forest-ext

Clea F. Rees[*]

2026/01/17

Abstract

forest-ext consists of various libraries for Sašo Živanović's package forest (2017). The aim of the libraries is to provide bug fixes or extensions currently unavailable in forest itself. I hope that this package — or at least many of its constituents — will eventually be rendered unnecessary by an updated forest and disappear.

Contents

[*]Bug tracker: codeberg.org/cfr/prooftrees/issues | Code: codeberg.org/cfr/prooftrees | Mirror: github.com/cfr42/prooftrees

1 Basic usage

This package currently provides the following libraries:

`ext.ling` (*lib.*) Experimental elementary support for trees involving multi-dominance, based on ext.multi. See section 4.

`ext.multi` (*lib.*) Experimental elementary support for nodes with multiple parents. See section 3.

`ext.tagging` (*lib.*) Experimental automatic tagging of forest trees. See section 2.

Although this relies only on documented public interfaces provided by forest — no forest internals are patched or redefined — the library does change the same PGF internals as the tagging support in latex-lab-tikz-testphase (LaTeX Project 2025).

`ext.utils` (*lib.*) Bits 'n bobs. See section 5.

For debugging, the following alternative libraries are provided:

`ext.ling-debug` (*lib.*) ext.ling plus debugging. See section 4.

`ext.multi-debug` (*lib.*) ext.multi plus debugging. See section 3.

`ext.tagging-debug` (*lib.*) ext.tagging plus debugging. See section 2.

`ext.utils-debug` (*lib.*) ext.utils plus debugging. See section 5.

Load the libraries in the same way as standard libraries:

```
\usepackage[<comma-separated-list of libraries>]{forest}
```

or

```
\usepackage{forest}
\useforestlibrary{<comma-separated-list of libraries>}
```

For example, the following line would load forest-lib-ext.multi and apply any defaults globally.

```
\usepackage[ext.multi]{forest}
```

The following lines would load the same library, but without applying any defaults.

```
\usepackage{forest}
\useforestlibrary{ext.multi}
```

Any default settings can then be applied locally using `\forestapplylibrarydefaults{`⟨*list of libraries*⟩`}`, if desired.

2 Tagging[1]

Note that this library requires ext.utils, described in section 5.

`ext.tagging` (*lib.*) Experimental semi-automatic tagging of forest trees.

`ext.tagging-debug` (*lib.*) ext.tagging plus debugging.

forest-lib-ext.tagging (and forest-lib-ext.tagging-debug) are based on the 'first-aid' in latex-lab-tikz-testphase by Ulrike Fischer (LaTeX Project 2025). Those patches do not work with forest because

[1] For an introduction to support for tagged PDF in LaTeX 2_ε, see Fischer (2025). For gorier details see, for example, International Organization for Standardization (2025) and PDF Association (2024a,b) and related publications.

a `forest` tree includes many `tikzpicture` environments, some of which may never be typeset and all of which are used only indirectly *via* low-level TEX boxes. Moreover, the latex-lab code depends on PGF's 'remember picture' feature, which is not compatible with forest with or without tagging.

In addition to making it possible to tag `forest` environments in tagged documents, the library produces an alternative text describing the tree semi-automatically. This is important because trees are unlike some other images, where relatively short summaries provide a reasonable alternative to the picture. To provide high quality access to the information contained in a typical tree, it is necessary to describe it in detail. Both the content of the nodes and their structural relationships must be described, together with any labels and annotations.

The current implementation does not do all of the work: it does not include information from regular labels or the content of annotations added using regular Ti*k*Z or PGF techniques. However, it does describe the main tree's structure, together with the content of its nodes and edge labels, though you may need to override the generated content for content which includes special characters, in a quite broad sense of 'special'.

The support for tagging adds the following forest *stages* which are executed in order, sandwiched between `compute xy stage` and `before drawing tree`.

If you redefine (or load code which redefines) the default implementation of stages, you must include or replace the additions from this library. For an example of how to do this, see prooftrees (Rees 2026), which includes, redefines, supplements or replaces these additions.

`before tagging nodes` (*keylist*) Empty by default. Analogous to `before typesetting nodes`, `before packing` etc.

`tag nodes` (*tag. keylist*) Executes code to assign tagging code to each node in the tree.

Note this is a ***tagging*** keylist. See section 5.3.

`before collating tags` (*keylist*) Empty by default. Analogous to `before typesetting nodes`, `before packing` etc.

`collate tags` (*tag. keylist*) Walks the tree to collate the tags into a single alternative text for the tree.

Note this is a ***tagging*** keylist. See section 5.3.

`before tagging tree` (*keylist*) Empty by default. Analogous to `before typesetting nodes`, `before packing` etc.

`tag tree stage` (*stage*) Calculates an approximate bounding box for the tree and inserts the collated tagging data into the document's tagging structure using tagpdf.

The code inserts a tagged structure analogous to (and heavily derived from) the `alt` plug provided by latex-lab-tikz-testphase. However, unlike the latex-lab plug, the library generates the `alt` text automatically by default. The result can be configured using a small number of keys. The keys' scope is the entire tree, except that the scope of `alt text` is *the current node*.

`alt text` (*auto. toks*) = ⟨*tokens*⟩

Override the automatic generation of alternative text for the current node.

Internally, the code uses the further key `node@ttoks`. In essence, if `alt text` is empty, `node@ttoks` is constructed from the node's `content`, `edge label` and any applicable structural descriptors, as specified by `is root`, `is branch` and so on. If `alt text` is not empty, it is used as-is. The reason for this indirect assignment — first constructing `node@ttoks` and only then assigning it to `alt text` — is that the value of `node@ttoks` is constructed incrementally (i.e. partially by `delayed` keys) and keeping `alt text` as-is makes it easy to test during every cycle.

`node@ttoks` is intended for purely internal use and should **NOT** be used outside the library code. `alt text` is the public face of this key.

Note that tagging content is always attached to nodes[2]. Labels, edge labels and structures

[2] I'm not altogether happy with this implementation, so this may change, but I want to keep things relatively simple for now.

are not (currently?) tagged independently. So, if you specify `alt text`, you replace not only the `content` of the node in the corresponding tag, but the content of any `edge label` and any relevant structural information. So if you want, say, a branch number prepended or an indication that the node is a '`child`' or '`leaf`', say, or that the tree forks from this node, you must include that information into the ⟨*tokens*⟩ when specifying `alt text`.

`is root` (*auto. toks reg.*) = ⟨*tokens*⟩

Specify text to insert when describing the root. Default is `root`.

`is child` (*auto. toks reg.*) = ⟨*tokens*⟩

Specify text to insert when describing a child. Default is `child`.

`is leaf` (*auto. toks reg.*) = ⟨*tokens*⟩

Specify text to insert when describing a leaf node. Default is `end branch`.

`is edge label` (*auto. toks reg.*) = ⟨*tokens*⟩

Specify text to insert when describing an edge label. Default is `edge label`.

`has branches` (*auto. toks reg.*) = ⟨*tokens*⟩

Specify text to insert when describing a parent's branches. Default is `branches`. A number is inserted before to indicate the number of branches.

`is branch` (*auto. toks reg.*) = ⟨*tokens*⟩

Specify text to insert when describing node's (and, hence, this subtree's) position in the tree. Default is `branch`. A number is appended to indicate which branch.

2.0.1 Customisation

Most users will not need the options explained in this section.

`tagging` (*bool. reg.*)

`tagging` may be used to make code conditional on the activation status of tagging. For this reason, it has a public name. However, it should **NOT** be changed.

More generally, you should not suspend, resume, enable or disable tagging inside a `forest` environment unless you understand what you are doing with respect to *both* the tagging code *and* forest[3].

`tag nodes uses` (*choice*) = `none|alt text`

Configures the keylist `tag nodes`. `alt text` installs the default auto-generation code which constructs a value if `alt text` is unspecified for a (non-phantom) node.

The order in which nodes are tagged may be set using `tag nodes processing order`. The default is `unique=tree`.

`collate tags uses` (*choice*) = `none|alt text`

Configures the keylist `collate tags`. `alt text` installs code to collate the values of the autowrapped toks option `alt text`.

The order of collation may be set using `collate tags processing order`. The default is `unique=tree depth first`.

`tag tree uses` (*choice*) = `none|alt text`

Configures the style `tag tree`. `alt text` installs the default keys used to calculate approximate dimensions for the bounding box and to pass the collated tags to the *plug* responsible for tagging the tree.

This style is used by the default implementation of `tag tree stage`:

[3]Possibly nobody currently meets both of these requirements.

```
  tag tree stage/.style={for root'=tag tree},
```

2.0.2 Custom plugs

By default, everything is `noop`. If the user does nothing and tagging is active, the `alt` plug is used. If this is not desired, it is sufficient to use , which will make everything (remain) `noop` or , which will allow the latex-lab patches to mix explosively with your forest trees. This is not recommended unless you plan to prevent such encounters yourself. In the worst cases, the combination will result in fatal compilation errors. In the best cases, the document will compile, but tagging will almost certainly be broken.

However, it is possible to strike a middle course and use the infrastructure provided by this library as the basis for custom tagging. Some approaches were explained in section 2.0.1. If those are not sufficient, you may define custom plugs. This section explains the minimal requirements for such plugs to be used by this library i.e. without using `custom tagging`.

Requirements Let `Percy` be the name of your custom plug. Then ext.tagging requires:

1. a plug named `Percy` for socket `tagsupport/forest/setup`;
2. a plug named `Percy` for socket `tagsupport/forest/tag`.

If both conditions are satisfied, writing

```
  \forestset{%
    plug=Percy,
  }
```

will not result in an immediate error.

In order to do something useful, of course, `Percy` must do rather more than this, so let's see what `alt` is used. `tagsupport/forest/setup` is used right at the start of the tree. This happens before any parenthetical argument is processed, before any star is used, before the default preamble and well before any tree-specific preamble[4]. In particular, the default values of *tagging keylists* may still be manipulated at this point, since the socket is used before they are transformed into regular keylist options. The `alt` plug exploits this using the following code:

```
\socket_new_plug:nnn {tagsupport/forest/setup}{alt}
{
  \forestset{
    plug=alt,
    tag nodes uses=alt text,
    collate tags uses=alt text,
    tag tree uses=alt,
  }
}
```

Note that it is good practice to set `plug` here, even if the code is already plug-specific, since the value is used later when calling the `tagsupport/forest/tag` socket. The content of the `alt` `tagsupport/forest/tag` plug is very similar to the latex-lab patch for .

So let's assume that `Percy` should use the same code as the `alt` plug for the `tagsupport/forest/tag` socket, but something different for `tagsupport/forest/setup`.

[4]It uses a generic hook to inject code before an internal macro. This ensures it works for both the environment and command forms without adding an additional TEX group, but is clearly not ideal.

As noted above, `tag nodes uses`, `collate tags uses` and `tag tree uses` are choice keys. Given the way pgfkeys implements such keys, `Percy` might do something like this:

```
\NewSocketPlug {tagsupport/forest/setup}{percy}
{%
  \forestset{%
    plug=percy,
    tag nodes uses=percy,
    collate tags uses=percy,
    tag tree uses=alt,
  }%
}
\forestset{%
  declare autowrapped toks={percy text}{},
  tag nodes uses/percy/.style={%
    redeclare tagging keylist={tag nodes}{%
      if percy text={}{%
        percy text/.option=content,
        +percy text={Percy: },
      }{%
        percy text+={: },
        percy text+/.option=content,
      },
    },
  },
  collate tags uses/percy/.style={%
    redeclare tagging keylist={collate tags}{%
      collate tag/.option=percy text,
    },
  },
}
\NewSocketPlug {tagsupport/forest/tag}{percy}
{%
  \AssignSocketPlug {tagsupport/forest/tag}{alt}%
  \UseSocket {tagsupport/forest/tag}%
}
```

This would result in each node in the tree contributing both its content and a prefix specified by option `percy text` to the alternative text provided in the tagging structure of the PDF. No structural information is added here i.e. there are no descriptions of branching or of the relationships between nodes[5].

2.0.3 Complete control

`custom tagging` (*code key*) `=true|false`

`not custom tagging` (*code key*)

If true, do not tag following trees in the current TEX group.

This key must be used BEFORE `\begin{forest}` or `\Forest`.

If you do not want to use the library's tagging code, you can easily avoid it by simply not using it. However, you might want to use it for only some trees or you might wish to use the pre-defined stages as a basis for a custom configuration. In such cases, `custom tagging` may be used to tell the library that it should not tag trees in the local TEX group even if tagging is active. In this

[5] For a more realistic implementation, see ?? for the code used for the `alt` plug. For a more elaborate example of customisation, see Rees (2026).

case, the user (or another package) is entirely responsible for tagging. The custom tagging code may nonetheless test `tagging` and use the additional stages, if desired. For example, it could redefine the stages which generate and concatenate the tags or it could install alternative plugs into appropriate sockets.

Note that latex-lab's code is still active in this scenario, so you are responsible for dealing with the patches it applies for `tikzpicture` environments. Note also that `custom tagging` is *not* a boolean register or option — it is simply designed to emulate one. It in fact uses the `.code` handler to set an expl3 boolean variable.

The default `alt` plug is implemented in modular fashion, so it is possible, with care, to take a pick-'n-mix approach.

2.1 Workflow

ext.tagging redefines forest's `stages`. If you just wish to use the library to tag ordinary trees, you can ignore the details of this definition. However, should you wish to use the library with a custom definition of `stages`, the details below should enable you to do so. As with `forest`'s own definitions, the various steps may be redefined, replaced, removed or extended as required. The library also follows the forest package's convention in providing `before` keylists reserved for user use i.e. all such keylists are empty by default.

Tagging is initialised and finalised by code added to the hooks `env/forest/begin` and `env/forest/end`.

```
stages/.style={
  for root'={
    process keylist register=default preamble,
    process keylist register=preamble
  },
  process keylist=given options,
  process keylist=before typesetting nodes,
  typeset nodes stage,
  process keylist=before packing,
  pack stage,
  process keylist=before computing xy,
  compute xy stage,
  process keylist=before tagging nodes,
  process keylist=tag nodes,
  process keylist=before collating tags,
  process keylist=collate tags,
  process keylist=before tagging tree,
  tag tree stage,
  process keylist=before drawing tree,
  draw tree stage
},
```

This describes the default implementation with `setup plug=alt` and `tag plug=alt`[6].

1. `default preamble` (see Živanović 2017)
2. `preamble` (see Živanović 2017)

[6]Strictly speaking, the non-trivial claims in items 10 to 15 are almost entirely false as stated. For example, `tag nodes` could construct an entirely new branch and put all the tagging information there, `collate tags` could then collect that information and write it to an external file and `tag tree stage` could embed or attach that file. But that is not very useful to know. The proof of this is simple: if such radical divergence features in your tagging plans, you do not need this package, while, if you do, it shouldn't. QED. It follows that you should skip this footnote.

3. `given options` (see Živanović 2017)
4. `before typesetting nodes` (see Živanović 2017)
5. `typeset nodes stage` (see Živanović 2017)
6. `before packing` (see Živanović 2017)
7. `pack stage` (see Živanović 2017)
8. `before computing xy` (see Živanović 2017)
9. `compute xy stage` (see Živanović 2017)
10. `before tagging nodes` Empty by default. Use in the same way as forest's `before` keylists.
11. `tag nodes` A *tagging keylist* which should, when processed, ensure that each node which requires tagging is correctly tagged in whichever way the installed `tag plug` and associated code requires e.g. for the default `alt` configuration, `alt text`.
12. `before collating tags` Empty by default. Use in the same way as forest's `before` keylists.
13. `collate tags` A *tagging keylist* which should, when processed, result in the collation of all tags for the tree in the form expected by `tag tree stage`.
14. `before tagging tree` Empty by default. Use in the same way as forest's `before` keylists.
15. `tag tree stage` Executes code to actually tag the tree using the data finalised in `collate tags` (possibly modified by `before tagging tree`).
16. `before drawing tree` (see Živanović 2017)
17. `draw tree stage` (see Živanović 2017)

2.2 Example

Here is a complete example[7]:

```
\DocumentMetadata{
 tagging=on,
 lang=en-GB,
 pdfversion=2.0,
 pdfstandard=ua-2,
}
\tagpdfsetup{
  math/mathml/structelem,
}
\documentclass{article}
\usepackage[ext.tagging]{forest}
\ifcsname directlua\endcsname
  \usepackage{unicode-math}
\else
  \usepackage[T1]{fontenc}
\fi
\title{This Test Needs No Title}
\begin{document}
  ABC apple banana pear
  \begin{forest}
    % This example is from Jasper Habicht.
    [VP
      [DP[John]]
      [V', alt text=V prime,
        [V[sent]]
        [DP[Mary]]
        [DP[D[a]][NP[letter]]]
      ]
    ]
  \end{forest}
  ABC apple banana pear
}
\end{document}
```

Note the use of `alt text` to avoid problems due to the use of `'` with PDFTEX. If the (LATEX Project recommended) engine LuaLATEX is used, you need not be quite so careful, but you should always check the content of the `alt` text for unpleasant surprises.

If compiled with pdfLATEX, the above example yields the following structure:

```
<PDF>
 <StructTreeRoot>
  <Document xmlns="http://iso.org/pdf2/ssn"
     id="ID.02"
    >
   <text-unit xmlns="https://www.latex-project.org/ns/dflt"
      id="ID.05"
```

[7]Note that the recommended syntax for invoking and using tagging support in LATEX 2ε changes very frequently. In particular, the recommended options for `\DocumentMetadata` and `\tagpdfsetup`, including whether to use the latter at all, are not at all stable. You should therefore check and use the recommended options at the time your document is written — there is nothing in the code before `\documentclass` which is in any way particular to using the libraries provided by this package. That is, deviations from documented best practice in the use of `\DocumentMetadata` and `\tagpdfsetup` are either due to mistakes on my part or the result of updates following the publication of this document. In either case, you should avoid replicating the deviations in your own code.

```
      rolemaps-to="Part"
     >
    <text xmlns="https://www.latex-project.org/ns/dflt"
       id="ID.06"
       xmlns:Layout="http://iso.org/pdf/ssn/Layout"
       Layout:TextAlign="Justify"
       rolemaps-to="P"
      >
     <?MarkedContent page="1" ?>ABC apple banana pear
     <Figure xmlns="http://iso.org/pdf2/ssn"
        id="ID.07"
        alt="root VP 2 branches   branch 1 DP  child John end branch  V prime V  child
        ↪  sent end branch  DP  child Mary end branch  DP 2 branches   branch 1 D  child
        ↪  a end branch   branch 2 NP  child letter end branch  "
        xmlns:Layout="http://iso.org/pdf/ssn/Layout"
        Layout:BBox="{ 259.4641, 542.90266, 407.35223, 667.19801 }"
       >
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
      <?MarkedContent page="1" ?>
     </Figure>
     <?MarkedContent page="1" ?> ABC apple banana pear
    </text>
   </text-unit>
  </Document>
 </StructTreeRoot>
</PDF>
```

A similar result is obtained with LuaLaTeX, but the output is a bit longer as it includes many empty `MarkedContent`s.

3 Multiple parents

This library provides some basic facilities for formatting trees which are not technically trees in forest's sense. In the (one) strict sense of 'tree', every node but one has exactly one parent, while the one has none.

However, in a different/looser sense of 'tree', every node but one has at least one parent, while the one has none. This library makes it a bit easier to draw such trees with forest.

The library began in response to a query from Alan Munn on TEX SE and initially focused entirely on *multi-dominance* structures in linguistics. Support for those structures is available in the ext.ling library, which now uses the more general ext.multi.

The styles in section 3.1 support drawing connections from a child to additional parents not currently in the tree, while those in section 3.2 support adding connections to additional extant parents.

Note that

- styles are always specified for the child node;
- the child must have exactly one 'natural' parent i.e. it must be part of the existing tree structure when the style is used.

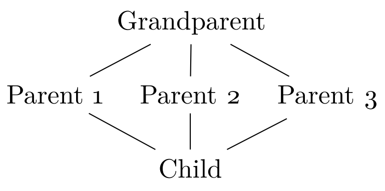
Load `ext.multi` or `ext.multi-debug` as described in section 1.

3.1 Creating multiple parents

Note:

- the child should be created as the *child* of its ultimate grandparent;
- the child's parents will all be children of the child's grandparent.

For example, consider the tree,



This structure can be conveniently created using `multi`, but to translate it into the bracket notation forest uses, all of Child's parents should first be omitted and Child should instead be specified as the child of Grandparent.

```
\begin{forest}
  [Grandparent [Child]]
\end{forest}
```

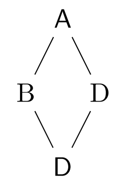
Parents 1, 2 and 3 should be specified as an option to Child:

```
\begin{forest}
  [Grandparent [Child, multi={Parent 1,Parent 2,Parent 3}]]
\end{forest}
```

`multi` (*style*) `={`⟨*content of parent 1, . . . , content of parent n*⟩`}` where $n \in \mathbb{N}, n > 1$

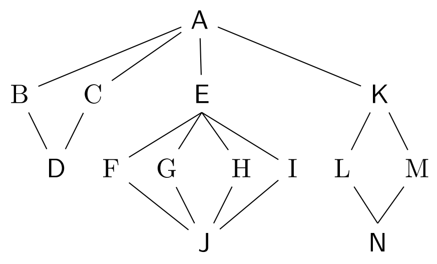
For every $i \in \mathbb{N}$ such that $0 < i \leq n$, create a new child of the current node's parent with content ⟨*content of parent i*⟩. Then detach the current node from its parent and attach it as the child of its n parents.

```
\begin{forest}
 [A
  [D,
   multi={B,D}
  ]
 ]
\end{forest}
```



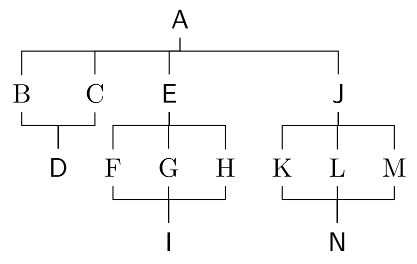
If `parent anchor` and/or `child anchor` are set, edges are drawn to/from these points as one would expect.

```
\begin{forest}
 [A
  [D, multi={B,C}, ]
  [E, parent anchor=children,
   [J, multi={F,G,H,I}, ]
  ]
  [K
   [N, multi={L,M}, child
   ↪  anchor=parent, ]
  ]
 ]
\end{forest}
```



If the edges library is loaded, the multi library loads the Ti*k*Z library, ext.paths.ortho and tries to emulate `forked edge` appropriately[8].

```
\begin{forest}
 forked edges,
 [A
  [D, multi={B,C} ]
  [E
   [I, multi={F,G,H} ]
  ]
  [J
   [N, multi={K,L,M} ]
  ]
 ]
\end{forest}
```

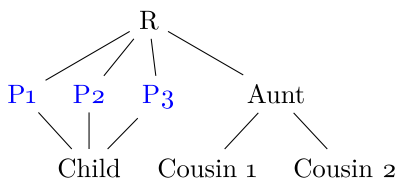


If we apply `forked edges` to only part of a tree, we can produce the rather ugly, but hopefully informative, structure below.

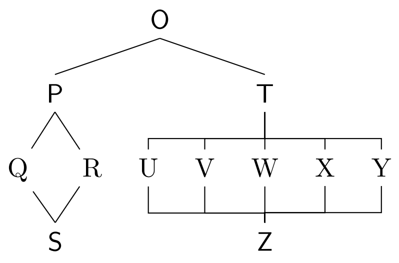
[8]The alignment seems to me to be close, but not always quite perfect, though I do not know why at the moment.

Box 3.5

```
\begin{forest}
  [R [Child, multi={P1,P2,P3},every parent=blue,]
  ↪   [Aunt [Cousin 1][Cousin 2]]]
\end{forest}
```



```
\begin{forest}
  for tree={%
    child anchor=parent,
    parent anchor=children,
    fork sep'=1em,
  },
  [O
    [P
      [S, multi={Q,R} ]
    ]
    [T, forked edges=descendants,
      [Z,multi={U,V,W,X,Y} ]
    ]
  ]
\end{forest}
```



Note that the change to `fork sep` for the tree in `forest`'s preamble affects the edges drawn from and to the nodes inserted by `multi`. This is because the library forwards values given to `fork sep` and applications of `forked edge` so that forest keys work in (hopefully) reasonably intuitive ways.

Should you *not* want such keys forwarded, either load the library without defaults (see section 1) or override the behaviour for the current TEX group with, say,

```
\forestset{%
  unautoforward=fork sep,
  null/.style={},
  forked edge'/.forward to=/forest/null,
}
```

The `phantom style` is needed because, unlike forest's provision for its own forwarding facilities, pgfkeys provides no easy way to undo the effects of the `.forward to` handler.

Since the library is currently experimental and implementation is complicated if one wants to avoid avoid using forest internals, configuration options are currently limited.

`every parent` (*keylist*) `=`{⟨*key-value list*⟩}

Apply ⟨*key-value list*⟩ to all the current node's parents. If `multi` is used, these are the parents created as a result; otherwise, it is the current node's singular parent or none, if the node has no parent.

Initial value: empty.

Box 3.5 illustrates usage with a simple example.

3.2 Connecting multiple parents

Sometimes one wants instead to give the current node an additional parent without removing the existing one and one does not wish to add the additional parent, but rather to specify some other extant node in the tree.

This kind of structure cannot be so easily automated, especially if one wants to avoid edges crossing each other or nodes. However, it is possible to provide some convenient styles to assist in manually specifying such structures.

`also parent` (*style*) = `{`⟨*dynamic tree operation*⟩`}{`⟨⟨*extant node*⟩:⟨*keylist*⟩⟩`}`

= `{`⟨*dynamic tree operation*⟩`}{`⟨*extant node*⟩`}`

`+also parent` (*style*) = `{`⟨⟨*extant node*⟩:⟨*keylist*⟩⟩`}`

= `{`⟨*extant node*⟩`}`

`also parent+` (*style*) = `{`⟨⟨*extant node*⟩:⟨*keylist*⟩⟩`}`

= `{`⟨*extant node*⟩`}`

Adds ⟨*extant node*⟩ as an additional parent of the current node. ⟨*keylist*⟩ specifies a list of key-values for the connecting node (see below).

The current node becomes ⟨*extant node*⟩'s *fosterling*, while ⟨*extant node*⟩ becomes the current node's *foster parent*.

The styles work by creating a new child of ⟨*extant node*⟩. This node affects the structure of the tree and can be configured in the usual way, but it is not visible. One might say it is 'semi-phantom': it is not quite `phantom` because, for instance, it has visible edges which serve to connect the current node with the additional parent.

For an illustration, see the (rather odd-looking) family tree in box 3.6[9].

`+also parent` prepends the new child to ⟨*extant node*⟩; `also parent+` appends it. These are just shorthand wrappers around `also parent` using the `prepend` and `append` dynamic tree operations.

Note that ⟨*dynamic tree operation*⟩ should create a new node, though this is not enforced.

`fosterlings` (*step*) Visit the current node's *fosterlings*.

`foster parents` (*step*) Visit the current node's *foster parents*.

`every fosterling` (*step*) = `{`⟨*nodewalk*⟩`}`

Visit every *fosterling* in ⟨*nodewalk*⟩.

`every foster parent` (*step*) = `{`⟨*nodewalk*⟩`}`

Visit every *foster parent* in ⟨*nodewalk*⟩.

`c fosterling` (*step*) Visit the *fosterling* which the current node connects to a *foster parent*.

`c foster parent` (*step*) Visit the *foster parent* which the current node connects to a *fosterling*.

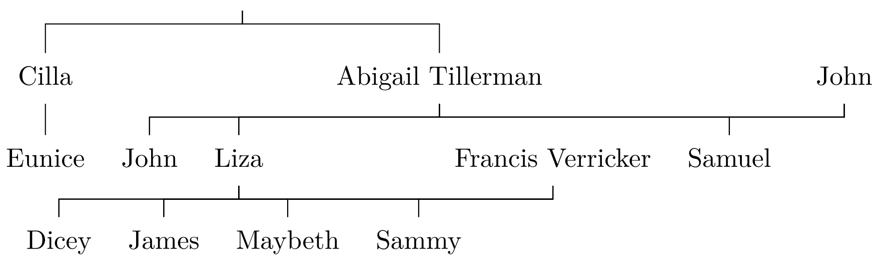
This last pair of steps are only really useful if you want to change `edge path`, since they are only accessible from a constructed, typically invisible node.

`debug multi phantoms` (*bool. reg.*) = `true|false`

`not debug multi phantoms` (*bool. reg.*) Render the normally invisible nodes created by `also parent` etc. visible for debugging purposes. If the nodes have no content, their borders are drawn in red; otherwise, their contents are rendered in red. Visible rendering does not change the remainder of the tree e.g. it does not alter the spacing of nodes or the paths of edges. However, if the nodes occur near the tree's boundaries, the bounding box may expand to accommodate them[10].

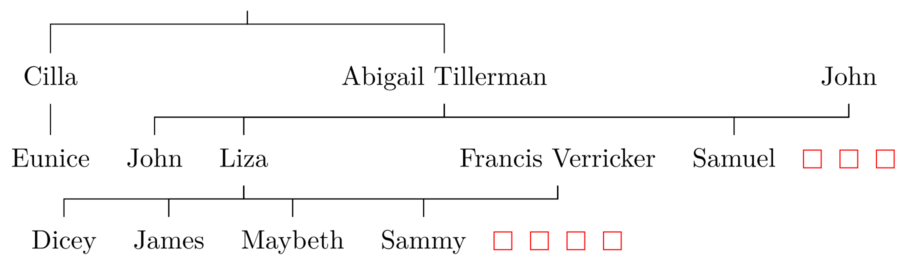
[9] The names are from the children's novels by Cynthia Voigt.
[10] It should not be hard to prevent this, but does not seem worth the trouble.

Box 3.6



```
\begin{forest}
  forked edges,
  delay={%
    for tree={%
      +content=\strut,
    },
  },
  [,coordinate,calign primary child=1,calign secondary child=2,calign=midpoint,
    [Cilla
      [Eunice]
    ]
    [Abigail Tillerman
      [John,also parent={append}{j}]
      [Liza, also parent={append}{!r3}, for children={also parent={append}{!un}}
        [Dicey]
        [James]
        [Maybeth]
        [Sammy]
      ]
      [Francis Verricker,no edge]
      [Samuel, also parent+={!r3}]
    ]
    [John, name=j, no edge
    ]
  ]
\end{forest}
```

Box 3.7



```
{%
 \forestset{debug multi phantoms}%
 \begin{forest}
  forked edges,
  delay={%
   for tree={%
    +content=\strut,
   },
  },
  [,coordinate,calign primary child=1,calign secondary child=2,calign=midpoint,
   [Cilla
    [Eunice]
   ]
   [Abigail Tillerman
    [John,also parent={append}{j}]
    [Liza, also parent={append}{!r3}, for children={also parent={append}{!un}}
     [Dicey]
     [James]
     [Maybeth]
     [Sammy]
    ]
    [Francis Verricker,no edge]
    [Samuel, also parent+={!r3}]
   ]
   [John, name=j, no edge
   ]
  ]
 \end{forest}%
}
```

Requires ext.multi-debug. If the debugging code is not loaded, use of these keys will do nothing but write a warning to the console and log.

For an example, see box 3.7. Note that the content of the `forest` environment is identical to that in box 3.6. The red squares are the effect of toggling `debug multi phantoms` beforehand.

4 Linguistics extensions

This library provides some elementary styles for formatting trees involving *multi-dominance*, together with a style for dealing with empty nodes resistant to the linguistics library's `nice empty nodes`. These former were developed in response to a query from Alan Munn on TEX SE.

See also section 3, especially for straight connections to multiple parents and dynamic creation of multiple parents as children of a single grandparent.

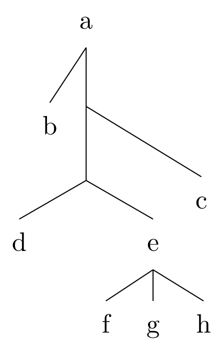
`pretty nice empty nodes` = {⟨*keylist*⟩}
(*style*)

Make empty nodes prettier in cases where `nice empty nodes` cannot be used. ⟨*keylist*⟩ permits supplementing or overriding what is done for empty nodes.

Note that `nice empty nodes` is preferable, so should be used where possible. For details, see the documentation of `nice empty nodes` in Živanović (2017).

For example[11],

```
\begin{forest}
  for tree={
    calign angle=60,
    align middle child,
  },
  pretty nice empty nodes={
    for current and
    ↪  siblings={anchor=parent},
    parent anchor=children,
    calign with current edge,
  },
  [a
    [b]
    [
      [
        [d]
        [e
          [f]
          [g]
          [h]
        ]
      ]
      [c]
    ]
  ]
\end{forest}
```



5 Utilities

This library provides *tagging keylists*, together with a few styles which do not really fit anywhere else.

5.1 Alignment

`align middle child` (*style*) = ⟨*option*⟩

If the current node has an odd number of children, sets `calign child` to the middle child and sets `calign=`⟨*option*⟩. ⟨*option*⟩ should, therefore, be a valid value for `calign`.

If ⟨*option*⟩ is omitted, a default of `child edge` is applied.

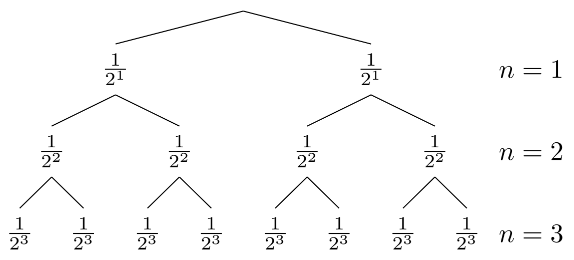
See box 4.1 for an example.

`align middle children` (*style*) = ⟨*option*⟩

Sets `align middle child=`⟨*option*⟩ for the tree.

[11]Based on TEX SE answer: 717677. Based on TEX SE question 717592 by argo.

Box 5.1



```
\begin{forest}
 for tree={
  parent anchor=children,
  child anchor=parent,
 },
 delay={
  for descendants={
   content/.process={Ow{level}{$\frac{1}{2^{#1}}$}},
  },
  for nodewalk={
   fake=root,
   while nodewalk valid={1}{1}%
  }{
   outer label/.process={Ow{level}{{$n=#1$}:{anchor=west}}}%
  },
 },
 [ [ [ [] [] ] [ [] [] ] ] [ [ [] [] ] [ [] [] ] ] ]
\end{forest}
```

5.2 Outer labels

Outer labels are nodes added after the tree is drawn, aligned with a boundary of the bounding box of the completed tree and nodes within the tree. The idea is to enable the addition of labels such as those shown in box 5.1.

`outer labels at` (*toks reg.*) = ⟨*anchor*⟩

Additional alignment point for any outer labels. ⟨*anchor*⟩ should be a valid anchor for the 'current bounding box' when the tree has been drawn, but additional code is not yet executed.

The default is `east`, which is probably what is wanted for most trees using the forest default value of `grow` etc.

Note that this is a *register*. You cannot use different values for different parts of a tree.

`outer labels` (*keylist reg.*) = `{`⟨*keylist*⟩`}`

PGF/Ti*k*Z key-values applied to all nodes where `outer label` is set. Options passed to `outer label` are applied later, so may override defaults for the tree.

The default is `anchor=base west`.

Note that this is a *register*. You cannot use different values for different parts of a tree.

`outer label` (*style*) = `{`⟨*content*⟩`}`

= `{`⟨*content*⟩`}:{`⟨*options*⟩`}`

Create a label aligned with the current node and the additional alignment point specified by `outer labels at` with content ⟨*content*⟩. If ⟨*options*⟩ are given, they are passed to the code responsible for creating the node.

5.3 ‘Tagging’ keylists

A ‘tagging keylist’ is very similar to a forest *keylist option*, but its default value can be changed and/or it can be redeclared[12]. For motivation, see section 2.

More specifically, *inside* a `forest` environment, it behaves exactly like a regular forest keylist option[13]. However, *outside* a `forest` environment, its default value can be modified and/or replaced. Where this is not a requirement, you should use a regular *keylist* option since *tagging keylists* are subject to additional limitations and the implementation is significantly less efficient.

Important:

1. These keys are not really tagging-specific and do not require tagging to be active, despite the names, so may be useful in other contexts.
2. These keys are only available *outside* `forest` environments.
3. *Tagging keylists* cannot be declared as registers[14]. Each tagging keylist corresponds to a *keylist option*. The option is automatically declared just before every `forest` environment in the current TEX group.
4. An additional TEX group is added to all `forest` environments. This ensures that the option declaration is properly localised, which in turn allows any tagging keylists’ default values to be further manipulated after the current `forest` is finished.
5. *Outside* `forest` environments, unlike forest keylists, tagging keylists are *not* ordered and do *not* store more than one instance of any key. The underlying implementation uses l3prop property lists.

   Inside `forest` environments, tagging keylists *are* ordered and behave as regular forest keylist options. l3prop property lists are *not* used inside `forest` environments.
6. *Outside the `forest` environment, they may be manipulated **only** using the keys defined by this library.*
7. *Inside the `forest` environment, they may be manipulated **only** using regular forest methods.*

Note that to actually influence a tree, any tagging keylist must be processed during the construction of that tree. Simply declaring a tagging keylist with some set of options will not, in itself, affect the typeset result in anyway. This is equally true of regular forest keylists. Please see Živanović (2017) for details.

`declare tagging keylist`,`redeclare tagging keylist` (*code key*) = `{`⟨*keylist*⟩`}{`⟨*key-value list*⟩`}`

Declares or redeclares a forest *keylist option.*

Available only outside `forest` environments.

Since keylists cannot actually be redeclared, what really happens is this:

- An internal property list is defined to hold ⟨*default*⟩. This may then be manipulated using the various keys explained below.
- At the start of each `forest` environment (within the current TEX group), a keylist option is declared. The default value passed to `declare keylist` is *not* necessarily ⟨*key-value list*⟩. It is, rather, a key-value list derived from the contents of the underlying property list at the time. Hence, the default may be further manipulated after the keylist option is declared.

[12]As far as I can tell, this is not possible for regular forest keylist options. Once declared, their default values are fixed.
[13]This is because it *is* a regular keylist option at this point.
[14]This is not a limitation since changing the default value of a *keylist register* is trivial.

Note that if you do not want the default be be manipulable after the keylist is declared, you should use the forest key `declare keylist=`{⟨*keylist*⟩}{⟨*key-value list*⟩} instead, as this will be far more efficient.

`tagging keylist put` (*code key*) = {⟨*keylist*⟩}{⟨*key-value list*⟩}

Adds the contents of ⟨*key-value list*⟩ to a ⟨*keylist*⟩ declared with `declare tagging keylist`.

Note that if ⟨*key-value list*⟩ includes an occurrence of a key already in the list, the key will be replaced, even if the value differs.

`tagging keylist remove key` (*code key*) = {⟨*keylist*⟩}{⟨*key*⟩}

Removes ⟨*key*⟩ from ⟨*keylist*⟩, where ⟨*keylist*⟩ was previously declared with `declare tagging keylist`.

Available only outside `forest` environments.

Note this removes the ⟨*key*⟩ regardless of its current value (if any).

`tagging keylist remove` (*code key*) = {⟨*keylist*⟩}{⟨*key-value list*⟩}

For each ⟨*key*⟩ or ⟨*key*⟩=⟨*value*⟩ pair in ⟨*key-value list*⟩, removes ⟨*key*⟩ from ⟨*keylist*⟩ iff it has the specified ⟨*value*⟩ (if given) or no value (otherwise), where ⟨*keylist*⟩ was previously declared with `declare tagging keylist`.

Available only outside `forest` environments.

Note that a valueless key is distinct from one with an empty value. To remove ⟨*key*⟩ iff it has no value, use ⟨`key`⟩. To remove ⟨*key*⟩ iff it's value is empty, use ⟨`key`⟩`=` or ⟨`key`⟩`=`.

References

Fischer, Ulrike (2025). *The tagpdf Package*. v0.99w. 31st Oct. 2025. CTAN: tagpdf.

International Organization for Standardization (2025). *Document management applications — Electronic document file format enhancement for accessibility —Part 2: Use of ISO 32000-2 (PDF/UA-2)*. 5th Apr. 2025.

PDF Association (2024a). *ISO 32000-2:2020 (PDF 2.0) including Errata Collection 2*. 24th Sept. 2024.

— (2024b). *Well-Tagged PDF (WTPDF) Using Tagged PDF for Accessibility and Reuse in PDF 2.0*. 28th Feb. 2024.

LaTeX Project (2025). *The latex-lab-tikz Package: Support for the Tagging of TikZ Pictures*. v0.80d. 27th Sept. 2025. CTAN: latex-lab.

Rees, Clea F. (2026). *prooftrees*. 0.9.2. 16th Jan. 2026. CTAN: prooftrees.

Živanović, Sašo (2017). *Forest: A PGF/TikZ-Based Package for Drawing Linguistic Trees*. 2.1.5. 14th July 2017. CTAN: forest.

Change History

v0.1
General: First public release. 1

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.