

The starray Package

Version 1.9

Alceu Frigeri*

March 2024

Abstract

This package implements vector like 'structures', alike 'C' and other programming languages. It's based on `expl3` and aimed at 'package writers', and not end users. The provided 'functions' are similar the ones provided for property (or sequence, or token) lists. Most of the provided functions have a companion 'branching version'.

Contents

1	Introduction	1
2	Package Options	2
3	Creating a starray	2
3.1	Conditionals	3
4	Defining and initialising a starray structure	3
4.1	Fixing an ill-instantiated starray	4
5	Instantiating starray terms	4
5.1	referencing terms	5
5.2	iterators	7
6	Changing and recovering starray properties	10
7	Additional Commands and Conditionals	11
8	Showing (debugging) starrays	12

1 Introduction

The main idea is to have an array like syntax when setting/recovering structured information, e.g. `\starray_get_prop:n` `{\student[2].work[3].reviewer[4]}` `{\name}` where "student" is the starray root, "work" is a sub-structure (an array in itself), "reviewer" is a sub-structure of "work" and so on, `\name` being a property of "reviewer". Moreover one can iterate over the structure, for instance `\starray_get_prop:n` `{\student.work.reviewer}` `{\name}` is also a possible reference in which one is using "student's", "work's" and "reviewer's" iterators.

Internally, a *starray* is stored as a collection of property lists. Each *starray* can contain a list of property pairs (key/value as in any `expl3` property lists) and a list of sub-structures. Each sub-structure, at it's turn, can also contain a list of property pairs and a list of sub-structures.

The construction/definition of a *starray* can be done piecewise (a property/sub-structure a time) or with a keyval interface or both, either way, one has to first "create a root starray" (`\starray_new:n`), define it's elements (properties and sub-structures), then instantiate them "as needed". An instance of a *starray* (or one of it's sub-structures) is referred, in this text, as a "term".

*<https://github.com/alceu-frigeri/starray>

Finally, almost all defined functions have a branching version, as per *expl3*: `T`, `F` and `TF` (note: no `_p` variants, see below). For simplicity, in the text below only the `TF` variant is described, as in `\starray_new:nTF`, keep in mind that all 3 variants are defined, e.g. `\starray_new:nT`, `\starray_new:nF` and `\starray_new:nTF`.

Note: Could it be implemented with a single property list? It sure could, but at a cost: 1. complexity; 2. access time. The current implementation, albeit also complex, tries to reach a balance between inherent structure complexity, number of used/defined auxiliary property lists and access time.

Important: *Expandability*, unfortunately most/all defined functions are not "fully expandable", in particular, most conditional/branching functions aren't, with just a few exceptions (marked with a star ★, as per *expl3* documentation convention).

2 Package Options

The package options (*key=value*) are:

`prefix` (default: `l__starray_`). Set the *prefix* used when declaring the property lists associated with any *starray*.

`msg-err` By default, the *starray* package only generates "warnings", with `msg-err` one can choose which cases will generate "package error" messages. There are 3 message classes: 1. *strict* relates to `\starray_new:n` cases (*starray* creation); 2. *syntax* relates to "term syntax" errors (student.work.reviewer in the above examples); finally 3. *reference* relates to cases whereas the syntax is correct but referring to non-existent terms/properties.

<code>none</code>	(default) no package message will raise an error.
<code>strict</code>	will raise an error on <i>strict</i> case alone.
<code>syntax</code>	will raise an error on <i>strict</i> and <i>syntax</i> cases.
<code>reference</code>	will raise an error on <i>strict</i> , <i>syntax</i> and <i>reference</i> cases.
<code>all</code>	will raise an error on all cases.

`msg-suppress` ditto, to suppress classes of messages:

<code>none</code>	(default) no package message will be suppressed.
<code>reference</code>	only <i>reference</i> level messages will be suppressed.
<code>syntax</code>	<i>reference</i> and <i>syntax</i> level messages will be suppressed.
<code>strict</code>	<i>reference</i> , <i>syntax</i> and <i>strict</i> level messages will be suppressed.
<code>all</code>	all messages will be suppressed.

3 Creating a starray

```

\starray_new:n \starray_new:n {<starray>}
\starray_new:nTF \starray_new:nTF {<starray>} {<if-true>} {<if-false>}

```

Creates a new `<starray>` or raises a warning if the name is already taken. The declaration (and associated property lists) is global. The given name is referred (in this text) as the `<starray-root>` or just `<root>`.

Note: A warning is raised (see 2) if the name is already taken. The branching version doesn't raise any warning.

3.1 Conditionals

```

\starray_if_exist_p:n ★ \starray_if_exist_p:n {⟨starray⟩}
\starray_if_exist:nTF ★ \starray_if_exist:nTF {⟨starray⟩} {⟨if-true⟩} {⟨if-false⟩}
\starray_if_valid_p:n ★ \starray_if_valid_p:n {⟨starray⟩}
\starray_if_valid:nTF ★ \starray_if_valid:nTF {⟨starray⟩} {⟨if-true⟩} {⟨if-false⟩}

```

new: 2023/05/20
update: 2024/03/28

`\starray_if_exist:nTF` only tests if `⟨starray⟩` (the base property) is defined. It doesn't verify if it really is a `starray`. `\starray_if_valid:nTF` is functionally equivalent since release 1.9. See `\starray_term_syntax:nTF`, section 7, for a more reliable validity test.

Note: The predicate versions, `_p`, expand to either `\prg_return_true:` or `\prg_return_false:`

4 Defining and initialising a starray structure

```

\starray_def_prop:nnn \starray_def_prop:nnn {⟨starray-ref⟩} {⟨prop-key⟩} {⟨initial-value⟩}
\starray_def_prop:nnnTF \starray_def_prop:nnnTF {⟨starray-ref⟩} {⟨prop-key⟩} {⟨initial-value⟩} {⟨if-true⟩}
\starray_def_prop:nnnTF {⟨if-false⟩}

```

Adds an entry, `⟨prop-key⟩`, to the `⟨starray-ref⟩` (see 5.1) definition and set its initial value. If `⟨prop-key⟩` is already present its initial value is updated. Both `⟨prop-key⟩` and `⟨initial-value⟩` may contain any `⟨balanced text⟩`. `⟨prop-key⟩` is an (`expl3`) property list `⟨key⟩` meaning that category codes are ignored.

The definition/assignment of a `⟨prop-key⟩` to a `⟨starray-ref⟩` is global.

Note: A warning is raised (see 2) in case of a `⟨starray-ref⟩` syntax/reference error. The branching version doesn't raise any warning.

```

\starray_def_structure:nn \starray_def_struct:nn {⟨starray-ref⟩} {⟨struct-name⟩}
\starray_def_structure:nnTF \starray_def_struct:nnTF {⟨starray-ref⟩} {⟨struct-name⟩} {⟨if-true⟩} {⟨if-false⟩}

```

Adds a sub-structure (a `starray` in itself) to `⟨starray-ref⟩` (see 5.1). If `⟨struct-name⟩` is already present nothing happens. The definition/assignment of a `⟨struct-name⟩` to a `⟨starray-ref⟩` is global.

Note: Do not use a dot when defining a (sub-)structure name, it might seem to work but it will break further down (see 5.1).

Note 2: A warning is raised (see 2) in case of a `⟨starray-ref⟩` syntax error. The branching version doesn't raise any warning.

```

\starray_def_from_keyval:nn \starray_def_from_keyval:nn {⟨starray-ref⟩} {⟨keyval-1st⟩}
\starray_def_from_keyval:nnTF \starray_def_from_keyval:nnTF {⟨starray-ref⟩} {⟨keyval-1st⟩} {⟨if-true⟩} {⟨if-false⟩}

```

Adds a set of `⟨keys⟩` / `⟨values⟩` and/or `⟨structures⟩` to `⟨starray-ref⟩` (see 5.1). The `⟨keyval-1st⟩` is pretty straightforward, the construction `⟨key⟩ . struct` denotes a nested structure :

```

\starray_def_from_keyval:nn {root.substructure}
{
  keyA = valA ,
  keyB = valB ,
  subZ . struct =
  {
    keyZA = valZA ,
    keyZB = valZB ,
  }
  subY . struct =
  {
    keyYA = valYA ,
    keyYB = valYB ,
    subYYY . struct =
    {
      keyYYYa = valYYYa ,
      keyYYYb = valYYYb
    }
  }
}

```

The definitions/assignments to $\langle\text{starray-ref}\rangle$ are all global.

Note: The non-branching version raises a warning (see 2) in case of a $\langle\text{starray-ref}\rangle$ syntax error. The branching version doesn't raise any warning. Also note that, syntax errors on the $\langle\text{keyval-1st}\rangle$ might raise low level (T_EX) errors.

4.1 Fixing an ill-instantiated starray

When instantiating (see 5) a *starray*, the associated structured will be constructed based on it's "current definition" (see 4). A problem that might arise, when one extends the definition of an already instantiated *starray* (better said, if one adds a sub-structure), is a *quark loop* (from *l3quark*). To avoid a *quark loop* it is necessary to "fix" the structure of already instantiated terms.

```

\starray_fix_terms:n \starray_fix_terms:n {\starray-ref}

```

The sole purpose of this function is to "fix" the already instantiated terms of a *starray*. Note, this can be an expensive operation depending on the number of terms (it has to crawl over all the terms of an instantiated *starray* adding any missing sub-structure references), but one doesn't need to run it "right away" it is possible to add a bunch of sub-structures and then run this just once.

5 Instantiating starray terms

```

\starray_new_term:n \starray_new_term:n {\starray-ref}
\starray_new_term:nn \starray_new_term:nn {\starray-ref} {\hash}
\starray_new_term:nTF \starray_new_term:nTF {\starray-ref} {\if-true} {\if-false}
\starray_new_term:nnTF \starray_new_term:nnTF {\starray-ref} {\hash} {\if-true} {\if-false}

```

This creates a new *term* (in fact a property list) of the (sub-)structure referenced by $\langle\text{starray-ref}\rangle$. Note that the newly created *term* will have all properties (key/values) as defined by the associated $\backslash\text{starray_prop_def:nn} \{\langle\text{starray-ref}\rangle\}$, with the respective "initial values". For instance, given the following

```

\starray_new:n {st-root}

\starray_def_from_keyval:nn {st-root}
{
  keyA = valA ,
  keyB = valB ,
  subZ . struct =
  {
    keyZA = valZA ,
    keyZB = valZB ,
  }
  subY . struct =
  {
    keyYA = valYA ,
    keyYB = valYB ,
    subYYY . struct =
    {
      keyYYYa = valYYYa ,
      keyYYYb = valYYYb
    }
  }
}

\starray_new_term:n {st-root}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subY}
\starray_new_term:nn {st-root}{hash-A}
\starray_new_term:n {st-root.subZ}

```

One will have created 6 *terms*:

1. 2 $\langle\text{st-root}\rangle$ *terms*
 - (a) the first one with index 1 and
 - i. 2 sub-structures $\langle\text{subZ}\rangle$ (indexes 1 and 2)
 - ii. 1 sub-structure $\langle\text{subY}\rangle$ (index 1)
 - (b) the second one with indexes 2 and "hash-A" and
 - i. 1 sub-structure $\langle\text{subZ}\rangle$ (index 1)

Note that, in the above example, it was used the "implicit" indexing (aka. iterator, see 5.1). Also note that no *term* of kind $\langle\text{subYYY}\rangle$ was created.

Note: A warning is raised (see 2) in case of a $\langle\text{starray-ref}\rangle$ syntax error. The branching version doesn't raise any warning.

5.1 referencing terms

When typing a $\langle\text{starray-ref}\rangle$ there are 3 cases to consider:

1. structure definition
2. term instantiation
3. getting/setting a property

The first case is the simplest one, in which, one (starting by $\langle\text{starray-root}\rangle$) will use a construct like $\langle\text{starray-root}\rangle.\langle\text{sub-struct}\rangle.\langle\text{sub-struct}\rangle\dots$ For example, an equivalent construct to the one shown in 5 :

```

\starray_new:n {st-root}

\starray_def_struct:nn {st-root}{subZ}

\starray_def_prop:nnn {st-root}{keyA}{valA}
\starray_def_prop:nnn {st-root}{keyB}{valB}

\starray_def_prop:nnn {st-root.subZ}{keyZA}{valZA}
\starray_def_prop:nnn {st-root.subZ}{keyZB}{valZB}

\starray_def_struct:nn {st-root}{subY}
\starray_def_prop:nnn {st-root.subY}{keyYA}{valYA}
\starray_def_prop:nnn {st-root.subY}{keyYB}{valYB}

\starray_def_struct:nn {st-root.subY}{subYYY}
\starray_def_prop:nnn {st-root.subY.subYYY}{keyYYYY}{valYYYY}
\starray_def_prop:nnn {st-root.subY.subYYY}{keyYYYYB}{valYYYYB}

```

Note that, all it's needed in order to be able to use $\langle\text{starray-root}\rangle.\langle\text{sub-A}\rangle$ is that $\langle\text{sub-A}\rangle$ is an already declared sub-structure of $\langle\text{starray-root}\rangle$. The property definitions can be made in any order.

In all other cases, term instantiation, getting/setting a property, one has to address/reference a specific instance/term, implicitly (using iterators) or explicitly using indexes. The general form, of a $\langle\text{starray-ref}\rangle$, is:

```

\langle\text{starray-root}\rangle\langle\text{idx}\rangle.\langle\text{sub-A}\rangle\langle\text{idxA}\rangle.\langle\text{sub-B}\rangle\langle\text{idxB}\rangle

```

In the case of term instantiation the last $\langle\text{sub-}\rangle$ cannot be indexed, after all one is creating a new term/index. Moreover, all $\langle\text{idx}\rangle$ are optional like:

```

\langle\text{starray-root}\rangle.\langle\text{sub-A}\rangle\langle\text{idxA}\rangle.\langle\text{sub-B}\rangle

```

in which case, one is using the "iterator" of $\langle\text{starray-root}\rangle$ and $\langle\text{sub-B}\rangle$ (more later, but keep in mind the $\langle\text{sub-B}\rangle$ iterator is the $\langle\text{sub-B}\rangle$ associated with the $\langle\text{sub-A}\rangle\langle\text{idxA}\rangle$).

Since one has to explicitly instantiate all (sub)terms of a starray, one can end with a highly asymmetric structure. Starting at the $\langle\text{starray-root}\rangle$ one has a first counter (representing, indexing the root structure terms), then for all sub-structures of $\langle\text{starray-root}\rangle$ one will have an additional counter for every term of $\langle\text{starray-root}\rangle$!

So, for example:

```

\starray_new:n {st-root}
\starray_def_struct:nn {st-root}{subZ}
\starray_def_struct:nn {st-root}{subY}
\starray_def_struct:nn {st-root.subY}{subYYY}

\starray_new_term:n {st-root}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subY}
\starray_new_term:n {st-root.subY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY}

\starray_new_term:n {st-root}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subY}

```

One has a single $\langle\text{st-root}\rangle$ iterator (pointing to one of the 3 $\langle\text{st-root}\rangle$ terms), then 3 " $\langle\text{subZ}\rangle$ iterators", in fact, one $\langle\text{subZ}\rangle$ iterator for each $\langle\text{st-root}\rangle$ term. Likewise there are 3 " $\langle\text{subY}\rangle$ iterators" and 4 (four) " $\langle\text{subYYY}\rangle$ iterators" one for each instance of $\langle\text{subY}\rangle$.

Every time a new term is created/instantiated, the corresponding iterator will points to it, which allows the notation used in this last example, keep in mind that one could instead, using explicit indexes:

```

\starray_new:n {st-root}
\starray_def_struct:nn {st-root}{subZ}
\starray_def_struct:nn {st-root}{subY}
\starray_def_struct:nn {st-root.subY}{subYYY}

\starray_new_term:n {st-root}
\starray_new_term:n {st-root[1].subZ}
\starray_new_term:n {st-root[1].subZ}
\starray_new_term:n {st-root[1].subY}
\starray_new_term:n {st-root[1].subY}
\starray_new_term:n {st-root[1].subY[2].subYYY}
\starray_new_term:n {st-root[1].subY}

\starray_new_term:n {st-root}
\starray_new_term:n {st-root[2].subZ}
\starray_new_term:n {st-root[2].subZ}
\starray_new_term:n {st-root[2].subY}

```

Finally, observe that, when creating a new term, one has the option to assign a "hash" to it, in which case that term can be referred to using an iterator, the explicit index or the hash:

```

\starray_new:n {st-root}
\starray_def_struct:nn {st-root}{subZ}
\starray_def_struct:nn {st-root}{subY}
\starray_def_struct:nn {st-root.subY}{subYYY}

\starray_new_term:nn {st-root}{hash-A}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root[1].subZ}
\starray_new_term:n {st-root[hash-A].subZ}

```

Will create 3 $\langle \text{subZ} \rangle$ terms associated with the first (index = 1) $\langle \text{st-root} \rangle$.

5.2 iterators

<u><code>\starray_set_iter:nn</code></u>	<code>\starray_set_iter:nn {<starray-ref>} {<int-val>}</code>
<u><code>\starray_set_iter:nnTF</code></u>	<code>\starray_set_iter:nnTF {<starray-ref>} {<int-val>} {<if-true>} {<if-false>}</code>
<code>\starray_reset_iter:nn</code>	<code>\starray_reset_iter:nn {<starray-ref>}</code>
<u><code>\starray_reset_iter:nnTF</code></u>	<code>\starray_reset_iter:nnTF {<starray-ref>} {<if-true>} {<if-false>}</code>
<code>\starray_next_iter:nn</code>	<code>\starray_next_iter:nn {<starray-ref>}</code>
<u><code>\starray_next_iter:nnTF</code></u>	<code>\starray_next_iter:nnTF {<starray-ref>} {<if-true>} {<if-false>}</code>

Those functions allows to **set** an iterator to a given $\langle \text{int-val} \rangle$, **reset** it (i.e. assign 1 to the iterator), or increase the iterator by one. An iterator might have a value between 1 and the number of instantiated terms (if the given (sub-)structure was already instantiated). If the (sub-)structure hasn't been instantiated yet, the iterator will always end being set to 0. The branching versions allows to catch those cases, like trying to set a value past its maximum, or a value smaller than one.

Important: Please observe that, when setting/resetting/incrementing the iterator of a (sub-)structure, all "descending" iterators will be also be reset.

Note: A warning is raised (see 2) in case of a $\langle \text{starray-ref} \rangle$ syntax error. The branching version doesn't raise any warning.

```

\starray_new:n {st-root}
\starray_def_struct:nn {st-root}{subZ}
\starray_def_struct:nn {st-root}{subY}
\starray_def_struct:nn {st-root.subY}{subYYY}

\starray_new_term:n {st-root}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subZ}
\starray_new_term:n {st-root.subY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY}
\starray_new_term:n {st-root.subY.subYYY}
\starray_new_term:n {st-root.subY.subYYY}

\starray_set_prop:nnn {st-root.subY.subYYY}{key}{val}
\starray_set_prop:nnn {st-root[2].subY[2].subYYY[2]}{key}{val}

\starray_reset_iter:n {st-root[2].subY}

\starray_set_prop:nnn {st-root.subY.subYYY}{key}{val}
\starray_set_prop:nnn {st-root[2].subY[1].subYYY[1]}{key}{val}

```

Before the reset $\langle st-root.subY.subYYY \rangle$ was equivalent to $\langle st-root[2].subY[2].subYYY[2] \rangle$, given that each iterator was pointing to the "last term", since the reset was of the $\langle subY \rangle$ iterator, only it and the descending ones (in this example just $\langle subYYY \rangle$) where reseted, and therefore $\langle st-root.subY.subYYY \rangle$ was then equivalent to $\langle st-root[2].subY[1].subYYY[1] \rangle$

```

\starray_set_iter_from_hash:nn \starray_set_iter_from_hash:nn {starray-ref} {hash}
\starray_set_iter_from_hash:nnTF \starray_set_iter_from_hash:nnTF {starray-ref} {hash} {if-true} {if-false}

```

new: 2023/11/04

$\backslash\text{starray_set_iter_from_hash:nn}$ $\langle starray-ref \rangle$ $\langle hash \rangle$ will set iter based on the $\langle hash \rangle$ used when instantiating a term (see 5).

Note: A warning is raised (see 2) in case of a $\langle starray-ref \rangle$ syntax error or invalid $\langle hash \rangle$. The branching version doesn't raise any warning.

```

\starray_get_iter:n \starray_get_iter:n {starray-ref}
\starray_get_iter:nN \starray_get_iter:nN {starray-ref} {int-var}
\starray_get_iter:nNTF \starray_get_iter:nNTF {starray-ref} {int-var} {if-true} {if-false}

```

$\backslash\text{starray_get_iter:n}$ $\langle starray-ref \rangle$ will type in the current value of a given iterator, whilst the other two functions will save it's value in a integer variable (*expl3*).

Note: A warning is raised (see 2) in case of a $\langle starray-ref \rangle$ syntax error. The branching version doesn't raise any warning.

```

\starray_parsed_get_iter: ★ \starray_parsed_get_iter:

```

new: 2023/05/20

$\backslash\text{starray_parsed_get_iter:}$ will place in the current iterator's value, using $\backslash\text{int_use:N}$, of the last parsed term in the input stream.

Warning: This can be used after any command which 'parses a term', for instance $\backslash\text{starray_term_syntax:n}$, see section 7, but it only makes sense (and returns a reliable/meaningful result) IF the last parser operation was successfully executed.

`\starray_parsed_get_iter:NN` ★ `\starray_parsed_get_iter:NN` {⟨parsed-refA⟩} {⟨parsed-refB⟩}

new: 2023/11/28

`\starray_parsed_get_iter:` will place in the current iterator's value associated with ⟨parsed-refA⟩ and ⟨parsed-refB⟩, using `\int_use:N`, in the input stream.

Warning: ⟨parsed-refA⟩ and ⟨parsed-refB⟩ are the values returned by `\starray_term_syntax:nNN`.

`\starray_get_cnt:n` `\starray_get_cnt:n` {⟨starray-ref⟩}
`\starray_get_cnt:nN` `\starray_get_cnt:nN` {⟨starray-ref⟩} {⟨integer⟩}
`\starray_get_cnt:nNTF` `\starray_get_cnt:nNTF` {⟨starray-ref⟩} {⟨integer⟩} {⟨if-true⟩} {⟨if-false⟩}

`\starray_get_cnt:n` {⟨starray-ref⟩} will type in the current number of terms of a given (sub-)structure, whilst the other two functions will save it's value in a integer variable (*expl3*).

Note: A warning is raised (see 2) in case of a ⟨starray-ref⟩ syntax error. The branching version doesn't raise any warning.

`\starray_parsed_get_cnt:` ★ `\starray_parsed_get_cnt:`

new: 2023/05/20

`\starray_parsed_get_cnt:` will place the current number of terms, using `\int_use:N`, of the last parsed term, in the input stream.

Warning: This can be used after any command which 'parses a term', for instance `\starray_term_syntax:n`, see section 7, but it only makes sense (and returns a reliable/meaningful result) IF the last parser operation was successfully executed.

`\starray_parsed_get_cnt:NN` ★ `\starray_parsed_get_cnt:NN` {⟨parsed-refA⟩} {⟨parsed-refB⟩}

new: 2023/11/28

`\starray_parsed_get_cnt:` will place in the current number of terms associated with ⟨parsed-refA⟩ and ⟨parsed-refB⟩, using `\int_use:N`, in the input stream.

Warning: ⟨parsed-refA⟩ and ⟨parsed-refB⟩ are the values returned by `\starray_term_syntax:nNN`.

`\starray_iterate_over:nm` `\starray_iterate_over:nm` {⟨starray-ref⟩} {⟨code⟩}
`\starray_iterate_over:nmTF` `\starray_iterate_over:nmTF` {⟨starray-ref⟩} {⟨code⟩} {⟨if-true⟩} {⟨if-false⟩}

new: 2023/11/04

`\starray_iterate_over:nm` will reset the ⟨starray-ref⟩ iterator, and then execute ⟨code⟩ for each valid value of `iter`. At the loop's end, the ⟨starray-ref⟩ iterator will point to the last element of it. The ⟨if-true⟩ is executed, at the loop's end if there is no syntax error, and the referenced structure was properly instantiated. Similarly ⟨if-false⟩ is only execute if a syntax error is detected or the referenced structure wasn't properly instantiated

Note: `\starray_iterate_over:nm` Creates a local group, so that one can recurse over sub-structures. Be aware, then, that ⟨code⟩ is executed in said local group.

Note: A warning is raised (see 2) in case of a ⟨starray-ref⟩ syntax error or the structure wasn't yet instantiated. The branching version doesn't raise any warning.

6 Changing and recovering starray properties

<code>\starray_set_prop:nnn</code>	<code>\starray_set_prop:nnn</code> {<starray-ref>} {<prop-key>} {<value>}
<code>\starray_set_prop:nnV</code>	<code>\starray_set_prop:nnV</code> {<starray-ref>} {<prop-key>} {<value>}
<code>\starray_set_prop:nnnTF</code>	<code>\starray_set_prop:nnnTF</code> {<starray-ref>} {<prop-key>} {<value>} {<if-true>} {<if-false>}
<code>\starray_set_prop:nnVTF</code>	<code>\starray_set_prop:nnVTF</code> {<starray-ref>} {<prop-key>} {<value>} {<if-true>} {<if-false>}
<code>\starray_gset_prop:nnn</code>	<code>\starray_gset_prop:nnn</code> {<starray-ref>} {<prop-key>} {<value>}
<code>\starray_gset_prop:nnV</code>	<code>\starray_gset_prop:nnV</code> {<starray-ref>} {<prop-key>} {<value>}
<code>\starray_gset_prop:nnnTF</code>	<code>\starray_gset_prop:nnnTF</code> {<starray-ref>} {<prop-key>} {<value>} {<if-true>} {<if-false>}
<code>\starray_gset_prop:nnVTF</code>	<code>\starray_gset_prop:nnVTF</code> {<starray-ref>} {<prop-key>} {<value>} {<if-true>} {<if-false>}

Those are the functions that allow to (g)set (change) the value of a term's property. If the <prop-key> isn't already present it will be added just for that term <starray-ref>. The <nnV> variants allow to save any variable like a token list, property list, etc...

Note: A warning is raised (see 2) in case of a <starray-ref> syntax error. The branching version doesn't raise any warning.

<code>\starray_set_from_keyval:nn</code>	<code>\starray_set_from_keyval:nnn</code> {<starray-ref>} {<keyval-1st>}
<code>\starray_set_from_keyval:nnnTF</code>	<code>\starray_set_from_keyval:nnnTF</code> {<starray-ref>} {<keyval-1st>} {<if-true>}
<code>\starray_gset_from_keyval:nn</code>	{<if-false>}
<code>\starray_gset_from_keyval:nnnTF</code>	<code>\starray_gset_from_keyval:nnn</code> {<starray-ref>} {<keyval-1st>}
	<code>\starray_gset_from_keyval:nnnTF</code> {<starray-ref>} {<keyval-1st>} {<if-true>}
	{<if-false>}

it is possible to set a collection of properties using a key/val syntax, similar to the one used to define a *starray* from keyvals (see 4), with a few distinctions:

1. when referring a (sub-)structure one can either explicitly use an index, or
2. implicitly use it's iterator
3. if a given key isn't already presented it will be added only to the given term

Note that, in the following example, TWO iterators are being used, the one for <st-root> and then <subY>.

```
\starray_set_from_keyval:nn {st-root}
{
  keyA = valA ,
  keyB = valB ,
  subZ[2] =
  {
    keyZA = valZA ,
    keyZB = valZB ,
  }
  subY =
  {
    keyYA = valYA ,
    keyYB = valYB ,
    subYYY[1] =
    {
      keyYYYa = valYYYa ,
      keyYYYb = valYYYb
    }
  }
}
```

Also note that the above example is fully equivalent to:

```

\starray_set_prop:nnn {st-root} {keyA} {valA}
\starray_set_prop:nnn {st-root} {keyB} {valB}
\starray_set_prop:nnn {st-root.subZ[2]} {keyZA} {valZA}
\starray_set_prop:nnn {st-root.subZ[2]} {keyZB} {valZB}
\starray_set_prop:nnn {st-root.subY} {keyYA} {valYA}
\starray_set_prop:nnn {st-root.subY} {keyYB} {valYB}
\starray_set_prop:nnn {st-root.subY.subYYY[1]} {keyYYYa} {valYYYa}
\starray_set_prop:nnn {st-root.subY.subYYY[1]} {keyYYYb} {valYYYb}

```

```

\starray_get_prop:nn    \starray_get_prop:nn {<starray-ref>} {<key>}
\starray_get_prop:nnN  \starray_get_prop:nnN {<starray-ref>} {<key>} {<t1-var>}
\starray_get_prop:nnNIF \starray_get_prop:nnNIF {<starray-ref>} {<key>} {<t1-var>} {<if-true>} {<if-false>}

```

`\starray_get_prop:nn` {<starray-ref>} {<key>} places the value of <key> in the input stream. `\starray_get_prop:nnN` {<starray-ref>} {<key>} {<t1-var>} recovers the value of <key> and places it in <t1-var> (a token list variable), this is specially useful in conjunction with `\starray_set_prop:nnV`, whilst the `\starray_get_prop:nnNIF` version branches accordingly.

Note: In case of a syntax error, or <key> doesn't exist, an empty value is left in the stream (or <t1-var>).

Note: A warning is raised (see 2) in case of a <starray-ref> syntax error. The branching version doesn't raise any warning.

```

\starray_parsed_get_prop:n ★ \starray_parsed_get_prop:n {<key>}

```

new: 2023/05/20

`\starray_parsed_get_prop:n` {<key>} places the value of <key>, if it exists, from the last parsed term, in the input stream.

Warning: This can be used after any command which 'parses a term', for instance `\starray_term_syntax:n`, see section 7, but it only makes sense (and returns a reliable/meaningful result) IF the last parser operation was successfully executed.

```

\starray_parsed_get_prop:NNn ★ \starray_parsed_get_prop:NNn {<parsed-refA>} {<parsed-refB>} {<key>}

```

new: 2023/11/28

`\starray_parsed_get_prop:NNn` places the value of <key>, if it exists, associated with <parsed-refA> and <parsed-refB>.

Warning: <parsed-refA> and <parsed-refB> should be the values returned by `\starray_term_syntax:nnN`.

7 Additional Commands and Conditionals

```

\starray_if_in:nnTF \starray_if_in:nnTF {<starray-ref>} {<key>} {<if-true>} {<if-false>}

```

The `\starray_if_in:nnTF` {<starray-ref>} {<key>} {<...>} {<...>} tests if a given <key> is present.

```

\starray_term_syntax:n    \starray_term_syntax:n {<starray-ref>}
\starray_term_syntax:nTF \starray_term_syntax:nTF {<starray-ref>} {<if-true>} {<if-false>}

```

new: 2023/05/20

This will just parse a <starray-ref> reference, and set interval variables so that commands like `\starray_parsed_` can be used.

Warning: The main idea is to allow some expandable commands, but be aware that all `\starray_` commands that use a <starray-ref> use the very same parser variables. In case one needs a more permanent/resilient `starray` reference, one should use the `\starray_term_syntax:nnN` variant.

Note: A warning is raised (see 2) in case of a <starray-ref> syntax error. The branching version doesn't raise any warning.

```

\starray_term_syntax:nNN \starray_term_syntax:nNN {⟨starray-ref⟩} {⟨parsed-refA⟩} {⟨parsed-refB⟩}
\starray_term_syntax:nNTF \starray_term_syntax:nNTF {⟨starray-ref⟩} {⟨parsed-refA⟩} {⟨parsed-refB⟩} {⟨if-true⟩}
new: 2023/11/28

```

Similar to the ones above (`\starray_term_syntax:n`). `⟨parsed-refA⟩` and `⟨parsed-refB⟩` (assumed to be two token list vars, `⟨tl-var⟩`) will receive two 'internal references' that can be used in commands like `\starray_parsed_...:NN` which expects such 'references', without having to worry about using another `\starray_` command.

Note: Once correctly parsed, `⟨parsed-refA⟩` and `⟨parsed-refB⟩` can be used at 'any time' (by those few `\starray_parsed_...:NN` associated commands).

Note: A warning is raised (see 2) in case of a `⟨starray-ref⟩` syntax error (in which case `⟨parsed-refA⟩` and `⟨parsed-refB⟩` will not hold a valid value). The branching version doesn't raise any warning.

```

\starray_parsed_if_in_p:n ★ \starray_parsed_if_in_p:nTF {⟨key⟩}
\starray_parsed_if_in:nTF ★ \starray_parsed_if_in:nTF {⟨key⟩} {⟨if-true⟩} {⟨if-false⟩}
new: 2023/05/20

```

This will test if the given `key` is present in the "last parsed term".

Note: The predicate version, `_p`, expands to either `\prg_return_true:` or `\prg_return_false:`.

Warning: This can be used after any command which 'parses a term', for instance `\starray_term_syntax:n`, but it only makes sense (and returns a reliable/meaningful result) IF the last parser operation was successfully executed.

```

\starray_parsed_if_in_p:NNn ★ \starray_parsed_if_in_p:nTF {⟨parsed-refA⟩} {⟨parsed-refB⟩} {⟨key⟩}
\starray_parsed_if_in:NNnTF ★ \starray_parsed_if_in:nTF {⟨parsed-refA⟩} {⟨parsed-refB⟩} {⟨key⟩} {⟨if-true⟩}
new: 2023/11/28

```

This will test if the given `key` is present/associated with `⟨parsed-refA⟩` and `⟨parsed-refB⟩`.

Note: The predicate version, `_p`, expands to either `\prg_return_true:` or `\prg_return_false:`.

Warning: `⟨parsed-refA⟩` and `⟨parsed-refB⟩` should be the values returned by `\starray_term_syntax:nNN`.

```

\starray_get_unique_id:nN \starray_get_unique_id:nN {⟨starray-ref⟩} {⟨tl-var⟩}
\starray_get_unique_id:nNTF \starray_get_unique_id:nNTF {⟨starray-ref⟩} {⟨tl-var⟩}
new: 2024/03/10

```

Gets an 'unique ID' for a given `⟨starray-ref⟩ term`, it should help defining/creating uniquely identified auxiliary structures, like auxiliary property or sequence lists, since one can't (better said shouldn't, as per l3kernel) store an anonymous property/sequence list using V-expansion.

Note: A warning is raised (see 2) in case of a `⟨starray-ref⟩` syntax error. The branching version doesn't raise any warning.

8 Showing (debugging) starrays

```

\starray_show_def:n \starray_show_def:n {⟨starray-ref⟩}
\starray_show_def_in_text:n \starray_show_def_in_text:n {⟨starray-ref⟩}

```

Displays the `<starray>` structure definition and initial property values in the terminal or directly in text.

<u><code>\starray_show_terms:n</code></u>	<code>\starray_show_terms:n {<starray-ref>}</code>
<u><code>\starray_show_terms_in_text:n</code></u>	<code>\starray_show_terms_in_text:n {<starray-ref>}</code>

Displays the `<starray>` instantiated terms and current property values in the terminal or directly in text.