS_new:cpn { foreach_I:n } {
  \seq_map_inline:Nn \g__bnvs_I_seq
}
```

`\__bnvs_foreach_T:nN`*TF* `\__bnvs_foreach_T:nNTF {`⟨*id*⟩`} `⟨*function:nn*⟩` {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_foreach_T:nn`*TF* `\__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.

```
\BNVS_new_conditional:cpnn { foreach_T:nN } #1 #2 { T, F, TF } {
  \seq_if_exist:cTF { g__bnvs_#1!_seq } {
    \seq_map_inline:cn { g__bnvs_#1!_seq } { #2 { #1 } { ##1 } }
    \prg_return_true:
  } { \prg_return_false: }
}

\BNVS_new_conditional:cpnn { foreach_T:nn } #1 #2 { T, F, TF } {
  \seq_if_exist:cTF { g__bnvs_#1!_seq } {
    \cs_set:Npn \BNVS_foreach_T_nn:nn ##1 ##2 { #2 }
    \seq_map_inline:cn { g__bnvs_#1!_seq }
      { \BNVS_foreach_T_nn:nn { #1 } { ##1 } }
    \prg_return_true:
  } { \prg_return_false: }
}
```

__bnvs_foreach_IT:N __bnvs_foreach_IT:N ⟨*function:nn*⟩
__bnvs_foreach_IT:n __bnvs_foreach_IT:n {⟨*code*⟩}

Execute the ⟨*function:nn*⟩ or the ⟨*code*⟩ for each combination of ⟨*id*⟩ and ⟨*tag*⟩.

```
\BNVS_new:cpn { foreach_IT:N } #1 {
  \__bnvs_foreach_I:n {
    \__bnvs_foreach_T:nNT { ##1 } #1 { }
  }
}

\BNVS_new:cpn { foreach_IT:n } #1 {
  \cs_set:Npn \BNVS_foreach_IT_n:nn ##1 ##2 { #1 }
  \__bnvs_foreach_I:n {
    \__bnvs_foreach_T:nNT { ##1 } \BNVS_foreach_IT_n:nn { }
  }
}
```

__bnvs_foreach_I:N __bnvs_foreach_key:N ⟨*function:n*⟩
__bnvs_foreach_I:n __bnvs_foreach_key:n {⟨*code*⟩}
__bnvs_foreach_key_main:N ⟨*function:n*⟩
__bnvs_foreach_key_main:n {⟨*code*⟩}
__bnvs_foreach_key_sub:N ⟨*function:n*⟩
__bnvs_foreach_key_sub:n {⟨*code*⟩}
__bnvs_foreach_key_cache:N ⟨*function:n*⟩
__bnvs_foreach_key_cache:n {⟨*code*⟩}

Execute the ⟨*function:n*⟩ or the ⟨*code*⟩ for each concerned key.

```
\BNVS_new:cpn { foreach_key_main:N } {
  \tl_map_function:nN { VAZL }
}

\BNVS_new:cpn { foreach_key_main:n } {
  \tl_map_inline:nn { VAZL }
}

\BNVS_new:cpn { foreach_key_sub:N } {
  \tl_map_function:nN { PNvn }
}

\BNVS_new:cpn { foreach_key_sub:n } {
  \tl_map_inline:nn { PNvn }
}

\BNVS_new:cpn { foreach_key:n } #1 {
  \__bnvs_foreach_key_main:n { #1 }
  \__bnvs_foreach_key_sub:n  { #1 }
}

\BNVS_new:cpn { foreach_key:N } #1 {
  \__bnvs_foreach_key_main:N #1
  \__bnvs_foreach_key_sub:N  #1
}
```

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

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

6.10.2 Basic functions

`\__bnvs_gset:nnnn` `\__bnvs_gset:nnnn {⟨key⟩} {⟨id⟩} {⟨tag⟩} {⟨spec⟩}`
`\__bnvs_gset:(nnnv|nvnn|nvvn|nvvv)`

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

```
\BNVS_new:cpn { gset:nnnn } #1 #2 #3 {
  \regex_match:nnTF { ^[a-z_]+$ } { #3 } {
    \use_none:n
  } {
    \__bnvs_register:nn { #2 } { #3 }
    \cs_gset:cpn { \__bnvs_name:nnn { #1 } { #2 } { #3 } }
  }
}

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

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

\BNVS_new:cpn { gset:nnnv } #1 #2 #3 {
  \BNVS_tl_use:nv {
    \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
  }
}

\BNVS_new:cpn { gset:nvvv } #1 {
  \BNVS_tl_use:nvvv { \__bnvs_gset:nnnn { #1 } }
}
```

`\__bnvs_gunset:nnn` `\__bnvs_gunset:nnn {⟨key⟩} {⟨id⟩} {⟨tag⟩}`
`\__bnvs_gunset:nn` `\__bnvs_gunset:nn {⟨id⟩} {⟨tag⟩}`
`\__bnvs_gunset:n` `\__bnvs_gunset:n {⟨id⟩}`
`\__bnvs_gunset:` `\__bnvs_gunset:`

Removes the specifications for the ⟨*key*⟩, ⟨*id*⟩, ⟨*tag*⟩ combination. In the variant, all possible ⟨*key*⟩s and ⟨*tag*⟩s are used.

```
\BNVS_new:cpn { gunset:nnn } #1 #2 #3 {
  \cs_undefine:c { \__bnvs_name:nnn { #1 } { #2 } { #3 } }
}
```

```
\BNVS_new:cpn { gunset:nvv } #1 {
  \BNVS_tl_use:nvv { \__bnvs_gunset:nnn { #1 } }
}

\BNVS_new:cpn { gunset:nn } #1 #2 {
  \__bnvs_if_registered:nnT { #1 } { #2 } {
    \tl_map_inline:nn {
      \__bnvs_foreach_key_main:n
      \__bnvs_foreach_key_sub:n
      \__bnvs_foreach_key_cache:n
    } {
      ##1 {
        \__bnvs_gunset:nnn { ####1 } { #1 } { #2 }
      }
    }
    \__bnvs_unregister:nn { #1 } { #2 }
  }
}

\BNVS_new:cpn { gunset_deep:nn } #1 #2 {
  \__bnvs_foreach_IT:n {
    \tl_if_eq:nnT { #1 } { ##1 } {
  \tl_if_in:nnT { .. ##2 } { .. #2 . } {
    \__bnvs_gunset:nn { #1 } { ##2 }
  }
    }
  }
}

\BNVS_new:cpn { gunset:vn } {
  \BNVS_tl_use:Nv \__bnvs_gunset:nn
}

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

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

\seq_new:N \l__bnvs_gunset_n_seq
\BNVS_new:cpn { gunset:n } #1 {
  \__bnvs_seq_clear:c { gunset_n }
  \__bnvs_foreach_I:n {
    \tl_if_eq:nnTF { ##1 } { #1 } {
      \__bnvs_foreach_T:nn { #1 } {
        \__bnvs_gunset:nn { #1  } { ####1 }
      }
    } {
      \__bnvs_seq_put_right:cn { gunset_n } { ##1 }
    }
  }
  \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_gunset_n_seq
}
```

```
\BNVS_new:cpn { gunset: } {
  \__bnvs_foreach_IT:N \__bnvs_gunset:nn
}
```

__bnvs_is_gset:nnn*TF*
__bnvs_is_gset:(nvv|nvx)*TF*
__bnvs_is_gset:nn*TF*
__bnvs_if_spec:nnn*TF*
__bnvs_if_spec:nn*TF*

__bnvs_is_gset:nnnTF {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_is_gset:nnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_spec:nnnTF {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_spec:nnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Convenient shortcuts to test for the existence of a ⟨*spec*⟩ for that ⟨*key*⟩, ⟨*id*⟩, ⟨*tag*⟩ combination. The version with no ⟨*key*⟩ is the or combination for keys `V`, `A` and `Z`.

The `_spec:...` variant is similar except that it uses ⟨*key*⟩, ⟨*id*⟩, ⟨*ref*⟩ or ⟨*key*⟩ empty ⟨*id*⟩, ⟨*tag*⟩ combinations.

```
\BNVS_new_conditional:cpnn { is_gset:nnn } #1 #2 #3 { T, F, TF } {
  \cs_if_exist:cTF { \__bnvs_name:nnn { #1 } { #2 } { #3 } } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { is_gset:nvx } #1 #2 #3 { T, F, TF } {
  \exp_args:Nnnx \BNVS_tl_use:nv {
    \__bnvs_is_gset:nnnTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { is_gset:nvv } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nvv {
    \__bnvs_is_gset:nnnTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { is_gset:nn } #1 #2 { T, F, TF } {
  \__bnvs_is_gset:nnnTF V { #1 } { #2 } {
    \prg_return_true:
  } {
    \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
      \prg_return_true:
    } {
      \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {
        \prg_return_true:
      } {
        \prg_return_false:
```

```
      }
    }
  }
}

\BNVS_new_conditional:cpnn { if_spec:nnn } #1 #2 #3 { T, F, TF } {
  \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \tl_if_empty:nTF { #2 } {
      \prg_return_false:
    } {
      \__bnvs_is_gset:nnnTF { #1 } { } { #3 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}

\BNVS_new_conditional:cpnn { if_spec:nn } #1 #2 { T, F, TF } {
  \__bnvs_is_gset:nnTF { #1 } { #2 } {
    \prg_return_true:
  } {
    \tl_if_empty:nTF { #1 } {
      \prg_return_false:
    } {
      \__bnvs_is_gset:nnTF { } { #2 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}
```

`\__bnvs_if_get:nnnc`*TF*
`\__bnvs_spec:nnnc`*TF*

`\__bnvs_if_get:nnncTF {`⟨*key*⟩`} {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*ans*⟩`}`
`{`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_spec:nnncTF {`⟨*key*⟩`} {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*ans*⟩`}`
`{`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

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

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

```
\BNVS_new_conditional:cpnn { if_get:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
    \exp_args:Nnc \use:n { \exp_args:Nno \cs_set:cpn { \BNVS_l:cn { #4 } { tl } } } } { \__bnv
    \prg_return_true:
  } {
    \prg_return_false:
```

```
  }
}

\BNVS_new_conditional:cpnn { if_get:nvvc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_tl_use:nvv {
    \__bnvs_if_get:nnncTF { #1 }
  } { #2 } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_spec:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \prg_return_true:
  } {
    \tl_if_empty:nTF { #2 } {
      \prg_return_false:
    } {
      \__bnvs_if_get:nnncTF { #1 } { } { #3 } { #4 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}
```

__bnvs_is_provide_gset:nnn*TF* __bnvs_is_provide_gset:nnnTF {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_is_provide_gset:nvv*TF*

Execute ⟨*yes code*⟩ when in provide mode and gset, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { is_provide_gset:nnn } #1 #2 #3 { T, F, TF } {
  \__bnvs_if:cTF { provide } {
    \cs_if_exist:cTF { \__bnvs_name:nnn { #1 } { #2 } { #3 } } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { is_provide_gset:nvv } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nvv { \__bnvs_is_provide_gset:nnnTF { #1 } } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_gprovide:TnnnnF
__bnvs_gprovide:TnvnnF
__bnvs_gprovide:TnvvnF

__bnvs_gprovide:TnnnnF {⟨*yes code*⟩} {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*value*⟩} {⟨*no code*⟩}

Execute ⟨*no code*⟩ exclusively when not in provide mode. Does nothing when something was set for the ⟨*id*⟩!⟨*tag*⟩/⟨*key*⟩ combination. Execute ⟨*yes code*⟩ before providing.

```
\BNVS_new:cpn { gprovide:TnnnnF } #1 #2 #3 #4 #5 {
  \__bnvs_if:cTF { provide } {
    \__bnvs_is_gset:nnnF { #2 } { #3 } { #4 } {
      #1
      \__bnvs_gset:nnnn { #2 } { #3 } { #4 } { #5 }
    }
  }
}

\BNVS_new:cpn { gprovide:TnvnnF } #1 #2 {
  \BNVS_tl_use:nv { \__bnvs_gprovide:TnnnnF { #1 } { #2 } }
}
\BNVS_new:cpn { gprovide:TnvvnF } #1 #2 {
  \BNVS_tl_use:nvv { \__bnvs_gprovide:TnnnnF { #1 } { #2 } }
}
```

6.10.3 Functions with cache

__bnvs_gset_cache:nnnn
__bnvs_gset_cache:(nnnv|nvvn)

__bnvs_gset_cache:nnnn {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*value*⟩}

Wrapper over the functions above for ⟨*key*⟩* instead of ⟨*key*⟩.

```
\BNVS_new:cpn { gset_cache:nnnn } #1 {
  \__bnvs_gset:nnnn { #1 * }
}

\BNVS_new:cpn { gset_cache:nvvn } #1 #2 {
  \BNVS_tl_use:nv {
    \BNVS_tl_use:nv {
      \__bnvs_gset_cache:nnnn { #1 }
    } { #2 }
  }
}

\BNVS_new:cpn { gset_cache:nnnv } #1 #2 #3 {
  \BNVS_tl_use:nv {
    \__bnvs_gset_cache:nnnn { #1 } { #2 } { #3 }
  }
}
```

__bnvs_if_get_cache:nnnc*TF*

__bnvs_if_get_cache:nnncTF {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*false code*⟩}

Wrapper over the functions above for ⟨*key*⟩* instead of ⟨*key*⟩.

```
\BNVS_new_conditional:cpnn { if_get_cache:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_get:nnncTF { #1 * } { #2 } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_gunset_cache:nnn
__bnvs_gunset_cache:nvv
__bnvs_gunset_cache:nn
__bnvs_gunset_cache:vv
__bnvs_gunset_cache:n
__bnvs_gunset_cache:

__bnvs_gunset_cache:nnn {⟨*key*⟩} {⟨*id*⟩} {⟨*tag*⟩}
__bnvs_gunset_cache:nn {⟨*id*⟩} {⟨*tag*⟩}
__bnvs_gunset_cache:n {⟨*id*⟩}
__bnvs_gunset_cache:

Wrapper over the functions above for ⟨*key*⟩* instead of ⟨*key*⟩.

```
\BNVS_new:cpn { gunset_cache:nnn } #1 {
  \__bnvs_gunset:nnn { #1 * }
}

\BNVS_new:cpn { gunset_cache:nvv } #1 {
  \__bnvs_gunset:nvv { #1 * }
}

\BNVS_new:cpn { gunset_cache:nn } #1 #2 {
  \__bnvs_foreach_key_cache:n {
    \__bnvs_gunset:nnn { ##1 } { #1 } { #2 }
  }
}

\BNVS_new:cpn { gunset_cache:vv } {
  \BNVS_tl_use:Nvv \__bnvs_gunset:nn
}

\BNVS_new:cpn { gunset_cache:n } #1 {
  \__bnvs_foreach_IT:n {
    \tl_if_eq:nnT { #1 } { ##1 } {
      \__bnvs_gunset_cache:nn { ##1 } { ##2 }
    }
  }
}

\BNVS_new:cpn { gunset_cache: } {
  \__bnvs_foreach_IT:n {
    \__bnvs_gunset_cache:nn { ##1 } { ##2 }
  }
}
```

6.11 Implicit value counter

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

__bnvs_v_gunset:nn __bnvs_v_gunset:n {⟨*id*⟩} {⟨*tag*⟩}
__bnvs_v_gunset:n __bnvs_v_gunset:n {⟨*id*⟩}
__bnvs_v_gunset: __bnvs_v_gunset:

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

```
\BNVS_new:cpn { v_gunset: } {
  \__bnvs_foreach_IT:n {
    \__bnvs_gunset:nnn   v { ##1 } { ##2 }
    \__bnvs_gunset_cache:nnn v { ##1 } { ##2 }
  }
}

\BNVS_new_conditional:cpnn { quark_if_nil:c } #1 { T, F, TF } {
  \BNVS_tl_use:nc { \exp_args:No \quark_if_nil:nTF } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { quark_if_no_value:c } #1 { T, F, TF } {
  \BNVS_tl_use:nc { \exp_args:No \quark_if_no_value:nTF } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_if_greset_all:nnn*TF* __bnvs_if_greset_all:nnnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*initial value*⟩} {⟨*yes code*⟩}
__bnvs_if_greset_all:vvn*TF* {⟨*no code*⟩}

If the ⟨*id*⟩!⟨*tag*⟩ combination is known, reset the value counter the given ⟨*initial value*⟩ and execute ⟨*yes code*⟩ otherwise ⟨*no code*⟩ is executed. The `..._all` variant also cleans the cached values and all the subvalues.

```
\BNVS_new_conditional:cpnn { if_greset_all:nnn } #1 #2 #3 { T, F, TF } {
  \__bnvs_is_gset:nnTF { #1 } { #2 } {
    \BNVS_begin:
    \__bnvs_foreach_key_main:n {
      \__bnvs_if_get:nnncT { ##1 } { #1 } { #2 } { a } {
        \__bnvs_quark_if_nil:cT { a } {
          \__bnvs_if_get_cache:nnncTF { ##1 } { #1 } { #2 } { a } {
            \__bnvs_gset:nnnv { ##1 } { #1 } { #2 } { a }
          } {
            \__bnvs_gset:nnnn { ##1 } { #1 } { #2 } { 1 }
          }
```

```
        }
      }
    }
    \BNVS_end:
    \__bnvs_gunset_cache:nn { #1 } { #2 }
    \__bnvs_foreach_key_sub:n {
      \__bnvs_gunset:nnn { ##1 } { #1 } { #2 }
    }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_greset_all:vvn } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \BNVS_tl_use:Nv \__bnvs_if_greset_all:nnnTF { #1 }
  } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
}
```

6.12 Regular expressions

\c__bnvs_short_regex

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

```
\regex_const:Nn \c__bnvs_short_regex {
  [[:alpha:]_][[:alnum:]_]*
}
```

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

\c__bnvs_path_regex

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

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

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

\c__bnvs_A_integer_Z_regex

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

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

\c__bnvs_A_index_Z_regex

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

```
\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`.)

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

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

(*End of definition for* \c__bnvs_A_ref_Z_regex.)

```
\regex_const:Nn \c__bnvs_A_ref_Z_regex {
```

1: the ⟨*frame id*⟩

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

2: The short name.

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

3: the path, if any.

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

\c__bnvs_A_IKT_Z_regex Matches the whole string, split into ⟨*id*⟩ and ⟨*tag*⟩.

(*End of definition for* \c__bnvs_A_IKT_Z_regex.)

1: The full match,

```
    \regex_const:Nn \c__bnvs_A_IKT_Z_regex {
```

2: the frame ⟨*id*⟩

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

3: The short name

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

\c__bnvs_A_ISP_Z_regex Matches the whole string.

(*End of definition for* \c__bnvs_A_ISP_Z_regex.)

1: The full match,

```
    \regex_const:Nn \c__bnvs_A_ISP_Z_regex {
```

2: the frame ⟨*id*⟩

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

3: The short name

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

4: The dotted path.

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

\c__bnvs_A_SP_Z_regex Matches the whole string.

(*End of definition for* \c__bnvs_A_SP_Z_regex.)

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

1: The full match,

2: the frame ⟨*id*⟩

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

3: The dotted path.

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

\c__bnvs_A_P_Z_regex Matches the whole string.

(*End of definition for* \c__bnvs_A_P_Z_regex.)

```
\regex_const:Nn \c__bnvs_A_P_Z_regex {
  \A \ur{c__bnvs_short_regex} (?: \. \ur{c__bnvs_short_regex} )* \Z
}
```

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

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

(*End of definition for* \c__bnvs_colons_regex.)

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

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

We start with '++' instrussions[4].

1 incrementation prefix

```
      \+\+
```

1.1: optional identifier: optional ⟨*frame id*⟩

1.2: followed by required `!`

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

1.3: ⟨*short name*⟩

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

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

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

2: without incement prefix

2.1: optional ⟨*frame id*⟩ followed by

2.2: required `!`

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

2.3: ⟨*short name*⟩

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

2.4: optionally followed by a dotted path

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

We continue with other expressions

2.5: the '`+`' in '`+=`' versus standalone '`=`'.

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

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

2.7: the post increment

```
      | (\+)\+


      )?
    ) \s*
}
```

"

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

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

6.13 **beamer.cls** interface

Work in progress.

```
\RequirePackage{keyval}

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

\AddToHook{env/beamer@frameslide/before}{
  \__bnvs_greset_call:
  \__bnvs_v_gunset:
  \__bnvs_set_true:c { in_frame }
}

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

6.14 Defining named slide ranges

`\__bnvs_range_if_set:cccn`*TF* `\__bnvs_range_if_set:cccnTF {`⟨*core first*⟩`} {`⟨*core end*⟩`} {`⟨*core length*⟩`} {`⟨*tl*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Parse ⟨*tl*⟩ as a range according to `\c__bnvs_colons_regex` and set the variables accordingly. ⟨*tl*⟩ is expected to only contain colons and integers.

```
\BNVS_new_conditional:cpnn { split_if_pop_left:c } #1 { T, F, TF } {
  \__bnvs_seq_pop_left:ccTF { split } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

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

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

\BNVS_new_conditional:cpnn { range_if_set:cccn } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_tl_clear:c A
  \__bnvs_tl_clear:c Z
  \__bnvs_tl_clear:c L
  \__bnvs_if_regex_split:cnTF { colons } { #4 } {
    \__bnvs_seq_pop_left:ccTF { split } A {
```

`A` may contain the ⟨*first*⟩, possibly empty, kept arround.

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

This is a one colon ⟨*A*⟩`:[^:]*`.

```
          \__bnvs_split_if_pop_left:cTF Z {
            \__bnvs_split_if_pop_left:cT L {
```

`Z` may contain the ⟨*last*⟩ and there is more material.

```
              \__bnvs_tl_if_empty:cTF L {
```

A `::` was expected:

```
                \BNVS_error:n { Invalid~range~expression(1):~#4 }
              } {
                \int_compare:nNnT { \__bnvs_tl_count:c L } > { 1 } {
                  \BNVS_error:n { Invalid~range~expression(2):~#4 }
                }
                \__bnvs_split_if_pop_left:cTF L {
```

`L` may contain the ⟨*length*⟩.

```
                  \__bnvs_seq_if_empty:cF { split } {
                    \BNVS_error:n { Invalid~range~expression(3):~#4 }
                  }
                } {
                  \BNVS_error:n { Unreachable~6~(range_if_set_cccnTF:nn) }
                }
              }
            }
          } {
            \BNVS_error:n { Unreachable~5~(range_if_set_cccnTF:nn) }
          }
        } {
```

This is a two colons ⟨*A*⟩`::...`, we expect a length.

```
          \int_compare:nNnT { \__bnvs_tl_count:c Z } > { 1 } {
            \BNVS_error:n { Too~many~colons(1):~#4 }
          }
          \__bnvs_split_if_pop_left:cTF L {
```

`L` may contain the ⟨*length*⟩.

```
            \__bnvs_split_if_pop_left:cTF Z {
              \__bnvs_tl_if_empty:cF Z {
                \BNVS_error:n { Too~many~colons(2):~#4 }
              }
              \__bnvs_split_if_pop_left:cTF Z {
```

`Z` may contain the ⟨*last*⟩.

```
                \__bnvs_seq_if_empty:cF { split } {
                  \BNVS_error:n { Invalid~range~expression(5):~#4 }
                }
              } {
                \BNVS_error:n { Invalid~range~expression(6):~#4 }
              }
            } {
              \__bnvs_tl_clear:c Z
            }
          } {
            \BNVS_error:n { Unreachable~3~(range_if_set_cccnTF:nn) }
          }
```

```
        }
      } {
        \BNVS_error:n { Unreachable~2~(range_if_set_cccnTF:nn) }
      }
    } {
      \BNVS_error:n { Unreachable~1~(range_if_set_cccnTF:nn) }
    }
```

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

```
    \__bnvs_tl_if_empty:cT A {
      \__bnvs_tl_if_empty:cTF Z {
        \__bnvs_tl_if_empty:cTF L {
          \BNVS_error:n { Invalid~range~expression(7):~#3 }
        } {
          \__bnvs_tl_set:cn A 1
        }
      } {
        \__bnvs_tl_if_empty:cT L {
          \__bnvs_tl_set:cn A 1
        }
      }
    }
    \cs_set:Npn \BNVS_range_if_set_cccnTF:w ##1 ##2 ##3 {
      \BNVS_end:
      \__bnvs_tl_set:cn { #1 } { ##1 }
      \__bnvs_tl_set:cn { #2 } { ##2 }
      \__bnvs_tl_set:cn { #3 } { ##3 }
    }
    \BNVS_exp_args:Nvvv \BNVS_range_if_set_cccnTF:w A Z L
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

__bnvs_parse_I:nn
__bnvs_parse_I:nv
__bnvs_parsed_IT:n

```
\__bnvs_parse_I:nn {⟨tag⟩} {⟨value⟩}
\__bnvs_parsed_IT:n {⟨value⟩}
```

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

__bnvs_range:nnnnn
__bnvs_range:nnvvv

```
\__bnvs_range:nnnnn {⟨id⟩} {⟨tag⟩} {⟨first⟩} {⟨last⟩} {⟨length⟩}
```

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

```
\BNVS_new:cpn { range:nnnnn } #1 #2 {
  \__bnvs_if:cTF { provide } {
    \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
      \use_none:nnn
    } {
      \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {
```

```
        \use_none:nnn
      } {
        \__bnvs_is_gset:nnnTF L { #1 } { #2 } {
          \use_none:nnn
        } {
          \__bnvs_do_range:nnnnn { #1 } { #2 }
        }
      }
    }
  } {
    \__bnvs_do_range:nnnnn { #1 } { #2 }
  }
}

\BNVS_new:cpn { range:nnvvv } #1 #2 {
  \BNVS_tl_use:nvvv {
     \__bnvs_range:nnnnn { #1 } { #2 }
  }
}

\BNVS_new:cpn { do_range:nnnnn } #1 #2 #3 #4 #5 {
  \__bnvs_gunset_deep:nn { #1 } { #2 }
  \__bnvs_gunset:nn { #1 } { #2 }
  \tl_if_empty:nTF { #5 } {
    \tl_if_empty:nTF { #3 } {
      \tl_if_empty:nTF { #4 } {
        \BNVS_error:n { Not~a~range:~#1!#2 }
      } {
        \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
        \__bnvs_gset:nnnn A { #1 } { #2 } {  1 }
        \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
      }
    } {
      \__bnvs_gset:nnnn A { #1 } { #2 } { #3 }
      \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
      \tl_if_empty:nF { #4 } {
        \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
        \__bnvs_gset:nnnn L { #1 } { #2 } { \q_nil }
      }
    }
  } {
    \tl_if_empty:nTF { #3 } {
      \__bnvs_gset:nnnn L { #1 } { #2 } { #5 }
      \tl_if_empty:nF { #4 } {
        \__bnvs_gset:nnnn Z { #1 } { #2 } { #4 }
        \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
        \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
      }
    } {
      \__bnvs_gset:nnnn A { #1 } { #2 } { #3 }
      \__bnvs_gset:nnnn L { #1 } { #2 } { #5 }
      \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
      \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
    }
  }
```

```
}

\BNVS_new:cpn { range_IT:vvv } {
  \__bnvs_range:nnvvv { id } { tag }
}
```

`\__bnvs_parsed:nnn` `\__bnvs_parsed:nnn {⟨id⟩} {⟨tag⟩} {⟨one spec⟩}`

Sets the internal model for the given ⟨*one spec*⟩, either a value or a range specification.

```
\BNVS_new:cpn { parsed:nnn } #1 #2 #3 {
  \__bnvs_range_if_set:cccnTF AZL { #3 } {
    \__bnvs_range:nnvvv { #1 } { #2 } AZL
  } {
    \__bnvs_is_provide_gset:nnnF V { #1 } { #2 } {
      \__bnvs_gunset_deep:nn { #1 } { #2 }
      \__bnvs_gunset:nn { #1 } { #2 }
      \tl_if_empty:nF { #3 } {
        \__bnvs_gset:nnnn V { #1 } { #2 } { #3 }
      }
    }
  }
}

\BNVS_new:cpn { parsed_IT:n } {
  \BNVS_tl_use:Nvv \__bnvs_parsed:nnn { id } { tag }
}
```

`\__bnvs_if_ref:n`*TF* `\__bnvs_if_ref:nTF {⟨name⟩} {⟨yes code⟩} {⟨no code⟩}`
`\__bnvs_if_ref:nn`*TF* `\__bnvs_if_ref:nnTF {⟨root⟩} {⟨relative⟩} {⟨yes code⟩} {⟨no code⟩}`
`\__bnvs_if_ref:vn`*TF* `\__bnvs_if_ref_relative:nnTF {⟨root⟩} {⟨relative⟩} {⟨yes code⟩} {⟨no code⟩}`
`\__bnvs_if_ref_relative:nn`*TF*

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

The second version calls the first one with ⟨*name*⟩ equals ⟨*relative*⟩ prepended with ⟨*root*⟩.

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

```
\BNVS_new_conditional:cpnn { if_ref:n } #1 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_match_if_once:NnTF \c__bnvs_A_ISP_Z_regex { #1 } {
    \__bnvs_if_match_pop_left:cTF { n } {
      \__bnvs_if_match_pop_left:cTF { id } {
        \__bnvs_if_match_pop_left:cTF { kri } {
          \__bnvs_if_match_pop_left:cTF { short } {
            \__bnvs_if_match_pop_left:cTF { path } {
              \cs_set:Npn \BNVS_aux_if_ref_nTF:nnn ##1 ##2 ##3 {
                \BNVS_end:
                \__bnvs_tl_set:cn { id } { ##1 }
                \__bnvs_tl_set:cn { short } { ##2 }
                \__bnvs_tl_set:cn { path } { ##3 }
              }
              \__bnvs_tl_if_empty:cTF { kri } {
```

```
                \BNVS_exp_args:Nvvv
                \BNVS_aux_if_ref_nTF:nnn
                  { id_last }
              } {
                \BNVS_exp_args:Nvvv
                \BNVS_aux_if_ref_nTF:nnn
                  { id }
              } { short } { path }
              \__bnvs_tl_set:cv { tag } { path }
              \__bnvs_tl_put_left:cv { tag } { short }
              \__bnvs_tl_set:cv { id_last } { id }
              \prg_return_true:
            } {
              \BNVS_end_unreachable_return_false:n { A_ISP_Z/path }
            }
          } {
            \BNVS_end_unreachable_return_false:n  { A_ISP_Z/short }
          }
        } {
          \BNVS_end_unreachable_return_false:n  { A_ISP_Z/kri }
        }
      } {
        \BNVS_end_unreachable_return_false:n  { A_ISP_Z/id }
      }
    } {
      \BNVS_end_unreachable_return_false:n  { A_ISP_Z/full_match }
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_ref_relative:nn } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_match_if_once:NnTF \c__bnvs_A_SP_Z_regex { #2 } {
    \__bnvs_if_match_pop_left:cTF { n } {
      \__bnvs_if_match_pop_left:cTF { short } {
        \__bnvs_if_match_pop_left:cTF { path } {
          \cs_set:Npn \BNVS_aux_if_ref_nTF:nn ##1 ##2 {
            \BNVS_end:
            \__bnvs_tl_put_right:cn { path } { . ##1 ##2 }
          }
          \BNVS_exp_args:Nvv
          \BNVS_aux_if_ref_nTF:nn { short } { path }
          \__bnvs_tl_set:cv { tag } { path }
          \__bnvs_tl_put_left:cv { tag } { short }
          \prg_return_true:
        } {
          \BNVS_end_unreachable_return_false:n { A_SP_Z/path }
        }
      } {
        \BNVS_end_unreachable_return_false:n  { A_SP_Z/short }
      }
```

```
    } {
      \BNVS_end_unreachable_return_false:n  { A_SP_Z/full_match }
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_ref:nn } #1 #2 { T, F, TF } {
  \tl_if_empty:nTF { #1 } {
    \__bnvs_if_ref:nTF { #2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \__bnvs_if_ref_relative:nnTF { #1 } { #2 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\BNVS_new_conditional:cpnn { if_ref:vn } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nv \__bnvs_if_ref:nnTF { #1 } { #2 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

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

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

```
\regex_const:Nn \c__bnvs_A_cln_Z_regex {
  \A \s* (?:
```

- 2 → V

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

- 3, 4, 5 → A : Z? or A :: L?

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

- 6, 7 → ::(L:Z)?

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

- 8, 9 → :(Z::L)?

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

\BNVS_int_new:c { next_IT }
\BNVS_tl_new:c { next_IT }
\BNVS_new:cpn { next_IT: } {
  \__bnvs_int_incr:c { next_IT }
  \__bnvs_tl_set_eq:cc { next_IT } { tag }
  \__bnvs_tl_put_right:cn { next_IT } { . }
  \BNVS_int_use:nv { \__bnvs_tl_put_right:cn { next_IT } } { next_IT }
  \__bnvs_is_gset:nvvT V { id } { next_IT } { \__bnvs_next_IT: }
}

\BNVS_new:cpn { next_IT:n } #1 {
  \BNVS_begin:
  \__bnvs_next_IT:
  \__bnvs_tl_put_right:cn { tag } { . }
  \BNVS_int_use:nv { \__bnvs_tl_put_right:cn { tag } } { next_IT }
  #1
  \BNVS_end_int_set:cv { next_IT } { next_IT }
}
```

6.14.1 Square brackets

See section 3.3.1.

__bnvs_bracket_parse_IT:n __bnvs_bracket_keyval_IT:n {⟨*definitions*⟩}
__bnvs_bracket_parse_IT:nn __bnvs_bracket_parse_IT:nn {⟨*name*⟩} {⟨*spec*⟩}
__bnvs_bracket_keyval_IT:n __bnvs_bracket_parse_IT:n {⟨*spec*⟩}

To parse what is inside square brackets.

```
\BNVS_new:cpn { bracket_assign:nnn } #1 #2 #3 {
  \__bnvs_gunset_deep:nn { #1 } { #2 }
  \__bnvs_gunset:nn { #1 } { #2 }
  \tl_if_empty:nF { #3 } {
    \__bnvs_gset:nnnn V { #1 } { #2 } { #3 }
  }
}
\BNVS_new:cpn { bracket_assign_I:cn } {
  \BNVS_tl_use:Nvv  \__bnvs_bracket_assign:nnn { id }
}
\BNVS_new:cpn { bracket_assign_IT:n } {
  \__bnvs_bracket_assign_I:cn { tag }
}
\BNVS_tl_new:c { bracket_assign_IT_nn }
\BNVS_new:cpn { bracket_assign_IT:nn } #1 {
  \__bnvs_tl_set_eq:cc { bracket_assign_IT_nn } { tag }
  \__bnvs_tl_put_right:cn { bracket_assign_IT_nn } { .#1 }
  \__bnvs_bracket_assign_I:cn { bracket_assign_IT_nn }
}
```

```
\BNVS_new:cpn { bracket_parse_IT:nn } #1 #2 {
  \__bnvs_match_if_once:NnTF \c__bnvs_colons_regex { #2 } {
    \BNVS_error:n { No~colon~allowed:~[...=#2...]}
  } {
    \__bnvs_match_if_once:NnTF \c__bnvs_A_integer_Z_regex { #1 } {
      \__bnvs_bracket_assign_IT:nn {#1 } { #2 }
    } {
      \BNVS_error:n { Not~an~integer:~#1 }
    }
  }
}

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

Find the first available index.

```
    \__bnvs_next_IT:n { \__bnvs_bracket_assign_IT:n { #1 } }
  }
}
```

For `X=[...]`.

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

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

```
  \__bnvs_tl_if_empty:cTF { tag } {
    \BNVS_error:n { Unexpected~list~at~top~level. }
  } {
    \__bnvs_is_provide_gset:nvvF V { id } { tag } {
      \BNVS_begin:
      \__bnvs_gunset_deep:vv { id } { tag }
      \__bnvs_gunset:vv { id } { tag }
      \__bnvs_int_zero:c { next_IT }
      \keyval_parse:nnn
        \__bnvs_bracket_parse_IT:n \__bnvs_bracket_parse_IT:nn { #1 }
      \BNVS_end:
    }
  }
}
```

6.14.2 Root level

`\__bnvs_root_parse:n` `\__bnvs_root_keyval:n {⟨definitions⟩}`
`\__bnvs_root_parse:nn` `\__bnvs_root_parse:nn {⟨name⟩} {⟨spec⟩}`
`\__bnvs_root_keyval:n` `\__bnvs_root_parse:n {⟨name⟩}`

To parse what is at the root level.

```
\BNVS_new:cpn { root_parse:n } #1 {
}

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

```
}
\BNVS_new:cpn { root_keyval:n } #1 {
```
This is a list, possibly at the top
```
  \BNVS_begin:
  \keyval_parse:nnn \BNVS_root_parse:n \BNVS_root_parse:nn { #1 }
  \BNVS_end:
```
This is a list, possibly at the top
```
}
```

`\l__bnvs_match_seq` Local storage for the match result.

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

The `a` tl auxiliary variable is used.

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

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

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

6.14.3 List specifiers

Here is the documentation. For ⟨*ref*⟩=⟨*spec*⟩.

`\__bnvs_list_keyval_IT:n` `\__bnvs_list_keyval_IT:n {`⟨*definitions*⟩`}`

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

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

```
\BNVS_new:cpn { list_keyval_IT:n } #1 {
  \BNVS_begin:
  \__bnvs_tl_clear:c V
  \__bnvs_tl_clear:c a
  \cs_set:Npn \BNVS: {
    \cs_set:Npn \BNVS:n ####1 {
      \__bnvs_tl_put_right:cn a { \BNVS:n { ####1 } }
    }
    \cs_set:Npn \BNVS:nn ####1 ####2 {
      \__bnvs_tl_put_right:cn a { \BNVS:nn { ####1 } { ####2 } }
    }
  }
  \cs_set:Npn \BNVS:n {
    \BNVS:
    \__bnvs_tl_set:cn V
  }
  \cs_set:Npn \BNVS:nn {
    \BNVS:
    \BNVS:nn
  }
  \keyval_parse:nnn \BNVS:n \BNVS:nn { #1 }
  \__bnvs_tl_if_empty:cTF a {
    \__bnvs_tl_if_empty:cTF V {
```

```
    } {
      \BNVS_tl_use:nv { \__bnvs_range_if_set:cccnTF A Z L } V {
      } {
        \__bnvs_is_provide_gset:nvvF V { id } { tag } {
          \__bnvs_gunset_deep:vv { id } { tag }
          \__bnvs_gunset:vv { id } { tag }
          \__bnvs_gset:nvvv V { id } { tag } V
        }
      }
    }
```

A single value or range specification.

```
  } {
  }
  \BNVS_end:
}
```

`\__bnvs_brace_parse:n` `\__bnvs_brace_parse:nn {⟨name⟩} {⟨spec⟩}`
`\__bnvs_brace_parse:nn` `\__bnvs_brace_parse:n {⟨named⟩}`

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

```
\exp_args_generate:n { nne }
\exp_args_generate:n { nnne }
\BNVS_new:cpn { brace_parse:n } #1 {
```

For X={{...,Y,...}}.

```
  \__bnvs_range_if_set:cccnTF A Z L { #1 } {
    \__bnvs_int_zero:c { next_IT }
    \__bnvs_next_IT:n {
      \__bnvs_set_true:c { deep }
      \BNVS_log_tl:c A
      \BNVS_log_tl:c Z
      \BNVS_log_tl:c L
      \typein { F00000000-#1 }
      \BNVS_int_use:Nv \__bnvs_parsed:nn { next_IT } { #1 }
    }
  } {
    \typein { F000000-#1 }
    \__bnvs_set_true:c { deep }
    \__bnvs_parsed:nn { #1 } { 1 }
    \typein { F000000-#1 }
  }
}

\BNVS_new:cpn { warn_up_to_q_stop:w } #1 \q_stop {
  \BNVS_warning:n { Ignored:~#1 }
}
\BNVS_new:cpn { scan_up_to_q_stop:w } {
  \peek_meaning:NTF \q_stop {
    \use_none:n
  } {
    \__bnvs_warn_up_to_q_stop:w
```

```
  }
}
```

Key only item.

```
\BNVS_new:cpn { parse_P_IT:n } #1 {
  \__bnvs_match_if_once:NnTF \c__bnvs_A_P_Z_regex { #1 } {
    \exp_args:Nnnx \BNVS_tl_use:nv { \__bnvs_gset:nnnn V } { id } {
      \BNVS_tl_use:c { root } #1
    } 1
  } {
    \BNVS_error:n { Unsupported~#1 }
  }
}

\BNVS_new:cpn { parse_Pd_IT:nn } #1 #2 {
  \__bnvs_match_if_once:NnTF \c__bnvs_A_Pd_Z_regex { #1 } {
    \exp_args:Nnnx \BNVS_tl_use:nv { \__bnvs_gset:nnnn V } { id } {
      \BNVS_tl_use:c { root } #1
    } 1
  } {
    \BNVS_error:n { Unsupported~#1 }
  }
}

\BNVS_new:cpn { brace_parse_IT:nw } #1 {
  \BNVS_begin:
```

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

```
  \__bnvs_tl_put_right:cv { root } { tag }
  \__bnvs_tl_put_right:cn { root } { . }

  \keyval_parse:nnn \BNVS_parse_P_IT:n \BNVS:nn { #1 }









  \BNVS_end:
  \__bnvs_scan_up_to_q_stop:w
}

\BNVS_new:cpn { brace_parse_IT:n } #1 {
  \BNVS_begin:
```

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

```
  \exp_args:Nc
  \peek_meaning:NTF { BNVS[]:w } {
```

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

```
    \BNVS_begin:
```

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

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

```
      \__bnvs_bracket_keyval_IT:n { ##1 }
      \BNVS_end:
    }
  } {
    \peek_meaning:NTF \c_group_begin_token {
      \BNVS_use:c { {{...}}:w }
    } {
      \peek_meaning:NTF \q_stop: {
        \use_none:n
      } {
        \BNVS_use:c { {...}:w }
      }
    }
  }
  #1 \q_stop
```

We export `\l__bnvs_id_last_tl`:

```
  \BNVS_end_tl_set:cv { id_last } { id_last }
}

\BNVS_new:cpn { brace_parse:nn } #1 #2 {
  \BNrocode}
% \begin{BNVS.gobble}
⟨/!debug⟩
% \end{BNVS.gobble}
% \end{BNVS.macrocode}
% We prepend the argument with |root|, in case we are recursive.
% \begin{BNVS.macrocode}
%    \begin{macrocode}
  \__bnvs_if_ref:vnTF { root } { #1 } {
    \exp_args:Nc
    \peek_meaning:NTF { BNVS[]:w } {
```

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

```
      \BNVS_begin:
```

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

```
      \cs_set:cpn { BNVS[]:w } ##1 ##2 \s_stop {
        \regex_match:nnT { \S } { ##2 } {
          \BNVS_warning:n { Ignoring~##2 }
        }
        \__bnvs_bracket_keyval_IT:n { ##1 }
        \BNVS_end:
      }
    } {
      \peek_meaning:NTF \s_stop: {
        \use_none:n
      } {
        \BNVS_use:c { X={...}:w }
      }
    }
    #2 \s_stop
  } {
    \BNVS_error:n { Invalid~name:~#2 }
  }
```

We export `\l__bnvs_id_last_tl`:

```
  \BNVS_end_tl_set:cv { id_last } { id_last }
}


\cs_new:Npn \BNVS_exp_args:NNcv #1 #2 #3 #4 {
  \BNVS_tl_use:nc { \exp_args:NNnV #1 #2 { #3 } }
    { #4 }
}
\cs_new:Npn \BNVS_end_tl_set:cv #1 {
  \BNVS_tl_use:nv {
    \BNVS_end: \__bnvs_tl_set:cn { #1 }
  }
}

\cs_new:Npn \BNVS_end_int_set:cv #1 {
  \BNVS_int_use:nv {
    \BNVS_end: \__bnvs_int_set:cn { #1 }
  }
}
```

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

- raw beamer lists `X=A` or `X={A,...}`,
- integer–value lists `X=[A,B]`.
- key–value lists `X={{A,B}}`,

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

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

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

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

```
      \BNVS_begin:
```

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

```
      \cs_set:cpn { BNVS[]:w } ##1 ##2 \s_stop {
        \regex_match:nnT { \S } { ##2 } {
          \BNVS_warning:n { Ignoring~##2 }
        }
        \__bnvs_bracket_keyval_IT:n { ##1 }
        \BNVS_end:
      }
    } {
      \peek_meaning:NTF \s_stop: {
        \use_none:n
      } {
        \BNVS_use:c { X={...}:w }
      }
    }
```

```
    #2 \s_stop
  } {
    \BNVS_error:n { Invalid~name:~#2 }
  }
```

We export `\l__bnvs_id_last_tl`:

```
  \__bnvs_match_if_once:NvT \c__bnvs_one_suffix_regex { tag } {
    \__bnvs_if_match_pop_left:cTF { a } {
      \__bnvs_if_match_pop_left:cTF { a } {
        \cs_set:Npn \BNVS_aux_parsed_nn: {
          \__bnvs_gset:nvvn V { id } { a } { #2 }
        }
        \__bnvs_if_get:nvvcT V { id } { a } { b } {
          \__bnvs_quark_if_nil:cF { b } {
            \cs_set:Npn \BNVS_aux_parsed_nn: { }
          }
        }
        \BNVS_aux_parsed_nn:
      } {
        \BNVS_error:n { Unreachable~2 }
      }
    } {
      \BNVS_error:n { Unreachable~1 }
    }
  }
```

We export `\l__bnvs_id_last_tl`:

```
  \BNVS_end_tl_set:cv { id_last } { id_last }
}
```

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

A `X={...}` list, for a ⟨*name*⟩–⟨*definition*⟩ dictionary.

```
  \BNVS_begin:
```

Remove the elements that contain a `=`.

```
  \__bnvs_tl_put_right:cv { root } { tag }
  \__bnvs_tl_put_right:cn { root } { . }
  \__bnvs_brace_keyval:n { #1 }
  \BNVS_end:
}
```

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

This is a `X=...`.

```
  \BNVS_begin:
  \__bnvs_tl_clear:c  { a }
  \__bnvs_seq_clear:c { a }
  \cs_set:Npn \BNVS_aux_parsed_nn:n ##1 {
    \__bnvs_tl_set:cn { a } { ##1 }
    \cs_set:Npn \BNVS_aux_parsed_nn:n ####1 {
      \__bnvs_seq_put_right:cn { a } { ####1 }
    }
  }
  \cs_set:Npn \BNVS_aux_parsed_nn:nn ##1 ##2 {
```

```
    \BNVS_warning:n { Ignored:~##1=##2 }
  }
  \keyval_parse:nnn \BNVS_aux_parsed_nn:n \BNVS_aux_parsed_nn:nn { #1 }
  \__bnvs_tl_if_empty:cTF { a } {
```

Clean everything, whether in provide mode or not.

```
   \__bnvs_gunset_deep:vv { id } { tag }
   \__bnvs_gunset:vv { id } { tag }
  } {
```

The first definition.

```
    \BNVS_tl_use:Nv \__bnvs_parsed_IT:n { a }
  }
  \BNVS_end:
}
```

`\__bnvs_parsed_IT[=...]:n` `\__bnvs_parsed_IT[=...]:n {⟨definitions⟩}`

Used by `\__bnvs_parse:nn` . ⟨*id*⟩ and ⟨*tag*⟩ are set. Store the associate values. ⟨*definitions*⟩ is a comma separated list of ⟨*definition*⟩'s, either for ranges or values. The first ⟨*definition*⟩ is for the `V` and `AZL` keys.

```
\BNVS_new:cpn { parsed_IT[=...]:n } #1 {
  \BNVS_begin:
  \__bnvs_tl_clear:c  { a }
  \__bnvs_seq_clear:c { a }
  \cs_set:Npn \BNVS_aux_parsed_nn:n ##1 {
    \__bnvs_tl_set:cn { a } { ##1 }
    \cs_set:Npn \BNVS_aux_parsed_nn:n ##1 {
      \__bnvs_seq_put_right:cn { a } { ##1 }
    }
  }
  \cs_set:Npn \BNVS_aux_parsed_nn:nn ##1 ##2 {
    \BNVS_warning:n { Ignored:~##1=##2 }
  }
  \keyval_parse:nnn \BNVS_aux_parsed_nn:n \BNVS_aux_parsed_nn:nn { #1 }
  \__bnvs_tl_if_empty:cTF { a } {
```

Clean everything, whether in provide mode or not.

```
    \__bnvs_gunset:vv { id } { tag }
  } {
```

The first definition.

```
    \__bnvs_seq_if_empty:cTF { a } {
      \BNVS_tl_use:Nv \__bnvs_parsed_IT:n { a }
    } {
      \__bnvs_int_zero:c { i }
      \__bnvs_seq_map_inline:cn { a } {
        \__bnvs_int_incr:c { i }
        {
          \__bnvs_tl_put_right:cn { tag } { . }
          \exp_args:Nnx \__bnvs_tl_put_right:cn { tag } {
            \__bnvs_int_use:c { i }
          }
          \__bnvs_parsed_IT:n { i }
```

```
        }
      }
    }
  }
  \BNVS_end:
}

\BNVS_new:cpn { parse_prepare:N } #1 {
  \tl_set:Nx #1 #1
  \__bnvs_set_false:c { parse }
  \bool_do_until:Nn \l__bnvs_parse_bool {
    \tl_if_in:NnTF #1 {%---[
    ]} {
      \regex_replace_all:nnNF { \[ ([^\]%---)
      ]*%---[(
      ) \] } { \c{BNVS[]:w} { \1 } } #1 {
        \__bnvs_set_true:c { parse }
      }
    } {
      \__bnvs_set_true:c { parse }
    }
  }
  \tl_if_in:NnTF #1 {%---[
  ]} {
    \BNVS_error:n { Unbalanced~%---[
    ]}
  } {
    \tl_if_in:NnT #1 { [%---]
    } {
      \BNVS_error:n { Unbalanced~[ %---]
      }
    }
  }
}
```

`\__bnvs_bracket_keyval_IT:n` `\__bnvs_bracket_keyval_IT:n {⟨definitions⟩}`
`\__bnvs_brace_keyval:n` `\__bnvs_brace_keyval:n {⟨definitions⟩}`

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

```
\BNVS_new:cpn { brace_keyval:n } {
  \keyval_parse:nnn \__bnvs_brace_parse:n \__bnvs_brace_parse:nn
}
```

`\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`.

```
\cs_new:Npn \BNVS_end_tl_put_right:cv #1 #2 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_tl_put_right:cn { #1 }
  } { #2 }
}
```

```
\cs_new:Npn \BNVS_end_gset:nnnv #1 #2 #3 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
  }
}

\NewDocumentCommand \Beanoves { sm } {
  \__bnvs_set_false:c { reset }
  \__bnvs_set_false:c { reset_all }
  \__bnvs_set_false:c { only }
  \tl_if_empty:NTF \@currenvir {
```

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

```
    \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
  } {
    \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, clear everything.

```
      \__bnvs_gunset:
    }
    \BNVS_begin:
    \__bnvs_tl_clear:c { root }
    \__bnvs_int_zero:c { i }
    \__bnvs_tl_set:cn { a } { #2 }
    \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, use the global definitions.

```
      \seq_if_empty:NF \g__bnvs_def_seq {
        \__bnvs_tl_put_left:cx { a } {
          \seq_use:Nn \g__bnvs_def_seq , ,
        }
      }
    }
    \__bnvs_parse_prepare:N \l__bnvs_a_tl
    \IfBooleanTF {#1} {
      \__bnvs_provide_on:
    } {
      \__bnvs_provide_off:
    }
    \BNVS_tl_use:Nv \__bnvs_brace_keyval:n { a }
    \BNVS_end_tl_set:cv { id_last } { id_last }
    \ignorespaces
  }
}
```

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

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

6.15 Scanning named overlay specifications

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

`\__bnvs@frame` `\__bnvs@frame {⟨overlay specification⟩}`
`\__bnvs@masterdecode` `\__bnvs@masterdecode {⟨overlay specification⟩}`

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

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

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

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

```
\makeatletter
\cs_set:Npn \__bnvs@frame < #1 > {
  \BNVS_begin:
  \__bnvs_tl_clear:c { ans }
  \__bnvs_scan:nNc { #1 } \__bnvs_if_resolve:ncTF { ans }
  \BNVS_set:cpn { :n } ##1 { \BNVS_end: \BNVS_saved@frame < ##1 > }
  \BNVS_tl_use:cv { :n } { ans }
}

\cs_set:Npn \__bnvs@masterdecode #1 {
  \BNVS_begin:
  \__bnvs_tl_clear:c { ans }
  \__bnvs_scan:nNc { #1 } \__bnvs_if_resolve_queries:ncTF { ans }
  \BNVS_tl_use:nv {
    \BNVS_end:
    \BNVS_saved@masterdecode
  } { ans }
}

\cs_new:Npn \BeanovesOff {
  \cs_set_eq:NN \beamer@frame \BNVS_saved@frame
  \cs_set_eq:NN \beamer@masterdecode \BNVS_saved@masterdecode
}

\cs_new:Npn \BeanovesOn {
  \cs_set_eq:NN \beamer@frame \__bnvs@frame
  \cs_set_eq:NN \beamer@masterdecode \__bnvs@masterdecode
}

\AddToHook{begindocument/before}{
  \cs_if_exist:NTF \beamer@frame {
    \cs_set_eq:NN \BNVS_saved@frame \beamer@frame
    \cs_set_eq:NN \BNVS_saved@masterdecode \beamer@masterdecode
  } {
    \cs_set:Npn \BNVS_saved@frame < #1 > {
      \BNVS_error:n {Missing~package~beamer}
    }
    \cs_set:Npn \BNVS_saved@masterdecode < #1 > {
      \BNVS_error:n {Missing~package~beamer}
    }
  }
```

```
  \BeanovesOn
}
\makeatother
```

`\__bnvs_scan:nNc` `\__bnvs_scan:nNc {⟨overlay query⟩} ⟨resolve⟩ {⟨ans⟩}`

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

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

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

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

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

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

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

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

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

`\l__bnvs_token_tl` Storage for just one token.

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

`\__bnvs_scan:nNc`*TF* `\__bnvs_scan:nNcTF {⟨overlay query⟩} ⟨resolve⟩ {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}`

Next are helpers.

`\__bnvs_scan_for_query_then_end_return:` `\__bnvs_scan_for_query_then_end_return:`

At top level state, scan the tokens of the ⟨*named overlay expression*⟩ looking for a '`?`' character. If a '`?(...)`' is found, then the ⟨*code*⟩ is executed.

```
\BNVS_new:cpn { scan_for_query_then_end_return: } {
  \__bnvs_seq_pop_left:ccTF { token } { token } {
    \__bnvs_tl_if_eq:cnTF { token } { ? } {
      \__bnvs_scan_require_open_end_return:
    } {
      \__bnvs_tl_put_right:cv { ans } { token }
      \__bnvs_scan_for_query_then_end_return:
    }
  } {
    \__bnvs_scan_end_return_true:
  }
}
```

`\__bnvs_scan_require_open_end_return:` `\__bnvs_scan_require_open_end_return:`

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

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

Get next token.

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

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

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

Record the forthcomming content in the `\l__bnvs_query_tl` variable, up to the next balancing '`)`'.

```
      \__bnvs_tl_clear:c { query }
      \__bnvs_scan_require_close_and_return:
    } {
```

Ignore this token and loop.

```
      \__bnvs_scan_require_open_end_return:
    }
  } {
```

Get next token.

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

```
    \BNVS_error:x {Missing~'('%---)
      ~after~a~? }
    \__bnvs_scan_end_return_true:
  }
}
```

`\__bnvs_scan_require_close_and_return:` `\__bnvs_scan_require_close_and_return:`

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

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

Get next token.

```
  \__bnvs_seq_pop_left:ccTF { token } { token } {
    \str_case:VnF \l__bnvs_token_tl {
      { ( %---)
      } {
```

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

```
        \__bnvs_int_incr:c { }
        \__bnvs_tl_put_right:cv { query } { token }
        \__bnvs_scan_require_close_and_return:
      }
      { %(---
        )
      } {
```

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

```
        \__bnvs_int_decr:c { }
        \int_compare:nNnTF { \__bnvs_int_use:c { } } = 0 {
```

The depth level has reached 0: we found our balancing parenthesis of the `?(...)` instruction. We can append the evaluated slide ranges token list to `ans` and look for the next ‘?’.

```
          \__bnvs_scan_handle_query_then_end_return:
        } {
```

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

```
          \__bnvs_tl_put_right:cv { query } { token }
          \__bnvs_scan_require_close_and_return:
        }
      }
    } {
```

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

```
        \__bnvs_tl_put_right:cv { query } { token }
      \__bnvs_scan_require_close_and_return:
    }
  } {
```

Above ends the code for Not a ‘(’. We reached the end of the sequence and the token list with no closing ‘)’. We raise and terminate. As recovery we feed `query` with the missing ‘)’.

```
    \BNVS_error:x { Missing~%(---
      `)' }
    \__bnvs_tl_put_right:cx { query } {
      \prg_replicate:nn { \l__bnvs_int } {%(---
      )}
    }
    \__bnvs_scan_end_return_true:
  }
}

\BNVS_new_conditional:cpnn { scan:nNc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \BNVS_set:cpn { error:x } ##1 {
    \msg_error:nnx { beanoves } { :n }
      { \tl_to_str:n { #1 }:~##1}
  }
  \__bnvs_tl_set:cn { scan } { #1 }
  \__bnvs_tl_clear:c { ans }
  \__bnvs_seq_clear:c { token }
```

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

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

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

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

```
      \__bnvs_scan_for_query_then_end_return:
    } {
      \BNVS_end_tl_put_right:cv { #3 } { ans }
```

Stop on the first error.

```
      \prg_return_false:
    }
  }
  \BNVS_set:cpn { scan_end_return_true: } {
    \BNVS_end_tl_put_right:cv { #3 } { ans }
    \prg_return_true:
  }
  \BNVS_set:cpn { scan_end_return_false: } {
    \BNVS_end_tl_put_right:cv { #3 } { ans }
    \prg_return_false:
  }
  \__bnvs_scan_for_query_then_end_return:
}
\BNVS_new:cpn { scan:nNc } #1 #2 #3 {
  \BNVS_use:c { scan:nNcTF } { #1 } #2 { #3 } {} {}
}
```

6.16 Resolution

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

`\__bnvs_if_TIP:ccc`*TF* `\__bnvs_if_TIP:cccTF {`⟨*name*⟩`} {`⟨*id*⟩`} {`⟨*path*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

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

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

This is a correct ⟨*tag*⟩, update the path sequence accordingly.

```
    \__bnvs_if_match_pop_TIP:cccTF { #1 } { #2 } { #3 } {
      \__bnvs_export_TIP:cccN { #1 } { #2 } { #3 }
        \BNVS_end:
      \prg_return_true:
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
    \BNVS_end:
```

```
    \prg_return_false:
  }
}
\quark_new:N \q__bnvs

\tl_new:N \l__bnvs_export_TIP_cccN_tl
\BNVS_new:cpn { export_TIP:cccN } #1 #2 #3 #4 {
  \cs_set:Npn \BNVS_export_TIP_cccN:w ##1 ##2 ##3 {
    #4
    \__bnvs_tl_set:cn { #1 } { ##1 }
    \__bnvs_tl_set:cn { #2 } { ##2 }
    \__bnvs_tl_set:cn { export_TIP_cccN } { ##3 }
  }
  \__bnvs_tl_set:cx { export_TIP_cccN }
    { \__bnvs_seq_use:cn { #1 } { \q__bnvs } }
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:Nv \BNVS_export_TIP_cccN:w { #1 }
  } { #2 } { export_TIP_cccN }
  \BNVS_tl_use:nv {
    \__bnvs_seq_set_split:cnn { #3 } { \q__bnvs }
  } { export_TIP_cccN }
  \__bnvs_seq_remove_all:cn { #3 } { }
}

\tl_new:N \l__bnvs_if_match_export_ISPn_cccc_tl
\BNVS_new:cpn { if_match_export_ISPn:ccccN } #1 #2 #3 #4 #5 {
  \cs_set:Npn \BNVS_if_match_export_ISPn_ccccN:w ##1 ##2 ##3 ##4 {
    #5
    \__bnvs_tl_set:cn { #1 } { ##1 }
    \__bnvs_tl_set:cn { #2 } { ##2 }
    \__bnvs_tl_set:cn { #3 } { ##3 }
    \__bnvs_tl_set:cn { #4 } { ##4 }
  }
  \__bnvs_tl_set:cx { if_match_export_ISPn_cccc }
    { \__bnvs_seq_use:cn { #1 } { \q__bnvs } }
  \BNVS_tl_use:nvvv {
    \BNVS_tl_use:Nv \BNVS_if_match_export_ISPn_ccccN:w { #1 }
  } { #2 } { if_match_export_ISPn_cccc } { #4 }
  \BNVS_tl_use:nv {
    \__bnvs_seq_set_split:cnn { #3 } { \q__bnvs }
  } { if_match_export_ISPn_cccc }
  \__bnvs_seq_remove_all:cn { #3 } { }
}

\BNVS_new_conditional:cpnn { if_match_pop_ISPn:cccc } #1 #2 #3 #4 { TF } {
  \BNVS_begin:
  \__bnvs_if_match_pop_left:cTF { #1 } {
    \__bnvs_if_match_pop_left:cTF { #1 } {
      \__bnvs_if_match_pop_left:cTF { #2 } {
        \__bnvs_if_match_pop_left:cTF { #3 } {
          \__bnvs_seq_set_split:cnv { #3 } { . } { #3 }
          \__bnvs_seq_remove_all:cn { #3 } { }
          \__bnvs_if_match_pop_left:cTF { #4 } {
            \__bnvs_if_match_export_ISPn:ccccN { #1 } { #2 } { #3 } { #4 }
              \BNVS_end:
```

```
            \prg_return_true:
          } {
            \BNVS_end_return_false:
          }
        } {
         \BNVS_end_return_false:
        }
      } {
        \BNVS_end_return_false:
      }
    } {
      \BNVS_end_return_false:
    }
  } {
    \BNVS_end_return_false:
  }
}
```

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.

```
\BNVS_new:cpn { seq_merge:cc } #1 #2 {
  \__bnvs_seq_if_empty:cF { #2 } {
    \__bnvs_seq_set_split:cnx { #1 } { \q__bnvs } {
      \__bnvs_seq_use:cn { #1 } { \q__bnvs }
      \exp_not:n { \q__bnvs }
      \__bnvs_seq_use:cn { #2 } { \q__bnvs }
    }
    \__bnvs_seq_remove_all:cn { #1 } { }
  }
}
```

6.17 Evaluation bricks

We start by helpers.

`\__bnvs_round:N` `\__bnvs_round:N` ⟨*tl variable*⟩
`\__bnvs_round:c` `\__bnvs_round:c` {⟨*tl core name*⟩}

Replaces the variable content with its rounded floating point evaluation.

```
\BNVS_new:cpn { round:N } #1 {
  \tl_if_empty:NTF #1 {
    \tl_set:Nn #1 { 0 }
  } {
    \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
  }
}
```

```
\BNVS_new:cpn { round:c } {
  \BNVS_tl_use:Nc \__bnvs_round:N
}
```

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

```
\BNVS_new_conditional:cpnn { if_assign_value:nnn } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve:ncTF { #3 } { a } {
    \__bnvs_gunset:nn { #1 } { #2 }
    \tl_map_inline:nn { V { V * } v } {
      \__bnvs_gset:nnnv { ##1 } { #1 } { #2 } { a }
    }
    \BNVS_end:
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_assign_value:nnv } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \__bnvs_if_assign_value:nnnTF { #1 } { #2 }
  } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_assign_value:vvv } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:Nv \_bnvs_if_assign_value:nnnTF { #1 }
  } { #2 } { #3 } { \prg_return_true: } { \prg_return_false: }
}
```

__bnvs_if_resolve_V:nnc*TF* __bnvs_if_resolve_V:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_resolve_V:nvc*TF* __bnvs_if_append_V:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_V:nnc*TF*
__bnvs_if_append_V:(nxc|nvc)*TF*

Resolve the content of the ⟨*id*⟩, ⟨*tag*⟩ value counter into the ⟨*ans*⟩ `tl` variable or append this value to the right of this variable. Execute ⟨*yes code*⟩ when there is a ⟨*value*⟩, {⟨*no code*⟩} otherwise. Inside the {⟨*no code*⟩} branch, the content of the ⟨*ans*⟩ `tl` variable is undefined. Implementation detail: in ⟨*ans*⟩ 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.

```
\BNVS_new:cpn { if_resolve_V_return:nnncT } #1 #2 #3 #4 #5 {
  \__bnvs_tl_if_empty:cTF { #4 } {
    \prg_return_false:
  } {
    \__bnvs_gset_cache:nnnv V { #2 } { #3 } { #4 }
    #5
    \prg_return_true:
  }
}

\makeatletter
\BNVS_new_conditional:cpnn { if_resolve_V:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get_cache:nnncTF V { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_get:nnncTF V { #1 } { #2 } { #3 } {
      \__bnvs_quark_if_nil:cTF { #3 } {
```

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

```
        \__bnvs_gset:nnnn V { #1 } { #2 } { \q_no_value }
        \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
          \__bnvs_if_resolve_V_return:nnncT A { #1 } { #2 } { #3 } {
            \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
          }
        } {
          \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
            \__bnvs_if_resolve_V_return:nnncT Z { #1 } { #2 } { #3 } {
              \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
            }
          } {
            \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
            \prg_return_false:
          }
        }
      } {
```

Possible recursive call.

```
        \__bnvs_quark_if_no_value:cTF { #3 } {
          \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
          \__bnvs_gset:nnnn V { #1 } { #2 } { 1 }
          \__bnvs_tl_set:cn { #3 } { 1 }
          \prg_return_true:
        } {
          \__bnvs_if_resolve:vcTF { #3 } { #3 } {
            \__bnvs_if_resolve_V_return:nnncT V { #1 } { #2 } { #3 } {
              \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
            }
          } {
            \__bnvs_gset:nnnn V { #1 } { #2 } { \q_nil }
            \prg_return_false:
          }
        }
      }
    } {
```

```
      \tl_if_eq:nnTF { #2 } { pauses } {
        \cs_if_exist:NTF \c@beamerpauses {
          \exp_args:Nnx \__bnvs_tl_set:cn { #3 } { \the\c@beamerpauses }
          \__bnvs_gunset:nn { #1 } { #2 }
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \tl_if_eq:nnTF { #2 } { slideinframe } {
          \cs_if_exist:NTF \beamer@slideinframe {
            \exp_args:Nnx \__bnvs_tl_set:cn { #3 } { \beamer@slideinframe }
          \__bnvs_gunset:nn { #1 } { #2 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        } {
          \prg_return_false:
        }
      }
    }
  }
}
\makeatother
\BNVS_new_conditional_vvc:cn { if_resolve_V } { T, F, TF }

\BNVS_new:cpn { end_put_right:vc } #1 #2 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_tl_put_right:cn { #2 }
  } { #1 }
}

\BNVS_new_conditional:cpnn { if_append_V:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { if_append_V } { T, F, TF }
```

`\__bnvs_if_resolve_A:nnc`*TF* `\__bnvs_if_resolve_A:nncTF {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_if_append_A:nnc`*TF* `\__bnvs_if_append_A:nncTF {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

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

```
\BNVS_new_conditional:cpnn { if_resolve_A:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get_cache:nnncTF A { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_get:nnncTF A { #1 } { #2 } { #3 } {
      \__bnvs_quark_if_nil:cTF { #3 } {
        \__bnvs_gset:nnnn A { #1 } { #2 } { \q_no_value }
```

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

```
        \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
          \__bnvs_tl_put_right:cn { #3 } { - }
          \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 } {
            \__bnvs_tl_put_right:cn { #3 } { + 1 }
            \__bnvs_round:c { #3 }
            \__bnvs_tl_if_empty:cTF { #3 } {
              \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
              \prg_return_false:
            } {
              \__bnvs_gset:nnnn   A { #1 } { #2 } { \q_nil }
              \__bnvs_gset_cache:nnnv A { #1 } { #2 } { #3 }
              \prg_return_true:
            }
          } {
            \BNVS_error:n {
Unavailable~length~for~#1~(\token_to_str:N\__bnvs_if_resolve_A:nncTF/2) }
            \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
            \prg_return_false:
          }
        } {
          \BNVS_error:n {
Unavailable~last~for~#1~(\token_to_str:N\__bnvs_if_resolve_A:nncTF/1) }
          \__bnvs_gset:nnnn A { #1 } { #2 } { \q_nil }
          \prg_return_false:
        }
      } {
        \__bnvs_quark_if_no_value:cTF { a } {
          \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
          \__bnvs_gset:nnnn A { #1 } { #2 } { 1 }
          \__bnvs_tl_set:cn { #3 } { 1 }
          \prg_return_true:
        } {
          \__bnvs_if_resolve:vcTF { #3 } { #3 } {
            \__bnvs_gset:nnnv A { #1 } { #2 } { #3 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      }
    } {
      \prg_return_false:
    }
  }
```

```
}

\BNVS_new_conditional:cpnn { if_append_A:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

`\__bnvs_if_resolve_Z:nnc`*TF* `\__bnvs_if_resolve_Z:nncTF {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_if_append_Z:nnc`*TF* `\__bnvs_if_append_Z:nncTF {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Resolve the last index of the ⟨*id*⟩!⟨*tag*⟩ range into or to the right of the ⟨*ans*⟩ `tl` variable, when possible. Execute ⟨*yes code*⟩ when a last index was given, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { if_resolve_Z:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get_cache:nnncTF Z { #1 } { #2 } { #3 } {
    \prg_return_true:
  }  {
    \__bnvs_if_get:nnncTF Z { #1 } { #2 } { #3 } {
      \__bnvs_quark_if_nil:cTF { #3 } {
        \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_no_value }
```

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

```
        \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
          \__bnvs_tl_put_right:cn { #3 } { + }
          \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 } {
            \__bnvs_tl_put_right:cn { #3 } { - 1 }
            \__bnvs_round:c { #3 }
            \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
            \__bnvs_gset:nnnn   Z { #1 } { #2 } { \q_nil }
            \prg_return_true:
          } {
            \BNVS_error:x {
 Unavailable~last~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
            \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
            \prg_return_false:
          }
        } {
          \BNVS_error:x {
Unavailable~first~for~#1~(\token_to_str:N \__bnvs_if_resolve_Z:ncTF/1) }
          \__bnvs_gset:nnnn Z { #1 } { #2 } { \q_nil }
          \prg_return_false:
        }
      } {
        \__bnvs_quark_if_no_value:cTF { #3 } {
          \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
          \__bnvs_tl_set:cn { #3 } { 1 }
          \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
          \prg_return_true:
```

```
        } {
          \__bnvs_if_resolve:vcTF { #3 } { #3 } {
            \__bnvs_gset_cache:nnnv Z { #1 } { #2 } { #3 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      } {
        \prg_return_false:
      }
    }
}
\BNVS_new_conditional_vvc:cn { if_resolve_Z } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_Z:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { if_append_Z } { T, F, TF }
```

__bnvs_if_resolve_L:nnc*TF* __bnvs_if_resolve_L:nncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_L:nnc*TF* __bnvs_if_append_L:nncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the length of the ⟨*id*⟩!⟨*tag*⟩ slide range into ⟨*ans*⟩ `tl` variable, or append the length of the ⟨*key*⟩ slide range to this variable. Execute ⟨*yes code*⟩ when there is a ⟨*length*⟩, ⟨*no code*⟩ otherwise.

```
\BNVS_new_conditional:cpnn { if_resolve_L:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get_cache:nnncTF L { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_get:nnncTF L { #1 } { #2 } { #3 } {
      \__bnvs_quark_if_nil:cTF { #3 } {
        \__bnvs_gset:nnnn L { #1 } { #2 } { \q_no_value }
```

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

```
        \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
          \__bnvs_tl_put_right:cn { #3 } { - }
          \__bnvs_if_append_A:nncTF { #1 } { #2 } { #3 } {
            \__bnvs_tl_put_right:cn { #3 } { + 1 }
            \__bnvs_round:c { #3 }
            \__bnvs_gset:nnnn   L { #1 } { #2 } { \q_nil }
            \__bnvs_gset_cache:nnnv L { #1 } { #2 } { #3 }
```

```
            \prg_return_true:
          } {
            \BNVS_error:n {
Unavailable~first~for~#1~(\__bnvs_if_resolve_L:nncTF/2) }
            \prg_return_false:
          }
        } {
          \BNVS_error:n {
Unavailable~last~for~#1~(\__bnvs_if_resolve_L:nncTF/1) }
          \prg_return_false:
        }
      } {
        \__bnvs_quark_if_no_value:cTF { #3 } {
          \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
          \__bnvs_gset_cache:nnnn L { #1 } { #2 } { 1 }
          \__bnvs_tl_set:cn { #3 } { 1 }
          \prg_return_true:
        } {
          \__bnvs_if_resolve:vcTF { #3 } { #3 } {
            \__bnvs_gset_cache:nnnv L { #1 } { #2 } { #3 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      }
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vvc:cn { if_resolve_L } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_L:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_L:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { if_append_L } { T, F, TF }
```

`\__bnvs_if_resolve_previous:nnc`*TF* `\__bnvs_if_append_previous:ncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩}
`\__bnvs_if_append_previous:nnc`*TF* {⟨*no code*⟩}

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

__bnvs_if_resolve_first:nncTF __bnvs_if_resolve_first:nncTF {⟨id⟩} {⟨tag⟩} ⟨ans⟩ {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_resolve_first:vvcTF
__bnvs_if_append_first:nncTF __bnvs_if_append_first:nncTF {⟨id⟩} {⟨tag⟩} ⟨ans⟩ {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_append_first:vvcTF

Resolve the first index starting the ⟨*id*⟩!⟨*tag*⟩ slide range into the ⟨*ans*⟩ `tl` variable, or append this index to that variable. Execute ⟨*yes code*⟩ when there is a ⟨*first*⟩ index, {⟨*no code*⟩} otherwise. In the latter case, on resolution only, the content of the ⟨*ans*⟩ `tl` variable is undefined.

```
\BNVS_new_conditional:cpnn { if_resolve_first:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve_V:nncTF { #1 } { #2.first } { #3 }
    { \prg_return_true: }
    { \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 }
        { \prg_return_true: }
        { \__bnvs_if_resolve_v:nncTF { #1 } { #2.1 } { #3 }
            { \prg_return_true: } { \prg_return_false: }
        }
    }
}
\BNVS_new_conditional_vvc:cn { if_resolve_first } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_first:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_append_index:nnncTF { #1 } { #2 } { 1 } { #3 } { \prg_return_true: } {
    \__bnvs_if_append_A:nncTF { #1 } { #2 } { #3 }
      { \prg_return_true: } { \prg_return_false: }
  }
}
\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 code⟩}
__bnvs_if_resolve_last:vvcTF
__bnvs_if_append_last:nncTF __bnvs_if_append_last:nncTF {⟨id⟩} {⟨tag⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_append_last:vvcTF

Resolve the last index of the ⟨*id*⟩!⟨*tag*⟩ slide range into the ⟨*ans*⟩ `tl` variable, or append this index to that variable. Execute ⟨*yes code*⟩ when there is a ⟨*last*⟩ index, {⟨*no code*⟩} otherwise. In the latter case, the content of the ⟨*ans*⟩ `tl` variable is undefined, on resolution only.

```
\BNVS_new_conditional:cpnn { if_resolve_last:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 }
    { \prg_return_true: } { \prg_return_false: }
}
\BNVS_new_conditional_vvc:cn { if_resolve_last } { T, F, TF }

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

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

__bnvs_if_resolve_length:nncTF
__bnvs_if_append_length:nncTF
__bnvs_if_append_length:vvcTF

```
\__bnvs_if_resolve_length:nncTF {⟨id⟩} {⟨tag⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
\__bnvs_if_append_length:nncTF  {⟨id⟩} {⟨tag⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
```

Resolve the length of the ⟨*id*⟩!⟨*tag*⟩ slide range into the ⟨*ans*⟩ `tl` variable, or append this number to that variable. Execute ⟨*yes code*⟩ when there is a ⟨*last*⟩ index, {⟨*no code*⟩} otherwise. In the latter case, the content of the ⟨*ans*⟩ `tl` variable is undefined, on resolution only.

```
\BNVS_new_conditional:cpnn { if_resolve_length:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve_L:nncTF { #1 } { #2 } { #3 }
    { \prg_return_true: } { \prg_return_false: }
}
\BNVS_new_conditional_vvc:cn { if_resolve_length } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_length:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_append_L:nncTF { #1 } { #2 } { #3 }
    { \prg_return_true: } {\prg_return_false: }
}
\BNVS_new_conditional_vvc:cn { if_append_length } { T, F, TF }
```

__bnvs_if_resolve_range:nncTF
__bnvs_if_append_range:nncTF

```
\__bnvs_if_resolve_range:nncTF {⟨id⟩} {⟨tag⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
\__bnvs_if_append_range:nncTF  {⟨id⟩} {⟨tag⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
```

Resolve the range of the ⟨*id*⟩!⟨*key*⟩ slide range into the ⟨*ans*⟩ `tl` variable or append this range to that variable. Execute ⟨*yes code*⟩ when there is a ⟨*range*⟩, ⟨*no code*⟩ otherwise, in that latter case the content the ⟨*ans*⟩ `tl` variable is undefined on resolution only.

```
\BNVS_new_conditional:cpnn { if_append_range:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { a } {
    \BNVS_tl_use:Nv \int_compare:nNnT { a } < 0 {
      \__bnvs_tl_set:cn { a } { 0 }
    }
    \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { b } {
```

Limited from above and below.

```
      \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
        \__bnvs_tl_set:cn { b } { 0 }
      }
      \__bnvs_tl_put_right:cn { a } { - }
      \__bnvs_tl_put_right:cv { a } { b }
      \BNVS_end_tl_put_right:cv { #3 } { a }
      \prg_return_true:
    } {
```

Limited from below.

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

```
      \prg_return_true:
    }
  } {
    \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { b } {
```

Limited from above.

```
      \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
        \__bnvs_tl_set:cn { b } { 0 }
      }
      \__bnvs_tl_put_left:cn { b } { - }
      \BNVS_end_tl_put_right:cv { #3 } { b }
      \prg_return_true:
    } {
      \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { b } {
      \BNVS_tl_use:Nv \int_compare:nNnT { b } < 0 {
        \__bnvs_tl_set:cn { b } { 0 }
      }
```

Unlimited range.

```
        \BNVS_end_tl_put_right:cv { #3 } { b }
        \__bnvs_tl_put_right:cn { #3 } { - }
        \prg_return_true:
      } {
        \BNVS_end:
        \prg_return_false:
      }
    }
  }
}
\BNVS_new_conditional_vvc:cn { if_append_range } { T, F, TF }

\BNVS_new_conditional:cpnn { if_resolve_range:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_tl_clear:c { #3 }
  \__bnvs_if_append_range:ncTF { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { if_resolve_range } { T, F, TF }
```

`\__bnvs_if_resolve_previous:nnc`*TF*
`\__bnvs_if_append_previous:nnc`*TF*

`\__bnvs_if_resolve_previous:nncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_if_append_previous:nncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

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

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

```
    \prg_return_true:
  } {
    \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
      \__bnvs_tl_put_right:cn { #3 } { -1 }
      \__bnvs_round:c { #3 }
      \__bnvs_gset_cache:nnnv P { #1 } { #2 } { #3 }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vvc:cn { if_resolve_previous } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_previous:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_previous:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { if_append_previous } { T, F, TF }
```

`\__bnvs_if_resolve_next:nnc`*TF*
`\__bnvs_if_append_next:nnc`*TF*

`\__bnvs_if_resolve_next:nncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_if_append_next:nncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

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

```
\BNVS_new_conditional:cpnn { if_resolve_next:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get_cache:nnncTF N { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
      \__bnvs_tl_put_right:cn { #3 } { +1 }
      \__bnvs_round:c { #3 }
      \__bnvs_gset_cache:nnnv N { #1 } { #2 } { #3 }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}
\BNVS_new_conditional_vvc:cn { if_resolve_next } { T, F, TF }

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

```
  \__bnvs_if_resolve_next:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_true:
  }
}
\BNVS_new_conditional_vvc:cn { if_append_next } { T, F, TF }
```

__bnvs_if_resolve_v:nnc*TF*	__bnvs_if_resolve_v:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_resolve_v:vvc*TF*	
__bnvs_if_append_v:nnc __bnvs_if_append_v:vvc*TF*	__bnvs_if_append_v:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

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

```
\BNVS_new_conditional:cpnn { if_resolve_v:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get:nnncTF v { #1 } { #2 } { #3 } {
    \__bnvs_quark_if_no_value:cTF { #3 } {
          \BNVS_error:n {Circular~definition:~#1!#2 (Error~recovery~1)}
          \__bnvs_gset:nnnn V { #1 } { #2 } { 1 }
          \__bnvs_tl_set:cn { #3 } { 1 }
          \prg_return_true:
        } {
      \prg_return_true:
    }
  } {
    \__bnvs_gset:nnnn v { #1 } { #2 } { \q_no_value }
    \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
      \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
      \prg_return_true:
    } {
      \__bnvs_if_resolve_A:nncTF { #1 } { #2 } { #3 } {
        \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
        \prg_return_true:
      } {
        \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #3 } {
        \__bnvs_gset:nnnv v { #1 } { #2 } { #3 }
          \prg_return_true:
        } {
          \__bnvs_gunset:nnn v { #1 } { #2 }
          \prg_return_false:
        }
      }
    }
  }
}
\BNVS_new_conditional_vvc:cn { if_resolve_v } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_v:nnc } #1 #2 #3 { T, F, TF } {
```

```
  \BNVS_begin:
  \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}

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

__bnvs_index_can:nn*TF*
__bnvs_index_can:vv*TF*
__bnvs_if_resolve_index:nnnc*TF*
__bnvs_if_resolve_index:vvvc*TF*
__bnvs_if_append_index:nnnc*TF*
__bnvs_if_append_index:vvvc*TF*

__bnvs_index_can:nnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_resolve_index:nnncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*integer*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_index:nnncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*integer*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

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

```
\BNVS_new_conditional:cpnn { index_can:nn } #1 #2 { T, F, TF } {
  \__bnvs_is_gset:nnnTF V { #1 } { #2 } {
    \prg_return_true:
  } {
    \__bnvs_is_gset:nnnTF A { #1 } { #2 } {
      \prg_return_true:
    } {
      \__bnvs_is_gset:nnnTF Z { #1 } { #2 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}

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

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

```
      \prg_return_true:
  } {
    \__bnvs_if_resolve_first:nncTF { #1 } { #2 } { #4 } {
      \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
      \__bnvs_round:c { #4 }
      \prg_return_true:
```

Limited overlay set.

```
    } {
      \__bnvs_if_resolve_Z:nncTF { #1 } { #2 } { #4 } {
        \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
        \__bnvs_round:c { #4 }
        \prg_return_true:
      } {
        \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
          \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
          \__bnvs_round:c { #4 }
          \prg_return_true:
        } {
          \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
            \__bnvs_tl_put_right:cn { #4 } { + #3 - 1 }
            \__bnvs_round:c { #4 }
            \prg_return_true:
          } {
            \prg_return_false:
          }
        }
      }
    }
  }
}

\BNVS_new_conditional:cpnn { if_resolve_index:nnvc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_tl_use:nv {
    \__bnvs_if_resolve_index:nnncTF { #1 } { #2 }
  } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_resolve_index:vvvc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:Nv \__bnvs_if_resolve_index:nnncTF { #1 }
  } { #2 } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

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

```
  \__bnvs_if_resolve_index:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \BNVS_end_tl_put_right:cv { #4 } { #4 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_append_index:vvvc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:Nv \__bnvs_if_append_index:nnncTF { #1 }
  } { #2 } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.18 Index counter

__bnvs_n_assign:nnn
__bnvs_n_assign:vvv

__bnvs_n_assign:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*value*⟩}

Assigns the resolved ⟨*value*⟩ to n counter ⟨*id*⟩!⟨*tag*⟩. Execute ⟨*yes code*⟩ when resolution succeeds, ⟨*no code*⟩ otherwise.

```
\BNVS_new:cpn { n_assign:nnn } #1 #2 #3 {
  \__bnvs_if_get:nnncF V { #1 } { #2 } { a } {
    \BNVS_warning:n { Unknwown~ #1!#2,~defaults~to~0 }
    \__bnvs_gset:nnnn V { #1 } { #2 } { 0 }
  }
  \__bnvs_if_resolve:ncTF { #3 } { a } {
    \__bnvs_gset:nnnv v { #1 } { #2 } { a }
  } {
    \BNVS_error:n { NO~resolution~of~#3,~defaults~to~0 }
    \__bnvs_gset:nnnn v { #1 } { #2 } { 0 }
  }
}

\BNVS_new:cpn { n_assign:vvv } #1 {
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:cv { n_assign:nn } { #1 }
  }
}
```

__bnvs_if_resolve_n:nc*TF*
__bnvs_if_append_n:nc*TF*
__bnvs_if_append_n:vc*TF*

__bnvs_if_resolve_n:nncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Evaluate the n counter associated to the ⟨*id*⟩!⟨*tag*⟩ overlay set into ⟨*ans*⟩ tl variable. Initialize this counter to 1 on the first use. ⟨*no code*⟩ is never executed.

```
\BNVS_new_conditional:cpnn { if_resolve_n:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get:nnncTF n { #1 } { #2 } { #3 } {
    \__bnvs_if_resolve:vcTF { #3 } { #3 } {
```

```
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \__bnvs_tl_set:cn { #3 } { 1 }
    \__bnvs_gset:nnnn n { #1 } { #2 } { 1 }
    \prg_return_true:
  }
}
\BNVS_new_conditional_vvc:cn { if_resolve_n } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_n:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}

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

__bnvs_if_resolve_n_index:nnc*TF*
__bnvs_if_append_n_index:nnc*TF*

__bnvs_if_resolve_n_index:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_n_index:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the index for the value of the n counter associated to the {⟨*tag*⟩} overlay set into the ⟨*ans*⟩ tl variable or append this value the right of that variable. Initialize this counter to 1 on the first use. If the computation is possible, ⟨*yes code*⟩ is executed, otherwise ⟨*no code*⟩ is executed. In the latter case, the content of the ⟨*ans*⟩ tl variable is undefined on resolution only.

```
\BNVS_new_conditional:cpnn { if_resolve_n_index:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #3 } {
    \__bnvs_tl_put_left:cn { #3 } { #1!#2. }
    \__bnvs_if_resolve:vcTF { #3 } { #3 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}

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

```
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvc:cn { if_append_n_index } { T, F, TF }
```

6.19 Value counter

__bnvs_if_resolve_v_incr:nnncTF
__bnvs_if_append_v_incr:nnncTF
__bnvs_if_append_v_incr:vvncTF

__bnvs_if_resolve_v_incr:nnnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*offset*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_v_incr:nnncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*offset*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Increment the value counter position accordingly. Put the result in the ⟨*ans*⟩ `tl` variable.

```
\BNVS_new_conditional:cpnn { if_resolve_v_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_resolve:ncTF { #3 } { #4 } {
    \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
      \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    } {
      \__bnvs_tl_put_right:cn { #4 } { + }
      \__bnvs_if_append_v:nncTF { #1 } { #2 } { #4 } {
        \__bnvs_round:c { #4 }
        \__bnvs_gset:nnnv v { #1 } { #2 } { #4 }
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_append_v_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_v_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \BNVS_end_tl_put_right:cv { #4 } { #4 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvnc:cn { if_append_v_incr } { T, F, TF }

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

```
  \__bnvs_if_resolve_v:nncTF { #1 } { #2 } { #4 } {
    \BNVS_begin:
    \__bnvs_if_resolve:ncTF { #3 } { a } {
      \BNVS_tl_use:Nv \int_compare:nNnTF { a } = 0 {
        \BNVS_end:
        \prg_return_true:
      } {
        \__bnvs_tl_put_right:cn { a } { + }
        \__bnvs_tl_put_right:cv { a } { #4 }
        \__bnvs_round:c { a }
        \BNVS_end_gset:nnnv v { #1 } { #2 } { a }
        \prg_return_true:
      }
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
      \prg_return_false:
  }
}
\BNVS_new_conditional_vvvc:cn { if_resolve_v_post } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_v_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_v_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \BNVS_end_tl_put_right:cv { #4 } { #4 }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvnc:cn { if_append_v_post } { T, F, TF }
\BNVS_new_conditional_vvvc:cn { if_append_v_post } { T, F, TF }
```

`\__bnvs_if_resolve_n_incr:nnnncTF` `\__bnvs_if_resolve_n_incr:nnnncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*base*⟩} {⟨*offset*⟩}
`\__bnvs_if_resolve_n_incr:nnncTF` {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_if_append_n_incr:nnnncTF` `\__bnvs_if_resolve_n_incr:nnncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*offset*⟩} {⟨*ans*⟩}
`\__bnvs_if_append_n_incr:nnncTF` {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_if_append_n_incr:vvncTF` `\__bnvs_if_append_n_incr:nnnncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*base*⟩} {⟨*offset*⟩}
`\__bnvs_if_resolve_n_post:nnncTF` {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_if_append_n_post:nnncTF` `\__bnvs_if_append_n_incr:nnncTF` {⟨*id*⟩} {⟨*tag*⟩} {⟨*offset*⟩} {⟨*ans*⟩}
`\__bnvs_if_append_n_post:vvncTF` {⟨*yes code*⟩} {⟨*no code*⟩}

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

```
\BNVS_new_conditional:cpnn { if_resolve_n_incr:nnnc } #1 #2 #3 #4 { T, TF } {
```

Resolve the ⟨*offset*⟩ into the ⟨*ans*⟩ variable.

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

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

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

```
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    } {
```

The ⟨*offset*⟩ does not resolve to 0.

```
      \__bnvs_tl_put_right:cn { #4 } { + }
      \__bnvs_if_append_n:nncTF { #1 } { #2 } { #4 } {
        \__bnvs_round:c { #4 }
        \__bnvs_gset:nnnv n { #1 } { #2 } { #4 }
        \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    }
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn
  { if_append_n_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_n_incr:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \BNVS_end_tl_put_right:cv { #4 } { #4 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvnc:cn { if_append_n_incr } { T, F, TF }
```

__bnvs_if_resolve_v_post:nnncTF __bnvs_if_resolve_v_post:nnncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*offset*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_v_post:nnncTF
__bnvs_if_append_v_post:vvncTF __bnvs_if_append_v_post:nnncTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*offset*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

Resolve the value of the free counter for the given ⟨*tag*⟩ into the ⟨*ans*⟩ `tl` variable then increment this free counter position accordingly. The append version, appends the value to the right of the ⟨*ans*⟩ `tl` variable. The content of ⟨*ans*⟩ is undefined while in the ⟨*no code*⟩ branch and on resolution only.

```
\BNVS_new_conditional:cpnn { if_resolve_n_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_resolve_n:nncTF { #1 } { #2 } { #4 } {
    \BNVS_begin:
```

```
    \__bnvs_if_resolve:ncTF { #3 } { #4 } {
      \BNVS_tl_use:Nv \int_compare:nNnTF { #4 } = 0 {
        \BNVS_end:
        \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \__bnvs_tl_put_right:cn { #4 } { + }
        \__bnvs_if_append_n:nncTF { #1 } { #2 } { #4 } {

          \__bnvs_round:c { #4 }
          \__bnvs_gset:nnnv n { #1 } { #2 } { #4 }
          \BNVS_end:
          \__bnvs_if_resolve_index:nnvcTF { #1 } { #2 } { #4 } { #4 } {
            \prg_return_true:
          } {
            \prg_return_false:
          }
        } {
          \BNVS_end:
          \prg_return_false:
        }
      }
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_append_n_post:nnnc } #1 #2 #3 #4{ T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_n_post:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \BNVS_end_tl_put_right:cv { #4 } { #4 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vvnc:cn { if_append_n_post } { T, F, TF }
```

6.20 Functions for the resolution

They manily start with `\__bnvs_if_resolve_` or `\__bnvs_split_`

`\__bnvs_split_pop_iksp:TFF`	`\__bnvs_split_pop_iksp:TFF {`⟨*black code*⟩`} {`⟨*blank code*⟩`}`
`\__bnvs_split_end_return_or_pop_complete:T {`⟨*end code*⟩`}`	
`\__bnvs_split_end_return_or_pop_void:T`	`\__bnvs_split_end_return_or_pop_complete:T {`⟨*blank code*⟩`}`
	`\__bnvs_split_end_return_or_pop_void:T {`⟨*black code*⟩`}`

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

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

For `\__bnvs_split_end_return_or_pop_void:T`: pops the eight heading items of the `split` sequence then execute ⟨*blank code*⟩.

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

```
\BNVS_new:cpn { split_pop_iksp:TFF } #1 #2 #3 {
  \__bnvs_split_if_pop_left:cTF { id } {
    \__bnvs_split_if_pop_left:cTF { kri } {
      \__bnvs_split_if_pop_left:cTF { short } {
        \__bnvs_split_if_pop_left:cTF { path } {
          \__bnvs_tl_if_blank:vTF { short } {
```

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

```
            #2
          } {
            \BNVS_tl_use:nv {
              \regex_match:NnT \c__bnvs_A_reserved_Z_regex
            } { short } {
              \__bnvs_tl_if_eq:cnF { short } { pauses } {
                \__bnvs_tl_if_eq:cnF { short } { slideinframe } {
\BNVS_error:x { Use~of~reserved~``\BNVS_tl_use:c { tag }'' }
                }
              }
            }
            \__bnvs_tl_if_blank:vTF { kri } {
              \__bnvs_tl_set:cv { id } { id_last }
            } {
              \__bnvs_tl_set:cv { id_last } { id }
            }
```

Build the `path` sequence and lowercase components conditionals.

```
            \__bnvs_seq_set_split:cnv { path } { . } { path }
            #1
          }
        } {
          \BNVS_fatal:n { split_pop_iksp:TFF/path }
        }
      } {
        \BNVS_fatal:n { split_pop_iksp:TFF/short }
      }
    } {
```

```
      \BNVS_fatal:n { split_pop_iksp:TFF/kri }
    }
  } {
    #3
  }
}
```

conditional variants.

```
\BNVS_new:cpn { split_end_return_or_pop_complete:T } #1 {
  \cs_set:Npn \BNVS_split_F:n ##1 {
    \BNVS_end_unreachable_return_false:n {
      split_end_return_or_pop_complete: ##1
    }
  }
  \__bnvs_split_if_pop_left_or:cT { n_incr } {
    \__bnvs_split_if_pop_left_or:cT { plus } {
      \__bnvs_split_if_pop_left_or:cT { rhs } {
        \__bnvs_split_if_pop_left_or:cT { post } {
          #1
        }
      }
    }
  }
}
```

```
\BNVS_new:cpn { split_end_return_or_pop_void:T } #1 {
  \cs_set:Npn \BNVS_split_F:n ##1 {
    \BNVS_end_unreachable_return_false:n {
      split_end_return_or_pop_void: ##1
    }
  }
  \__bnvs_split_if_pop_left:cTn { a } {
    \__bnvs_split_if_pop_left:cTn { a } {
      \__bnvs_split_if_pop_left:cTn { a } {
        \__bnvs_split_if_pop_left:cTn { a } {
          \__bnvs_split_if_pop_left:cTn { a } {
            \__bnvs_split_if_pop_left:cTn { a } {
              \__bnvs_split_if_pop_left:cTn { a } {
                \__bnvs_split_if_pop_left:cTn { a } {
                  \__bnvs_split_if_pop_left:cTn { a } {
                    #1
                  } { T/8 }
                } { T/7 }
              } { T/6 }
            } { T/5 }
          } { T/4 }
        } { T/3 }
      } { T/2 }
    } { T/1 }
}
```

__bnvs_if_resolve:ncTF __bnvs_if_resolve:ncTF {⟨*expression*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_resolve:vcTF __bnvs_if_append:ncTF {⟨*expression*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append:ncTF
__bnvs_if_append:vcTF

Resolves the ⟨*expression*⟩, replacing all the named overlay specifications by their static counterpart then put the rounded result in ⟨*ans*⟩ `tl` variable when resolving or to the right of this variable when appending.

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

\l__bnvs_ans_tl To feed ⟨*tl variable*⟩ with.

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

\l__bnvs_split_seq The sequence of catched query groups and non queries.

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

\l__bnvs_split_int Is the index of the non queries, before all the catched groups.

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

```
\BNVS_int_new:c { split }
```

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

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

\l__bnvs_path_tl Storage for `split` sequence items that represent integer paths.

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

Catch circular definitions. Open a main TEX group to define local functions and variables, sometimes another grouping level is used. The main TEX group is closed in the various `\...end_return...` functions.

```
\BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
  \BNVS_begin:
  \__bnvs_if_resolve:ncTF { #1 } { #2 } {
    \BNVS_end_tl_put_right:cv { #2 } { #2 }
    \prg_return_true:
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { if_append } { T, F, TF }
```

Heavily used.

```
\cs_new:Npn \BNVS_end_unreachable_return_false:n #1 {
  \BNVS_error:n { UNREACHABLE/#1 }
  \BNVS_end:
  \prg_return_false:
}
\cs_new:Npn \BNVS_end_unreachable_return_false:x #1 {
  \BNVS_error:x { UNREACHABLE/#1 }
  \BNVS_end:
```

```
  \prg_return_false:
}

\BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { TF } {
  \__bnvs_if_call:TF {
    \BNVS_begin:
```

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

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

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

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

Normal and unique end of the loop.

```
        \__bnvs_if_resolve_round_ans:
        \BNVS_end_tl_set:cv { #2 } { ans }
        \prg_return_true:
      }
```

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

```
      \BNVS_set:cpn { if_resolve_round_ans: } { \__bnvs_round:c { ans } }
      \__bnvs_tl_clear:c { ans }
      \__bnvs_split_loop_or_end_return:
    } {
```

There is not reference.

```
      \__bnvs_tl_set:cn { ans } { #1 }
      \__bnvs_round:c { ans }
      \BNVS_end_tl_set:cv { #2 } { ans }
      \prg_return_true:
    }
  } {
    \BNVS_error:n { TOO_MANY_NESTED_CALLS/Resolution }
    \BNVS_end:
    \prg_return_false:
  }
}
\BNVS_new_conditional_vc:cn { if_resolve } { T, F, TF }


\BNVS_new:cpn { build_tag: } {
  \__bnvs_tl_set_eq:cc { tag } { short }
  \__bnvs_seq_map_inline:cn { path } {
    \__bnvs_tl_put_right:cn { tag} { . ##1 }
  }
}
\BNVS_new:cpn { build_tag_head: } {
  \__bnvs_tl_set_eq:cc { tag } { short }
  \__bnvs_seq_map_inline:cn { path_head } {
    \__bnvs_tl_put_right:cn { tag } { . ##1 }
  }
}
```

`\__bnvs_split_loop_or_end_return:` `\__bnvs_split_loop_or_end_return:`

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

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

\BNVS_new_conditional:cpnn { if:c } #1 { p, T, F, TF } {
  \bool_if:cTF { l__bnvs_#1_bool } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { bool_if_exist:c } #1 { p, T, F, TF } {
  \bool_if_exist:cTF { l__bnvs_#1_bool } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new:cpn { prepare_context:N } #1 {
  \clist_map_inline:nn {
    n, v, reset, reset_all, first, last, length,
    previous, next, range, assign, only
  } {
    \__bnvs_set_false:c { ##1 }
  }
  \__bnvs_seq_clear:c { path_head }
  \__bnvs_seq_clear:c { path_tail }
  \__bnvs_tl_clear:c { index }
  \__bnvs_tl_clear:c { suffix }
  \BNVS_set:cpn { :n } ##1 {
    \tl_if_blank:nF { ##1 } {
      \__bnvs_tl_if_empty:cF { index } {
         \__bnvs_seq_put_right:cv { path_head } { index }
        \__bnvs_tl_clear:c { index }
      }
      \__bnvs_seq_put_right:cn { path_head } { ##1 }
    }
  }
  \__bnvs_seq_map_inline:cn { path } {
    \__bnvs_bool_if_exist:cTF { ##1 } {
      \__bnvs_set_true:c { ##1 }
      \clist_if_in:nnF { n, v, reset, reset_all } { ##1 } {
```

```
        \bool_if:NT #1 {
          \BNVS_error:n {Unexpected~##1~in~assignment }
        }
        \__bnvs_tl_set:cn { suffix } { ##1 }
      }
      \BNVS_set:cpn { :n } ####1 {
        \tl_if_blank:nF { ####1 } {
          \BNVS_error:n {Unexpected~####1 }
        }
      }
    } {
      \regex_match:NnTF \c__bnvs_A_index_Z_regex { ##1 } {
        \__bnvs_tl_if_empty:cF { index } {
         \__bnvs_seq_put_right:cv { path_head } { index }
        }
        \__bnvs_tl_set:cn { index } { ##1 }
      } {
        \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { ##1 } {
          \BNVS_error:n { Unsupported~##1 }
        } {
          \__bnvs_:n { ##1 }
        }
      }
    }
  }
  \__bnvs_seq_set_eq:cc { path } { path_head }
}

\BNVS_new:cpn { split_loop_or_end_return: } {
  \__bnvs_split_if_pop_left:cTF { a } {
    \__bnvs_tl_put_right:cv { ans } { a }
    \__bnvs_split_pop_iksp:TFF {
      \__bnvs_split_end_return_or_pop_void:T {
        \__bnvs_prepare_context:N \c_true_bool
        \__bnvs_build_tag:
        \__bnvs_split_loop_or_end_return_iadd:n { 1 }
      }
    } {
      \__bnvs_split_pop_iksp:TFF {
        \__bnvs_split_end_return_or_pop_complete:T {
          \__bnvs_tl_if_blank:vTF { n_incr } {
            \__bnvs_tl_if_blank:vTF { plus } {
              \__bnvs_tl_if_blank:vTF { rhs } {
                \__bnvs_tl_if_blank:vTF { post } {
                  \__bnvs_prepare_context:N \c_false_bool
                  \__bnvs_build_tag:
```

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

```
                  \__bnvs_tl_if_empty:cTF { index } {
                    \__bnvs_tl_if_empty:cTF { suffix } {
                      \__bnvs_split_loop_or_end_return_v:
                    } {
                      \__bnvs_split_loop_or_end_return_suffix:
                    }
                  } {
```

```
                  \__bnvs_split_loop_or_end_return_index:
                }
              } {
                \__bnvs_prepare_context:N \c_true_bool
                \__bnvs_build_tag:
                \BNVS_use:c { split_loop_or_end_return[...++]: }
              }
            } {
              \__bnvs_prepare_context:N \c_true_bool
              \__bnvs_build_tag:
              \__bnvs_split_loop_or_end_return_assign:
            }
          } {
            \__bnvs_if_resolve:vcTF { rhs } { rhs } {
              \__bnvs_prepare_context:N \c_true_bool
              \__bnvs_build_tag:
              \BNVS_tl_use:Nv
                \__bnvs_split_loop_or_end_return_iadd:n { rhs }
            } {
              \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs }}
              \__bnvs_split_loop_or_end_return:
            }
          }
        } {
          \__bnvs_prepare_context:N \c_true_bool
          \__bnvs_build_tag:
          \__bnvs_set_true:c { n }
          \__bnvs_split_loop_or_end_return_iadd:n { 1 }
        }
      }
    } {
\BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/3 }
    } {
\BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/2 }
    }
  } {
```

The `split` sequence is empty.

```
    \__bnvs_if_resolve_end_return_true:
  }
 } {
\BNVS_end_unreachable_return_false:n { split_loop_or_end_return:/1 }
 }
}


\BNVS_new_conditional:cpnn { if_suffix: } { T, F, TF } {
 \__bnvs_tl_if_empty:cTF { suffix } {
  \__bnvs_seq_pop_right:ccTF { path } { suffix } {
   \prg_return_true:
  } {
   \prg_return_false:
  }
 } {
```

```
    \prg_return_true:
  }
}
```

Implementation detail: `tl` variable `a` is used.

```
\BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {

}

\BNVS_new:cpn { error_end_return_false:n } #1 {
  \__bnvs_build_tag:
  \__bnvs_tl_set:cx { a } {
     \BNVS_tl_use:c { tag } . \BNVS_tl_use:c { suffix }
  }
  \__bnvs_if_resolve_v:vvcTF { id } { a } { a } {
    \__bnvs_tl_put_right:cv { ans } { a }
    \__bnvs_split_loop_or_end_return:
  } {
    \__bnvs_if_resolve_V:vvcTF { id } { a } { a } {
      \__bnvs_tl_put_right:cv { ans } { a }
      \__bnvs_split_loop_or_end_return:
    } {
      #1
    }
  }
}

\BNVS_new:cpn { path_branch_loop_or_end_return: } {
  \__bnvs_if_call:TF {
    \__bnvs_if_path_branch:TF {
      \__bnvs_path_branch_end_return:
    } {
      \__bnvs_if_get:nvvcTF V { id } { tag } { a } {
        \__bnvs_if_TIP:cccTF { id } { a } { path } {
          \__bnvs_tl_set_eq:cc { tag } { a }
          \__bnvs_seq_merge:cc { path } { path_tail }
          \__bnvs_seq_clear:c { path_tail }
          \__bnvs_seq_set_eq:cc { path_head } { path }
          \__bnvs_path_branch_TIPn_loop_or_end_return:
        } {
          \__bnvs_path_branch_head_to_tail_end_return:
        }
      } {
        \__bnvs_path_branch_head_to_tail_end_return:
      }
    }
  } {
    \__bnvs_path_branch_end_return_false:n {
      Too~many~calls.
    }
  }
}

\BNVS_new:cpn { path_branch_end_return: } {
  \__bnvs_split_loop_or_end_return:
}
```

```
\BNVS_new:cpn { set_if_path_branch:n } {
  \prg_set_conditional:Npnn \__bnvs_if_path_branch: { TF }
}

\BNVS_new:cpn { path_branch_head_to_tail_end_return: } {
  \__bnvs_seq_pop_right:ccTF { path_head } { a } {
    \__bnvs_seq_put_left:cv { path_tail } { a }
    \__bnvs_build_tag_head:
    \__bnvs_path_branch_TIPn_loop_or_end_return:
  } {
    \__bnvs_build_tag:
    \__bnvs_seq_set_eq:cc { path_head } { path_tail }
    \__bnvs_seq_clear:c { path_tail }
    \__bnvs_is_gset:nvxTF V { id } {
      \__bnvs_tl_use:c { tag }.1
    } {
      \__bnvs_tl_set:cn { index } { 1 }
      \__bnvs_split_loop_or_end_return_index:
    } {
      \__bnvs_gset:nvvn       V { id } { tag } { 0 }
      \__bnvs_gset_cache:nvvn V { id } { tag } { 0 }
      \__bnvs_path_branch_TIPn_loop_or_end_return:
    }
  }
}
```

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

```
\BNVS_new:cpn { path_branch_TIPn_loop_or_end_return: } {
  \__bnvs_build_tag_head:
  \__bnvs_if_resolve_v:vvcTF { id } { tag } { a } {
    \__bnvs_tl_put_right:cv { ans } { a }
    \__bnvs_split_loop_or_end_return:
  } {
    \__bnvs_if_resolve_V:vvcTF { id } { tag } { a } {
      \__bnvs_tl_put_right:cv { ans } { a }
      \__bnvs_split_loop_or_end_return:
    } {
      \__bnvs_path_branch_loop_or_end_return:
    }
  }
}
```

- Case `....`⟨*index*⟩.

```
\BNVS_new:cpn { split_loop_or_end_return_index: } {
  % known, id, tag, path, suffix
  \__bnvs_set_if_path_branch:n {
    \__bnvs_if_append_index:vvvcTF { id } { tag } { index } { ans } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
```

```
  \__bnvs_path_branch_loop_or_end_return:
}

\BNVS_new:cpn { split_loop_reset: } {
  \__bnvs_if:cT { reset_all } {
    \__bnvs_set_false:c { reset_all }
    \__bnvs_greset_cache:
  }
  \__bnvs_if:cT { reset } {
    \__bnvs_set_false:c { reset }
    \__bnvs_gunset:nvv v { id } { tag }
  }
}
```

- Case `....`

```
\BNVS_new:cpn { split_loop_or_end_return_v: } {
  \__bnvs_split_loop_reset:
  \__bnvs_set_if_path_branch:n {
    \__bnvs_if_append_v:vvcTF { id } { tag } { ans } {
      \prg_return_true:
    } {
      \__bnvs_if_append_V:vvcTF { id } { tag } { ans } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
  \__bnvs_path_branch_loop_or_end_return:
}
```

- Case `....<suffix>`.

```
\BNVS_new:cpn { split_loop_or_end_return_suffix: } {
 \__bnvs_if_resolve_V_loop_or_end_return_true:F {
    \__bnvs_set_if_path_branch:n {
      \BNVS_use:c {
        if_append_ \__bnvs_tl_use:c { suffix } :vvcTF
      } { id } { tag } { ans } {
        \__bnvs_if:cT { range } {
          \BNVS_set:cpn { if_resolve_round_ans: } { }
        }
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}
```

- Case `...++`.

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

- Case `....reset++`.

```
    \cs_set:Npn \BNVS_split_loop: {
      NO~....reset++~for
        ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
    }
  } {
```

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

```
    \cs_set:Npn \BNVS_split_loop: {
      \BNVS_error_ans:x {
        NO~...(.reset_all)++~for
          ~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }
      }
    }
  }
  \__bnvs_build_tag:
  \__bnvs_split_loop_reset:
  \__bnvs_if_append_v_post:vvncTF { id } { tag } { 1 } { ans } {
  } {
    \BNVS_error_ans:x {
      Problem~with~\BNVS_tl_use:c { id }!\BNVS_tl_use:c { tag }~use.
    }
  }
  \__bnvs_split_loop_or_end_return:
}

\BNVS_new:cpn { split_loop_or_end_return_assign: } {
```

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

```
  \__bnvs_if_resolve:vcTF { rhs } { rhs } {
    \__bnvs_gset:nvvv v { id } { tag } { rhs }
    \__bnvs_if_append_v:vvcTF { id } { tag } { ans } {
    } {
      \BNVS_error_ans:n { No~...=... }
    }
  } {
    \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c { rhs }. }
  }
  \__bnvs_split_loop_or_end_return:
}
```

- Case `...+=...`.

```
\BNVS_new:cpn { split_loop_or_end_return_iadd:n } #1 {
  \__bnvs_if_resolve:ncTF { #1 } { rhs } {
    \__bnvs_split_loop_reset:
    \__bnvs_if_append_v_incr:vvncTF { id } { tag } { #1 } { ans } {
```

```
    } {
      \BNVS_error_ans:n { No~...+=... }
    }
  } {
    \BNVS_error_ans:x { Error~in~\BNVS_tl_use:c { rhs } }
  }
  \__bnvs_split_loop_or_end_return:
}
```

`\__bnvs_if_resolve_query:ncTF` `\__bnvs_if_resolve_query:ncTF {⟨overlay query⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}`

Evaluates the single ⟨*overlay query*⟩, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of te ⟨*ans*⟩ `tl` variable. Ranges are supported with the colon syntax. This is executed within a local 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`.)

```
\BNVS_new:cpn { resolve_query_end_return_true: } {
  \BNVS_end:
  \prg_return_true:
}

\BNVS_new:cpn { resolve_query_end_return_false: } {
  \BNVS_end:
  \prg_return_false:
}

\BNVS_new:cpn { resolve_query_end_return_false:n } #1 {
  \BNVS_end:
  \prg_return_false:
}

\BNVS_new:cpn { if_resolve_query_return_false:n } #1 {
  \prg_return_false:
}
```

```
\BNVS_new:cpn { resolve_query_error_return_false:n } #1 {
  \BNVS_error:n { #1 }
  \__bnvs_if_resolve_query_return_false:
}
\BNVS_generate_variant:cn { resolve_query_error_return_false:n } { x }

\BNVS_new:cpn { if_resolve_query_return_unreachable: } {
  \__bnvs_resolve_query_error_return_false:n { UNREACHABLE }
}

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

\BNVS_new_conditional:cpnn { if_match_pop_left:c } #1 { T, F, TF } {
  \BNVS_tl_use:nc {
    \BNVS_seq_use:Nc \seq_pop_left:NNTF { match }
  } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

`\__bnvs_if_resolve_query_branch:`*TF* `\__bnvs_if_resolve_query_branch:TF {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Called by `\__bnvs_if_resolve_query:ncTF` that just filled `\l__bnvs_match_seq` after the `c__bnvs_A_cln_Z_regex`. Puts the proper items of `\l__bnvs_match_seq` into the variables `\l__bnvs_V_tl`, `\l__bnvs_A_tl`, `\l__bnvs_Z_tl`, `\l__bnvs_L_tl` then branches accordingly on one of the returning
`\__bnvs_if_resolve_query_return[`⟨*description*⟩`]:`
functions. All these functions properly set the `\l__bnvs_ans_tl` variable and they end with either `\prg_return_true:` or `\prg_return_false:`. This is used only once but is not inlined for readability.

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

At start, we ignore the whole match.

```
  \__bnvs_if_match_pop_left:cT V {
    \__bnvs_if_match_pop_left:cT V {
      \__bnvs_if_blank:cTF V {
        \__bnvs_if_match_pop_left:cT A {
          \__bnvs_if_match_pop_left:cT Z {
            \__bnvs_if_match_pop_left:cT L {
              \__bnvs_if_blank:cTF A {
                \__bnvs_if_match_pop_left:cT L {
                  \__bnvs_if_match_pop_left:cT Z {
                    \__bnvs_if_blank:cTF L {
                      \__bnvs_if_match_pop_left:cT Z {
                        \__bnvs_if_match_pop_left:cT L {
                          \__bnvs_if_blank:cTF L {
                            \BNVS_use:c { if_resolve_query_return[:Z]: }
                          } {
                            \BNVS_use:c { if_resolve_query_return[:Z::L]: }
                          }
```

```
                        }
                      }
                    } {
                      \__bnvs_if_blank:cTF Z {
\__bnvs_resolve_query_error_return_false:n { Missing~first~or~last }
                      } {
                        \BNVS_use:c { if_resolve_query_return[:Z::L]: }
                      }
                    }
                  }
                }
              } {
                \__bnvs_if_blank:cTF Z {
                  \__bnvs_if_blank:cTF L {
                    \BNVS_use:c { if_resolve_query_return[A:]: }
                  } {
                    \BNVS_use:c { if_resolve_query_return[A::L]: }
                  }
                } {
                  \__bnvs_if_blank:cTF L {
                    \BNVS_use:c { if_resolve_query_return[A:Z]: }
                  } {
```

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

```
                    \__bnvs_if_resolve_query_return_unreachable:
                  }
                }
              }
            }
          }
        }
      } {
        \BNVS_use:c { if_resolve_query_return[V]: }
      }
    }
  }
}
```

Single value

```
\BNVS_new:cpn { if_resolve_query_return[V]: } {
  \__bnvs_if_resolve:vcTF { V } { ans } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

⟨*first*⟩:⟨*last*⟩ range

```
\BNVS_new:cpn { if_resolve_query_return[A:Z]: } {
  \__bnvs_if_resolve:vcTF { A } { ans } {
    \__bnvs_tl_put_right:cn { ans } { - }
    \__bnvs_if_append:vcTF { Z } { ans } {
      \prg_return_true:
    } {
```

```
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
```

⟨*first*⟩::⟨*length*⟩ range

```
\BNVS_new:cpn { if_resolve_query_return[A::L]: } {
  \__bnvs_if_resolve:vcTF { A } { A } {
    \__bnvs_if_resolve:vcTF { L } { ans } {
      \__bnvs_tl_put_right:cn { ans } { + }
      \__bnvs_tl_put_right:cv { ans } { A }
      \__bnvs_tl_put_right:cn { ans } { -1 }
      \__bnvs_round:c { ans }
      \__bnvs_tl_put_left:cn { ans } { - }
      \__bnvs_tl_put_left:cv { ans } { A }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
```

⟨*first*⟩: and ⟨*first*⟩:: range

```
\BNVS_new:cpn { if_resolve_query_return[A:]: } {
  \__bnvs_if_resolve:vcTF { A } { ans } {
    \__bnvs_tl_put_right:cn { ans } { - }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

:⟨*last*⟩::⟨*length*⟩ or ::⟨*length*⟩:⟨*last*⟩ range

```
\BNVS_new:cpn { if_resolve_query_return[:Z::L]: } {
  \__bnvs_if_resolve:vcTF { Z } { Z } {
    \__bnvs_if_resolve:vcTF { L } { ans } {
      \__bnvs_tl_put_left:cn  { ans } { 1-}
      \__bnvs_tl_put_right:cn { ans } { + }
      \__bnvs_tl_put_right:cv { ans } { Z }
      \__bnvs_round:c { ans }
      \__bnvs_tl_put_right:cn { ans } { - }
      \__bnvs_tl_put_right:cv { ans } { Z }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
```

: or :: range

```
\BNVS_new:cpn { if_resolve_query_return[:]: } {
  \__bnvs_tl_set:cn { ans } { - }
  \prg_return_true:
}
```

:⟨*last*⟩ range

```
\BNVS_new:cpn { if_resolve_query_return[:Z]: } {
  \__bnvs_tl_set:cn { ans } { - }
  \__bnvs_if_append:vcTF { Z } { ans } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_if_resolve_query:nc*TF* __bnvs_if_resolve_query:ncTF {⟨*query*⟩} {⟨*tl core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

Evaluate only one query.

```
\BNVS_new_conditional:cpnn { if_resolve_query:nc } #1 #2 { T, F, TF } {
  \__bnvs_greset_call:
  \__bnvs_match_if_once:NnTF \c__bnvs_A_cln_Z_regex { #1 } {
    \BNVS_begin:
    \__bnvs_if_resolve_query_branch:TF {
      \BNVS_end_tl_set:cv { #2 } { ans }
      \prg_return_true:
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
    \BNVS_error:n { Syntax~error:~#1 }
    \BNVS_end:
    \prg_return_false:
  }
}
```

__bnvs_if_resolve_queries:nc*TF* __bnvs_if_resolve_queries:ncTF {⟨*overlay query list*⟩} {⟨*ans*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

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

\l__bnvs_query_seq Storage for a sequence of ⟨*query*⟩'s obtained by splitting a comma separated list.

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

\l__bnvs_ans_seq Storage for the evaluated result.

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

\c__bnvs_comma_regex Used to parse slide range overlay specifications.

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

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

No other variable is used.

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

Local variables cleared

```
  \__bnvs_seq_clear:c { ans }
```

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

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

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

```
  \BNVS_set:cpn { end_return: } {
    \__bnvs_seq_if_empty:cTF { ans } {
      \BNVS_end:
    } {
```

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

```
      \exp_args:Nnx
      \use:n {
        \BNVS_end:
        \__bnvs_tl_put_right:cn { #2 }
      } { \__bnvs_seq_use:cn { ans } , }
    }
    \prg_return_true:
  }
  \__bnvs_seq_map_inline:cn { query } {
    \__bnvs_tl_clear:c { ans }
```

```
    \__bnvs_if_resolve_query:ncTF { ##1 } { ans } {
      \__bnvs_tl_if_empty:cF { ans } {
        \__bnvs_seq_put_right:cv { ans } { ans }
      }
    } {
      \seq_map_break:n {
        \BNVS_set:cpn { end_return: } {
          \BNVS_error:n { Circular/Undefined~dependency~in~#1}
          \exp_args:Nnx
          \use:n {
            \BNVS_end:
            \__bnvs_tl_put_right:cn { #2 }
          } { \__bnvs_seq_use:cn { ans } , }
          \prg_return_false:
        }
      }
    }
  }
  \__bnvs_end_return:
}


\NewDocumentCommand \BeanovesResolve { O{} m } {
  \BNVS_begin:
  \keys_define:nn { BeanovesResolve } {
    in:N .tl_set:N = \l__bnvs_resolve_in_tl,
    in:N .initial:n = { },
    show .bool_set:N = \l__bnvs_resolve_show_bool,
    show .default:n = true,
    show .initial:n = false,
  }
  \keys_set:nn { BeanovesResolve } { #1 }
  \__bnvs_tl_clear:c { ans }
  \__bnvs_if_resolve_queries:ncTF { #2 } { ans } {
    \__bnvs_tl_if_empty:cTF { resolve_in } {
      \bool_if:nTF { \l__bnvs_resolve_show_bool } {
        \BNVS_tl_use:Nv \BNVS_end: { ans }
      } {
        \BNVS_end:
      }
    } {
      \bool_if:nTF { \l__bnvs_resolve_show_bool } {
        \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
          \BNVS_end:
          \tl_set:Nn ##1 { ##2 }
          ##2
        }
        \BNVS_tl_use:nv {
          \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
        } { ans }
      } {
        \cs_set:Npn \BNVS_end:Nn ##1 ##2 {
          \BNVS_end:
          \tl_set:Nn ##1 { ##2 }
```

```
        }
        \BNVS_tl_use:nv {
          \exp_last_unbraced:NV \BNVS_end:Nn \l__bnvs_resolve_in_tl
        } { ans }
      }
    }
  } {}
}
```

6.21 Reseting counters and values

```
\BNVS_new:cpn { reset:n } #1 {
  \BNVS_begin:
  \__bnvs_set_true:c { reset }
  \__bnvs_set_false:c { provide }
  \__bnvs_tl_clear:c { root }
  \__bnvs_int_zero:c { i }
  \__bnvs_tl_set:cn { a } { #1 }
  \__bnvs_provide_off:
  \BNVS_tl_use:Nv \__bnvs_brace_keyval:n { a }
  \BNVS_end_tl_set:cv { id_last } { id_last }
}

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

\makeatletter
\NewDocumentCommand \BeanovesReset { O{} m } {
  \tl_if_empty:NTF \@currenvir {
```

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

```
    \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}~in~the~preamble.}
  } {
    \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, clear everything.

```
     \BNVS_error:x {No~\token_to_str:N \BeanovesReset{}~at~the~top~level.}
    }
    \BNVS_begin:
    \__bnvs_set_true:c { reset }
    \__bnvs_set_false:c { provide }
    \keys_define:nn { BeanovesReset } {
      all .bool_set:N = \l__bnvs_reset_all_bool,
      all .default:n = true,
      all .initial:n = false,
      only .bool_set:N = \l__bnvs_only_bool,
      only .default:n = true,
      only .initial:n = false,
    }
    \keys_set:nn { BeanovesReset } { #1 }
    \__bnvs_tl_clear:c { root }
    \__bnvs_int_zero:c { i }
    \__bnvs_tl_set:cn { a } { #2 }
```

```
    \__bnvs_provide_off:
    \BNVS_tl_use:Nv \__bnvs_brace_keyval:n { a }
    \BNVS_end_tl_set:cv { id_last } { id_last }
    \ignorespaces
  }
}
\makeatother

\ExplSyntaxOff
```