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[1].

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.

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[2].

`beanoves`

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

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. Notice that ⟨*ref*⟩`=1` can be shortened to ⟨*ref*⟩.

3.3.1 Value specifiers

Hereafter ⟨*value*⟩ denotes a numerical expression.

Standalone

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

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

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

[1] This will avoid collisions with the `fp` module of `expl3`.

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

3.3.2 Range specifiers

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

Standalone

⟨*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

⟨*ref*⟩=[⟨def_1⟩,...,⟨def_j⟩], where ⟨def_k⟩, $1 \leq k \leq j$, is one of

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

The first step is to remove previous ⟨*ref*⟩ related definitions, then execute the various ⟨*ref*⟩.⟨*i*⟩=⟨*value*⟩ definitions in the order given. Here is the implementation.

⟨*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. In a final step, the ⟨*name*⟩'s are collected in a comma sepated list to initialize ⟨*ref*⟩ with. ⟨*spec*⟩ is any specifier. Here is the implementation.

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 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*⟩, ⟨*length*⟩ and ⟨*i expr*⟩ 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*⟩`::` define 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.

Define 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*⟩`{`⟨*i expr*⟩`}`	⟨*first*⟩ + ⟨*i expr*⟩ − 1
⟨*ref*⟩`.previous`	⟨*first*⟩ − 1
⟨*ref*⟩`.first`	⟨*first*⟩
⟨*ref*⟩`.last`	⟨*last*⟩
⟨*ref*⟩`.next`	⟨*last*⟩ + 1
⟨*ref*⟩`.length`	⟨*length*⟩

Notice that the resolution of the ⟨*ref*⟩ overlay query is a beamer range and not an algebraic difference, negative integers do not make sense there while in beamer context.

In the frame example below, we use the `\BeanovesResolve` command for the demonstration. It is mainly used for debugging and testing purposes.

```
\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***⟩ `=` `[...]`
⟨***ref***⟩ `=` `{...}` See the list range specifiers in section 3.3.3.

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***⟩`{`⟨*i expr*⟩`}`	⟨*value*⟩ + ⟨*i expr*⟩ − 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[3].

⟨***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.

IN PROGRESS

This can be used for an indirection.

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

```
\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](A.C)       == \BeanovesResolve[show](A.20) == 20,
\BeanovesResolve[show](A.C+=10)   == \BeanovesResolve[show](A.20) == 20+10,
\BeanovesResolve[show](A.C)       == \BeanovesResolve[show](A.20) == 30,
\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 the last value it was initialized with, and that it is meant to be used inside a `frame` environment. When the `all` option is provided, some internals that were cached for performance reasons are cleared as well.

4.3 The `beamer` counters

While inside a `frame` environment, it is possible to save the current value of the `beamerpauses` counter that controls whether elements should appear on the current slide. For that, we can execute one of `\Beanoves{`⟨*ref*⟩`=pauses}` or in a query `?(...(`⟨*ref*⟩`=pauses)...)`. Then later on, we can use `?(...`⟨*ref*⟩`...)` to refer to this saved value in the same frame[4]. Next frame source is an example of usage.

```
\begin{frame}
\visible<+->{A}\\
\visible<+->{B\Beanoves{afterB=pauses}}\\
\visible<+->{C}\\
\visible<?(afterB)>{other C}\\
\visible<?(afterB.previous)>{other B}\\
\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}`

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

or inside a query `?(⟨...⟩(⟨ref⟩=slideinframe)⟨...⟩)`. It uses the numerical value of `\insertslideinframe`.

4.4 Multiple queries

It is possible to replace the comma separated list of queries `?(⟨query$_1$⟩),...,?(⟨query$_j$⟩)` with the shorter single query `?(⟨query$_1$⟩,...,⟨query$_j$⟩)`.

4.5 Frame id

Except for very special situations, the *frame ids* can be left unspecified. When no *frame id* was explicitly provided, beanoves uses the *last frame id* and if the resolution fails an empty *frame id*. At the beginning of each frame, the *last frame id* is set to the *frame id* of the current frame, which is denoted *current frame id* and is empty by default. Then it gets updated after each named reference resolution where a *frame id* is explicitly given. For example, the first time `A.1` reference is resolved within a given frame, it is first translated to ⟨*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. Another possibility offered by the beanoves package to share named overlay sets is a fall back mechanism, for example when `X!A` cannot be resolved, resolve `!A` instead.

4.6 Resolution command

`\BeanovesResolve` `\BeanovesResolve [⟨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).

```
1 ⟨@@=bnvs⟩
```

6.1 Package declarations

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

6.2 Overview

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

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

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

6.3 Facility layer: definitions and naming

In order to make the code shorter and easier to read during development, we add a layer over LaTeX3. The `c` and `v` argument specifiers take a slightly different meaning when used in a function which name contains `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 internally used to focus on the discrimining part of variable or function names.

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

```
\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_args:NNc \BNVS_use_raw:NN #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: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 } }
}

\tl_new:N \l__bnvs_last_unbraced_tl
\cs_new:Npn \BNVS_tl_last_unbraced:nv #1 {
  \tl_set:Nn \l__bnvs_last_unbraced_tl { #1 }
  \BNVS_tl_use:nc { \exp_last_unbraced:NV \l__bnvs_last_unbraced_tl }
}
```

```
\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 }
```

6.3.1 Debug

\BNVS_DEBUG_on:
\BNVS_DEBUG_off:
\BNVS_DEBUG_push:nn
\BNVS_DEBUG_pop:

\BNVS_DEBUG_on:
\BNVS_DEBUG_off:
\BNVS_DEBUG_push:nn {⟨*types on*⟩} {⟨*types off*⟩}
\BNVS_DEBUG_pop:

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

6.3.2 Facility layer: Functions

\BNVS_new:cpn
\BNVS_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 } }
}
```

6.3.3 logging

\BNVS_warning:n
\BNVS_warning:x
\BNVS_error:n
\BNVS_error:x
\BNVS_fatal:n
\BNVS_fatal:x

\BNVS_warning:n {⟨*message*⟩}
\BNVS_error:n {⟨*message*⟩}
\BNVS_fatal:n {⟨*message*⟩}

Very rudimentary. Non long message for error recovery.

```
\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_generate_variant:Nn \BNVS_error:n { x }

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

6.3.4 Facility layer: Variables

`\BNVS_N_new:c` `\BNVS_N_new:c {`⟨*type*⟩`}`
`\BNVS_v_new:c` `\BNVS_v_new:c {`⟨*type*⟩`}`

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

```
\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 #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 } }
}
\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 } }
}
```

`\BNVS_new_conditional:cpnn` `\BNVS_new_conditional:cpnn {⟨core⟩} ⟨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 }
}
```

6.3.5 Regex

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

6.3.6 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 under the hood.

```
\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_empty:cTF` {⟨*core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_tl_if_blank:v`*TF* `\__bnvs_tl_if_blank:vTF` {⟨*core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
`\__bnvs_tl_if_eq:cn`*TF* `\__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 }
```

6.3.7 Strings

`\__bnvs_str_if_eq:vvTF` `\__bnvs_str_if_eq:vvTF {⟨core⟩} {⟨tl⟩} {⟨yes code⟩} {⟨no code⟩}`

These are shortcuts to

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

```
\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 }
```

6.3.8 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_if_empty:cTF {⟨*seq core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_seq_get_right:cc*TF* __bnvs_seq_get_right:ccTF {⟨*seq core*⟩} {⟨*tl core*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_seq_pop_left:cc*TF*
__bnvs_seq_pop_right:cc*TF*

```
\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 }
```

6.3.9 Integers

__bnvs_int_new:c __bnvs_int_new:c {⟨*core*⟩}
__bnvs_int_use:c __bnvs_int_use:c {⟨*core*⟩}
__bnvs_int_zero:c __bnvs_int_incr:c {⟨*core*⟩}
__bnvs_int_inc:c __bnvs_int_decr:c {⟨*core*⟩}
__bnvs_int_decr:c __bnvs_int_set:cn {⟨*core*⟩} {⟨*value*⟩}
__bnvs_int_set:cn
__bnvs_int_set:cv

These are shortcuts to

- \int_new:c {l__bnvs_⟨*core*⟩_int}
- \int_use:c {l__bnvs_⟨*core*⟩_int}
- \int_incr:c {l__bnvs_⟨*core*⟩_int}
- \int_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 }
```

6.4 Debug facilities

Typesetting file `beanoves.dtx` creates both beanoves and beanoves-debug style files. The former is intended for everyday use whereas the latter contains supplemental debugging and testing facilities which are intentionally left undocumented. For example, we have aliases for `\group_begin:` and `\group_end:` to allow the display of supplemental informations while debugging. We also have some techniques for coverage as well as inline testing.

6.5 Local variables

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

\l___bnvs__I__tl: A frame identifier, contains a regex catched group.

\l___bnvs__J__tl: The last frame identifier

\l___bnvs__K__tl: `!` as regex catched group, if non empty.

\l___bnvs__S__tl: A short name, a regex catched group.

\l___bnvs__P__tl: A path, a regex catched group.

\l___bnvs__G__tl: `-` as regex catched group, if non empty.

\l___bnvs__N__tl: The representation of an unsigned decimal integer.

\l___bnvs__U__tl: The representation of an unsigned decimal integer.

\l___bnvs__R__tl: The rhs for assignment.

\l___bnvs__O__tl: The rhs for post assignment.

```
\tl_new:N \l__bnvs_J_tl
\tl_new:N \l__bnvs_I_tl
\tl_new:N \l__bnvs_K_tl
\tl_new:N \l__bnvs_S_tl
\tl_new:N \l__bnvs_P_tl
\tl_new:N \l__bnvs_G_tl
\tl_new:N \l__bnvs_N_tl
\tl_new:N \l__bnvs_U_tl
\tl_new:N \l__bnvs_R_tl
\tl_new:N \l__bnvs_O_tl
\tl_new:N \l__bnvs_T_tl
\tl_new:N \l__bnvs_V_tl
\tl_new:N \l__bnvs_A_tl
\tl_new:N \l__bnvs_F_tl
\tl_new:N \l__bnvs_L_tl
\tl_new:N \l__bnvs_Z_tl
\tl_new:N \l__bnvs_a_tl
\tl_new:N \l__bnvs_z_tl
```

```
\tl_new:N \l__bnvs_l_tl
\tl_new:N \l__bnvs_r_tl
\tl_new:N \l__bnvs_n_tl
\tl_new:N \l__bnvs_ref_tl
\tl_new:N \l__bnvs_v_tl
\tl_new:N \l__bnvs_b_tl
\tl_new:N \l__bnvs_c_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_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_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_P_seq
\seq_new:N \l__bnvs_P_head_seq
\seq_new:N \l__bnvs_P_tail_seq
\seq_new:N \l__bnvs_query_seq
\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
\bool_new:N \l__bnvs_no_range_bool
\bool_set_false:N \l__bnvs_no_range_bool

\BNVS_new:cpn { tl_clear_ans: } {
  \__bnvs_tl_clear:c { ans }
}

\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

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

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

6.6 Infinite loop management

Unending recursivity is managed here.

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

(*End of definition for* \g__bnvs_call_int.)

```
\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:
  }
}
```

6.7 Overlay specification

6.7.1 Registration

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

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

Create a unique name from the arguments.

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

\g__bnvs_I_seq List of registered identifiers.

(*End of definition for* \g__bnvs_I_seq.)

```
\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.

__bnvs_register:NNnn `\__bnvs_register:NNnn ⟨cs⟩ ⟨seq⟩ {⟨id⟩} {⟨tag⟩}`

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

```
\seq_new:N \l__bnvs_register_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 }
      \cs_set:Npn \BNVS_register: {
        \__bnvs_seq_put_right:cn { register } { #4 }
        \cs_set:Npn \BNVS_register: { }
      }
      \cs_set:Npn \BNVS_register:w ##1 ##2 {
        \str_compare:nNnTF { ##2 } < { #4 } {
          \__bnvs_seq_put_right:cn { register } { ##2 }
        } {
          \BNVS_register:
          \__bnvs_seq_put_right:cn { register } { ##2 }
          \cs_set:Npn \BNVS_register:w ####1 ####2 {
            \__bnvs_seq_put_right:cn { register } { ####2 }
          }
        }
      }
      \__bnvs_foreach_T:nNTF { #3 } \BNVS_register:w {
        \BNVS_register:
        \seq_gset_eq:NN #2 \l__bnvs_register_seq
      } {
        \BNVS_error:n { Unreachable/register:NNnn~id~#3 }
      }
    } {
      \seq_new:N #2
      \seq_gput_right:Nn #2 { #4 }
      \__bnvs_seq_clear:c { register }
      \cs_set:Npn \BNVS_register: {
        \__bnvs_seq_put_right:cn { register } { #3 }
        \cs_set:Npn \BNVS_register: {}
      }
      \cs_set:Npn \BNVS_register:w ##1 {
        \str_compare:nNnTF { ##1 } < { #3 } {
          \__bnvs_seq_put_right:cn { register } { ##1 }
        } {
          \BNVS_register:
          \__bnvs_seq_put_right:cn { register } { ##1 }
```

```
          \cs_set:Npn \BNVS_register:w ####1 {
            \__bnvs_seq_put_right:cn { register } { ####1 }
          }
        }
      }
      \__bnvs_foreach_I:N \BNVS_register:w
      \BNVS_register:
      \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_register_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, through `\__bnvs_name:nn` and `\__bnvs_name:nnn`.

```
\seq_new:N \l__bnvs_unregister_seq
\BNVS_new:cpn { unregister:NNnn } #1 #2 #3 #4 {
  \cs_if_exist:NT #1 {
    \cs_undefine:N #1
    \__bnvs_seq_clear:c { unregister }
    \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 } { ##2 }
        \cs_set:Npn \BNVS_unregister_NNnn:n ####1 { }
      } {
        \cs_set:Npn \BNVS_unregister_NNnn:w ####1 ####2 {
          \__bnvs_seq_put_right:cn { unregister } { ####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_seq
    } {
      \BNVS_error:n { Unreachable / unregister:NNnn~#3!#4 }
    }
    \BNVS_unregister_NNnn:n {
      \__bnvs_seq_clear:c { unregister }
      \cs_set:Npn \BNVS_unregister_NNnn:w ##1 {
        \str_compare:nNnTF { ##1 } < { #3 } {
          \__bnvs_seq_put_right:cn { unregister } { ##1 }
        } {
          \cs_set:Npn \BNVS_unregister_NNnn:n ####1 {
            \__bnvs_seq_put_right:cn { unregister } { ####1 }
          }
```

```
        }
      }
      \__bnvs_foreach_I:N \BNVS_unregister_NNnn:w
      \seq_gset_eq:NN \g__bnvs_I_seq \l__bnvs_unregister_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_registered: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*⟩`}`
`\__bnvs_foreach_break:` `\__bnvs_foreach:,`
`\__bnvs_foreach_break:n` `\__bnvs_foreach_break:n {`⟨*code*⟩`}`

Execute the ⟨*function:n*⟩ or the ⟨*code*⟩ for each declared identifier. The break commands work like `\seq_map_break:` and `\seq_map_break:n`.

```
\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
}

\cs_set_eq:NN \__bnvs_foreach_break:  \seq_map_break:
\cs_set_eq:NN \__bnvs_foreach_break:n \seq_map_break:n
```

`\__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. Both are the arguments passed to ⟨*function:nn*⟩ or ⟨*code*⟩ (through `##1` and `##2`).

```
\BNVS_new_conditional:cpnn { foreach_T:nN } #1 #2 { T, F, TF } {
  \if_cs_exist:w \__bnvs_id_seq:n { #1 } \cs_end:
    \seq_map_inline:cn { \__bnvs_id_seq:n { #1 } } { #2 { #1 } { ##1 } }
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi
}

\BNVS_new_conditional:cpnn { foreach_T:nn } #1 #2 { T, F, TF } {
  \if_cs_exist:w \__bnvs_id_seq:n { #1 } \cs_end:
    \cs_set:Npn \BNVS_foreach_T_nn:nn ##1 ##2 { #2 }
    \seq_map_inline:cn { \__bnvs_id_seq:n { #1 } }
      { \BNVS_foreach_T_nn:nn { #1 } { ##1 } }
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi
}
```

`\__bnvs_foreach_IT:N` `\__bnvs_foreach_IT:nNn` ⟨*before*⟩ ⟨*function:nn*⟩ ⟨*after*⟩
`\__bnvs_foreach_IT:n` `\__bnvs_foreach_IT:N` ⟨*function:nn*⟩
`\__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:NT } #1 #2 {
  \__bnvs_foreach_I:n {
    \__bnvs_foreach_T:nNT { ##1 } #1 { #2 }
  }
}

\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_key:N` `\__bnvs_foreach_key:N` ⟨*function:n*⟩
`\__bnvs_foreach_key:n` `\__bnvs_foreach_key:n` {⟨*code*⟩}
`\__bnvs_foreach_key_init:N` `\__bnvs_foreach_key_init:N` ⟨*function:n*⟩
`\__bnvs_foreach_key_init:n` `\__bnvs_foreach_key_init:n` {⟨*code*⟩}
`\__bnvs_foreach_key_main:N` `\__bnvs_foreach_key_main:N` ⟨*function:n*⟩
`\__bnvs_foreach_key_main:n` `\__bnvs_foreach_key_main:n` {⟨*code*⟩}
`\__bnvs_foreach_key_sub:N` `\__bnvs_foreach_key_sub:N` ⟨*function:n*⟩
`\__bnvs_foreach_key_sub:n` `\__bnvs_foreach_key_sub:n` {⟨*code*⟩}

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

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

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

\BNVS_new:cpn { foreach_key_init:N } {
  \tl_map_function:nN { 0 }
}

\BNVS_new:cpn { foreach_key_init:n } {
  \tl_map_inline:nn { 0 }
}

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

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

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

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

6.7.2 Basic model functions

__bnvs_gset:nnnn
__bnvs_gset:(nnnv|nnnx|vnnn|nvnn|vvnn|vvnv)
__bnvs_gset:nnv

__bnvs_gset:nnnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩} {⟨*spec*⟩}

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 \c__bnvs_A_reserved_Z_regex { #2 } {
    \use_none:n
  } {
    \__bnvs_register:nn { #1 } { #2 }
    \tl_gclear_new:c { g__bnvs_#1!#2/#3_tl }
    \tl_gset:cn { g__bnvs_#1!#2/#3_tl }
  }
}
```

```
\BNVS_new:cpn { gset:vnnn } {
  \BNVS_tl_use:cv { gset:nnnn }
}

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

\BNVS_new:cpn { gset:vvnn } {
  \BNVS_tl_use:Nvv \__bnvs_gset:nnnn
}

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

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

\BNVS_new:cpn { gset:vvnv } #1 #2 #3 {
  \BNVS_tl_use:nv {
    \__bnvs_gset:vvnn { #1 } { #2 } { #3 }
  }
}

\cs_generate_variant:Nn \__bnvs_gset:nnnn { nnnx }
```

__bnvs_gunset:nnn
__bnvs_gunset:vvn
__bnvs_gunset:nn
__bnvs_gunset:nv

__bnvs_gunset:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩}
__bnvs_gunset:nn {⟨*id*⟩} {⟨*tag*⟩}

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

```
\BNVS_new:cpn { gunset:nnn } #1 #2 #3 {
  \cs_undefine:c { g__bnvs_#1!#2/#3_tl }
}

\BNVS_new:cpn { gunset:vvn } {
  \BNVS_tl_use:Nvv \__bnvs_gunset:nnn
}

\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
    } {
      ##1 {
        \__bnvs_gunset:nnn { #1 } { #2 } { ####1 }
      }
    }
  }
}

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

\BNVS_new:cpn { gunset_deep:nn } #1 #2 {
  \__bnvs_foreach_T:nnT { #1 } {
    \tl_if_in:nnT { .. ##2 } { .. #2 . } {
      \__bnvs_gunset:nn { #1 } { ##2 }
    }
  } { }
  \__bnvs_gunset:nn { #1 } { #2 }
}

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

\BNVS_new:cpn { gunset:vn } {
  \BNVS_tl_use:cv { 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:nnT { #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_gclear_init:nn*TF*
__bnvs_gclear:nn*TF*
__bnvs_gclear:nv*TF*
__bnvs_gclear:*TF*
__bnvs_greset:nn*TF*
__bnvs_greset:nv*TF*
__bnvs_greset:*TF*

__bnvs_gclear:nn {⟨*id*⟩} {⟨*tag*⟩}
__bnvs_gclear:n {⟨*id*⟩}
__bnvs_gclear:
__bnvs_greset:nn {⟨*id*⟩} {⟨*tag*⟩}
__bnvs_greset:n {⟨*id*⟩}
__bnvs_greset:

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

```
\BNVS_new:cpn { greset:nn } #1 #2 {
  \__bnvs_foreach_T:nnT { #1 } {
    \tl_if_in:nnT { .. ##2 } { .. #2 . } {
      \tl_map_inline:nn {
        \__bnvs_foreach_key_main:n
        \__bnvs_foreach_key_sub:n
      } {
        ####1 {
          \__bnvs_gunset:nnn { #1 } { ##2 } { ########1 }
        }
      }
    }
  } { }
  \__bnvs_if_registered:nnT { #1 } { #2 } {
    \tl_map_inline:nn {
      \__bnvs_foreach_key_main:n
      \__bnvs_foreach_key_sub:n
    } {
      ##1 {
        \__bnvs_gunset:nnn { #1 } { #2 } { ####1 }
      }
    }
  }
}
```

Clears out every data associated to ⟨*id*⟩!⟨*tag*⟩, and ⟨*id*⟩!⟨*tag*⟩.⟨*subtag*⟩.

```
\seq_new:N \l__bnvs_gclear_nn_seq
\BNVS_new:cpn { gclear:nn } #1 #2 {
  \__bnvs_seq_clear:c { gclear_nn }
  \__bnvs_foreach_T:nnT { #1 } {
    \tl_if_in:nnT { .. ##2 } { .. #2 . } {
      \tl_map_inline:nn {
        \__bnvs_foreach_key_init:n
        \__bnvs_foreach_key_main:n
        \__bnvs_foreach_key_sub:n
      } {
        ####1 {
```

```
            \__bnvs_gunset:nnn { #1 } { ##2 } { ########1 }
          }
        }
        \__bnvs_seq_put_right:cn { gclear_nn } {
          \__bnvs_unregister:nn { #1 } { ##2 }
        }
      }
    } { \__bnvs_seq_use:cn { gclear_nn } {} }
    \__bnvs_if_registered:nnT { #1 } { #2 } {
      \tl_map_inline:nn {
        \__bnvs_foreach_key_init:n
        \__bnvs_foreach_key_main:n
        \__bnvs_foreach_key_sub:n
      } {
        ##1 {
          \__bnvs_gunset:nnn { #1 } { #2 } { ####1 }
        }
      }
      \__bnvs_unregister:nn { #1 } { #2 }
    }
}

\BNVS_new:cpn { gclear:n } #1 {
  \__bnvs_foreach_T:nNT { #1 } \__bnvs_gclear:nn {}
}

\BNVS_new:cpn { gclear: } {
  \__bnvs_foreach_IT:N \__bnvs_gclear:nn
}

\BNVS_new:cpn { greset:n } #1 {
  \__bnvs_foreach_T:nNT { #1 } \__bnvs_greset:nn {}
}

\BNVS_new:cpn { greset: } {
  \__bnvs_foreach_IT:N \__bnvs_greset:nn
}
```

__bnvs_is_gset:nnn*TF*	__bnvs_is_gset:nnnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_is_gset:(nvn|vvn|vxn)*TF*	__bnvs_is_gset:nnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_is_gset:nn*TF*	__bnvs_if_spec:nnnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_spec:nnn*TF*	__bnvs_if_spec:nnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_spec:nn*TF*	

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

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

```
\BNVS_new_conditional:cpnn { is_gset:nnn } #1 #2 #3 { T, F, TF } {
  \if_cs_exist:w \__bnvs_name:nnn { #1 } { #2 } { #3 } \cs_end:
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
}
```

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

\BNVS_new_conditional:cpnn { is_gset:vvn } #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 { #1 } { #2 } V {
    \prg_return_true:
  } {
    \__bnvs_is_gset:nnnTF { #1 } { #2 } A {
      \prg_return_true:
    } {
      \__bnvs_is_gset:nnnTF { #1 } { #2 } Z {
        \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 { #1 } {
      \prg_return_false:
    } {
      \__bnvs_is_gset:nnnTF { } { #2 } { #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:nnncTF
__bnvs_if_get:nncTF
__bnvs_if_get:vvcTF
__bnvs_if_spec:nnncTF

__bnvs_if_get:nnncTF {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_spec:nnncTF {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_get:nncTF {⟨id⟩} {⟨tag⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}

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

The `...spec...` variant is similar except that it uses what was stored for ⟨*id*⟩`!`⟨*tag*⟩`/`⟨*key*⟩ or `!`⟨*tag*⟩`/`⟨*key*⟩ (with an empty ⟨*id*⟩).

`\__bnvs_if_get:nncTF` is a shortcut for `\__bnvs_if_get:nnncTF` when ⟨*key*⟩ and ⟨*ans*⟩ are the same.

```
\BNVS_new_conditional:cpnn { if_get:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_is_gset:nnnTF { #1 } { #2 } { #3 } {
    \tl_set_eq:cc { \BNVS_l:cn { #4 } { tl } } {
      \__bnvs_name:nnn { #1 } { #2 } { #3 }
    }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_get:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_get:vvc } #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:Nvv \__bnvs_if_get:nncTF { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
```

```
  }
}
```

__bnvs_if_resolved:nnncTF __bnvs_if_resolved:nnncTF {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨ans⟩} {⟨yes code⟩} {⟨no code⟩}

Execute ⟨yes code⟩ if resolution succeeded for either ⟨id⟩!⟨tag⟩/⟨key⟩ or !⟨tag⟩/⟨key⟩.

```
\BNVS_new_conditional:cpnn { if_resolved:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 } { #4 } {
    \__bnvs_if_unresolvable:cTF { #3 } {
      \__bnvs_if_resolved:nnnc { #1 } { #2 } { #3 } { #4 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    } {
      \__bnvs_is_will_gset:cTF { #3 } {
        \prg_return_false:
      } {
        \prg_return_true:
      }
    }
  } {
    \tl_if_empty:nTF { #1 } {
      \prg_return_false:
    } {
      \__bnvs_if_resolved:nnnc { #1 } { #2 } { #3 } { #4 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}
```

__bnvs_require:nnnc __bnvs_require:nnnc {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨ans⟩}
__bnvs_require:nnc __bnvs_require:nnc {⟨id⟩} {⟨tag⟩} {⟨ans⟩}
__bnvs_require:vvc

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

```
\BNVS_new:cpn { require:nnnc } #1 #2 #3 #4 {
  \__bnvs_if_get:nnncF { #1 } { #2 } { #3 } { #4 } {
    \BNVS_fatal_unreachable:
  }
}

\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 { #1 } {
      \prg_return_false:
    } {
```

```
      \__bnvs_if_get:nnncTF { } { #2 } { #3 } { #4 } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}
```

6.7.3 The provide mode

```
\bool_new:N \l__bnvs_provide_bool

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

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

__bnvs_is_provide_and_gset:nn*TF*
__bnvs_is_provide_and_gset:nnn*TF*
__bnvs_is_provide_and_gset:(nvn|vvn)*TF*

__bnvs_is_provide_and_gset:nnnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

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

```
\BNVS_new_conditional:cpnn { if_should_provide:nn } #1 #2 { T, F, TF } {
  \__bnvs_if:cTF { provide } {
    \if_cs_exist:w \__bnvs_name:nnn { #1 } { #2 } 0 \cs_end:
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  } {
    \prg_return_true:
  }
}

\BNVS_new_conditional:cpnn { is_provide_and_gset:nnn }
                           #1 #2 #3 { T, F, TF } {
  \__bnvs_if:cTF { provide } {
    \if_cs_exist:w \__bnvs_name:nnn { #1 } { #2 } { #3 } \cs_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  } {
    \prg_return_false:
  }
}
```

```
\BNVS_new_conditional:cpnn { is_provide_and_gset:nvn }
                           #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:nv {
    \__bnvs_is_provide_and_gset:nnnTF { #1 }
  } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { is_provide_and_gset:vvn }
                           #1 #2 #3 { T, F, TF } {
  \BNVS_tl_use:Nvv \__bnvs_is_provide_and_gset:nnnTF { #1 } { #2 } { #3 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_gprovide:TnnnnF
__bnvs_gprovide:TvnnnF
__bnvs_gprovide:TvvnnF

__bnvs_gprovide:TnnnnF {⟨*pre code*⟩} {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩} {⟨*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 ⟨*pre code*⟩ before setting.

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

\BNVS_new:cpn { gprovide:TvnnnF } #1 {
  \BNVS_tl_use:nv { \__bnvs_gprovide:TnnnnF { #1 } }
}
\BNVS_new:cpn { gprovide:TvvnnF } #1 {
  \BNVS_tl_use:nvv { \__bnvs_gprovide:TnnnnF { #1 } }
}
```

6.7.4 Initialize

__bnvs_ginit:nnn
__bnvs_ginit:nnv
__bnvs_ginit:nnnn

__bnvs_ginit:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definition*⟩}
__bnvs_ginit:nnnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*index*⟩} {⟨*definition*⟩}
__bnvs_ginit_V:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definition*⟩}

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

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

```
  \__bnvs_is_provide_and_gset:nnnF { #1 } { #2 } 0 {
    \__bnvs_gclear:nn { #1 } { #2 }
    \__bnvs_gset:nnnn { #1 } { #2 } 0 { #3 }
  }
}
\BNVS_new:cpn { ginit:nnv } #1 #2 {
  \BNVS_tl_use:nv { \__bnvs_ginit:nnn { #1 } { #2 } }
}
\BNVS_new:cpn { ginit:nnnn } #1 #2 #3 #4 {
  \__bnvs_is_gset:nnnT { #1 } { #2 } 0 {
    \__bnvs_gclear:nn { #1 } { #2 }
  }
  \__bnvs_is_provide_and_gset:nnnF { #1 } { #2.#3 } 0 {
    \__bnvs_gclear:nn { #1 } { #2 }
    \__bnvs_gset:nnnn { #1 } { #2.#3 } 0 { #4 }
  }
  \__bnvs_is_gset:nnnF { #1 } { #2 } 0 {
    \__bnvs_gset:nnnx { #1 } { #2 } 0 {
      \exp_not:n { #4 - } \int_eval:n { #3 - 1 }
    }
  }
}
```

`\__bnvs_will_gset:nnn` `\__bnvs_will_gset:nnn {⟨id⟩} {⟨tag⟩} {⟨key⟩}`

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

```
\BNVS_new:cpn { will_gset:nnn } #1 #2 #3 {
  \__bnvs_gset:nnnn { #1 } { #2 } { #3 } { \q_no_value }
}
```

`\__bnvs_is_will_gset:c`*TF*
`\__bnvs_if_will_get:nnnc`*TF*
`\__bnvs_is_will_gset:cTF {⟨in⟩} {⟨yes code⟩} {⟨no code⟩}`

To manage circular dependencies. See `\__bnvs_will_gset:nnn` above.

```
\__bnvs_will_gset:nnn {⟨id⟩} {⟨tag⟩} {⟨key⟩}
...
\__bnvs_if_get:nnncT {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨in⟩} {
  ...
  \__bnvs_is_will_gset:cTF {⟨in⟩} {
    {⟨yes code⟩} } { {⟨no code⟩} }
  ...
}
```

```
\BNVS_new_conditional:cpnn { is_will_gset:c } #1 { T, F, TF } {
  \BNVS_tl_use:Nv \quark_if_no_value:nTF { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.8 Implicit value counter

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

__bnvs_V_gunset:nn
__bnvs_V_gunset:n
__bnvs_V_gunset:

```
\__bnvs_V_gunset:nn {⟨id⟩} {⟨tag⟩}
\__bnvs_V_gunset:n  {⟨id⟩}
\__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 { ##1 } { ##2 } V
  }
}
```

6.8.1 Unresolvable

If resolution has failed for ⟨*id*⟩!⟨*tag*⟩/⟨*key*⟩, it is marked unresolvable in order to save some computation time.

__bnvs_gset_unresolvable:nnn

See below for the usage.

```
\BNVS_new:cpn { gset_unresolvable:nnn } #1 #2 #3 {
  \__bnvs_gset:nnnn { #1 } { #2 } { #3 } { \q_nil }
}
```

__bnvs_if_unresolvable:c*TF*

```
\__bnvs_gset_unresolvable:nnn {⟨id⟩} {⟨tag⟩} {⟨key⟩}
...
\__bnvs_if_get:nnncT {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨in⟩} {
  ...
  \__bnvs_if_unresolvable:cTF {⟨in⟩} {
    {⟨yes code⟩} } { {⟨no code⟩} }
  ...
}
```

```
\BNVS_new_conditional:cpnn { if_unresolvable:c } #1 { T, F, TF } {
  \BNVS_tl_use:nc { \exp_args:NV \quark_if_nil:nTF } { #1 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_is_provided:nnnTF __bnvs_is_provided:nnnTF {⟨id⟩} {⟨tag⟩} {⟨key⟩} {⟨yes code⟩} {⟨no code⟩}

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

```
\tl_new:N \l__bnvs_is_provided_tl
\BNVS_new_conditional:cpnn { is_provided:nnn } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_get:nnncTF { #1 } { #2 } { #3 }  { is_provided } {
    \__bnvs_if_unresolvable:cTF { is_provided } {
      \prg_return_false:
    } {
      \__bnvs_is_will_gset:cTF { is_provided } {
        \prg_return_false:
      } {
        \prg_return_true:
      }
    }
  } {
    \prg_return_false:
  }
}
```

6.9 Regular expressions

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

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

(*End of definition for* \c__bnvs_S_regex.)

\c__bnvs_P_regex A possibly empty list of .⟨short name⟩ items representing a path.

```
\regex_const:Nn \c__bnvs_P_regex {
  (?: \. \ur{c__bnvs_S_regex} )*
}
```

(*End of definition for* \c__bnvs_P_regex.)

\c__bnvs_A_GN_Z_regex

(*End of definition for* \c__bnvs_A_GN_Z_regex.)

```
\regex_const:Nn \c__bnvs_A_GN_Z_regex {
  \A (?: \. (?: [+] | ( [-] ) )? 0*? ( [1-9] \d* | 0 ) ) \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_XP_Z_regex A qualified dotted name is the qualified name of an overlay set possibly followed by a dotted path. Matches the whole string.

(*End of definition for* \c__bnvs_A_XP_Z_regex.)

```
\regex_const:Nn \c__bnvs_A_XP_Z_regex {
```

1: the frame ⟨*id*⟩

2: the exclamation mark

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

3: The short name.

```
  ( \ur{c__bnvs_S_regex} )
```

4: the path, if any.

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

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

(*End of definition for* \c__bnvs_A_XP_Z_regex.)

```
\regex_const:Nn \c__bnvs_A_XPGN_Z_regex {
```

1: the frame ⟨*id*⟩

2: the exclamation mark

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

3: The short name.

```
  ( \ur{c__bnvs_S_regex} )
```

4: the path, if any.

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

5: The otional - sign

6: and unsigned value of the last optional .⟨*index*⟩ component, no leading 0 except for 0 itself.

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

\c__bnvs_A_XP_Z_regex Matches the whole string.

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

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

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

1: The full match,

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

2,3: the optional frame ⟨*id*⟩ and `!`

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

4: The tag (short name and dotted path)

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

5: The sign

6: and unsigned value of the last optional .⟨*index*⟩ component, no leading `0` except for 0 itself.

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

3: The dotted path.

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

`\c__bnvs_A_PGN_Z_regex` Matches the whole string.

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

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

1: The full match,

2: the path

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

3: The sign

3: abd the value of the trailing digits component.

```
      (?: \. (?: [+] | ( [-] ) )? 0*? ( [1-9] \d* | 0 ) )? \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_S_regex} \ur{c__bnvs_P_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_XPXPGNURO_regex Used to parse slide list overlay specifications in queries. Next are the 12 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

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

We start with '++' instrussions[5].

1 incrementation prefix

```
    \+\+
```

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

1.2: followed by required `!`

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

1.3: ⟨*short name*⟩

```
    ( \ur{c__bnvs_S_regex} )
```

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

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

2: without incement prefix

2.1: optional frame ⟨*id*⟩ followed by

2.2: required `!`

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

2.3: ⟨*short name*⟩

```
      ( \ur{c__bnvs_S_regex} )
```

2.4: optionally followed by a dotted path

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

2.5: The sign

2.6: and unsigned value of the last optional .⟨*index*⟩ component, no leading 0 except for 0 itself.

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

We continue with other expressions

2.7: the '+' in '+=' versus standalone '='.

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

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

2.9: the post increment

```
    | (\+)\+
    )?
  ) \s*
}
```

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

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

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

```
\regex_const:Nn \c__bnvs_A_VAZLLZZL_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
}
```

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

6.10 End group setter

```
\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_tl_gset:nnnv #1 #2 #3 {
  \BNVS_tl_use:nv {
    \BNVS_end:
    \__bnvs_gset:nnnn { #1 } { #2 } { #3 }
  }
}

\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 }
  }
}
```

6.11 **beamer.cls** interface

Work in progress.

```
\RequirePackage{keyval}

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

\bool_new:N \l__bnvs_in_frame_bool
\bool_set_false:N \l__bnvs_in_frame_bool
\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 }
}
```

6.12 Utilities

6.12.1 Split utilities

__bnvs_if_regex_split:cnc*TF* {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_regex_split:cn*TF* __bnvs_if_regex_split:cncTF {⟨*regex core*⟩} {⟨*expression*⟩} ⟨*seq core*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_regex_split:cnTF {⟨*regex core*⟩} {⟨*expression*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

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

```
\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:
  }
}

\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:
  }
}

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

\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_pop_XPGNURO:T } #1 {
```

```
  \__bnvs_split_if_pop_left:cT a {
    \__bnvs_split_if_pop_left:cT a {
      \__bnvs_split_if_pop_left:cT a {
        \__bnvs_split_if_pop_left:cT a {
          \__bnvs_split_if_pop_left:cT G {
            \__bnvs_split_if_pop_left:cT N {
              \__bnvs_split_if_pop_left:cT U {
                \__bnvs_split_if_pop_left:cT R {
                  \__bnvs_split_if_pop_left:cT O {
                    #1
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

\BNVS_new:cpn { split_if_pop_GNURO:T } #1 {
  \__bnvs_split_if_pop_left:cT G {
    \__bnvs_split_if_pop_left:cT N {
      \__bnvs_split_if_pop_left:cT U {
        \__bnvs_split_if_pop_left:cT R {
          \__bnvs_split_if_pop_left:cT O {
            #1
          }
        }
      }
    }
  }
}
```

6.12.2 Match utilities

__bnvs_match_if_once:NnTF __bnvs_match_if_once:NnTF ⟨*regex variable*⟩ {⟨*expression*⟩}
__bnvs_match_if_once:NvTF {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_match_if_once:nnTF __bnvs_match_if_once:nnTF {⟨*regex*⟩} {⟨*expression*⟩}
{⟨*yes code*⟩} {⟨*no code*⟩}

These are shortcuts to

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

```
\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:
  }
}
```

`\l__bnvs_match_seq` Local storage for the match result.

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

```
\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 { match_if_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:
  }
}

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

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

`\__bnvs_if_ISP:n`*TF* `\__bnvs_if_ISP:nTF {`⟨*name*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_if_ISP:nnTF {`⟨*root*⟩`} {`⟨*relative*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

If ⟨*name*⟩ is a reference, put the frame id it defines, or the current frame id, into `I` the short name into `S`, the dotted path into `P`, then execute ⟨*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_ISP:n } #1 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_match_if_once:NnTF \c__bnvs_A_XP_Z_regex { #1 } {
    \__bnvs_match_if_pop_left:cT I {
      \__bnvs_match_if_pop_left:cT I {
        \__bnvs_match_if_pop_left:cT K {
          \__bnvs_match_if_pop_left:cT S {
            \__bnvs_match_if_pop_left:cT P {
              \cs_set:Npn \BNVS_if_ISP:nnn ##1 ##2 ##3 {
                \BNVS_end:
                \__bnvs_tl_set:cn I { ##1 }
                \__bnvs_tl_set:cn S { ##2 }
                \__bnvs_tl_set:cn P { ##3 }
              }
              \__bnvs_tl_if_empty:cTF K {
                \BNVS_tl_use:Nvvv \BNVS_if_ISP:nnn J
              } {
                \BNVS_tl_use:Nvvv \BNVS_if_ISP:nnn I
              } S P
              \__bnvs_tl_set:cv T P
              \__bnvs_tl_put_left:cv T S
              \__bnvs_tl_set:cv J I
```

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

6.12.3 Utilities

```
\BNVS_new:cpn { warn_until_s_stop:w } #1 \s_stop {
  \tl_if_empty:nF { #1 } {
    \BNVS_warning:n { Ignored:~#1 }
  }
}
\BNVS_new:cpn { scan_until_s_stop:w } {
  \peek_meaning:NTF \s_stop {
    \use_none:n
  } {
    \__bnvs_warn_until_s_stop:w
  }
}
```

`\__bnvs_peek_branch_until_s_stop:nnnnw` `\__bnvs_peek_branch_until_s_stop:nnnnw {⟨square:n⟩} {⟨curly:n⟩} {⟨non empty:n⟩} {⟨empty:⟩} ⟨...⟩ \s_stop`

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

```
\BNVS_new:cpn { peek_branch_until_s_stop:nnnnw } #1 #2 #3 #4 {
  \peek_meaning:NTF \BNVS_square:n {
    \cs_set:Npn \BNVS_peek_branch:w ##1 ##2 {
      \cs_set:Npn \BNVS_peek_branch:w ####1 {
        #1
      }
      \BNVS_peek_branch:w { ##2 }
      \__bnvs_warn_until_s_stop:w
    }
    \BNVS_peek_branch:w
  } {
    \peek_meaning:NTF \BNVS_curly:n {
      \cs_set:Npn \BNVS_peek_branch:w ##1 ##2 {
        \cs_set:Npn \BNVS_peek_branch:w ####1 {
          #2
        }
        \BNVS_peek_branch:w { ##2 }
        \__bnvs_warn_until_s_stop:w
      }
```

```
      \BNVS_peek_branch:w
    } {
      \peek_meaning:NTF \s_stop {
        #4
        \use_none:n
      } {
        \cs_set:Npn \BNVS_peek_branch:w ##1 \s_stop {
          #3
        }
        \BNVS_peek_branch:w
      }
    }
  }
}
```

__bnvs_keyval:ncc `\__bnvs_keyval:ncc {⟨definitions⟩} {⟨V⟩} {⟨a⟩}`

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

```
\BNVS_new:cpn { keyval:ncc } #1 #2 #3 {
  \cs_set:Npn \BNVS: {
    \cs_set:Npn \BNVS:n ####1 {
      \__bnvs_tl_put_right:cn { #3 } { \BNVS:n { ####1 } }
    }
    \cs_set:Npn \BNVS:nn ####1 ####2 {
      \__bnvs_tl_put_right:cn { #3 } { \BNVS:nn { ####1 } { ####2 } }
    }
  }
  \cs_set:Npn \BNVS:n ##1 {
    \BNVS:
    \__bnvs_tl_set:cn { #2 } { ##1 }
  }
  \cs_set:Npn \BNVS:nn {
    \BNVS:
    \BNVS:nn
  }
  \__bnvs_tl_set:cn { #2 } { #1 }
  \BNVS_tl_use:nc {
    \regex_replace_all:nnN { \cB. } { \c{BNVS:} \0 }
  } { #2 }
  \__bnvs_tl_clear:c { #3 }
  \BNVS_tl_use:nv {
    \__bnvs_tl_clear:c { #2 }
    \keyval_parse:NNn \BNVS:n \BNVS:nn
  } { #2 }
  \BNVS_tl_use:nc {
    \regex_replace_all:nnN { \c{BNVS:} } { }
  } { #2 }
  \BNVS_tl_use:nc {
    \regex_replace_all:nnN { \c{BNVS:} } { }
  } { #3 }
```

```
}

\BNVS_new:cpn { normalize_GN: } {
  \__bnvs_tl_if_eq:cnF N 0 {
    \__bnvs_tl_if_empty:cF G {
      \__bnvs_tl_put_left:cn N { - }
    }
  }
}
```

6.12.4 Next index

```
\BNVS_int_new:c { next_index }
\BNVS_new:cpn { next_index:nn } #1 #2 {
  \__bnvs_int_incr:c { next_index }
  \__bnvs_tl_set:cn a { #2 . }
  \BNVS_int_use:nv { \__bnvs_tl_put_right:cn a } { next_index }
  \__bnvs_is_gset:nvnT { #1 } a 0 {
    \__bnvs_next_index:nn { #1 } { #2 }
  }
}
```

`\__bnvs_next_index:nnn` `\__bnvs_next_index:nnn {⟨id⟩} {⟨tag⟩} {⟨code:n⟩}`

⟨*code:n*⟩ takes one argument: the next available tag.

```
\BNVS_new:cpn { next_index:nnn } #1 #2 #3 {
  \BNVS_begin:
  \__bnvs_int_zero:c { next_index }
  \__bnvs_next_index:nn { #1 } { #2 }
  \cs_set:Npn \BNVS:n ##1 {
    \BNVS_end:
    \__bnvs_int_set:cn { next_index } { ##1 }
    #3
  }
  \BNVS_tl_use:Nv \BNVS:n { next_index }
}
```

6.13 Parsing

Workflows:

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

6.13.1 Square brackets

Here is the documentation.

__bnvs_square_parse:nnn
__bnvs_square_parse:nnnn
__bnvs_square_keyval:nnn

__bnvs_square_keyval:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definitions*⟩}
__bnvs_square_parse:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definition*⟩}
__bnvs_square_parse:nnnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*index*⟩} {⟨*definition*⟩}

To parse what is inside square brackets.

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

Find the first available index.

```
    \__bnvs_next_index:nnn { #1 } { #2 } {
      \__bnvs_ginit:nnn { #1 } { #2.##1 } { \BNVS_value:n { #3 } }
    }
  }
}
```

Why no colon allowed?

```
\BNVS_new:cpn { square_parse:nnnn } #1 #2 #3 #4 {
  \__bnvs_match_if_once:NnTF \c__bnvs_colons_regex { #4 } {
    \BNVS_error:n { No~colon~allowed:~...#3=#4 }
  } {
    \__bnvs_match_if_once:NnTF \c__bnvs_A_GN_Z_regex { #3 } {
      \__bnvs_match_if_pop_left:cT G {
        \_bnvs_match_if_pop_left:cT G {
          \_bnvs_match_if_pop_left:cT N {
            \__bnvs_normalize_GN:
            \exp_args:Nnx \__bnvs_ginit:nnn { #1 } {
              #2.\l__bnvs_N_tl
            } { #4 }
          }
        }
      }
    } {
      \BNVS_error:n { Not~an~integer:~#3~(=#4) }
    }
  }
}
```

For `X=[...]`.

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

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

```
  \tl_if_empty:nTF { #2 } {
    \BNVS_error:n { Unexpected~list~at~top~level. }
  } {
    \tl_if_empty:nF { #3 } {
      \__bnvs_is_provide_and_gset:nnnF { #1 } { #2 } V {
        \BNVS_begin:
```

```
        \__bnvs_gunset_deep:nn { #1 } { #2 }
        \keyval_parse:nnn
          { \__bnvs_square_parse:nnn  { #1 } { #2 } }
          { \__bnvs_square_parse:nnnn { #1 } { #2 } }
          { #3 }
        \BNVS_end:
      }
    }
  }
}

\BNVS_new:cpn { square_keyval:nvn } #1 {
  \BNVS_tl_use:nv { \__bnvs_square_keyval:nnn { #1 } }
}

\BNVS_new:cpn { square_keyval:vvn } {
  \BNVS_tl_use:Nvv \__bnvs_square_keyval:nnn
}

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

\BNVS_new:cpn { square_keyval_I:vn } {
  \BNVS_tl_use:cv { square_keyval_I:nn }
}

\BNVS_new:cpn { square_keyval_IT:n } {
  \BNVS_tl_use:Nvv \__bnvs_square_keyval:nnn I T
}
```

6.13.2 Range or value lists

`\__bnvs_indexed_keyval:nnnn` `\__bnvs_indexed_keyval:nnnn {⟨id⟩} {⟨tag⟩} {⟨index⟩} {⟨definitions⟩}`

Parses the definitions as a standalone value. Other values are errors. It does not mean that standalone values are free from errors by themselves. Called by `\__bnvs_root_parsed:nn`.

```
\BNVS_new:cpn { indexed_keyval:nnnn } #1 #2 #3 #4 {
  \tl_if_empty:nTF { #4 } {
    \__bnvs_gclear:nn { #1 } { #2.#3 }
  } {
    \BNVS_begin:
    \__bnvs_keyval:ncc { #4 } V a
    \BNVS_tl_last_unbraced:nv {
      \__bnvs_peek_branch_until_s_stop:nnnnw {
        \BNVS_error:n { No~`[...]`~allowed~with~index. }
      } {
```

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

```
        \__bnvs_indexed_keyval:nnnn { #1 } { #2 } { #3 } { ##1 }
      } {
        \__bnvs_ginit:nnnn { #1 } { #2 } { #3 } { \BNVS:n { #4 } }
```

Support \BNVS_value:n

```
        \__bnvs_ginit:nnn { #1 } { #2.#3 } { #4 }
        \__bnvs_is_gset:nnnF { #1 } { #2 } 0 {
          \__bnvs_gset:nnnx { #1 } { #2 } 0 {
            \exp_not:n { \BNVS_value:n { #4 } - } \int_eval:n { #3 - 1 }
          }
        }
      } {
      }
    } V \s_stop
    \__bnvs_tl_if_empty:cF a {
      \cs_set:Npn \BNVS:n  ##1 { \exp_not:n { ##1, } }
      \cs_set:Npn \BNVS:nn ##1 ##2 { \exp_not:n { ##1 = ##2, } }
      \BNVS_error:x { Ignored:~\__bnvs_tl_use:c a }
    }
    \BNVS_end:
  }
}
```

6.13.3 List specifiers

__bnvs_list_keyval:nnn

__bnvs_list_keyval:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definitions*⟩}

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

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

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

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

Standalone key stored in `V`.

```
    \BNVS_tl_last_unbraced:nv {
      \__bnvs_peek_branch_until_s_stop:nnnnw {
        \BNVS_error:n { No~`[...]`~allowed. }
      } {
```

For ⟨*id*⟩`!`⟨*tag*⟩{{⟨...⟩}}.

```
        \__bnvs_list_keyval:nnn { #1 } { #2 } { ##1 }
      } {
        \__bnvs_gunset_deep:nn { #1 } { #2 }
        \__bnvs_gset:nnnv { #1 } { #2 } 0 V
      } {
      }
    } V \s_stop
  } {
```

A single value or range specification.

```
    \__bnvs_gunset_deep:nn { #1 } { #2 }
    \__bnvs_tl_clear:c b
    \cs_set:Npn \BNVS:nn ##1 ##2 {
      \__bnvs_list_keyval:nnn { #1 } { #2.##1 } { ##2 }
      \__bnvs_tl_put_right:cn b { \BNVS:n { ##1 } }
    }
    \cs_set:Npn \BNVS:n ##1 {
      \__bnvs_next_index:nnn { #1 } { #2 } {
        \BNVS:nn { ####1 } { ##1 }
      }
    }
    \__bnvs_tl_if_empty:cF V {
      \BNVS_tl_use:Nv \BNVS:n V
    }
    \__bnvs_tl_use:c a
    \__bnvs_gset:nnnv { #1 } { #2 } 0 b
  }
  \BNVS_end:
}
```

`\__bnvs_keyval:nnn` `\__bnvs_keyval:vvn`

`\__bnvs_keyval:nnn {⟨id⟩} {⟨tag⟩} {⟨definition⟩}`

Calls `\keyval_parse:nnn`. ⟨*definition*⟩ is the corresponding definition. Workflow:

- `\__bnvs_root_parsed:nn`
- `\__bnvs_keyval:nnn`

`\__bnvs_keyval:nnn` `\__bnvs_keyval:nnn {⟨id⟩} {⟨key⟩} {⟨definition⟩}`

```
\BNVS_new:cpn { keyval:nnn } #1 #2 #3 {
  \tl_if_empty:nTF { #3 } {
    \__bnvs_gclear:nn { #1 } { #2 }
  } {
    \__bnvs_if_should_provide:nnT { #1 } { #2 } {
      \BNVS_begin:
      \__bnvs_keyval:ncc { #3 } V a
      \__bnvs_tl_if_empty:cTF a {
```

Standalone key stored in `V`.

```
        \BNVS_tl_last_unbraced:nv {
          \__bnvs_peek_branch_until_s_stop:nnnnw {
            \tl_if_empty:nTF { ##1 }
              \__bnvs_keyval:nnn
              \__bnvs_square_keyval:nnn
                { #1 } { #2 } { ##1 }
          } {
```

For `ID!TAG={{FOO}}`.

```
            \tl_if_empty:nTF { ##1 }
              \__bnvs_keyval:nnn
              \__bnvs_curly_keyval:nnn
                { #1 } { #2 } { ##1 }
          } {
            \__bnvs_ginit:nnv { #1 } { #2 } V
          } {
          }
        } V \s_stop
      } {
```

A single value or range specification.

```
        \__bnvs_gunset_deep:nn { #1 } { #2 }
        \__bnvs_tl_clear:c b
        \cs_set:Npn \BNVS:nn ##1 ##2 {
          \__bnvs_keyval:nnn { #1 } { #2.##1 } { ##2 }
          \__bnvs_tl_put_right:cn b { \BNVS:n { ##1 } }
        }
```

Tiny subtlety with great impact:

```
        \__bnvs_tl_if_empty:cF V {
          \BNVS_tl_use:Nv \BNVS:nn V 1
        }
        \__bnvs_tl_use:c a
        \__bnvs_gset:nnnv { #1 } { #2 } 2 b
      }
      \BNVS_end:
    }
  }
}

\BNVS_new:cpn { keyval:vvn } {
  \BNVS_tl_use:Nvv \__bnvs_keyval:nnn
}
```

6.13.4 Items between braces

Workflow:

- __bnvs_keyval:nnn or __bnvs_curly_parse:nnnn
- __bnvs_curly_keyval:nnn
- __bnvs_curly_parse:nnn or __bnvs_curly_parse:nnnn
- __bnvs_curly_keyval:nnn, __bnvs_square_keyval:nnn or __bnvs_ginit:nnn.

__bnvs_curly_keyval:nnn	__bnvs_curly_keyval:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definitions*⟩}
__bnvs_curly_keyval:(vvn|nvn)	__bnvs_curly_parse:nnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*definition*⟩}
__bnvs_curly_parse:nnn	__bnvs_curly_parse:nnnn {⟨*id*⟩} {⟨*tag*⟩} {⟨*key*⟩} {⟨*definition*⟩}
__bnvs_curly_parse:nnnn	

For ⟨*ref*⟩={⟨*def_1*⟩,...,⟨*def_j*⟩}. Deep first traversal.

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

```
}

\BNVS_new:cpn { curly_parse:nnnn } #1 #2 #3 #4 {
  \BNVS_begin:
  \__bnvs_match_if_once:NnTF \c__bnvs_A_PGN_Z_regex { #3 } {
    \__bnvs_match_if_pop_left:cT P {
      \__bnvs_match_if_pop_left:cT P {
        \__bnvs_tl_put_left:cn P { #2. }
        \__bnvs_match_if_pop_left:cT G {
          \__bnvs_match_if_pop_left:cT N {
            \__bnvs_tl_if_empty:cTF { N } {
```

This is not a `A...`⟨*number*⟩ refererence.

```
              \__bnvs_peek_branch_until_s_stop:nnnnw {
                \__bnvs_square_keyval:nvn { #1 } P { ##1 }
              } {
                \__bnvs_curly_keyval:nvn { #1 } P { ##1 }
              } {
                \BNVS_tl_use:nv { \__bnvs_ginit:nnn { #1 } } P { ##1 }
              } {
```

No specification: `A=` nothing, unset everything.

```
                \__bnvs_gunset_deep:nv { #1 } P
              }
            } {
              \__bnvs_normalize_GN:
              \__bnvs_peek_branch_until_s_stop:nnnnw {
                \BNVS_error:n { Unexpected~[]:~##1 }
              } {
                \BNVS_error:n { Unexpected~{}:~##1 }
              } {
                \__bnvs_if_set:ncccTF A Z L { ##1 } {
                  \BNVS_error:n { Unexpected~colon:~##1 }
                } {
                  \__bnvs_is_gset:nvnTF { #1 } P 0 {
                    \__bnvs_tl_put_right:cn P .
                    \__bnvs_tl_put_right:cv P N
                    \__bnvs_is_provide_and_gset:nvnF { #1 } P 0 {
                      \__bnvs_ginit:nvn { #1 } P { ##1 }
                    }
                  } {
                    \__bnvs_tl_if_eq:cnTF N 1 {
                      \__bnvs_ginit:nvn { #1 } P { ##1 }
                    } {
                      \exp_args:Nnnx \__bnvs_ginit:nvn { #1 } P {
                        \exp_not:n { ##1 -} \int_eval:n { \l__bnvs_N - 1 }
                      }
                    }
                    \__bnvs_tl_put_right:cn P .
                    \__bnvs_tl_put_right:cv P N
                    \__bnvs_ginit:nvn { #1 } P { ##1 }
                  }
                }
              } {
                \__bnvs_tl_put_right:cn P .
```

```
                \__bnvs_tl_put_right:cv P N
                \__bnvs_gunset_deep:nv { #1 } P
                \__bnvs_gunset:nv { #1 } P
              }
            }
            #4 \s_stop
          }
        }
      }
    }
  } {
    \BNVS_error:n { Unsupported~#3 }
  }
  \BNVS_end:
}
```

Here is the documentation.

```
\BNVS_new:cpn { curly_keyval:nnn } #1 #2 #3 {
  \tl_if_empty:nTF { #3 } {
  } {
    \BNVS_begin:
%     \__bnvs_curly_keyval:nc { #3 } a
    \keyval_parse:nnn {
      \__bnvs_curly_parse:nnn  { #1 } { #2 }
    } {
      \__bnvs_curly_parse:nnnn { #1 } { #2 }
    } { #3 }
    \BNVS_end:
  }
}

\BNVS_new:cpn { curly_keyval:nvn } #1 {
  \BNVS_tl_use:nv { \__bnvs_curly_keyval:nnn { #1 } }
}

\BNVS_new:cpn { curly_keyval:vvn } {
  \BNVS_tl_use:Nvv \__bnvs_curly_keyval:nnn
}
```

6.13.5 High level and root parser

These are the first functions called. Workflow:

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

__bnvs_root_parsed:nn

__bnvs_root_parsed:nn {⟨*name*⟩} {⟨*definition*⟩}

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

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

```
  \__bnvs_match_if_once:NnTF \c__bnvs_A_IKTGN_Z_regex { #1 } {
    \__bnvs_match_if_pop_left:cT a {
      \__bnvs_match_if_pop_left:cT I {
        \__bnvs_match_if_pop_left:cT K {
          \__bnvs_tl_if_empty:cTF K {
            \__bnvs_tl_set_eq:cc I J
          } {
            \__bnvs_tl_set_eq:cc J I
          }
          \__bnvs_match_if_pop_left:cT T {
            \__bnvs_match_if_pop_left:cT G {
              \__bnvs_match_if_pop_left:cT N {
                \__bnvs_tl_if_empty:cTF N {
                  \__bnvs_keyval:vvn IT { #2 }
                } {
                  \__bnvs_normalize_GN:
                  \BNVS_tl_use:Nvvv \__bnvs_indexed_keyval:nnnn ITN { #2 }
                }
              }
            }
          }
        }
      }
    }
  } {
    \BNVS_error:n { Unexpected~ref:~#1 }
  }
}
```

`\__bnvs_root_keyval:nc` `\__bnvs_root_keyval:nc {⟨definitions⟩} {⟨aux⟩}`

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

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

Does not remove .

```
\BNVS_new:cpn { root_keyval:nc } #1 #2 {
  \__bnvs_tl_set:cx { #2 } { #1 }
  \BNVS_tl_use:nc {
    \regex_replace_all:nnN { \cB. } { \c { BNVS_curly:n } \0 }
  } { #2 }
  \cs_set:Npn \BNVS:N ##1 {
    \cs_set:Npn \BNVS: {
      \regex_replace_all:nnNT { \[ (.*?) \] } {
```

```
        \c { BNVS_square:n } \cB\{ \1 \cE \}
      } ##1 {
        \BNVS:
      }
    }
    \BNVS:
    \regex_match:nNT { \[|\] } ##1 {
      \BNVS_error:n { Unbalanced~[|] }
    }
  }
  \BNVS_tl_use:Nc \BNVS:N { #2 }
  \cs_set:Npn \BNVS:n ##1 {
    \__bnvs_root_parsed:nn { ##1 } 1
  }
  \BNVS_tl_use:nv {
    \keyval_parse:nnn {
      \exp_after:wN \__bnvs_root_parsed:nn \use_ii_i:nn 1
    } {
      \__bnvs_root_parsed:nn
    }
  } { #2 }
}
```

`\__bnvs_root_keyval:NNn` `\__bnvs_root_keyval:NNn` ⟨*is at top*⟩ ⟨*on provide mode*⟩ {⟨*definitions*⟩}

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

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

At the document level, clear everything.

```
    \__bnvs_gclear:
  }
  \BNVS_begin:
  \bool_if:NTF #2 {
    \__bnvs_provide_on:
  } {
    \__bnvs_provide_off:
  }
  \__bnvs_int_zero:c { i }
  \__bnvs_tl_set:cn a { #3 }
  \bool_if:NT #1 {
```

At the document 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_tl_use:Nv \__bnvs_root_keyval:nc a a
```

This is a list, possibly at the top

```
  \BNVS_end_tl_set:cv J J
}
```

\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`.

```
\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:NnTF \@currenvir { document } {
      \IfBooleanTF {#1} {
        \__bnvs_root_keyval:NNn \c_true_bool \c_true_bool
      } {
        \__bnvs_root_keyval:NNn \c_true_bool \c_false_bool
      }
    } {
      \IfBooleanTF {#1} {
        \__bnvs_root_keyval:NNn \c_false_bool \c_true_bool
      } {
        \__bnvs_root_keyval:NNn \c_false_bool \c_false_bool
      }
    } { #2 }
    \ignorespaces
  }
}
```

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

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

6.14 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, ⟨...⟩ 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_resolve_queries:nc { #1 } { 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_resolve_queries:nc { #1 } { 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
```

6.14.1 Top level

__bnvs_resolve_queries:nc
__bnvs_prepare_queries:c

__bnvs_resolve_queries:nc {⟨*queries*⟩} {⟨*ans*⟩}
__bnvs_prepare_queries:c {⟨*ans*⟩}

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

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

```
  \__bnvs_tl_set:cn { #2 } { #1 }
  \__bnvs_prepare_queries:c { #1 }
  \__bnvs_if_resolve_queries:cF {
    \BNVS_error:n { Failure~#1 }
  }
}

\BNVS_new:cpn { prepare_queries:c } #1 {
  \BNVS_begin:
  \__bnvs_tl_set:cn L { #1 }
  \cs_set:Npn \BNVS: {
    \regex_replace_once:nnNT { \( (?:(\?)|(!)/)? ([%(
            ^)]*)\)
    } {
      \c{BNVS_round\1\2:n } \cB\{ \3 \cE\}
    } \l__bnvs_L_tl \BNVS:
  }
  \BNVS:
  \regex_replace_all:nnN
    { \c{ BNVS_round(?:\?|!):n } } { \c{BNVS_query:n} } \l__bnvs_L_tl

  \cs_set:cpn \BNVS_query:n { \exp_not:N \BNVS_query:n  }
  \cs_set:Npn \BNVS_round:n ##1 { ( ##1 ) }
  \__bnvs_tl_set:cx L { \l__bnvs_L_tl }
  \BNVS_end_tl_set:cv { #1 } L
}
```

`\__bnvs_if_resolve_flat:nc`*TF* `\__bnvs_if_resolve_flat:ncTF {`⟨*query*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Called by `\__bnvs_if_resolve_queries:ncTF`

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

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

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

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

```
      \BNVS_set:cpn { if_resolve_flat_end_return_true: } {
```

Normal and unique end of the loop.

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

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

```
      \BNVS_set:cpn { if_resolve_flat_end_return_false: } {
```

Normal and unique end of the loop.

```
        \BNVS_error:n { Unsupported~query: #1}
        \BNVS_end:
        \prg_return_false:
      }
```

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

```
      \BNVS_set:cpn { resolution_round_ans: } { \__bnvs_round:c { ans } }
      \__bnvs_tl_clear_ans:
      \__bnvs_resolve_init_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_if_resolve_query:nc`*TF* `\__bnvs_if_resolve_query:ncTF {`⟨*query*⟩`} {`⟨*tl core*⟩`} {`⟨*yes code*⟩`} {`⟨*no*
`\__bnvs_if_resolve_query:vc`*TF* *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_VAZLLZZL_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_new_conditional_vc:cn { if_resolve_query } { T, F, TF }
```

`\__bnvs_if_resolve_queries:c`*TF* `\__bnvs_if_resolve_queries:c {`⟨*queries*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Resolve the queries in place and branch to ⟨*yes code*⟩ otherwise leave the variable untouched and branch to ⟨*no code*⟩. The queries were prepared beforehands.

```
\tl_new:N \l__bnvs_if_resolve_queries_tl
\BNVS_new:cpn { if_resolve_queries:c } #1 {
  \BNVS_begin:
  \__bnvs_tl_clear:c { if_resolve_queries }
  \BNVS_tl_use:Nv \clist_map_inline:nn { #1 } {
    \regex_match:nnTF { \c{ BNVS_query:n } } { ##1 } {
      \cs_set:Npn \BNVS:w ####1 \BNVS_query:n ####2 ####3 \s_stop {
        \__bnvs_tl_put_right:cn { if_resolve_queries } { ##1 }
        \clist_map_inline:nn { ####2 } {
          \__bnvs_if_append:ncF { ########1 } { if_resolve_queries } {
            \BNVS_error:x { Bad~query:~\tl_to_str:n { ########1 } }
          }
        }
        \regex_if_match:nnNTF { \c{ BNVS_query:n } } { ####3 } {
          \BNVS:w ####3 \s_stop
        } {
          \__bnvs_tl_put_right:cn { if_resolve_queries } { ####3 }
        }
      }
      \regex_if_match:nnTF { \c{ BNVS_query:n } } { ####3 } {
        \BNVS:w ####3 \s_stop
      } {
        \__bnvs_tl_put_right:cn { if_resolve_queries } { ####3 }
      }
      \cs_set:Npn \BNVS:w ####1 \BNVS_query:n ####2 ####3 \s_stop {
        \__bnvs_tl_put_right:cn A { ####1 }
        \clist_map_inline:nn { ####2 } {
          \__bnvs_if_append:ncF { ########1 } A {
            \BNVS_error:x { Bad~query:~\tl_to_str:n { ########1 } }
          }
        }
        \regex_if_match:nnNTF { \c{ BNVS_query:n } } { ####3 } {
          \BNVS:w ####3 \s_stop
        } {
          \__bnvs_tl_put_right:cn A { ####3 }
        }
      }
      \BNVS:w ##1 \s_stop
    } {
      \__bnvs_tl_put_right:cn { if_resolve_queries } { ##1 }
    }
  }
  \BNVS_end_tl_set:cv { #1 } { if_resolve_queries }
}
```

A group is created to use local variables:

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

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

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

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

6.15 Resolution

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

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

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

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

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

```
    \__bnvs_match_if_pop_ISP:cccTF { #1 } { #2 } { #3 } {
      \__bnvs_export_ISP:cccN { #1 } { #2 } { #3 }
        \BNVS_end:
      \prg_return_true:
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
    \prg_return_false:
  }
}
\quark_new:N \q__bnvs_X

\tl_new:N \l__bnvs_export_ISP_cccN_tl
\BNVS_new:cpn { export_ISP:cccN } #1 #2 #3 #4 {
  \cs_set:Npn \BNVS_export_ISP_cccN:w ##1 ##2 ##3 {
    #4
    \__bnvs_tl_set:cn { #1 } { ##1 }
    \__bnvs_tl_set:cn { #2 } { ##2 }
    \__bnvs_tl_set:cn { export_ISP_cccN } { ##3 }
  }
  \__bnvs_tl_set:cx { export_ISP_cccN }
    { \__bnvs_seq_use:cn { #1 } { \q__bnvs_X } }
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:Nv \BNVS_export_ISP_cccN:w { #1 }
  } { #2 } { export_ISP_cccN }
  \BNVS_tl_use:nv {
    \__bnvs_seq_set_split:cnn { #3 } { \q__bnvs_X }
  } { export_ISP_cccN }
  \__bnvs_seq_remove_all:cn { #3 } { }
}
```

```
\tl_new:N \l__bnvs_match_if_export_XP_cccc_tl
\BNVS_new:cpn { match_if_export_XP:ccccN } #1 #2 #3 #4 #5 {
  \cs_set:Npn \BNVS_match_if_export_XP_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 { match_if_export_XP_cccc }
    { \__bnvs_seq_use:cn { #1 } { \q__bnvs_X } }
  \BNVS_tl_use:nvvv {
    \BNVS_tl_use:Nv \BNVS_match_if_export_XP_ccccN:w { #1 }
  } { #2 } { match_if_export_XP_cccc } { #4 }
  \BNVS_tl_use:nv {
    \__bnvs_seq_set_split:cnn { #3 } { \q__bnvs_X }
  } { match_if_export_XP_cccc }
  \__bnvs_seq_remove_all:cn { #3 } { }
}
\BNVS_new_conditional:cpnn { match_if_pop_XP:cccc } #1 #2 #3 #4 { TF } {
  \BNVS_begin:
  \__bnvs_match_if_pop_left:cTF { #1 } {
    \__bnvs_match_if_pop_left:cTF { #1 } {
      \__bnvs_match_if_pop_left:cTF { #2 } {
        \__bnvs_match_if_pop_left:cTF { #3 } {
          \__bnvs_seq_set_split:cnv { #3 } { . } { #3 }
          \__bnvs_seq_remove_all:cn { #3 } { }
          \__bnvs_match_if_pop_left:cTF { #4 } {
            \__bnvs_match_if_export_XP: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_X } {
      \__bnvs_seq_use:cn { #1 } { \q__bnvs_X }
      \exp_not:n { \q__bnvs_X }
      \__bnvs_seq_use:cn { #2 } { \q__bnvs_X }
    }
    \__bnvs_seq_remove_all:cn { #1 } { }
  }
}
```

6.16 Evaluation bricks

6.16.1 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
}
```

6.16.2 Resolve from initial values

Lower level resolution functions.

`\__bnvs_resolve_V_or_R:nnncc` `\__bnvs_resolve_V_or_R:nnn {`⟨*id*⟩`} {`⟨*tag*⟩`} {`⟨*spec*⟩`} {`⟨*V*⟩`} {`⟨*R*⟩`}`

Auxiliary function used by `\__bnvs_resolve:nn`. It does not change the ⟨*id*⟩`!`⟨*tag*⟩ data model . It resolves ⟨*spec*⟩ into either `{`⟨*V*⟩`}` or `{`⟨*R*⟩`}` in the ⟨*id*⟩`!`⟨*tag*⟩ context.

```
\regex_const:Nn \c__bnvs_A_R_Z_regex { \A ([^-]*)[-]([^-]*)\Z}
\tl_new:N \l__bnvs_vr_tl
\tl_new:N \l__bnvs_VR_tl
\BNVS_new:cpn { resolve_V_or_R:nnncc } #1 #2 #3 #4 #5 {
  \__bnvs_tl_clear:c { VR }
  \cs_set:Npn \BNVS:n ##1 {
    \__bnvs_tl_clear:c { vr }
    \__bnvs_if_append:nnncF { #1 } { #2 } { ##1 } { vr } {
```

```
      \__bnvs_if_resolve:ncTF { ##1 } { vr } {
        \BNVS_tl_use:Nv \clist_map_inline:nn { vr } {
          \tl_if_empty:nF { ####1 } {
            \__bnvs_tl_if_empty:cF { VR } {
              \__bnvs_tl_put_right:cn { VR } { , }
            }
            \__bnvs_match_if_once:NnTF \c__bnvs_A_R_Z_regex { ####1 } {
              \__bnvs_match_if_pop_left:cT a {
                \_bnvs_tl_if_empty:cT a {
                  \__bnvs_tl_set:cn a 1
                }
                \__bnvs_match_if_pop_left:cT z { }
              }
              \__bnvs_gset_azl:nn { #1 } { #2 }
              \__bnvs_tl_put_right:cx { VR } {
                \l__bnvs_a_tl - \l__bnvs_z_tl
              }
            } {
              \__bnvs_tl_put_right:cn { VR } { ####1 }
            }
          }
        }
      } {
        \__bnvs_tl_if_empty:cF { VR } {
          \__bnvs_tl_put_right:cn { VR } { , }
        }
        \__bnvs_tl_put_right:cn { VR } { 1 }
      }
    }
  }
  \tl_if_head_eq_meaning:nNTF { #3 } \BNVS:n {
    #3
  } {
    \BNVS:n { #3 }
  }
  \__bnvs_tl_if_empty:cTF { VR } {
    \tl_if_empty:nF { #4 } { \__bnvs_tl_clear:c { #4 } }
    \tl_if_empty:nF { #5 } { \__bnvs_tl_clear:c { #5 } }
  } {
    \__bnvs_gunset:nn { #1 } { #2 }
    \BNVS_tl_use:nv { \regex_match:nnTF { [-,] } } { VR } {
      \__bnvs_gset:nnnv  { #1 } { #2 } R { VR }
      \tl_if_empty:nF { #4 } { \__bnvs_tl_clear:c { #4 }        }
      \tl_if_empty:nF { #5 } { \__bnvs_tl_set:cv  { #5 } { VR } }
    } {
      \__bnvs_gset:nnnv  { #1 } { #2 } V { VR }
      \tl_if_empty:nF { #4 } { \__bnvs_tl_set:cv  { #4 } { VR } }
      \tl_if_empty:nF { #5 } { \__bnvs_tl_clear:c { #5 }        }
    }
  }
}
```

__bnvs_if_resolve_init:nn*TF* __bnvs_if_resolve_init:nnTF {⟨*id*⟩} {⟨*tag*⟩} {⟨*yes code*⟩} {⟨*no code*⟩}

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

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

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

```
\BNVS_new_conditional:cpnn { if_resolve_init:nn } #1 #2 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_get:nnncTF { #1 } { #2 } 0 { VR } {
    \__bnvs_tl_clear:c { VR }
    \cs_set:Npn \BNVS:n ##1 {
      \__bnvs_tl_clear:c { vr }
      \__bnvs_if_append:nnncF { #1 } { #2 } { ##1 } { vr } {
        \__bnvs_if_resolve:ncTF { ##1 } { vr } {
          \BNVS_tl_use:Nv \clist_map_inline:nn { vr } {
            \tl_if_empty:nF { ####1 } {
              \__bnvs_tl_if_empty:cF { VR } {
                \__bnvs_tl_put_right:cn { VR } { , }
              }
              \__bnvs_match_if_once:NnTF \c__bnvs_A_R_Z_regex { ####1 } {
                \__bnvs_match_if_pop_left:cT a {
                  \_bnvs_tl_if_empty:cT a {
                    \__bnvs_tl_set:cn a 1
                  }
                  \__bnvs_match_if_pop_left:cT z { }
                }
                \__bnvs_gset_azl:nn { #1 } { #2 }
                \__bnvs_tl_put_right:cx { VR } {
                  \l__bnvs_a_tl - \l__bnvs_z_tl
                }
              } {
                \__bnvs_tl_put_right:cn { VR } { ####1 }
              }
            }
          }
        } {
          \__bnvs_tl_if_empty:cF { VR } {
            \__bnvs_tl_put_right:cn { VR } { , }
          }
          \__bnvs_tl_put_right:cn { VR } { 1 }
        }
      }
    }
    \__bnvs_will_gset:nnn { #1 } { #2 } V
    \__bnvs_will_gset:nnn { #1 } { #2 } R
    \cs_set:Npn \BNVS_if_resolve_init:n ##1 {
      \__bnvs_tl_clear:c { VR }
      \tl_if_head_eq_meaning:nNTF { ##1 } \BNVS:n {
        ##1
      } {
```

```
        \BNVS:n { ##1 }
      }
    }
    \BNVS_tl_use:Nv \BNVS_if_resolve_init:n { VR }
    \BNVS_tl_use:nv { \regex_match:nnTF { [-,] } } { VR } {
      \__bnvs_gunset:nn { #1 } { #2 }
      \__bnvs_gset_unsolvable:nnn { #1 } { #2 } V
      \__bnvs_gset:nnnv { #1 } { #2 } R { VR }
      \BNVS_end:
      \prg_return_true:
    } {
      \__bnvs_tl_if_empty:cTF { VR } {
        \__bnvs_gunset:nnn { #1 } { #2 } V
        \__bnvs_gunset:nnn { #1 } { #2 } R
        \BNVS_end:
        \prg_return_false:
      } {
        \__bnvs_gunset:nn { #1 } { #2 }
        \__bnvs_gset:nnnv { #1 } { #2 } V { VR }
        \__bnvs_gset_unsolvable:nnn { #1 } { #2 } R
        \BNVS_end:
        \prg_return_true:
      }
    }
  } {
    \BNVS_end:
    \prg_return_false:
  }
}
```

6.16.3 V for value

__bnvs_if_resolve_V:nnc*TF*
__bnvs_if_resolve_V:nvc*TF*
__bnvs_if_append_V:nnc*TF*
__bnvs_if_append_V:(nxc|nvc)*TF*

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

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.

If a range is associated to ⟨*id*⟩`!`⟨*tag*⟩, ⟨*no code*⟩ is executed.

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

We already tried to compute but failed.

```
      \prg_return_false:
    } {
      \__bnvs_is_will_gset:cTF { #3 } {
        \BNVS_error:n { Circular~definition:~#1!#2/V (Error~recovery~1) }
        \__bnvs_gunset:nn { #1 } { #2 }
        \__bnvs_gset:nnnn { #1 } { #2 } 1 1
        \__bnvs_gset:nnnn { #1 } { #2 } V 1
        \__bnvs_tl_set:cn { #3 } 1
        \prg_return_true:
```

Circular definition call during resolution.

```
      } {
        \prg_return_true:
      }
    }
  } {
    \__bnvs_is_provided:nnnTF { #1 } { #2 } R {
      \__bnvs_set_unresolvable:nnn { #1 } { #2 } V
      \prg_return_false:
    } {
      \__bnvs_if_resolve_init:nnTF { #1 } { #2 } {
        \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #3 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \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 }
```

6.16.4 R for range

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

Auxiliary function only called by __bnvs_if_resolve_R:nncTF.

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

We already tried to compute but failed.

```
      \prg_return_false:
    } {
      \__bnvs_is_will_gset:cTF { if_resolve_R_nnc } {
        \BNVS_error:n { Circular~definition:~#1!#2/r (Error~recovery~1) }
        \__bnvs_gset:nnnn { #1 } { #2 } R 1,
        \__bnvs_gset:nnnn { #1 } { #2 } A 1
        \__bnvs_gset:nnnn { #1 } { #2 } Z 1
        \__bnvs_gset:nnnn { #1 } { #2 } L 1
        \__bnvs_tl_set:cn { #3 } { 1, }
        \prg_return_true:
```

Circular definition call during resolution.

```
      } {
        \__bnvs_tl_set:cv { #3 } { if_resolve_R_nnc }
        \prg_return_true:
      }
    }
  } {
    \__bnvs_is_provided:nnnTF { #1 } { #2 } V {
      \__bnvs_set_unresolvable:nnn { #1 } { #2 } R
      \prg_return_false:
    } {
      \__bnvs_if_resolve_init:nnTF { #1 } { #2 } {
        \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
          \prg_return_true:
        } {
          \prg_return_false:
        }
      } {
        \prg_return_false:
      }
    }
  }
}

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

6.16.5 Already complete

__bnvs_if_resolve:nnc*TF* __bnvs_if_resolve:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append:nnc*TF* __bnvs_if_append:nncTF {⟨*id*⟩} {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}

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

Next fonction is only called by `\__bnvs_if_resolve:nncTF` when

```
\tl_new:N \l__bnvs_if_resolve_nnc_tl
\BNVS_new_conditional:cpnn { if_resolve:nnc } #1 #2 #3 { T, F, TF } {
  \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { if_resolve_nnc } {
    \__bnvs_tl_set:cv { #3 } { if_resolve_nnc }
    \prg_return_true:
  } {
    \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { if_resolve_nnc } {
      \BNVS_set:cpn { resolution_round_ans: } { }
      \__bnvs_tl_set:cv { #3 } { if_resolve_nnc }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\BNVS_new_conditional:cpnn { if_append:nnc } #1 #2 #3 { T, F, TF } {
  \BNVS_group_begin:
  \__bnvs_if_resolve:nncTF { #1 } { #2 } { #3 } {
    \BNVS_end_tl_put_right:cv { #3 } { #3 }
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.16.6 Assignment

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

```
\BNVS_new_conditional:cpnn { if_assign_value:nnn } #1 #2 #3 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_set_true:c { no_range }
  \__bnvs_if_resolve:ncTF { #3 } a {
    \__bnvs_gclear:nn { #1 } { #2 }
    \__bnvs_gset:nnnv { #1 } { #2 } V 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: }
}
```

6.16.7 beamer counters

__bnvs_if_resolve_counter:nc*TF* __bnvs_if_resolve_counter:ncTF {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_resolve_counter:vc*TF* __bnvs_if_append_special:ncTF {⟨*tag*⟩} ⟨*ans*⟩ {⟨*yes code*⟩} {⟨*no code*⟩}
__bnvs_if_append_special:nc*TF*
__bnvs_if_append_special:vc*TF*

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

```
\makeatletter
\BNVS_new_conditional:cpnn { if_resolve_counter:nc } #1 #2 { T, F, TF } {
  \tl_if_eq:nnTF { #1 } { pauses } {
    \cs_if_exist:NTF \c@beamerpauses {
      \exp_args:Nnx \__bnvs_tl_set:cn { #2 } { \the\c@beamerpauses }
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \tl_if_eq:nnTF { #1 } { slideinframe } {
      \cs_if_exist:NTF \beamer@slideinframe {
        \exp_args:Nnx \__bnvs_tl_set:cn { #2 } { \beamer@slideinframe }
        \prg_return_true:
      } {
        \prg_return_false:
      }
    } {
      \prg_return_false:
    }
  }
}
\makeatother
```

__bnvs_if_set:ncccTF __bnvs_if_set:ncccTF {⟨A⟩} {⟨Z⟩} {⟨L⟩} {⟨query⟩} {⟨yes code⟩} {⟨no code⟩}

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

```
\BNVS_new_conditional:cpnn { if_set:nccc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_regex_split:cnTF { colons } { #1 } {
    \BNVS_begin:
    \tl_map_inline:nn { AZL } {
      \__bnvs_tl_clear:c ##1
    }
    \__bnvs_split_if_pop_left:cT A {
```

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

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

This is a single colon ⟨A⟩:[^:]*.

```
          \__bnvs_split_if_pop_left:cT Z {
            \__bnvs_seq_pop_left:ccT { split } L {
```

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

```
              \__bnvs_tl_if_empty:cTF L {
                \BNVS_error:n { Invalid~range~expression(1)/#1 }
```

A :: was expected:

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

A :: was expected:

```
                }
                \__bnvs_split_if_pop_left:cT L {
```

L may contain the ⟨length⟩.

```
                  \__bnvs_seq_if_empty:cF { split } {
                    \BNVS_error:n { Invalid~range~expression(2)/#1 }
```

No more material expected.

```
                  }
                }
              }
            }
          }
        } {
```

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

```
          \int_compare:nNnT { \__bnvs_tl_count:c Z } > { 1 } {
            \BNVS_warning:n { Too~many~colons(2)/#1 }
          }
          \__bnvs_split_if_pop_left:cT 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(3)/#1 }
```

```
              }
              \__bnvs_split_if_pop_left:cT Z {
```

Z may contain the ⟨*last*⟩.

```
                \__bnvs_seq_if_empty:cF { split } {
                  \BNVS_error:n { Invalid~range~expression(3)/#1 }
                }
              }
            } {
              \__bnvs_tl_clear:c Z
            }
          }
        }
      }
    }
```

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_set:cn A 1
      } {
         \__bnvs_tl_if_empty:cT L {
          \__bnvs_tl_set:cn A 1
        }
      }
    }
    \cs_set:Npn \BNVS_if_set_ncccTF:w ##1 ##2 ##3 {
      \BNVS_end:
      \__bnvs_tl_set:cn { #2 } { ##1 }
      \__bnvs_tl_set:cn { #3 } { ##2 }
      \__bnvs_tl_set:cn { #4 } { ##3 }
    }
    \BNVS_tl_use:Nvvv \BNVS_if_set_ncccTF:w A Z L
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

__bnvs_if_resolve:nnnn*TF* `\__bnvs_if_resolve:nnnnTF {⟨query⟩} {⟨A⟩} {⟨Z⟩} {⟨L⟩} {⟨yes code⟩} {⟨no code⟩}`

Calls `\__bnvs_if_set:ncccTF` and resolves each variable.

```
\BNVS_new:cpn { will_gset_azl: } {
  \__bnvs_tl_if_empty:cT a {
    \__bnvs_tl_if_empty:cTF z {
      \__bnvs_tl_set:cn a 1
     } {
      \__bnvs_tl_if_empty:cTF l {
        \__bnvs_tl_set:cn a 1
      } {
        \__bnvs_tl_set:cn a { max ( 0,
          \l__bnvs_z_tl - \l__bnvs_l_tl
        ) + 1 }
```

```
        \__bnvs_round:c a
      }
    }
  }
  \__bnvs_tl_if_empty:cT z {
    \__bnvs_tl_if_empty:cF l {
      \__bnvs_tl_set:cn z {
        \l__bnvs_a_tl + \l__bnvs_l_tl - 1
      }
      \__bnvs_round:c z
    }
  }
```

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

```
}

\BNVS_new:cpn { gset_azl:nn } #1 #2 {
  \__bnvs_will_gset_azl:
  \tl_map_inline:nn { AZL } {
    \__bnvs_tl_clear:c ##1
  }
  \__bnvs_tl_if_empty:cTF z {
    \__bnvs_if_get:nncTF { #1 } { #2 } A {
      \int_compare:nNnTF \l__bnvs_a_tl < \l__bnvs_A_tl {
        \__bnvs_tl_set:cv A a
      } {
      }
    } {
      \__bnvs_tl_set:cv A a
    }
  } {
    \int_compare:nNnF \l__bnvs_z_tl < \l__bnvs_a_tl {
```

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

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

`A` will diminish.

```
          \__bnvs_tl_set:cv A a
        }
        \__bnvs_require:nnc { #1 } { #2 } Z
        \__bnvs_tl_if_empty:cTF Z {
        } {
          \int_compare:nNnTF \l__bnvs_z_tl > \l__bnvs_Z_tl {
            \__bnvs_tl_set:cv Z z
          } {
          }
        }
      } {
```

```
        \__bnvs_tl_set:cv A a
        \__bnvs_tl_set:cv Z z
      }
    }
  }
  \tl_if_empty:cF A {
    \tl_map_inline:nn { AZ } {
      \__bnvs_gset:nnv { #1 } { #2 } ##1
    }
    \__bnvs_tl_if_empty:cTF Z {
      \__bnvs_gset:nnnn { #1 } { #2 } L {}
    } {
      \__bnvs_tl_set:cn L {
        \l__bnvs_Z_tl - \l__bnvs_A_tl + 1
      }
      \__bnvs_round:c L
      \__bnvs_gset:nnv { #1 } { #2 } L
    }
  }
}

\BNVS_new_conditional:cpnn { if_append:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_set:ncccTF { #3 } azl {
    \tl_map_inline:nn { azl } {
      \__bnvs_tl_if_empty:cF ##1 {
        \__bnvs_if_resolve:vcTF ##1 ##1 {
          \BNVS_tl_use:Nv \regex_match:nnT { [-,] } ##1 {
            \BNVS_warning:n { Unexpected~range~. }
            \__bnvs_tl_set:cn ##1 0
          }
        } {
          \BNVS_warning:n { Something~got~wrong. }
          \__bnvs_tl_set:cn ##1 0
        }
      }
    }
    \__bnvs_gset_azl:nn { #1 } { #2 }
    \__bnvs_tl_set:cv r a
    \__bnvs_tl_if_empty:cTF z {
      \__bnvs_tl_put_right:cn r { - }
    } {
      \__bnvs_tl_if_eq:cvF a z {
        \__bnvs_tl_put_right:cn r { - }
        \__bnvs_tl_put_right:cv r z
      }
    }

    \_bnvs_tl_if_empty:cF { #4 } {
      \__bnvs_tl_put_right:cn { #4 } { , }
    }
    \BNVS_end_tl_put_right:cv { #4 } r
    \prg_return_true:
  } {
```

```
    \BNVS_end:
    \prg_return_false:
  }
}
```

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

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:cpn { if_resolve_return:cnnc } #1 #2 #3 #4 {
  \tl_if_empty:nTF { #2 } {
    \prg_return_false:
  } {
    \__bnvs_use:c { if_resolve_#1:nncTF } {} { #3 } { #4 } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
}

\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_resolved:nnncTF { #1 } { #2 } A { #3 } {
      \prg_return_true:
    } {
      \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
        \__bnvs_require:nnnc { #1 } { #2 } A { #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_begin:
  \__bnvs_if_resolve_first: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_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_V:nncTF { #1 } { #2.last } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_resolved:nnncTF { #1 } { #2 } Z { #3 } {
      \prg_return_true:
    } {
      \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
        \__bnvs_require:nnnc { #1 } { #2 } Z { #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_begin:
  \__bnvs_if_resolve_last: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_last } { T, F, TF }
```

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

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_V:nncTF { #1 } { #2.length } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_resolved:nnncTF { #1 } { #2 } L { #3 } {
      \prg_return_true:
    } {
      \__bnvs_if_resolve_R:nncTF { #1 } { #2 } { #3 } {
        \__bnvs_require:nnnc { #1 } { #2 } L { #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_begin:
  \__bnvs_if_resolve_length: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_length } { 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_resolve_V:nncTF { #1 } { #2.previous } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_resolve_first:nncTF { #1 } { #2 } { #3 } {
      \__bnvs_tl_put_right:cn { #3 } { -1 }
      \__bnvs_round:c { #3 }
      \__bnvs_gset:nnnv { #1 } { #2 } P { #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_resolve_V:nncTF { #1 } { #2.next } { #3 } {
    \prg_return_true:
  } {
    \__bnvs_if_resolve_last:nncTF { #1 } { #2 } { #3 } {
      \__bnvs_tl_put_right:cn { #3 } { +1 }
      \__bnvs_round:c { #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_index_can:nnTF __bnvs_index_can:nnTF {⟨id⟩} {⟨tag⟩} {⟨yes code⟩} {⟨no code⟩}
__bnvs_index_can:vvTF __bnvs_if_resolve_N:nnncTF {⟨id⟩} {⟨tag⟩} {⟨integer⟩} {⟨ans⟩} {⟨yes code⟩}
__bnvs_if_resolve_N:nnncTF {⟨no code⟩}
__bnvs_if_resolve_N:vvvcTF __bnvs_if_append_N:nnncTF {⟨id⟩} {⟨tag⟩} {⟨integer⟩} {⟨ans⟩} {⟨yes code⟩}
__bnvs_if_append_N:nnncTF {⟨no code⟩}
__bnvs_if_append_N:vvvcTF

Resolve the index associated to the ⟨*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 { #1 } { #2 } V {
    \prg_return_true:
  } {
    \__bnvs_is_gset:nnnTF { #1 } { #2 } A {
      \prg_return_true:
    } {
      \__bnvs_is_gset:nnnTF { #1 } { #2 } Z {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
}

\BNVS_new_conditional:cpnn { index_can:vv } #1 #2 { T, F, TF } {
  \BNVS_tl_use:Nvv \__bnvs_index_can:nnTF { #1 } { #2 }
    { \prg_return_true: } { \prg_return_false: }
}

\BNVS_new_conditional:cpnn { if_resolve_N:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__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_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_N:nnvc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_tl_use:nv {
    \__bnvs_if_resolve_N:nnncTF { #1 } { #2 }
  } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}

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

\BNVS_new_conditional:cpnn { if_append_N:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_N: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_N:vvvc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_tl_use:nvv {
    \BNVS_tl_use:Nv \__bnvs_if_append_N:nnncTF { #1 }
  } { #2 } { #3 } { #4 } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

6.17 Value counter

__bnvs_if_resolve_incr:nnncTF
__bnvs_if_append_incr:nnncTF
__bnvs_if_append_incr:vvncTF

__bnvs_if_resolve_incr:nnnTF {⟨id⟩} {⟨tag⟩} {⟨offset⟩} ⟨ans⟩ {⟨yes code⟩} {⟨no code⟩}
__bnvs_if_append_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_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_resolve_query: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 { #1 } { #2 } V { #4 }
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  } {
    \prg_return_false:
  }
}

\BNVS_new_conditional:cpnn { if_append_incr:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_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_incr } { T, F, TF }

\BNVS_new_conditional:cpnn
  { if_resolve_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \__bnvs_if_resolve_V:nncTF { #1 } { #2 } { #4 } {
    \BNVS_begin:
    \__bnvs_if_resolve_query: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_tl_gset:nnnv { #1 } { #2 } V a
        \prg_return_true:
      }
    } {
      \BNVS_end:
      \prg_return_false:
    }
  } {
      \prg_return_false:
  }
}
\BNVS_new_conditional_vvvc:cn { if_resolve_post } { T, F, TF }

\BNVS_new_conditional:cpnn { if_append_post:nnnc } #1 #2 #3 #4 { T, F, TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_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_post } { T, F, TF }
\BNVS_new_conditional_vvvc:cn { if_append_post } { T, F, TF }
```

6.18 Functions for the resolution

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

`\__bnvs_split_pop_XP:TFF`
`\__bnvs_split_if_pop_GNURO_or_end_return:T`
`\__bnvs_split_pop_XPGNURO:T`

`\__bnvs_split_pop_XP:TFF {`⟨*black code*⟩`} {`⟨*blank code*⟩`} {`⟨*end code*⟩`}`
`\__bnvs_split_if_pop_GNURO_or_end_return:T {`⟨*blank code*⟩`}`
`\__bnvs_split_pop_XPGNURO:T {`⟨*black code*⟩`}`

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

For `\__bnvs_split_if_pop_GNURO_or_end_return:T`: pops the four heading items of the `split` sequence into the five variables `G`, `N`, `U`, `R` and `O`. Then execute ⟨*black code*⟩.

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

Next is called after a query has been splitted.

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

```
  \__bnvs_split_if_pop_left:cTF I {
    \cs_set:Npn \BNVS_split_F:n ##1 {
      \BNVS_fatal:n { split_pop_XP:TFF/##1 }
    }
    \__bnvs_split_if_pop_left:cTn K {
      \__bnvs_split_if_pop_left:cTn S {
        \__bnvs_split_if_pop_left:cTn P {
          \__bnvs_tl_if_blank:vTF S {
```

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

```
            #2
          } {
            \BNVS_tl_use:nv {
              \regex_match:NnT \c__bnvs_A_reserved_Z_regex
            } S {
              \__bnvs_tl_if_eq:cnF S { pauses } {
                \__bnvs_tl_if_eq:cnF S { slideinframe } {
\BNVS_error:x { Use~of~reserved~``\__bnvs_tl_use:c T'' }
                }
              }
            }
            \__bnvs_tl_if_blank:vTF K {
              \__bnvs_tl_set:cv I J
            } {
              \__bnvs_tl_set:cv J I
            }
```

Build the `path` sequence and lowercase components conditionals.

```
            \__bnvs_tl_if_empty:cTF P {
              \__bnvs_seq_clear:c P
            } {
              \__bnvs_seq_set_split:cnv P { . } P
            }
            #1
          }
        } P
      } S
    } K
  } {
    #3
  }
}


\BNVS_new:cpn { split_if_pop_GNURO_or_end_return:T } #1 {
  \__bnvs_split_if_pop_left:cT G {
    \__bnvs_split_if_pop_left:cT N {
      \__bnvs_split_if_pop_left:cT U {
        \__bnvs_split_if_pop_left:cT R {
          \__bnvs_split_if_pop_left:cT O {
```

```
            #1
          }
        }
      }
    }
  }
}
```

`\__bnvs_if_resolve_query:nc`*TF* `\__bnvs_if_resolve_query:ncTF {`⟨*query*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_if_resolve_query:vc`*TF* `\__bnvs_if_append_query:ncTF {`⟨*query*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`
`\__bnvs_if_append:nc`*TF*
`\__bnvs_if_append:vc`*TF*

Resolves the ⟨*query*⟩, 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 the function `\__bnvs_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_T_tl` Storage for `split` sequence items that represent names.

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

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

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

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

```
\BNVS_new_conditional:cpnn { if_append:nc } #1 #2 { TF } {
  \BNVS_begin:
  \__bnvs_if_resolve_query: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:
}
```

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

```
\BNVS_new:cpn { split_pop_spec_or_end_return: } {
  \__bnvs_split_pop_XP:TFF {
    \__bnvs_split_pop_XPGNURO:T {
      \__bnvs_prepare_context:N \c_true_bool
      \__bnvs_build_T:
      \__bnvs_resolve_loop_or_end_return_iadd:n { 1 }
    }
  } {
    \__bnvs_split_pop_XP:TFF {
      \__bnvs_split_if_pop_GNURO_or_end_return:T {
        \__bnvs_tl_if_eq:cnTF N 0 {
          \__bnvs_build_T:
          \__bnvs_resolve_loop_N_or_end_return:
        } {
          \__bnvs_tl_if_empty:cTF N {
            \__bnvs_tl_if_blank:vTF U {
              \__bnvs_tl_if_blank:vTF R {
                \__bnvs_tl_if_blank:vTF 0 {
                  \__bnvs_prepare_context:N \c_false_bool
                  \__bnvs_build_T:
                  \BNVS_tl_use:Nv
                  \__bnvs_if_resolve_counter:ncTF T V {
                    \__bnvs_tl_put_right:cv { ans } V
                    \__bnvs_resolve_loop_or_end_return:
                  } {
                    \__bnvs_if_resolve_V:vvcTF IT V {
                      \__bnvs_tl_put_right:cv { ans } V
                      \__bnvs_resolve_loop_or_end_return:
                    } {
```

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

```
                    \__bnvs_if_resolve_R:vvcTF IT r {
                      \__bnvs_tl_put_right:cv { ans } r
```

```
                        \__bnvs_resolve_loop_or_end_return:
                      } {
                        \__bnvs_tl_if_empty:cTF { suffix } {
                          \__bnvs_resolve_loop_or_end_return_V:
                        } {
                          \__bnvs_resolve_loop_or_end_return_suffix:
                        }
                      }
                    }
                  }
                } {
                  \__bnvs_prepare_context:N \c_true_bool
                  \__bnvs_build_T:
                  \BNVS_use:c { resolve_loop_or_end_return[...++]: }
                }
              } {
                \__bnvs_prepare_context:N \c_true_bool
                \__bnvs_build_T:
                \__bnvs_resolve_loop_R_or_end_return:
              }
            } {
              \__bnvs_if_resolve_query:vcTF R R {
                \__bnvs_prepare_context:N \c_true_bool
                \__bnvs_build_T:
                \BNVS_tl_use:Nv
                  \__bnvs_resolve_loop_or_end_return_iadd:n R
              } {
                \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c R }
                \__bnvs_resolve_loop_or_end_return:
              }
            }
          } {
            \__bnvs_normalize_GN:
            \__bnvs_build_T:
            \__bnvs_resolve_loop_N_or_end_return:
          }
        }
      }
    } {
\BNVS_end_unreachable_return_false:n { resolve_loop_or_end_return:/3 }
    } {
\BNVS_end_unreachable_return_false:n { resolve_loop_or_end_return:/2 }
    }
  } {
```

The `split` sequence is empty.

```
    \__bnvs_if_resolve_end_return_true:
  }
}
```

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

```
\BNVS_new:cpn { resolve_init_end_return: } {
```

```
  \__bnvs_split_if_pop_left:cTF a {
    \__bnvs_tl_put_right:cv { ans } a
    \__bnvs_split_pop_spec_or_end_return:
  } {
\BNVS_end_unreachable_return_false:n { resolve_init_end_return:/1 }
  }
}
```

Next is a main entry point.

```
\tl_new:N \l__bnvs_if_resolved_A_tl
\tl_new:N \l__bnvs_if_resolved_B_tl
\BNVS_new_conditional:cpnn { if_resolve:nc } #1 #2 { T, F, TF } {
  \__bnvs_if_call:TF {
    \BNVS_begin:
```

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

```
    \regex_match:nnTF { \c { BNVS_query:n } } { #1 } {
      \__bnvs_tl_clear:c { if_resolved_A }
      \cs_set:Npn \BNVS_if_resolve_return: {
        \BNVS_end_tl_put_right:cv { #2 } { if_resolved_A }
        \prg_return_true:
      }
      \cs_set:Npn \BNVS:w ##1 \BNVS_query:n ##2 ##3 \s_stop {
        \__bnvs_tl_put_right:cn { if_resolved_A } { ##1 }
        \__bnvs_if_append:ncF { ##2 } { if_resolved_A } {
          \cs_set:Npn \BNVS_if_resolve_return: {
            \prg_return_false:
          }
        }
        \regex_match:nnTF { \c { BNVS_query:n } } { ##3 } {
          \BNVS:w #1 \s_stop
        } {
          \__bnvs_if_resolve_flat:ncTF { ##3 } { if_resolved_B } {
            \__bnvs_tl_put_right:cv { if_resolved_A } { if_resolved_B }
          } {
            \cs_set:Npn \BNVS_if_resolve_return: {
              \prg_return_false:
            }
          }
        }
      }
      \BNVS:w #1 \s_stop
      \BNVS_if_resolve_return:
    } {
      \__bnvs_if_resolve_flat:ncTF { #1 } { if_resolved_A } {
        \BNVS_end_tl_put_right:cv { #2 } { if_resolved_A }
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  } {
    \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_T: } {
  \__bnvs_tl_set_eq:cc T S
  \__bnvs_seq_map_inline:cn P {
    \__bnvs_tl_put_right:cn T { . ##1 }
  }
}
```

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

```
\BNVS_new:cpn { build_tag_head: } {
  \__bnvs_tl_set_eq:cc T S
  \__bnvs_seq_map_inline:cn { P_head } {
    \__bnvs_tl_put_right:cn T { . ##1 }
  }
}
```

`\__bnvs_resolve_loop_or_end_return:` `\__bnvs_resolve_loop_or_end_return:`

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

```
\tl_map_inline:nn {
  {n}{reset}{reset_all}{v}{first}{last}{length}
  {previous}{next}{range}{assign}{only}
} {
  \bool_new:c { \BNVS_l:cn { #1 } { bool } }
}

\BNVS_new_conditional:cpnn { if:c } #1 { p, T, F, TF } {
  \bool_if:cTF { \BNVS_l:cn { #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 { \BNVS_l:cn { #1 } { bool } } {
    \prg_return_true:
  } {
    \prg_return_false:
  }
}
```

```
\BNVS_new:cpn { prepare_context:N } #1 {
  \clist_map_inline:nn {
    reset, reset_all, first, last, length,
    previous, next, range, assign, only
  } {
    \__bnvs_set_false:c { ##1 }
  }
  \__bnvs_seq_clear:c { P_head }
  \__bnvs_seq_clear:c { P_tail }
  \__bnvs_tl_clear:c N
  \__bnvs_tl_clear:c { suffix }
  \BNVS_set:cpn { prepare_context_N:n } ##1 {
    \tl_if_blank:nF { ##1 } {
      \__bnvs_tl_if_empty:cF N {
       \__bnvs_seq_put_right:cv { P_head } N
        \__bnvs_tl_clear:c N
      }
      \__bnvs_seq_put_right:cn { P_head } { ##1 }
    }
  }
  \__bnvs_seq_map_inline:cn P {
    \__bnvs_bool_if_exist:cTF { ##1 } {
      \__bnvs_set_true:c { ##1 }
      \clist_if_in:nnF { reset, reset_all } { ##1 } {
        \bool_if:NT #1 {
          \BNVS_error:n {Unexpected~##1~in~assignment }
        }
        \__bnvs_tl_set:cn { suffix } { ##1 }
      }
      \BNVS_set:cpn { prepare_context_N: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 N {
           \__bnvs_seq_put_right:cv { P_head } N
        }
        \__bnvs_tl_set:cn N { ##1 }
      } {
        \regex_match:NnTF \c__bnvs_A_reserved_Z_regex { ##1 } {
          \BNVS_error:n { Unsupported~##1 }
        } {
          \__bnvs_prepare_context_N:n { ##1 }
        }
      }
    }
  }
  \__bnvs_seq_set_eq:cc P { P_head }
}

\BNVS_new:cpn { resolve_loop_or_end_return: } {
```

```
  \__bnvs_split_if_pop_left:cTF a {
    \__bnvs_tl_put_right:cv { ans } a
    \__bnvs_tl_if_empty:cTF a {
```

This is where the difference with `\resolve_loop_or_end_return:` lives.

```
      \__bnvs_seq_if_empty:cTF { split } {
        \__bnvs_if_resolve_end_return_true:
      } {
        \__bnvs_if_resolve_end_return_false:
      }
    } {
      \__bnvs_split_pop_spec_or_end_return:
    }
  } {
\BNVS_end_unreachable_return_false:n { resolve_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 P { suffix } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  } {
    \prg_return_true:
  }
}
```

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

```
\BNVS_set:cpn { if_resolve_V_loop_or_end_return_true:F } #1 {
  \__bnvs_build_T:
  \__bnvs_tl_set:cx a {
     \__bnvs_tl_use:c T . \__bnvs_tl_use:c { suffix }
  }
  \__bnvs_if_resolve_v:vvcTF I a a {
    \__bnvs_tl_put_right:cv { ans } a
    \__bnvs_resolve_loop_or_end_return:
  } {
    \__bnvs_if_resolve_V:vvcTF I a a {
      \__bnvs_tl_put_right:cv { ans } a
      \__bnvs_resolve_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:vvcTF I T V {
```

```
        \__bnvs_if_ISP:cccTF I V P {
          \__bnvs_tl_set_eq:cc T V
          \__bnvs_seq_merge:cc P { P_tail }
          \__bnvs_seq_clear:c { P_tail }
          \__bnvs_seq_set_eq:cc { P_head } P
          \__bnvs_path_branch_IT_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_resolve_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 { P_head } a {
    \__bnvs_seq_put_left:cv { P_tail } a
    \__bnvs_build_tag_head:
    \__bnvs_path_branch_IT_loop_or_end_return:
  } {
    \__bnvs_build_T:
    \__bnvs_seq_set_eq:cc { P_head } { P_tail }
    \__bnvs_seq_clear:c { P_tail }
    \STOP
    \__bnvs_is_gset:vxnTF I { \__bnvs_tl_use:c T.1 } V {
      \__bnvs_tl_set:cn N 1
      \__bnvs_resolve_loop_N_or_end_return:
    } {
      \__bnvs_gset:vvnn I T V 1
      \__bnvs_path_branch_IT_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_IT_loop_or_end_return: } {
  \__bnvs_build_tag_head:
  \__bnvs_if_resolve_V:vvcTF I T V {
```

```
    \__bnvs_tl_put_right:cv { ans } V
    \__bnvs_resolve_loop_or_end_return:
  } {
    \__bnvs_path_branch_loop_or_end_return:
  }
}
```

- Case `....`⟨*index*⟩.

```
\BNVS_new:cpn { resolve_loop_N_or_end_return: } {
  % known, id, tag, path, suffix
  \__bnvs_set_if_path_branch:n {
    \__bnvs_if_append_N:vvvcTF I T N { ans } {
      \prg_return_true:
    } {
      \prg_return_false:
    }
  }
  \__bnvs_path_branch_loop_or_end_return:
}
```

```
\BNVS_new:cpn { resolve_loop_reset: } {
  \__bnvs_if:cTF { reset_all } {
    \__bnvs_set_false:c { reset_all }
    \__bnvs_set_false:c { reset }
    \__bnvs_gunset:vvn I T v
  } {
    \__bnvs_if:cT { reset } {
      \__bnvs_set_false:c { reset }
      \__bnvs_gunset:vvn I T v
    }
  }
}
```

- Case `....`

```
\BNVS_new:cpn { resolve_loop_or_end_return_V: } {
  \__bnvs_resolve_loop_reset:
  \__bnvs_set_if_path_branch:n {
    \__bnvs_if_append_V:vvcTF I T { ans } {
      \prg_return_true:
    } {
      \__bnvs_if_append_V:vvcTF I T { ans } {
        \prg_return_true:
      } {
        \prg_return_false:
      }
    }
  }
  \__bnvs_path_branch_loop_or_end_return:
}
```

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

```
\BNVS_new:cpn { resolve_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
     } I T { ans } {
       \__bnvs_if:cT { range } {
         \BNVS_set:cpn { resolution_round_ans: } { }
       }
       \prg_return_true:
     } {
       \prg_return_false:
     }
   }
   \__bnvs_path_branch_loop_or_end_return:
 }
}
```

- Case `...++`.

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

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

```
    \cs_set:Npn \BNVS_resolve_loop: {
      NO~....reset++~for
        ~\__bnvs_tl_use:c I!\__bnvs_tl_use:c T
    }
  } {
```

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

```
    \cs_set:Npn \BNVS_resolve_loop: {
      \BNVS_error_ans:x {
        NO~...(.reset_all)++~for
          ~\__bnvs_tl_use:c I!\__bnvs_tl_use:c T
      }
    }
  }
  \__bnvs_build_T:
  \__bnvs_resolve_loop_reset:
  \__bnvs_if_append_post:vvncTF I T { 1 } { ans } {
  } {
    \BNVS_error_ans:x {
      Problem~with~\__bnvs_tl_use:c I!\__bnvs_tl_use:c T~use.
    }
  }
  \__bnvs_resolve_loop_or_end_return:
}
```

```
\BNVS_new:cpn { resolve_loop_R_or_end_return: } {
```

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

```
  \__bnvs_if_resolve_query:vcTF R R {
```

```
    \__bnvs_is_gset:vvnF I T V {
      \__bnvs_gset:vvnv I T V R
    }
    \__bnvs_gset:vvnv I T V R
    \__bnvs_if_append_V:vvcTF I T { ans } {
    } {
      \BNVS_error_ans:n { No~...=... }
    }
  } {
    \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c R. }
  }
  \__bnvs_resolve_loop_or_end_return:
}
```

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

```
\BNVS_new:cpn { resolve_loop_or_end_return_iadd:n } #1 {
  \__bnvs_if_resolve_query:ncTF { #1 } { rhs } {
    \__bnvs_resolve_loop_reset:
    \__bnvs_if_append_incr:vvncTF I T { #1 } { ans } {
    } {
      \BNVS_error_ans:n { No~...+=... }
    }
  } {
    \BNVS_error_ans:x { Error~in~\__bnvs_tl_use:c { rhs } }
  }
  \__bnvs_resolve_loop_or_end_return:
}
```

6.18.1 Resolve one query

`\__bnvs_if_resolve_query:nc`*TF* `\__bnvs_if_resolve_query:ncTF {`⟨*query*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

Evaluates the single ⟨*query*⟩, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of 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_if_resolve_query_branch:`*TF* `\__bnvs_if_resolve_query_branch:TF {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

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

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

At start, we ignore the whole match.

```
  \__bnvs_match_if_pop_left:cT V {
```

```
    \__bnvs_match_if_pop_left:cT V {
      \__bnvs_if_blank:cTF V {
        \__bnvs_match_if_pop_left:cT A {
          \__bnvs_match_if_pop_left:cT Z {
            \__bnvs_match_if_pop_left:cT L {
              \__bnvs_if_blank:cTF A {
                \__bnvs_match_if_pop_left:cT L {
                  \__bnvs_match_if_pop_left:cT Z {
                    \__bnvs_if_blank:cTF L {
                      \__bnvs_match_if_pop_left:cT Z {
                        \__bnvs_match_if_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_query: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_query:vcTF A A {
    \__bnvs_if_resolve_query:vcTF L { ans } {
      \__bnvs_tl_put_right:cn { ans } { + \l__bnvs_A_tl - 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_query: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_query:vcTF Z Z {
    \__bnvs_if_resolve_query: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_queries:nc`*TF* `\__bnvs_if_resolve_queries:ncTF {`⟨*queries*⟩`} {`⟨*ans*⟩`} {`⟨*yes code*⟩`} {`⟨*no code*⟩`}`

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

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

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

`\l__bnvs_ans_seq` Storage for the evaluated result.

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

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

```
\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 the `query` sequence.

```
  \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 { if_resolve_queries_end_return: } {
    \__bnvs_seq_if_empty:cTF { ans } {
      \BNVS_end:
      \__bnvs_tl_clear_ans:
    } {
```

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

```
      \exp_args:NNnx
      \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_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 { if_resolve_queries_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_if_resolve_queries_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_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 }
      }
    }
  } {}
}
```

6.19 Reseting counters and values

```
\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_int_zero:c { i }
    \__bnvs_provide_off:
    \__bnvs_root_keyval:nc { #2 } a
    \BNVS_end_tl_set:cv J J
    \ignorespaces
  }
}
\makeatother

\ExplSyntaxOff
```