FOREST: a PGF/Ti*k*Z-based package for drawing linguistic trees

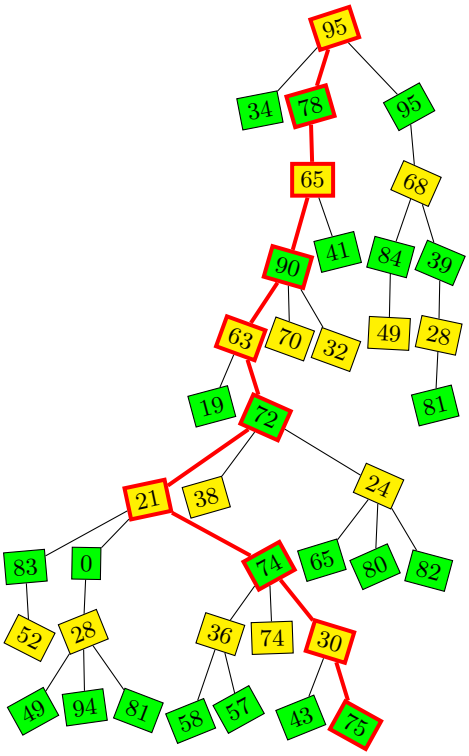
v2.1

Sašo Živanović*

December 5, 2016

Abstract

FOREST is a PGF/Ti*k*Z-based package for drawing linguistic (and other kinds of) trees. Its main features are (i) a packing algorithm which can produce very compact trees; (ii) a user-friendly interface consisting of the familiar bracket encoding of trees plus the key–value interface to option-setting; (iii) many tree-formatting options, with control over option values of individual nodes and mechanisms for their manipulation; (iv) a powerful mechanism for travesing the tree; (v) the possibility to decorate the tree using the full power of PGF/Ti*k*Z; (vi) an externalization mechanism sensitive to code-changes.



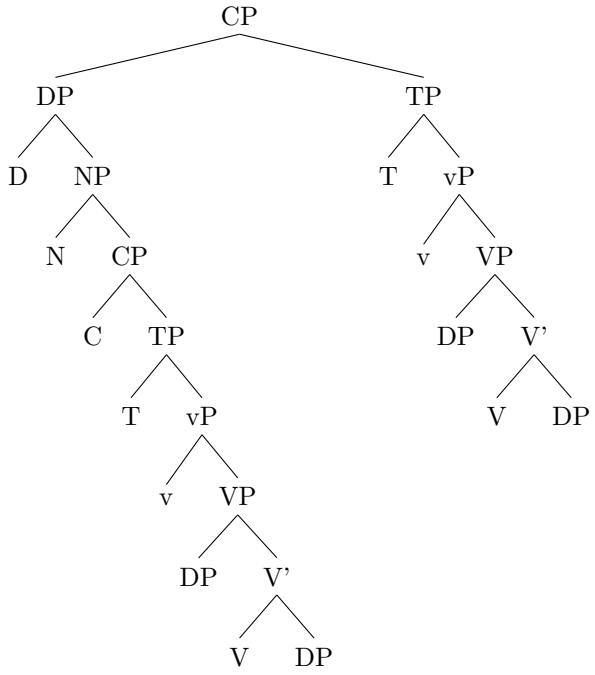

```
\pgfmathsetseed{14285}
\begin{forest}
  random tree/.style n args={3}{% #1 = max levels, #2 = max children, #3 = max content
    content/.pgfmath={random(0,#3)},
    if={#1>0}{repeat={random(0,#2)}{append={[,random tree={#1-1}{#2}{#3}]}}}{}},
  before typesetting nodes={for tree={draw,s sep=2pt,rotate={int(30*rand)},l+={5*rand},
                                      if={isodd(level())}{fill=green}{fill=yellow}}},
  important/.style={draw=red,line width=1.5pt,edge={red,line width=1.5pt}},
  before drawing tree={sort by=y, for nodewalk={min=tree,ancestors}{important,typeset node}}
  [,random tree={9}{3}{100}]
\end{forest}
```

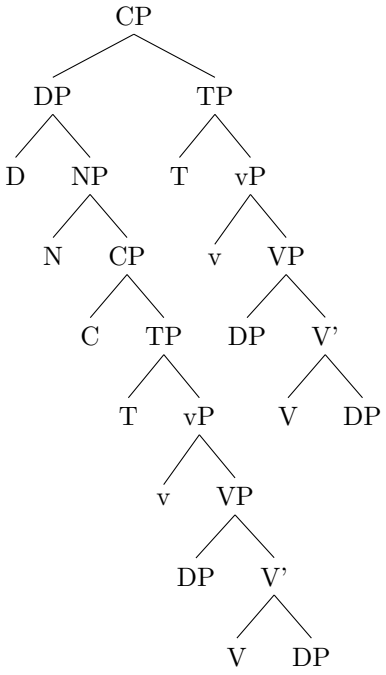
*e-mail: saso.zivanovic@guest.arnes.si; web: http://spj.ff.uni-lj.si/zivanovic/

Contents

1 Introduction

Over several years, I had been a grateful user of various packages for typesetting linguistic trees. My main experience was with `qtree` and `synttree`, but as far as I can tell, all of the tools on the market had the same problem: sometimes, the trees were just too wide. They looked something like the tree on the left, while I wanted something like the tree on the right.





Luckily, it was possible to tweak some parameters by hand to get a narrower tree, but as I quite dislike constant manual adjustments, I eventually started to develop Forest. It started out as `xyforest`, but lost the `xy` prefix as I became increasingly fond of pgf/Ti*k*Z, which offered not only a drawing package but also a 'programming paradigm.' It is due to the awesome power of the supplementary facilities of pgf/Ti*k*Z that Forest is now, I believe, the most flexible tree typesetting package for LaTeX you can get.

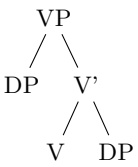
The latest stable version of Forest is available at CTAN. Development version(s) can be found at GitHub. Comments, criticism, suggestions and code are all very welcome! If you find the package useful, you can show your appreciation by making a PayPal donation to `saso.zivanovic@guest.arnes.si`.

2 Tutorial

This short tutorial progresses from basic through useful to obscure ... fortunately, it is not the only newcomer's source of information on Forest: check out Forest Quickstart Guide for Linguists. Another very useful source of information (and help!) about Forest and TeX in general is TeX StackExchange. Check out the questions tagged `forest`!

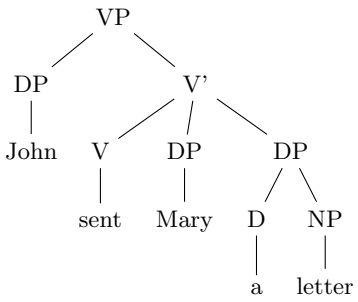
2.1 Basic usage

A tree is input by enclosing its specification in a `forest` environment. The tree is encoded by *the bracket syntax*: every node is enclosed in square brackets; the children of a node are given within its brackets, after its content.



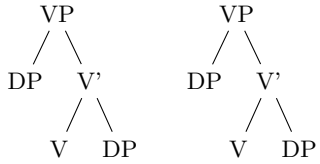

```
\begin{forest}
  [VP
    [DP]
    [V'
      [V]
      [DP]
    ]
  ]
\end{forest}
```
(1)

Binary trees are nice, but not the only thing this package can draw. Note that by default, the children are vertically centered with respect to their parent, i.e. the parent is vertically aligned with the midpoint between the first and the last child.



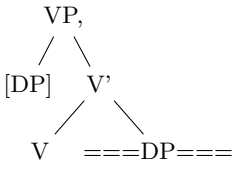

```
\begin{forest}                                                    (2)
  [VP
    [DP[John]]
    [V'
      [V[sent]]
      [DP[Mary]]
      [DP[D[a]][NP[letter]]]
    ]
  ]
\end{forest}
```

Spaces around brackets are ignored — format your code as you desire!



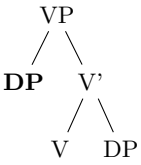

```
\begin{forest}                                                    (3)
  [VP[DP][V'[V][DP]]]
\end{forest}
\quad
\begin{forest}[VP
  [DP ] [ V'[V][ DP]]
  ]\end{forest}
```

If you need a square bracket as part of a node's content, use braces. The same is true for the other characters which have a special meaning in the Forest package, like comma , and equality sign =.



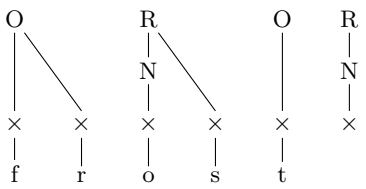

```
\begin{forest}                                                    (4)
  [V{P,}
    [{[DP]}]
    [V'
      [V]
      [{===DP===}]]]
\end{forest}
```

Macros in a node specification will be expanded when the node is drawn — you can freely use formatting commands inside nodes!




```
\begin{forest}                                                    (5)
  [VP
    [{\textbf{DP}}]
    [V'
      [V]
      [DP]]]
\end{forest}
```

All the examples given above produced top-down trees with centered children. The other sections of this manual explain how various properties of a tree can be changed, making it possible to typeset radically different-looking trees. However, you don't have to learn everything about this package to profit from its power. Using styles, you can draw predefined types of trees with ease. For example, a phonologist can use the GP1 style from library linguistics to easily typeset (Government Phonology) phonological representations. The style is applied simply by writing its name before the first (opening) bracket of the tree.




```
\usepackage[linguistics]{forest}                                  (6)
% ...
\begin{forest} GP1 [
  [O[x[f]][x[r]]]
  [R[N[x[o]]][x[s]]]
  [O[x[t]]]
  [R[N[x]]]
]\end{forest}
```

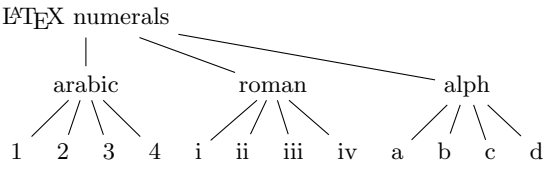
Of course, someone needs to develop the style — you, me, your local TEXnician ... Fortunately, designing styles is not very difficult once you get the hang of Forest, if you write one, please contribute! Some

macros relating to various fields are collected in *libraries* that are distributed alongside the main package. This is the case for the `GP1` style used above, which is defined in the `linguistics` library. The simplest way to load a library is as shown in the example, by loading the package with an optional argument. For more information on loading libraries, see §3.1.

2.2 Options

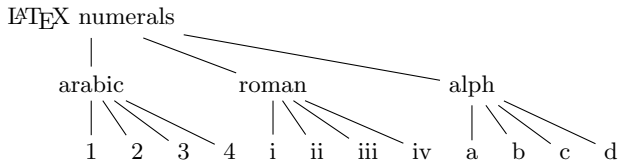
A node can be given various options, which control various properties of the node and the tree. For example, at the end of section 2.1, we have seen that the `GP1` style vertically aligns the parent with the first child. This is achieved by setting option `calign` (for *c*hild-*align*ment) to `first` (child).

Let's try. Options are given inside the brackets, following the content, but separated from it by a comma. (If multiple options are given, they are also separated by commas.) A single option assignment takes the form ⟨option name⟩=⟨option value⟩. (There are also options which do not require a value or have a default value: these are given simply as ⟨option name⟩.)




```
\begin{forest}                                          (7)
  [\LaTeX\ numerals, calign=first
    [arabic[1][2][3][4]]
    [roman[i][ii][iii][iv]]
    [alph[a][b][c][d]]
  ]
\end{forest}
```

The experiment has succeeded only partially. The root node's children are aligned as desired (so `calign=first` applied to the root node), but the value of the `calign` option didn't get automatically assigned to the root's children! *An option given at some node applies only to that node.* In FOREST, the options are passed to the node's relatives via special keys, called *propagators*. What we need above is the `for tree` propagator. Observe:

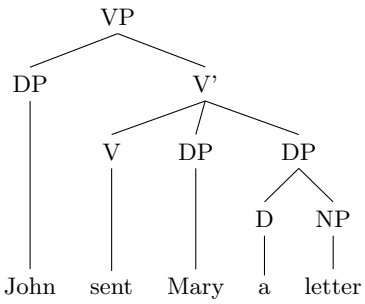



```
\begin{forest}                                          (8)
  [\LaTeX\ numerals,
      for tree={calign=first}
    [arabic[1][2][3][4]]
    [roman[i][ii][iii][iv]]
    [alph[a][b][c][d]]
  ]
\end{forest}
```

The value of propagator `for tree` is a list of keys that we want to process. This keylist is propagated to all the nodes in the subtree[1] rooted in the current node (i.e. the node where `for tree` was given), including the node itself. (Propagator `for descendants` is just like `for tree`, only that it excludes the node itself. There are many other `for` ⟨step⟩ propagators; for the complete list, see sections 3.5.1 and 3.8.)

Some other useful options are `parent anchor`, `child anchor` and `tier`. The `parent anchor` and `child anchor` options tell where the parent's and child's endpoint of the edge between them should be, respectively: usually, the value is either empty (meaning a smartly determined border point [see 2, §16.11]; this is the default) or a compass direction [see 2, §16.5.1]. (Note: the `parent anchor` determines where the edge from the child will arrive to this node, not where the node's edge to its parent will start!)

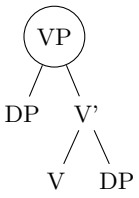
Option `tier` is what makes the skeletal points × in example (6) align horizontally although they occur at different levels in the logical structure of the tree. Using option `tier` is very simple: just set `tier=tier name` at all the nodes that you want to align horizontally. Any tier name will do, as long as the tier names of different tiers are different ... (Yes, you can have multiple tiers!)




```
\begin{forest}                                          (9)
  [VP, for tree={parent anchor=south, child anchor=north}
    [DP[John,tier=word]]
    [V'
      [V[sent,tier=word]]
      [DP[Mary,tier=word]]
      [DP[D[a,tier=word]][NP[letter,tier=word]]]
    ]
  ]
\end{forest}
```

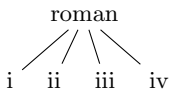
[1] It might be more precise to call `for tree` `for subtree` ... but this name at least saves some typing.

Before discussing the variety of Forest's options, it is worth mentioning that Forest's node accepts all options [2, see §16] that Ti*k*Z's node does — mostly, it just passes them on to Ti*k*Z. For example, you can easily encircle a node like this:[2]



(10)
```
\begin{forest}
  [VP,circle,draw
    [DP][V'[V][DP]]
  ]
\end{forest}
```

Let's have another look at example (6). You will note that the skeletal positions were input by typing `xs`, while the result looks like this: × (input as `\times` in math mode). Obviously, the content of the node can be changed. Even more, it can be manipulated: added to, doubled, boldened, emphasized, etc. We will demonstrate this by making example (8) a bit fancier: we'll write the input in the arabic numbers and have LaTeX convert it to the other formats. We'll start with the easiest case of roman numerals: to get them, we can use the (plain) TeX command `\romannumeral`. To change the content of the node, we use option `content`. When specifying its new value, we can use `#1` to insert the current content.[3]



(11)
```
\begin{forest}
  [roman, delay={for children={content=\romannumeral#1}}
    [1][2][3][4]
  ]
\end{forest}
```

This example introduces another option: `delay`. Without it, the example wouldn't work: we would get arabic numerals. This is so because of the order in which the options are processed. First, the processing proceeds through the tree in a depth-first, parent-first fashion (first the parent is processed, and then its children, recursively; but see `processing order`). Next, the option string of a node is processed linearly, in the order they were given. Option `content` is specified implicitely and is always the first. If a propagator is encountered, the options given as its value are propagated *immediately*. The net effect is that if the above example contained simply `roman,for children={content=...}`, the `content` option given there would be processed *before* the implicit content options given to the children (i.e. numbers `1`, `2`, `3` and `4`). Thus, there would be nothing for the `\romannumeral` to change — it would actually crash; more generally, the content assigned in such a way would get overridden by the implicit content. Key `delay` is true to its name. It delays the processing of the keylist given as its argument until the whole tree was processed. In other words, it introduces cyclical option processing. Whatever is delayed in one cycle, gets processed in the next one. The number of cycles is not limited — you can nest `delay`s as deep as you need.

Unlike `for` ⟨step⟩ keys we have met before, `delay` is not a spatial, but a temporal propagator. Several other temporal propagators options exist, see §3.4.1.

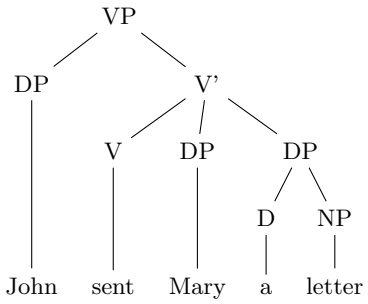
We are now ready to learn about simple conditionals.[4] Every node option has the corresponding `if` ⟨option⟩ and `where` ⟨option⟩ keys. `if` ⟨option⟩=⟨value⟩⟨true options⟩⟨false options⟩ checks whether the value of ⟨option⟩ equals ⟨value⟩. If so, ⟨true options⟩ are processed, otherwise ⟨false options⟩. The `where` ⟨option⟩ keys are the same, but do this for the every node in the subtree; informally speaking, `where` = `for tree` + `if`. To see this in action, consider the rewrite of the `tier` example (9) from above. We don't set the tiers manually, but rather put the terminal nodes (option `n children` is a read-only option containing the number of children) on tier `word`.[5]

[2] If option `draw` was not given, the shape of the node would still be circular, but the edge would not be drawn. For details, see [2, §16].

[3] This mechanism is called *wrapping*. By default, `content` is the only ⟨autowrapped toks⟩ option, i.e. option where wrapping works implicitely (simply because I assume that wrapping will be almost exclusively used with this option). To wrap values of other options, use handler `.wrap value`; see §3.12.

[4] See §3.9 for further information on conditionals, including the generic `if` and `where`.

[5] We could omit the braces around `0` because it is a single character. If we were hunting for nodes with 42 children, we'd have to write `where n children={42}...`.

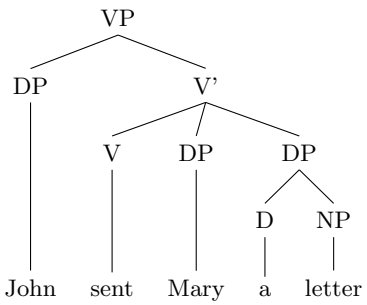



```
\begin{forest}                                                    (12)
  where n children=0{tier=word}{}
  [VP
    [DP[John]]
    [V'
      [V[sent]]
      [DP[Mary]]
      [DP[D[a]][NP[letter]]]
    ]
  ]
\end{forest}
```

→ Note that you usually don't want to embed a `where ...` conditional in a `for tree`, as this will lead to a multiple traversal of many nodes, resulting in a slower execution. If you're inside a `for tree`, you probably want to use `if`.

Finally, let's talk about styles. (They are not actually defined in the FOREST package, but rather inherited from `pgfkeys`.)

At the first approximation, styles are abbreviations: if you often want to have non-default parent/child anchors, say south/north as in example (9), you could save some typing by defining a style. Styles are defined using PGF's handler `.style`, like shown below.[6]




```
\begin{forest}                                                    (13)
  sn edges/.style={for tree={
          parent anchor=south, child anchor=north}},
  sn edges
  [VP,
    [DP[John,tier=word]]
    [V'
      [V[sent,tier=word]]
      [DP[Mary,tier=word]]
      [DP[D[a,tier=word]][NP[letter,tier=word]]]]]
\end{forest}
```

If you want to use a style in more than one tree, you have to define it outside the `forest` environment. Use macro `\forestset` to do this.

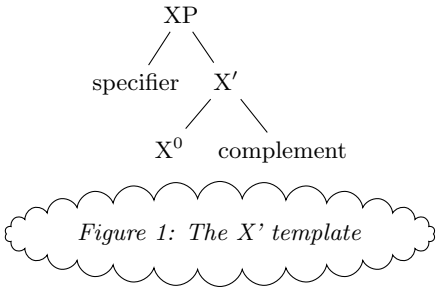
```
\forestset{
  sn edges/.style={for tree={parent anchor=south, child anchor=north}},
  background tree/.style={for tree={
              text opacity=0.2,draw opacity=0.2,edge={draw opacity=0.2}}}
}
```

You might have noticed that in the last two examples, some keys occurred even before the first opening bracket, contradicting was said at the beginning of this section. This is mainly just syntactic sugar (it can separate the design and the content): such *preamble* keys behave as if they were given in the root node, the only difference (which often does not matter) being that they get processed before all other root node options, even the implicit `content`.

If you find yourself writing the same preamble for every tree in your document, consider modifying `default preamble`, which is implicitely included at the beginning of every preamble.

2.3 Decorating the tree

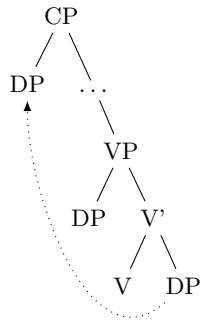
The tree can be decorated (think movement arrows) with arbitrary Ti*k*Z code.




```
\begin{forest}                                                    (14)
  [XP
    [specifier]
    [X$'$
      [X$^0$]
      [complement]
    ]
  ]
  \node at (current bounding box.south)
    [below=1ex,draw,cloud,aspect=6,cloud puffs=30]
    {\emph{Figure 1: The X' template}};
\end{forest}
```

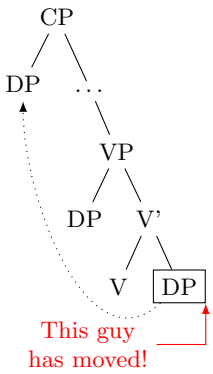
[6] Style `sn edges` is actually already defined by library `linguistics`. The definition there is a bit more generic.

However, decorating the tree would make little sense if one could not refer to the nodes. The simplest way to do so is to give them a Ti*k*Z name using the `name` option, and then use this name in Ti*k*Z code as any other (Ti*k*Z) node name.



(15)
```
\begin{forest}
  [CP
    [DP,name=spec CP]
    [\dots
      [,phantom]
      [VP
        [DP]
        [V'
          [V]
          [DP,name=object]]]]]
  \draw[->,dotted] (object) to[out=south west,in=south] (spec CP);
\end{forest}
```

It gets better than this, however! In the previous examples, we put the Ti*k*Z code after the tree specification, i.e. after the closing bracket of the root node. In fact, you can put Ti*k*Z code after *any* closing bracket, and Forest will know what the current node is. (Putting the code after a node's bracket is actually just a special way to provide a value for option `tikz` of that node.) To refer to the current node, simply use an empty node name. This works both with and without anchors [see 2, §16.11]: below, `(.south east)` and `()`.



(16)
```
\begin{forest}
  [CP
    [DP,name=spec CP]
    [\dots
      [,phantom]
      [VP
        [DP]
        [V'
          [V]
          [DP,draw] {
            \draw[->,dotted] () to[out=south west,in=south] (spec CP);
            \draw[<-,red] (.south east)--++(0em,-4ex)--++(-2em,0pt)
                 node[anchor=east,align=center]{This guy\\has moved!};
            }
        ]]]]
\end{forest}
```

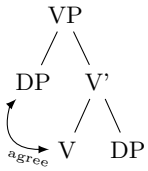
Important: *the* Ti*k*Z *code should usually be enclosed in braces* to hide it from the bracket parser. You don't want all the bracketed code (e.g. `[->,dotted]`) to become tree nodes, right? (Well, they probably wouldn't anyway, because TEX would spit out a thousand errors.)

Finally, the most powerful tool in the node reference toolbox: *relative nodes*. It is possible to refer to other nodes which stand in some (most often geometrical) relation to the current node. To do this, follow the node's name with a ! and a *nodewalk* specification.

A nodewalk is a concise[7] way of expressing node relations. It is simply a string of steps, which are represented by single characters, where: `u` stands for the parent node (up); `p` for the previous sibling; `n` for the next sibling; `s` for *the* sibling (useful only in binary trees); `1`, `2`, ... `9` for first, second, ... ninth child; `l`, for the last child, etc. For the complete specification, see section 3.8.7.

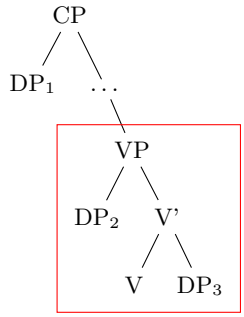
To see the nodewalk in action, consider the following examples. In the first example, the agree arrow connects the V node, specified simply as `()`, since the Ti*k*Z code follows `[V]`, and the DP node, which is described as "a sister of V's parent": `!us` = up + sibling.

[7] Actually, Forest distinguishes two kinds of steps in node walks: long-form and short-form steps. This section introduces only short-form steps. See §3.8.




```
\begin{forest}                                                          (17)
  [VP
    [DP]
    [V'
      [V] {\draw[<->] ()
           .. controls +(left:1cm) and +(south west:0.4cm) ..
           node[very near start,below,sloped]{\tiny agree}
           (!us);}
      [DP]
    ]
  ]
\end{forest}
```

The second example uses Ti*k*Z's fitting library (automatically loaded by Forest) to compute the smallest rectangle containing node VP, its first child (DP_2) and its last grandchild (DP_3). The example also illustrates that the Ti*k*Z code can be specified via the "normal" option syntax, i.e. as a value to option `tikz`.[8]

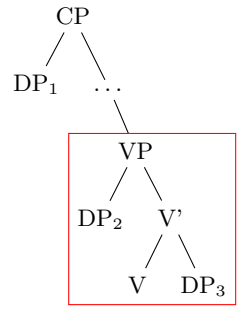



```
\begin{forest}                                                          (19)
  [CP
    [DP$_1$]
    [\dots
      [,phantom]
      [VP,tikz={\node [draw,red,fit=()(!1)(!ll)] {};}
        [DP$_2$]
        [V'
          [V]
          [DP$_3$]
  ]]]]
\end{forest}
```

2.4 Node positioning

Forest positions the nodes by a recursive bottom-up algorithm which, for every non-terminal node, computes the positions of the node's children relative to their parent. By default, all the children will be aligned horizontally some distance down from their parent: the "normal" tree grows down. More generally, however, the direction of growth can change from node to node; this is controlled by option `grow`=⟨direction⟩.[9] The system computes and stores the positions of children using a coordinate system dependent on the parent, called an *ls-coordinate system*: the origin is the parent's anchor; l-axis is in the direction of growth in the parent; s-axis is orthogonal to the l-axis (positive side in the counter-clockwise direction from *l*-axis); l stands for *l*evel, s for *s*ibling. The example shows the ls-coordinate system for a node with `grow=45`.[10]

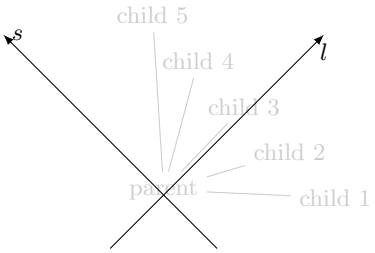
[8] Actually, there's a simpler way to do this: use `/tikz/fit to=tree`!




```
\begin{forest}                                                          (18)
  [CP
    [DP$_1$]
    [\dots
      [,phantom]
      [VP,tikz={\node [draw,red,inner sep=0,fit to=tree]{};}
        [DP$_2$]
        [V'
          [V]
          [DP$_3$]
  ]]]]
\end{forest}
```

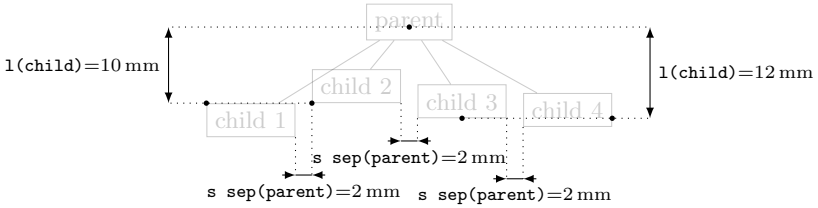
[9] The direction can be specified either in degrees (following the standard mathematical convention that 0 degrees is to the right, and that degrees increase counter-clockwise) or by the compass directions: `east`, `north east`, `north`, etc.

[10] The axes are drawn using coordinates given in `forest cs` coordinate system; the "manually" given polar coordinate equivalent is shown in the comment.




```
\begin{forest} background tree                                                    (20)
  [parent, grow=45
    [child 1][child 2][child 3][child 4][child 5]
  ]
  %\draw[,->](-135:1cm)--(45:3cm) node[below]{$l$};
  \draw[,->](forest cs:l=-1cm,s=0)--(forest cs:l=3cm,s=0) node[below]{$l$};
  %\draw[,->](-45:1cm)--(135:3cm) node[right]{$s$};
  \draw[,->](forest cs:s=-1cm,l=0)--(forest cs:s=3cm,l=0) node[right]{$s$};
\end{forest}
```

The l-coordinate of children is (almost) completely under your control, i.e. you set what is often called the level distance by yourself. Simply set option `l` to change the distance of a node from its parent.[11] More precisely, `l`, and the related option `s`, control the distance between the (node) anchors of a node and its parent. The anchor of a node can be changed using option `anchor`: by default, nodes are anchored at their base; see [2, §16.5.1].) In the example below, positions of the anchors are shown by dots: observe that anchors of nodes with the same `l` are aligned and that the distances between the anchors of the children and the parent are as specified in the code.[12]




```
\begin{forest} background tree,                                                   (21)
  for tree={draw,tikz={\fill[](.anchor)circle[radius=1pt];}}
  [parent
    [child 1, l=10mm, anchor=north west]
    [child 2, l=10mm, anchor=south west]
    [child 3, l=12mm, anchor=south]
    [child 4, l=12mm, anchor=base east]
  ]
  \measureydistance[\texttt{l(child)}=#1]{(!2.anchor)}{(.anchor)}{(!1.anchor)+(-5mm,0)}{left}
  \measureydistance[\texttt{l(child)}=#1]{(!3.anchor)}{(.anchor)}{(!4.anchor)+(5mm,0)}{right}
  \measurexdistance[\texttt{s sep(parent)}=#1]{(!1.south east)}{(!2.south west)}{+(0,-5mm)}{below}
  \measurexdistance[\texttt{s sep(parent)}=#1]{(!2.south east)}{(!3.south west)}{+(0,-5mm)}{below}
  \measurexdistance[\texttt{s sep(parent)}=#1]{(!3.south east)}{(!4.south west)}{+(0,-8mm)}{below}
\end{forest}
```

Positioning the chilren in the s-dimension is the job and *raison d'etre* of the package. As a first approximation: the children are positioned so that the distance between them is at least the value of option `s sep` (s-separation), which defaults to double PGF's `inner xsep` (and this is 0.3333em by default). As you can see from the example above, s-separation is the distance between the borders of the nodes, not their anchors!

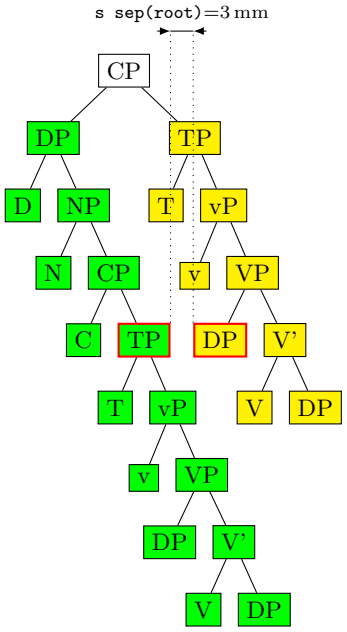
[11] If setting `l` seems to have no effect, read about `l sep` further down this section.

[12] Here are the definitons of the macros for measuring distances. Args: the x or y distance between points #2 and #3 is measured; #4 is where the distance line starts (given as an absolute coordinate or an offset to #2); #5 are node options; the optional arg #1 is the format of label. (Lengths are printed using package `printlen`.)

```
\newcommand\measurexdistance[5][####1]{\measurexorydistance{#2}{#3}{#4}{#5}{\x}{-|}{(5pt,0)}{#1}}
\newcommand\measureydistance[5][####1]{\measurexorydistance{#2}{#3}{#4}{#5}{\y}{|-}{(0,5pt)}{#1}}
\tikzset{dimension/.style={<->,>=latex,thin,every rectangle node/.style={midway,font=\scriptsize}},
  guideline/.style=dotted}
\newdimen\absmd
\def\measurexorydistance#1#2#3#4#5#6#7#8{%
  \path #1 #3 #6 coordinate(md1) #1; \draw[guideline] #1 --  (md1);
  \path (md1) #6 coordinate(md2) #2; \draw[guideline] #2 -- (md2);
  \path let \p1=($(md1)-(md2)$), \n1={abs(#51)} in \pgfextra{\xdef\md{#51}\global\absmd=\n1\relax};
  \def\distancelabelwrapper##1{#8}%
  \ifdim\absmd>5mm
    \draw[dimension] (md1)--(md2) node[#4]{\distancelabelwrapper{\uselengthunit{mm}\rndprintlength\absmd}};
  \else
    \ifdim\md>0pt
      \draw[dimension,<-] (md1)--+#7; \draw[dimension,<-] let \p1=($(0,0)-#7$) in (md2)--+(\p1);
    \else
      \draw[dimension,<-] let \p1=($(0,0)-#7$) in (md1)--+(\p1); \draw[dimension,<-] (md2)--+#7;
    \fi
    \draw[dimension,-] (md1)--(md2) node[#4]{\distancelabelwrapper{\uselengthunit{mm}\rndprintlength\absmd}};
  \fi}
```

A fuller story is that `s sep` does not control the s-distance between two siblings, but rather the distance between the subtrees rooted in the siblings. When the green and the yellow child of the white node are s-positioned in the example below, the horizontal distance between the green and the yellow subtree is computed. It can be seen with the naked eye that the closest nodes of the subtrees are the TP and the DP with a red border. Thus, the children of the root CP (top green DP and top yellow TP) are positioned so that the horizontal distance between the red-bordered TP and DP equals `s sep`.

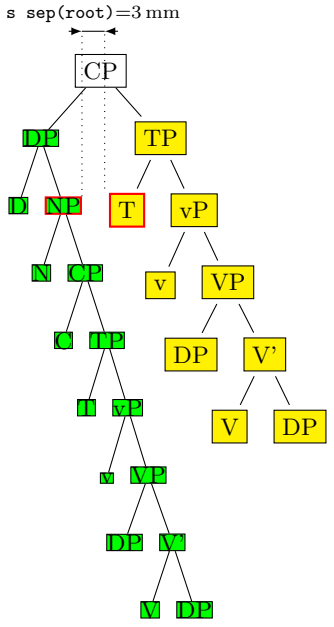
(22)




```
\begin{forest}
  important/.style={name=#1,draw={red,thick}}
  [CP, s sep=3mm, for tree=draw
    [DP, for tree={fill=green}
      [D][NP[N][CP[C][TP,important=left
      [T][vP[v][VP[DP][V'[V][DP]]]]]]]]
    [TP,for tree={fill=yellow}
      [T][vP[v][VP[DP,important=right][V'[V][DP]]]]]
  ]
  \measurexdistance[\texttt{s sep(root)}=#1]
    {(left.north east)}{(right.north west)}{(.north)+(0,3mm)}{above}
\end{forest}
```

Note that FOREST computes the same distances between nodes regardless of whether the nodes are filled or not, or whether their border is drawn or not. Filling the node or drawing its border does not change its size. You can change the size by adjusting Ti*k*Z's `inner sep` and `outer sep` [2, §16.2.2], as shown below:

(23)

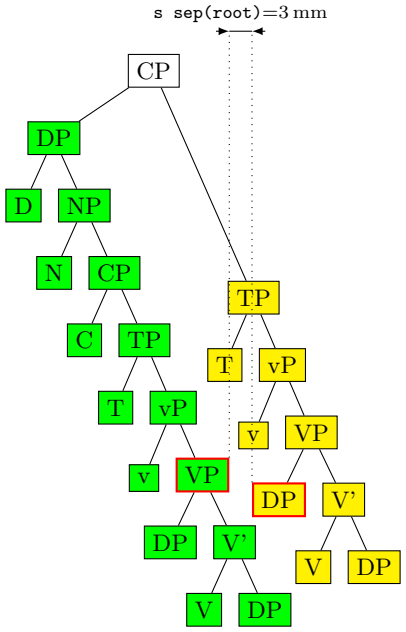



```
\begin{forest}
  important/.style={name=#1,draw={red,thick}}
  [CP, s sep=3mm, for tree=draw
    [DP, for tree={fill=green,inner sep=0}
      [D][NP,important=left[N][CP[C][TP[T][vP[v]
      [VP[DP][V'[V][DP]]]]]]]]
    [TP,for tree={fill=yellow,outer sep=2pt}
      [T,important=right][vP[v][VP[DP][V'[V][DP]]]]]
  ]
  \measurexdistance[\texttt{s sep(root)}=#1]
    {(left.north east)}{(right.north west)}{(.north)+(0,3mm)}{above}
\end{forest}
```

(This looks ugly!) Observe that having increased `outer sep` makes the edges stop touching borders of the nodes. By (PGF's) default, the `outer sep` is exactly half of the border line width, so that the edges start and finish precisely at the border.

Let's play a bit and change the `l` of the root of the yellow subtree. Below, we set the vertical distance of the yellow TP to its parent to 3 cm: and the yellow submarine sinks diagonally ... Now, the closest nodes are the higher yellow DP and the green VP.

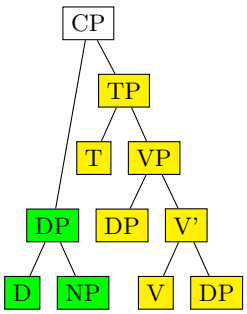
(24)




```
\begin{forest}
  important/.style={name=#1,draw={red,thick}}
  [CP, s sep=3mm, for tree=draw
    [DP, for tree={fill=green}
      [D][NP[N][CP[C][TP
      [T][vP[v][VP,important=left[DP][V'[V][DP]]]]]]]]
    [TP,for tree={fill=yellow}, l=3cm
      [T][vP[v][VP[DP,important=right][V'[V][DP]]]]]
  ]
  \measurexdistance[\texttt{s sep(root)}=#1]
    {(left.north east)}{(right.north west)}{(.north)+(0,3mm)}{above}
\end{forest}
```

Note that the yellow and green nodes are not vertically aligned anymore. The positioning algorithm has no problem with that. But you, as a user, might have, so here's a neat trick. (This only works in the "normal" circumstances, which are easier to see than describe.)

(25)

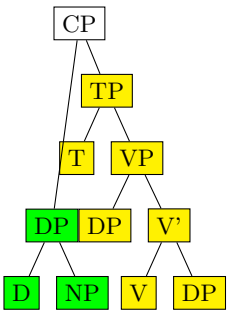



```
\begin{forest}
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D][NP]]
    [TP,for tree={fill=yellow}
      [T][VP[DP][V'[V][DP]]]]
  ]
\end{forest}
```

We have changed DP's `l`'s value via "augmented assignment" known from many programming languages: above, we have used `l*=3` to triple `l`'s value; we could have also said `l+=5mm` or `l-=5mm` to increase or decrease its value by 5 mm, respectively. This mechanism works for every numeric and dimensional option in Forest.

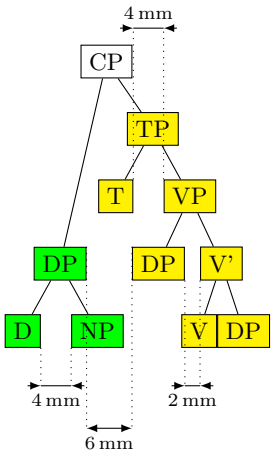
Let's now play with option `s sep`.

(26)




```
\begin{forest}
  [CP, for tree=draw, s sep=0
    [DP, for tree={fill=green},l*=3
      [D][NP]]
    [TP,for tree={fill=yellow}
      [T][VP[DP][V'[V][DP]]]]
  ]
\end{forest}
```

Surprised? You shouldn't be. The value of `s sep` at a given node controls the s-distance *between the subtrees rooted in the children of that node*! It has no influence over the internal geometry of these subtrees. In the above example, we have set `s sep=0` only for the root node, so the green and the yellow subtree are touching, although internally, their nodes are not. Let's play a bit more. In the following example, we set the `s sep` to: 0 at the last branching level (level 3; the root is level 0), to 2 mm at level 2, to 4 mm at level 1 and to 6 mm at level 0.

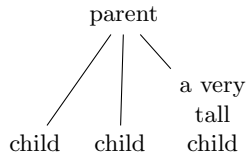


(27)
```
\begin{forest}
  for tree={s sep=(3-level)*2mm}
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D][NP]]
    [TP,for tree={fill=yellow}
      [T][VP[DP][V'[V][DP]]]]
  ]
  \measurexdistance{(!11.south east)}{(!12.south west)}{+(0,-5mm)}{below}
  \path(md2)-|coordinate(md)(!221.south east);
  \measurexdistance{(!221.south east)}{(!222.south west)}{(md)}{below}
  \measurexdistance{(!21.north east)}{(!22.north west)}{+(0,2cm)}{above}
  \measurexdistance{(!1.north east)}{(!221.north west)}{+(0,-2.4cm)}{below}
\end{forest}
```

As we go up the tree, the nodes "spread." At the lowest level, V and DP are touching. In the third level, the `s sep` of level 2 applies, so DP and V' are 2 mm apart. At the second level we have two pairs of nodes, D and NP, and T and TP: they are 4 mm apart. Finally, at level 1, the `s sep` of level 0 applies, so the green and yellow DP are 6 mm apart. (Note that D and NP are at level 2, not 4! Level is a matter of structure, not geometry.)

As you have probably noticed, this example also demostrated that we can compute the value of an option using an (arbitrarily complex) formula. This is thanks to PGF's module `pgfmath`. FOREST provides an interface to `pgfmath` by defining `pgfmath` functions for every node option, and some other information, like the `level` we have used above, the number of children `n children`, the sequential number of the child `n`, etc. For details, see §3.18.

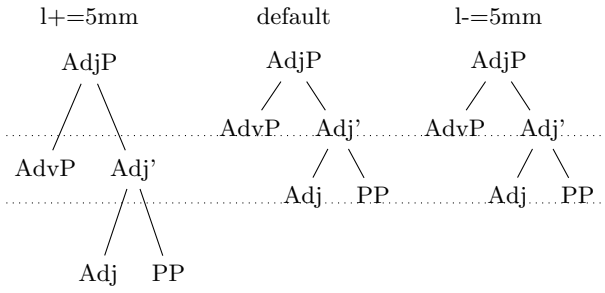
The final separation parameter is `l sep`. It determines the minimal separation of a node from its descendants. It the value of `l` is too small, then *all* the children (and thus their subtrees) are pushed away from the parent (by increasing their `l`s), so that the distance between the node's and each child's subtree boundary is at least `l sep`. The initial `l` can be too small for two reasons: either some child is too high, or the parent is too deep. The first problem is easier to see: we force the situation using a bottom-aligned multiline node. (Multiline nodes can be easily created using \\ as a line-separator. However, you must first specify the horizontal alignment using option `align` (see §3.7.1). Bottom vertical alignment is achieved by setting `base=bottom`; the default, unlike in Ti*k*Z, is `base=top`).

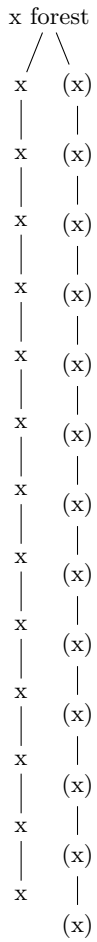


(28)
```
\begin{forest}
  [parent
    [child]
    [child]
    [a very\\tall\\child, align=center, base=bottom]
  ]
\end{forest}
```

The defaults for `l` and `l sep` are set so that they "cooperate." What this means and why it is necessary is a complex issue explained in §2.4.1, which you will hopefully never have to read ... You might be out of luck, however. What if you needed to decrease the level distance? And nothing happened, like below on the left? Or, what if you used lots of parenthesis in your nodes? And got a strange vertical misalignment, like below on the right? Then rest assured that these (at least) are features not bugs and read §2.4.1.

(29)





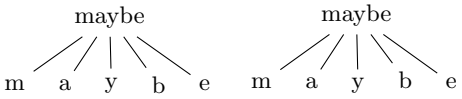

```
\begin{forest}
  [,phantom,for children={l sep=1ex,fit=band,
    for 1={edge'=,l=0},baseline}
    [{l+=5mm},for descendants/.pgfmath=content
      [AdjP[AdvP][Adj'[Adj][PP]]]]
    [default
      [AdjP[AdvP][Adj'[Adj][PP]]]]
    [{l-=5mm},for descendants/.pgfmath=content
      [AdjP[AdvP][Adj'[Adj][PP]]]]
  ]
  \path (current bounding box.west)|-coordinate(l1)(!212.base);
  \path (current bounding box.west)|-coordinate(l2)(!2121.base);
  \path (current bounding box.east)|-coordinate(r1)(!212.base);
  \path (current bounding box.east)|-coordinate(r2)(!2121.base);
  \draw[dotted] (l1)--(r1) (l2)--(r2);
\end{forest}
\hspace{5cm}
\begin{forest}
  [x forest, baseline
    [x[x[x[x[x[x[x[x[x[x[x]]]]]]]]]]]]
    [(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)]]]]]]]]]]]]
  ]
\end{forest}
```

2.4.1 The defaults, or the hairy details of vertical alignment

In this section we discuss the default values of options controlling the l-alignment of the nodes. The defaults are set with top-down trees in mind, so l-alignment is actually vertical alignment. There are two desired effects of the defaults. First, the spacing between the nodes of a tree should adjust to the current font size. Second, the nodes of a given level should be vertically aligned (at the base), if possible.

Let us start with the base alignment: Ti*k*Z's default is to anchor the nodes at their center, while FOREST, given the usual content of nodes in linguistic representations, rather anchors them at the base [2, §16.5.1]. The difference is particularly clear for a "phonological" representation:

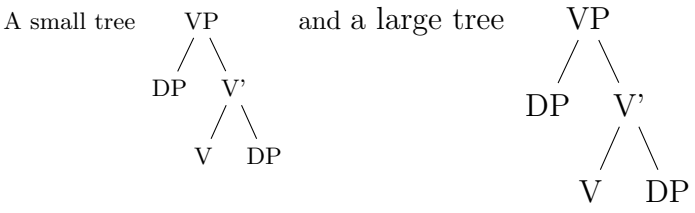
(30)




```
\begin{forest} for tree={anchor=center}
  [maybe[m][a][y][b][e]]
\end{forest}\quad
\begin{forest}
  [maybe[m][a][y][b][e]]
\end{forest}
```

The following example shows that the vertical distance between nodes depends on the current font size.

(31)

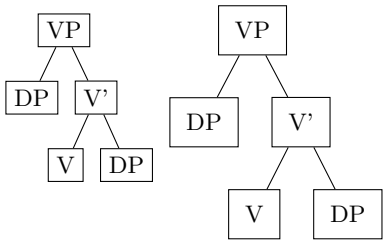



```
\hbox{\small A small tree
\begin{forest} baseline
  [VP[DP][V'[V][DP]]]
\end{forest}
\normalsize and
\large
a large tree
\begin{forest} baseline
  [VP[DP][V'[V][DP]]]
\end{forest}}
```

Furthermore, the distance between nodes also depends on the value of PGF's `inner sep` (which also depends on the font size by default: it equals 0.3333 em).

$$\texttt{l sep} = \text{height(strut)} + \texttt{inner ysep}$$

The default value of `s sep` depends on `inner xsep`: more precisely, it equals double `inner xsep`).

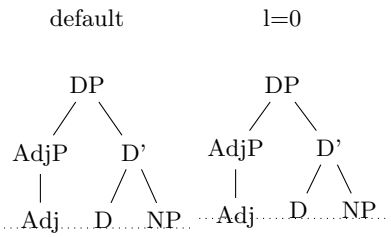



```
\begin{forest} baseline,for tree=draw
  [VP[DP][V'[V][DP]]]
\end{forest}
\pgfkeys{/pgf/inner sep=0.6666em}
\begin{forest} baseline,for tree=draw
  [VP[DP][V'[V][DP]]]
\end{forest}
```
(32)

Now a hairy detail: the formula for the default `l`.

$$\texttt{l} = \texttt{l sep} + 2 \cdot \texttt{outer ysep} + \text{total height('dj')}$$

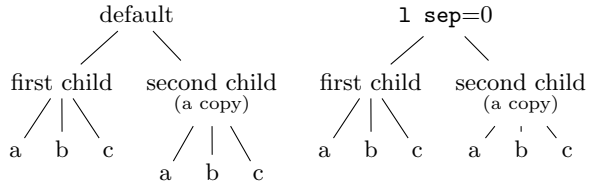
To understand what this is all about we must first explain why it is necessary to set the default `l` at all? Wouldn't it be enough to simply set `l sep` (leaving `l` at 0)? The problem is that not all letters have the same height and depth. A tree where the vertical position of the nodes would be controlled solely by (a constant) `l sep` could result in a ragged tree (although the height of the child–parent edges would be constant).




```
\begin{forest}
  [default,baseline,for children={no edge}
    [DP
      [AdjP[Adj]]
      [D'[D][NP,name=np]]]]
  \path (current bounding box.west)|-coordinate(l)(np.base);
  \path (current bounding box.east)|-coordinate(r)(np.base);
  \draw[dotted] (l)--(r);
\end{forest}
\begin{forest}
  [{l=0},baseline,for children={no edge}
    [DP,for descendants={l=0}
      [AdjP[Adj]]
      [D'[D][NP,name=np]]]]
  \path (current bounding box.west)|-coordinate(l)(np.base);
  \path (current bounding box.east)|-coordinate(r)(np.base);
  \draw[dotted] (l)--(r);
\end{forest}
```
(33)

The vertical misalignment of Adj in the right tree is a consequence of the fact that letter j is the only letter with non-zero depth in the tree. Since only `l sep` (which is constant throughout the tree) controls the vertical positioning, Adj, child of Ad*j*P, is pushed lower than the other nodes on level 2. If the content of the nodes is variable enough (various heights and depths), the cumulative effect can be quite strong, see the right tree of example (29).

Setting only a default `l sep` thus does not work well enough in general. The same is true for the reverse possibility, setting a default `l` (and leaving `l sep` at 0). In the example below, the depth of the multiline node (anchored at the top line) is such that the child–parent edges are just too short if the level distance is kept constant. Sometimes, misalignment is much preferred ...




```
\mbox{}\begin{forest}                                                    (34)
  [default,baseline
    [first child[a][b][c]]
    [{second child\\[-1ex]\scriptsize(a copy)},
     align=center[a][b][c]]
  ]
\end{forest}\quad
\begin{forest} for tree={l sep=0}
  [{\texttt{l sep}=0},baseline
    [first child[a][b][c]]
    [{second child\\[-1ex]\scriptsize(a copy)},
                  align=center[a][b][c]]
  ]
\end{forest}
```

Thus, the idea is to make `l` and `l sep` work as a team: `l` prevents misalignments, if possible, while `l sep` determines the minimal vertical distance between levels. Each of the two options deals with a certain kind of a "deviant" node, i.e. a node which is too high or too deep, or a node which is not high or deep enough, so we need to postulate what a *standard* node is, and synchronize them so that their effect on standard nodes is the same.

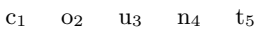
By default, FOREST sets the standard node to be a node containing letters d and j. Linguistic representations consist mainly of letters, and in the TEX's default Computer Modern font, d is the highest letter (not character!), and j the deepest, so this decision guarantees that trees containing only letters will look nice. If the tree contains many parentheses, like the right tree of example (29), the default will of course fail and the standard node needs to be modified. But for many applications, including nodes with indices, the default works.

The standard node can be changed using macro `\forestStandardNode`; see 3.19.

2.5 Advanced option setting

We have already seen that the value of options can be manipulated: in (11), we have converted numeric content from arabic into roman numerals using the *wrapping* mechanism `content=\romannumeral#1`; in (25), we have tripled the value of `l` by saying `l*=3`. In this section, we will learn more about the mechanisms for setting options and referring to their values.

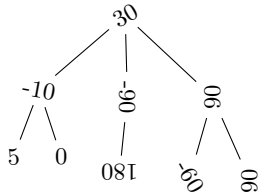
One other way to access an option value is using macro `\forestoption`. The macro takes a single argument: an option name. In the following example, the node's child sequence number is appended to the existing content. (This is therefore also an example of wrapping.)




```
\begin{forest}                                                           (35)
  [,phantom,delay={for descendants={
    content=#1$_{\forestoption{n}}$}}
  [c][o][u][n][t]]
\end{forest}
```

However, only options of the current node can be accessed using `\forestoption`. Possibly the simplest way to access option values of other nodes is to use FOREST's extensions to the PGF's mathematical library `pgfmath`, documented in [2, part VI]. To see `pgfmath` in action, first take a look at the crazy tree on the title page, and observe how the nodes are rotated: the value given to option `rotate` is a full-fledged `pgfmath` expression yielding an integer in the range from −30 to 30. Similiarly, `l+` adds a random float in the $[-5, 5]$ range to the current value of `l`.

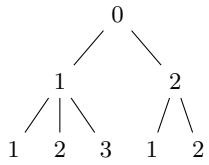
Example (27) demonstrated that information about the node, like the node's level, can be accessed within `pgfmath` expressions. All options are accessible in this way, i.e. every option has a corresponding `pgfmath` function. For example, we could rotate the node based on its content:




```
\begin{forest}                                                           (36)
  delay={for tree={rotate=content}}
  [30[-10[5][0]][-90[180]][90[-60][90]]]
\end{forest}
```

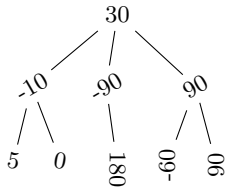
All numeric, dimensional and boolean options of FOREST automatically pass the given value through `pgfmath`. If you need pass the value through `pgfmath` for a string option, use the `.pgfmath` handler. The

following example sets the node's content to its child sequence number (the root has child sequence number 0).



(37)
```
\begin{forest}
  delay={for tree={content/.pgfmath=int(n)}}
  [[[][][]][[][]]]
\end{forest}
```

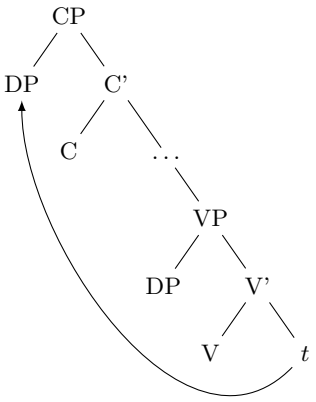
As mentioned above, using `pgfmath` it is possible to access options of non-current nodes. This is achieved by providing the option function with a ⟨relative node name⟩ (see §3.15) argument.[13] In the next example, we rotate the node based on the content of its parent (`u` means 'up').



(38)
```
\begin{forest}
  delay={for descendants={rotate=content("!u")}}
  [30[-10[5][0]][-90[180]][90[-60][90]]]
\end{forest}
```

Note that the argument of the option function is surrounded by double quotation marks: this is to prevent evaluation of the relative node name as a `pgfmath` function — which it is not.

For further ways to access option values, see §2.6. Here, we continue by introducing *relative node setting*: write ⟨relative node name⟩`.`⟨option⟩`=`⟨value⟩ to set the value of ⟨option⟩ of the specified relative node. Important: computation of the value is done in the context of the original node. The following example defines style `move` which not only draws an arrow from the source (the current node) to the target, but also moves the content of the source to the target (leaving a trace). Note the difference between `#1` and `##1`: `#1` is the argument of the style `move` (a node walk determining the target), while `##1` is the original option (in this case, `content`) value.



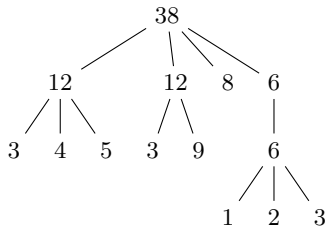
(39)
```
\begin{forest}
  for tree={calign=fixed edge angles},
  move/.style={
    tikz={\draw[->] () to[out=south west,in=south] (#1);},
    delay={#1.content={##1},content=$t$}},
  [CP[][C'[C][\dots[,phantom][VP[DP][V'[V][DP,move=!r1]]]]]]
\end{forest}
```

In the following example, the content of the branching nodes is computed by Forest: a branching node is a sum of its children. The algorithm visits each node (but the root node) and adds its content to the content of the parent. Note that as the computation must proceed bottom-up, `for descendants children-first` propagator is used to walk through the tree.[14]

[13] The form without parentheses `option_name` that we have been using until now to refer to an option of the current node is just a short-hand notation for `option_name()` — note that in some contexts, like preceding `+` or `-`, the short form does not work! (The same seems to be true for all pgfmath functions with "optional" arguments.)

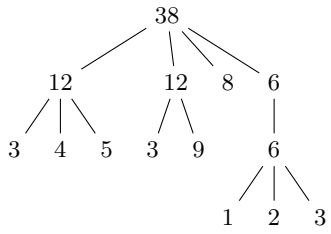
[14] It would be possible to emulate `for descendants children-first` by defining a recursive style, as was done in this manual for versions of the package prior to introduction of the bottom-up propagator. The following code produces identical result as the code in the main text.

```
\begin{forest}
  calc/.style={if n children={0}{}{content=0,for children={calc,!u.content/.pgfmath=int(content("!u")+content())}}},
  delay=calc,
  [[[3][4][5]][[3][9]][8][[[1][2][3]]]]
\end{forest}
```




```
\begin{forest}                                                    (40)
  delay={
    where n children={0}{}{content=0},
    for descendants children-first={
      !u.content/.pgfmath=int(content("!u")+content())}
  }
  [[[3][4][5]][[3][9]][8][[[1][2][3]]]
\end{forest}
```

Actually, for common computations such as summing things up, Forest provides an easier way to do it: aggregate functions (§3.14). Below, aggregate function `.sum`, defined as `pgfkeys` handler, walks through the `children` (second argument) of the current node, summing up their `content` (first argument) and stores the result as the `content` of the current node (because `content` is the handled key).

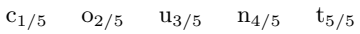



```
\begin{forest}                                                    (41)
  delay={
    aggregate postparse=int,
    for tree children-first={
      if n children={0}{}{
        content/.sum={content}{children}
      }
    }
  }
  [[[3][4][5]][[3][9]][8][[[1][2][3]]]
\end{forest}
```

2.6 Wrapping

We have already seen examples of inserting option values into other expressions. In example (11), we have wrapped the value of the option being assigned to (`#1` stood for the current value of option `content`); example (35) additionally wrapped the value of option `n` (of the current node) using macro `\forestoption`. In general, Forest offers two ways to perform computations (from simple option value lookups to complicated formulas) and insert their results into another expression (of any kind: TEX code, `pgfkeys` keylist, `pgfmath` expression, etc.).

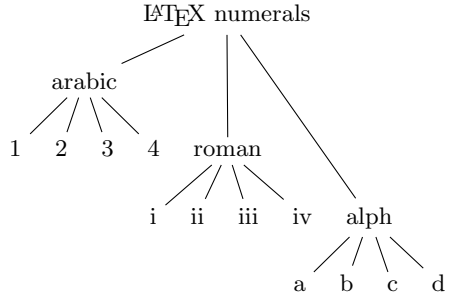
Historically, the first Forest's mechanism that offered wrapping of computed values were handlers `.wrap pgfmath arg` and `.wrap` n `pgfmath args` (for $n = 2, \ldots, 8$), which combine the wrapping mechanism with the `pgfmath` evaluation. The idea is to compute (most often, just access option values) arguments using `pgfmath` and then wrap them into the given macro body (marked below) using TEX's parameters (`#1` etc.). Below, this is used to subscript the contents of a node with its sequential number and the number of parent's children.




```
\begin{forest} [,phantom,delay={for descendants={                 (42)
            content/.wrap 3 pgfmath args=
            {#1$_{#2/#3}$}
            {content}{n}{n_children("!u")}}}
  [c][o][u][n][t]]
\end{forest}
```

Note the underscore `_` character in `n_children`: in `pgfmath` function names, spaces, apostrophes and other non-alphanumeric characters from option names are all replaced by underscores.

As another example, let's make the numerals example (7) a bit fancier. The numeral type is read off the parent's content and used to construct the appropriate control sequence (`\@arabic`, `\@roman` and `\@alph`). (The numbers are not specified in content anymore: we simply read the sequence number `n`.)



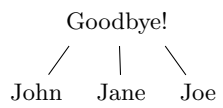

```
\begin{forest}                                                    (43)
  delay={where level={2}{content/.wrap 2 pgfmath args=
      {\csname @#1\endcsname{#2}}
      {content("!u")}{n}}{}},
  for children={l*=n},
  [\LaTeX\ numerals,
    [arabic[][][][]]
    [roman[][][][]]
    [alph[][][][]]
  ]
\end{forest}
```

Invoking `pgfmath` is fairly time consuming and using it to do nothing but retrieve an option value seems a bit of an overkill. To remedy the situation, argument processor (§3.13) was introduced in FOREST v2.0 and considerably expanded in v2.1. One way to invoke it is using handler `.process`.

The argument processor takes a sequence of instructions and an arbitrary number of arguments, transforms the given arguments according to the instructions, and feeds the resulting list of arguments into the handled key.

An instruction is given by a single-character code. The simplest instructions are: `0`, which expects its argument to be an option name (possibly preceded by a ⟨relative node name⟩`.` to access the option value of a non-current node) and returns the value of the option; `R`, which does the same for registers; and `noop`, which leaves the argument unchanged.

In the following example, we define style `test` taking four arguments and call it by providing the arguments via `.process`. The instruction string `R00_` tells the argument processor that the first argument is the value of (scratch) register `temptoksa`, the second the value of option `n children` at the current node, the third the value of option `content` of the second child of the current node, and the fourth just a plain string. Macro `test` is thus actually invoked with argument list `{Hello}{3}{Jane}{Goodbye}`.




```
\begin{forest}                                                          (44)
  test/.style n args={4}{align=center,
    content={#1!\\I have #2 children.\\One of them is #3.\\#4!}}
  [,delay={temptoksa=Hello,
    test/.process={R00_}{temptoksa}{n children}{!2.content}{Goodbye}}
    [John][Jane][Joe]]
\end{forest}
```

To wrap using the argument processor, use instruction `w`. Unless wrapping a single argument, this instruction should be followed by a number indicating the number of arguments consumed. `w` will take the required number of arguments from the list of already processed arguments and wrap them in the macro body given as the next (yet unprocessed) argument.

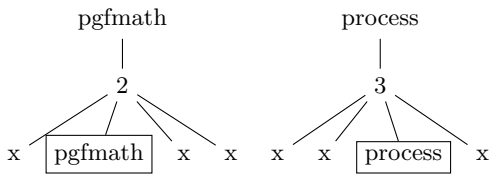
The following example has the same result as example (42). Note that the order of the wrapper-macro body and the arguments is different for `.process` and `.wrap` *n* `pgfmath args`. (Experience shows that `.process`'s order is easier on the eyes.) The example also illustrates that (i) the instructions need not be enclosed in braces and (ii) that repetition of an argument processor instruction can be indicated by appending a number to the instruction: thus `O3` below means the same as `OOO`.

$c_{1/5}$ $o_{2/5}$ $u_{3/5}$ $n_{4/5}$ $t_{5/5}$

```
\begin{forest} [,phantom,delay={for descendants={                       (45)
            content/.process=O3 w3
              {content}{n}{!u.n children}
              {#1$_{#2/#3}$}
            }}
  [c][o][u][n][t]]
\end{forest}
```

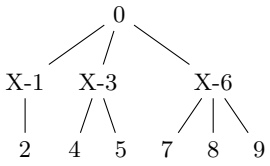
Note that the order of the wrapper-macro body and the arguments is different for `.process` and `.wrap` *n* `pgfmath args`. Experience shows that `.process`'s order is easier on the eyes. The example also illustrates that the instructions need not be enclosed in braces and that repetition of an argument processor instruction can be indicated by appending a number to the instruction: `O3` above is equivalent to `OOO`.

`.wrap` *n* `pgfmath args` always returns a single braced expression and is thus a bit cumbersome to use when the handled key expects multiple arguments: the trick is to enclose the expected argument list in extra braces (marked in the code below). As `.process` can return multiple arguments, there is no need for such a workaround. See the following example for comparison of the two methods.




```
\begin{forest}                                                          (46)
  [,phantom
    [pgfmath[2,delay={for n/.wrap 2 pgfmath args=
        {{#1}{content=#2,draw}}
        {content}{content("!u")}
      } [x][x][x][x]]]
    [process[3, delay={for n/.process=
        {0 0w1}{content}
        {!u.content}{content=#1,draw}
      } [x][x][x][x]]]
  ]
\end{forest}
```

A single `.process` invocation can perform multiple wrappings. The numbering of arguments of each wrapping starts at `#1`. In the example below, `for nodewalk` takes two arguments, a nodewalk and a list of nodekeys. Each is produced by an independent wrapping (wrap bodies are marked in the code).



(47)
```
\begin{forest}
  declare toks register=prefix,
  declare count register=level to prefix,
  prefix=X-,
  level to prefix=1,
  delay={
    for nodewalk/.process=Rw Rw
    {level to prefix}{level=#1}
    {prefix}{+content=#1}
  }
  [0[1[2]][3[4][5]][6[7][8][9]]]
\end{forest}
```

2.7 Externalization

Forest can be quite slow, due to the slowness of both pgf/Ti*k*Z and its own computations. However, using *externalization*, the amount of time spent in Forest in everyday life can be reduced dramatically. The idea is to typeset the trees only once, saving them in separate PDFs, and then, on the subsequent compilations of the document, simply include these PDFs instead of doing the lenghty tree-typesetting all over again.

Forest's externalization mechanism is built on top of Ti*k*Z's `external` library. It enhances it by automatically detecting the code and context changes: the tree is recompiled if and only if either the code in the `forest` environment or the context (arbitrary parameters; by default, the parameters of the standard node) changes.

To use Forest's externalization facilities, say:[15]

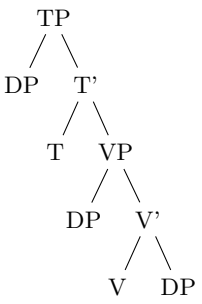
```
\usepackage[external]{forest}
\tikzexternalize
```

If your `forest` environment contains some macro, you will probably want the externalized tree to be recompiled when the definition of the macro changes. To achieve this, use `\forestset{external/depends on macro=\macro}`. The effect is local to the TEX group.

Ti*k*Z's externalization library promises a `\label` inside the externalized graphics to work out-of-box, while `\ref` inside the externalized graphics should work only if the externalization is run manually or by `make` [2, §32.4.1]. A bit surprisingly perhaps, the situation is roughly reversed in Forest. `\ref` inside the externalized graphics will work out-of-box. `\label` inside the externalized graphics will not work at all. Sorry. (The reason is that Forest prepares the node content in advance, before merging it in the whole tree, which is when Ti*k*Z's externalization is used.)

2.8 Expansion control in the bracket parser

By default, macros in the bracket encoding of a tree are not expanded until nodes are being drawn — this way, node specification can contain formatting instructions, as illustrated in section 2.1. However, sometimes it is useful to expand macros while parsing the bracket representation, for example to define tree templates such as the X-bar template, familiar to generative grammarians:[16]



(48)
```
\bracketset{action character=@}
\def\XP#1#2#3{#1P[#2][#1'[#1][#3]]}
\begin{forest}
  [@\XP T{DP}{@\XP V{DP}{DP}}]
\end{forest}
```

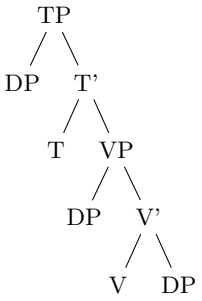
[15] When you switch on the externalization for a document containing many `forest` environments, the first compilation can take quite a while, much more than the compilation without externalization. (For example, more than ten minutes for the document you are reading!) Subsequent compilations, however, will be very fast.

[16] Honestly, dynamic node creation might be a better way to do this; see §3.11.

In the above example, the `\XP` macro is preceded by the *action character* `@`: as the result, the token following the action character was expanded before the parsing proceeded.

The action character is not hard coded into FOREST. Actually, there is no action character by default. (There's enough special characters in FOREST already, anyway, and the situations where controlling the expansion is preferable to using the `pgfkeys` interface are not numerous.) It is defined at the top of the example by processing key `action character` in the `/bracket` path; the definition is local to the TEX group.

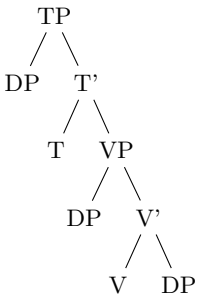
Let us continue with the description of the expansion control facilities of the bracket parser. The expandable token following the action character is expanded only once. Thus, if one defined macro `\VP` in terms of the general `\XP` and tried to use it in the same fashion as `\XP` above, he would fail. The correct way is to follow the action character by a braced expression: the braced expression is fully expanded before bracket-parsing is resumed.



(49)

```
\bracketset{action character=@}
\def\XP#1#2#3{#1P[#2][#1'[#1][#3]]}
\def\VP#1#2{\XP V{#1}{#2}}
\begin{forest}
  [@\XP T{DP}{@{\VP{DP}{DP}}}]
\end{forest}
```

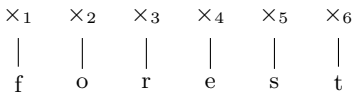
In some applications, the need for macro expansion might be much more common than the need to embed formatting instructions. Therefore, the bracket parser provides commands `@+` and `@-`: `@+` switches to full expansion mode — all tokens are fully expanded before parsing them; `@-` switches back to the default mode, where nothing is automatically expanded.



(50)

```
\bracketset{action character=@}
\def\XP#1#2#3{#1P[#2][#1'[#1][#3]]}
\def\VP#1#2{\XP V{#1}{#2}}
\begin{forest} @+
  [\XP T{DP}{\VP{DP}{DP}}]
\end{forest}
```

All the action commands discussed above were dealing only with TEX's macro expansion. There is one final action command, `@@`, which yields control to the user code and expects it to call `\bracketResume` to resume parsing. This is useful to e.g. implement automatic node enumeration:



(51)

```
\bracketset{action character=@}
\newcount\xcount
\def\x#1{@@\advance\xcount1
  \edef\xtemp{[$\noexpand\times_{\the\xcount}$[#1]]}%
  \expandafter\bracketResume\xtemp
}
\begin{forest}
  phantom,
  delay={where level=1{content={\strut #1}}{}}
  @+
  [\x{f}\x{o}\x{r}\x{e}\x{s}\x{t}]
\end{forest}
```

This example is fairly complex, so let's discuss how it works. `@+` switches to the full expansion mode, so that macro `\x` can be easily run. The real magic hides in this macro. In order to be able to advance the node counter `\xcount`, the macro takes control from FOREST by the `@@` command. Since we're already in control, we can use `\edef` to define the node content. Finally, the `\xtemp` macro containing the node specification is expanded with the resume command sticked in front of the expansion.

3 Reference

This section documents all publicly exposed keys and macros defined by the core package. All other commands defined by the package (see the implementation typeset in `forest.pdf`) are considered internal and might change without prior notice or compatibility support.

3.1 Package loading and options

Load the package by writing `\usepackage{forest}` in the document preamble.

Field-specific definitions and defaults are stored in separate libraries. Use `\usepackage[`⟨library name⟩`]{forest}` to load library ⟨library name⟩ and its defaults alongside the main package. Loading several libraries in this way is allowed: however, if you need more control over loading the defaults, use the following macros.

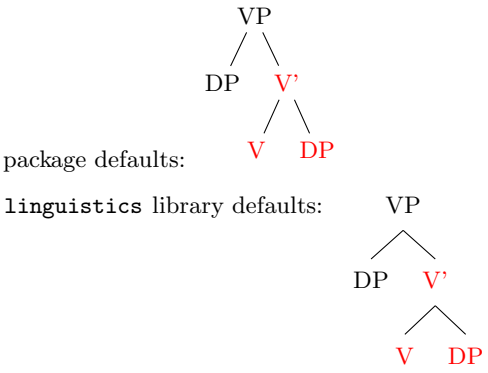
macro `\useforestlibrary`[*][[⟨options⟩]]{⟨library⟩,...} Loads the given libraries.

The starred version applies their defaults as well, while the starless does not. Multiple library names can be given, separated by commas. Libraries can receive ⟨options⟩. This macro can only be used in the preamble.

macro `\forestapplylibrarydefaults{`⟨library name⟩,...`}` Loads the default settings of ⟨library⟩.

Multiple library names can be given, separated by commas. This macro can be used either in the preamble or in the document body. Its effect is local to the current TEX scope.

For example, the `linguistics` library defines c-command related nodewalks, changes the default parent–child edges to south–north (the main package default is border–border) and sets the baseline to the root node. Thus, if you write `\usepackage[linguistics]{forest}` in your preamble, or use macro `\forestapplylibrarydefaults` like below, you get the following:




```
package defaults:                                                    (52)
\begin{forest}
  [VP % cannot use "for c-commanded" below!
    [DP, for sibling={for tree=red}]
    [V'[V][DP]]
  ]
\end{forest}\\[1ex]
|linguistics| library defaults:
\forestapplylibrarydefaults{linguistics}
\begin{forest}
  [VP
    [DP, for c-commanded={red}]
    [V'[V][DP]]
  ]
\end{forest}
```

package option `external=true|false` `false`

Enable/disable externalization, see §3.20.

package option `compat=`⟨keylist⟩ Enter compatibility mode with previous versions of the package. `most`

If at all possible, each backwards incompatible change is given a key in the `compat` path, e.g. `compat=1.0-forstep` reverts to the old behaviour of spatial propagators `for` ⟨step⟩, where a propagator could not fail.

While each compatibility feature can be enabled individually, they are grouped for ease of use. To load compatibility features since the last version of form `x[.y[.z]]`, write `compat=x[.y[.z]]-all` or `compat=x[.y[.z]]-most`. The former enables all compatibility features since that release, the latter only those that are guaranteed to not disrupt any new functionality of the package.

To load all compatibility features since the last major release (`x` in `x.y.z`), write `compat=all`; to load most of them, write `compat=most` or simply `compat`.

To enable multiple compatibility features, either use this option multiple times, or provide it with a comma-separated list of compatibility features. (Surround the list by braces.)

Specifying this option also defines macro \forestcompat (taking the same arguments as the package option) which can be used to enable compatibility features locally, within the document body. To enable compatibility mode but not enable any specific compatibility feature for the entire document, write compat=none as a package option.

For a list of compatibility features, see §6.1.

By default, the package warns when a compatibility feature is used. Disable this behaviour by compat=silent.

package option tikzcshack=true|false — true

Enable/disable the hack into Ti*k*Z's implicit coordinate syntax, see §3.15.

package option tikzinstallkeys=true|false — true

Install certain keys into the /tikz path. Currently: /tikz/fit to.

package option debug=⟨debug category⟩[, ⟨debug category⟩]*

Prints out some debugging info to the log file. When given no argument, prints out all the available information, otherwise only the information on the listed (comma-separated) debug categories. The available categories are listed below.

debug value nodewalks

debug value dynamics

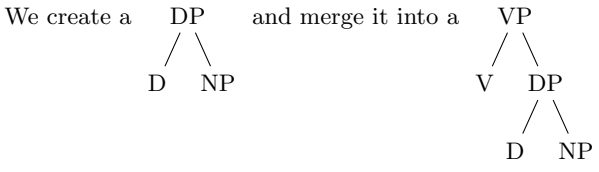
debug value process

3.2 Invocation

environment \begin{forest}[(⟨config⟩)]⟨tree⟩\end{forest}

macro \Forest[*][(⟨config⟩)]{⟨tree⟩}

The environment and the starless version of the macro introduce a group; the starred macro does not, so the created nodes can be used afterwards, like in the example below. (Note that this will leave a lot of temporary macros lying around. This shouldn't be a problem, however, since all of them reside in the \forest "namespace".)




```
We create a                                   (53)
\Forest*{
  [DP,name=DP,baseline
    [D]
    [NP]
  ]
}
and merge it into a
\Forest*{
  [VP,baseline
    [V]
    [,replace by=DP
    ]
  ]
}
```

⟨config⟩ is a keylist that configures the behaviour of the environment/macro. The configuration is the first operation that the environment/macro does; it precedes even the reading of the tree specification. Currently, ⟨config⟩ accepts only one key:

forest option stages=⟨keylist⟩

By default, after reading the tree specification, Forest executes style stages. If key stages is used in ⟨config⟩, ⟨keylist⟩ is executed instead.

macro \forestset{⟨keylist⟩}

Execute ⟨keylist⟩ (of node keys) with the default path set to /forest.

→ This macro is usually used to define Forest styles.

→ Usually, no current node is set when this macro is called. Thus, executing most node keys in this place will fail. However, if you have some nodes lying around, you can use propagator `for name=`⟨node name⟩ to set the node with the given name as current.

3.3 The bracket representation

A bracket representation of a tree is a token list with the following syntax:

$$
\begin{aligned}
\langle \text{tree} \rangle &= [\langle \text{preamble} \rangle]\, \langle \text{node} \rangle \\
\langle \text{node} \rangle &= \texttt{[}\, [\langle \text{content} \rangle]\, [\texttt{,}\langle \text{keylist} \rangle]\, [\langle \text{children} \rangle]\, \texttt{]}\, \langle \text{afterthought} \rangle \\
\langle \text{preamble} \rangle &= \langle \text{keylist} \rangle \\
\langle \text{keylist} \rangle &= \langle \text{key–value} \rangle\, [, \langle \text{keylist} \rangle] \\
\langle \text{key–value} \rangle &= \langle \text{key} \rangle \mid \langle \text{key} \rangle\texttt{=}\langle \text{value} \rangle \\
\langle \text{children} \rangle &= \langle \text{node} \rangle\, [\langle \text{children} \rangle]
\end{aligned}
$$

The ⟨preamble⟩ keylist is stored into keylist register `preamble`. The ⟨keylist⟩ of a ⟨node⟩ is stored into keylist option `given options`. ⟨content⟩ and ⟨afterthought⟩ are normally stored by prepending and appending `content'=`⟨content⟩ and `afterthought=`⟨afterthought⟩ to `given options`, respectively; this is customizable via `content to` and redefining style `afterthought`.

Normally, the tokens in the bracket representation are not expanded while the input is parsed. However, it is possible to control expansion. Expansion control sequences of Forest's bracket parser are shown below. Note that by default, there is no `action character`.

⟨action character⟩`-`	no-expansion mode (default): nothing is expanded
⟨action character⟩`+`	expansion mode: everything is fully expanded
⟨action character⟩⟨token⟩	expand ⟨token⟩
⟨action character⟩⟨TEX-group⟩	fully expand ⟨TEX-group⟩
⟨action character⟩⟨action character⟩	yield control; upon finishing its job, user's code should call `\bracketResume`

To customize the bracket parser, call `\bracketset`⟨keylist⟩, where the keys can be the following.

bracket key `opening bracket=`⟨character⟩ `[`

bracket key `closing bracket=`⟨character⟩ `]`

bracket key `action character=`⟨character⟩ `none`

By redefining the following two keys, the bracket parser can be used outside Forest.

bracket key `new node=`⟨preamble⟩⟨node specification⟩⟨csname⟩. Required semantics: create a new node given the preamble (in the case of a new root node) and the node specification and store the new node's id into ⟨csname⟩.

bracket key `set afterthought=`⟨afterthought⟩⟨node id⟩. Required semantics: store the afterthought in the node with given id.

3.4 The workflow

3.4.1 Stages

Forest does its job in several stages. The default course of events is the following:

1. The bracket representation of the tree (§3.3) is parsed and stored in a data structure.
2. The keys given in the bracket representation are processed. In detail, `default preamble` is processed first, then the given `preamble` (both in the context of the (formal) root node) and finally the keylists given to individual nodes. The latter are processed recursively, in a depth-first, parent-first fashion.
3. Each node is typeset in its own `tikzpicture` environment, saved in a box and its measures are taken.
4. The nodes of the tree are *packed*, i.e. the relative positions of the nodes are computed so that the nodes don't overlap. That's difficult. The result: option `s` is set for all nodes. (Sometimes, the value of `l` is adjusted as well.)

5. Absolute positions, or rather, positions of the nodes relative to the root node are computed. That's easy. The result: options `x` and `y` are set.

6. The Ti*k*Z code that will draw the tree is produced and executed. (The nodes are drawn by using the boxes typeset in step 3.)

Stage 1 collects user input and is thus "fixed". However, the other stages, which do the actual work, are under user's control.

First, hooks exist between the individual stages which make it possible (and easy) to change the properties of the tree between the processing stages. For a simple example, see example (71): the manual adjustment of `y` can only be done after the absolute positions have been computed, so the processing of this option is deferred by `before drawing tree`. For a more realistic example, see the definition of style `GP1`: before packing, `outer xsep` is set to a high (user determined) value to keep the ×s uniformly spaced; before drawing the tree, the `outer xsep` is set to `0pt` to make the arrows look better.

Second, the execution of the processing stages 2–6 is *completely* under user's control. To facilitate adjusting the processing flow, the approach is twofold. The outer level: FOREST initiates the processing by executing style `stages`, which by default executes the processing stages 2–6, preceding the execution of each but the first stage by processing the keys embedded in temporal propagators `before ...` (see §3.4.2). The inner level: each processing step is the sole resident of a stage-style, which makes it easy to adjust the workings of a single step. What follows is the default content of style `stages`, including the default content of the individual stage-styles. Both nicely readable and ready to copy-paste versions are given.

style `stages`

```
for root'={
  process keylist register=default preamble,
  process keylist register=preamble
}
process keylist=given options
process keylist=before typesetting nodes
```

style `typeset nodes stage` {for root'=typeset nodes}

```
process keylist=before packing
```

style `pack stage` {for root'=pack}

```
process keylist=before computing xy
```

style `compute xy stage` {for root'=compute xy}

```
process keylist=before drawing tree
```

style `draw tree stage` {for root'=draw tree}

```
\forestset{
  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 drawing tree,
    draw tree stage
  },
  typeset nodes stage/.style={for root'=typeset nodes},
  pack stage/.style={for root'=pack},
  compute xy stage/.style={for root'=compute xy},
  draw tree stage/.style={for root'=draw tree},
}
```

Both style `stages` and the individual stage-styles may be freely modified by the user. Obviously, as a style must be redefined before it is processed, `stages` should be redefined (using macro `\forestset`) outside the `forest` environment; alternatively, stages can be given as the (parenthesized) optional argument of the environment (see §3.2). A stage style can also be redefined in the preamble or in any of the keylists processed prior to entering that stage.

Here's the list of keys used either in the default processing or useful in an alternative processing flow.

stage `typeset nodes`

stage `typeset nodes'`

Typesets each node of the current node's subtree in its own `tikzpicture` environment. The result is saved in a box (which is used later, in the `draw tree stage`) and its measures are taken.

In the `typeset nodes'` variant, the node box's content is not overwritten if the box already exists.

The order in which the nodes are typeset is controlled by nodewalk style `typeset nodes processing order` or, if this style is not defined, by `processing order`.

`typeset node` Typesets the *current* node, saving the result in the node box.

This key can be useful also in the default `stages`. If, for example, the node's content is changed and the node retypeset just before drawing the tree, the node will be positioned as if it contained the "old" content, but have the new content: this is how the constant distance between ×s is implemented in the `GP1` style.

stage `pack` The nodes of the tree are *packed*, i.e. the relative positions of the nodes are computed so that the nodes don't overlap. The result: option `s` is set for all nodes; sometimes (in tier alignment and for some values of `calign`), the value of some nodes' `l` is adjusted as well.

`pack'` "Non-recursive" packing: packs the children of the current node only. (Experimental, use with care, especially when combining with tier alignment.)

stage `compute xy` Computes the positions of the nodes in the subtree relative to the current node. The results are stored into options `x` and `y`. The current node's `x` and `y` remain unchanged.

stage `draw tree`

stage `draw tree'` Produces and executes the Ti*k*Z code that draws the (sub)tree rooted in the current node.

The procedure uses the node boxes typeset by `typeset nodes` or friends. The `'` variant includes the node boxes in the picture using `\copy`, not `\box`, thereby preserving them.

For details and customization, see §3.4.3.

`draw tree box`=[⟨TEX box⟩] The picture drawn by the subsequent invocations of `draw tree` and `draw tree'` is put into ⟨TEX box⟩. If the argument is omitted, the subsequent pictures are typeset normally (the default).

`process keylist`=⟨keylist option⟩ For each node in the entire tree, the keylist saved in ⟨keylist option⟩ of the node is processed (in the context of that node).

Note that this key is not sensitive to the current node: it processes the keylists for the whole tree. Actually, it is possible to control which nodes are visited: ⟨keylist option⟩ `processing order` is walked if it is defined, otherwise `processing order`. In both cases, the processing nodewalk starts at the formal root of the tree (see `root'` and `set root`), which is reevaluated at the beginning of each internal cycle (see below). By default, ⟨keylist option⟩ `processing order` is indeed undefined, while the `processing order` defauls to `tree`, which means that *all the nodes in the entire tree* are processed.

Keylist-processing proceeds in cycles. In a given cycle, the value of option ⟨keylist option name⟩ is processed for every node visited by the processing nodewalk. During a cycle, keys may be *delayed* using key `delay`. Keys delayed in a cycle are processed in the next cycle. The number of cycles in unlimited.

Dynamic creation of nodes happens between the cycles. The options given to the dynamically created nodes are implicitely delayed and thus processed at the end of the next cycle.

This key is primarily intended for use within `stages`. The calls of this key should *not* be nested, and it should not be embedded under `process keylist'` or `process keylist register`.

When changing the processing nodewalk, note that delayed keys will be executed only for nodes visited by the processing nodewalk. Delayed spatially propagated keys will be remembered, though, and executed when the given keylist is processed for the target node. Using spatial propagators without delaying cannot result in a non-processed key.

`process keylist'`=⟨keylist option⟩⟨nodewalk⟩

This key is a variant of `process keylist`. The differences are as follows.

The processing nodewalk is given explicitely (by ⟨nodewalk⟩) and starts at the current node (in each internal cycle).

There is no dynamic creation of nodes between the delay cycles. Any dynamic node instructions will be remembered and executed after the next cycle of `process keylist`, or an explicit call to `do dynamics`.

It is safe to embed this key within `process keylist` and (all) friends.

`process keylist''`=⟨keylist option⟩⟨nodewalk⟩

This key is a variant of `process keylist` which executes neither dynamic node operations nor delayed keys (there are thus no internal cycles). Any delayed keys will not be processed during the execution of this key. They will be remembered and executed at the end of the next cycle of `process keylist` or `process keylist'`.

As for `process keylist'`, the processing nodewalk is given explicitely (by ⟨nodewalk⟩) and starts at the current node.

It is safe to embed this key within `process keylist` and (all) friends.

`process keylist register`=⟨register⟩

Process the keylist saved in ⟨register⟩ in the context of the current node.

Any delayed keys will not be processed during the execution of this key. They will be remembered and executed at the end of the next cycle of `process keylist` or `process keylist'`.

It is safe to embed this key within `process keylist` or `process keylist'`.

`process delayed`=⟨nodewalk⟩ Process delayed keys.

Keylist `delay` cannot be processed using `process keylist` or `process keylist'`. Thus this key.

Like `process keylist` or `process keylist'`, this key uses internal cycles. Thus, any embedded `delay`s will be processed.

There is no dynamic creation of nodes between the delay cycles. Any dynamic node instructions will be remembered and executed after the next cycle of `process keylist` or `process keylist'`, or an explicit call to `do dynamics`.

This key is safe to use within `process keylist`, `process keylist'` and `process keylist register`.

nodewalk style `processing order/.nodewalk style`=⟨nodewalk⟩ `tree`

Redefine this style to change the default order in which `process keylist` processes a keylist option. For example, to process the nodes in a child-first fashion, write

```
processing order/.nodewalk style=tree children first
```

Note that this is a *nodewalk* style, so it must be defined either using `.style` handler during a nodewalk or using `.nodewalk style`.

nodewalk style ⟨keylist option⟩ `processing order/.nodewalk style`=⟨nodewalk⟩ `processing order`

Redefine this style to change the `process keylist` processing order for a specific ⟨keylist option⟩. For example, to process `before drawing tree` options in the child-first fashion, leaving the processing of other `before ...` keylists untouched, write

```
before drawing tree processing order/.nodewalk style=tree children first
```

`do dynamics` Experimental. Perform pending dynamic tree operations.

Do not use this key within `process keylist` or `process keylist'`.

3.4.2 Temporal propagators

Temporal propagators delay processing of given keys until some other point in the processing of the tree. There are three kinds of temporal propagators. Most of the propagators have the form `before ...` and defer the processing of the given keys to a hook just before some stage in the workflow (§3.4.1). `before packing node` and `after packing node` are special as they fire *during* the packing stage. The `delay` propagator is "internal" to the current hook: the keys in the hook are processed cyclically, and `delay` delays the processing of the given keys until the next cycle.

Formally, temporal propagators are keylist options (except `delay n`, which is a style), so augmented assignments are possible (§3.6.1).

All temporal propagators can be nested without limit.

→ A note on typos.
By default, all keys unknown to Forest are appended to keylist option `node options` The value of `node options` is fed to Ti*k*Z when typesetting a node, so any typos are caught by Ti*k*Z. However, as nodes are normally typeset in stage `typeset nodes stage`, any typos in keys temporally propagated past that stage will not be noticed, simply because noone will use the value of `node options` where they end up (the exception being nodes which are explicitely retypeset by the user using `typeset node`).
To sum up, typos in any keys temporally propagated by `before packing`, `before packing node`, `after packing node`, `before computing xy` and `before drawing tree` will be silently ignored. This is probably not what you want, so double-check everything you write there.
Using `unknown to=unknown key error`, it is possible to change the default behaviour. You will catch all typos if you append the command to `pack stage`, as shown below. This can be done either in the tree or by `\forestset`.

```
typeset nodes stage/.append style={unknown to=unknown key error}
```

Of course, this makes it impossible to write simply `before drawing tree={inner sep=5pt, typeset node}`. Any `tikz`'s options must be given explicitely via `node options`: `before drawing tree={node options={inner sep=5pt}, typeset node}`.

propagator `delay`=⟨keylist⟩ Defers the processing of the ⟨keylist⟩ until the next cycle.

Internally, `delay` is a keylist option, so augmented operators of the ⟨keylist⟩ type can be used.

To check whether any keys were delayed, use conditional `if have delayed`.

propagator `delay n`=⟨integer⟩⟨keylist⟩ Defers the processing of the ⟨keylist⟩ for n cycles. n may be 0, and it may be given as a `pgfmath` expression.

propagator `given options`

When `stages` processing starts, this list holds the keys given by the user in the bracket representation.

propagator `before typesetting nodes`=⟨keylist⟩ Defers the processing of the ⟨keylist⟩ to until just before the nodes are typeset.

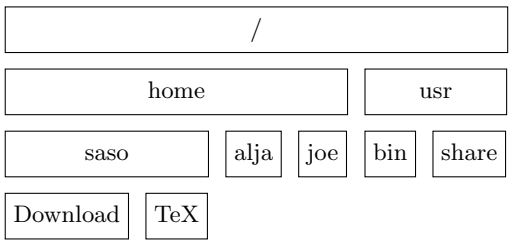
propagator `before packing`=⟨keylist⟩

propagator `before packing node`=⟨keylist⟩

Defers the processing of the ⟨keylist⟩ given to the node to until just before/after the subtree of *this specific node* is packed. Even before packing node, the (subtrees of the) children of the node have already been packed.[17]

propagator `after packing node`=⟨keylist⟩ Defers the processing of the ⟨keylist⟩ given to the node to until just after *this specific node* is packed.

[17] Forest employs two variants of the packing algorithm: the faster one is used for (parts of) trees with uniform growth, i.e. subtrees where `grow` does not change; the slower, generic variant is used in where this is not the case. Now, the fast method works by dealing with l and s dimension separately, and it is able to do this for the entire (sub)tree, without needing to invoke the packing method for its constituents. The consequence is that there is no place where `before packing node` could be called meaningfully, as the node's constituents are not packed individually, "just before packing the current node" is the same as "just before packing the tree", and for many nodes packing is not called anyway in the fast method. As the rationale behind `before packing node` is to be able to adjust the options of the subtree based on the information gained by packing its constituents, specifying `before packing node` automatically switches to the generic method.




```
\forestset{box/.style={                                          (54)
  draw, no edge, l=0, l sep=1.5ex,
  calign=first, anchor=base west,
  content format={\strut\forestoption{content}},
  if n children=0{}{
    after packing node={
      minimum width/.pgfmath=
        {s("!1")+max_x("!1")-s("!1")-min_x("!1")},
      for children/.wrap pgfmath arg={s+={##1}}{0},
      typeset node}}}}
\begin{forest} for tree={box} [/
  [home[saso[Download][TeX]][alja][joe]]
  [usr[bin][share]]]
\end{forest}
```

→ Remember to typeset or pack the node using `pack'` if you have changed options influencing the typesetting or packing process.

propagator `before computing xy`=⟨keylist⟩ Defers the processing of the ⟨keylist⟩ to until just before the absolute positions of the nodes are computed.

propagator `before drawing tree`=⟨keylist⟩ Defers the processing of the ⟨keylist⟩ to until just before the tree is drawn.

3.4.3 Drawing the tree

This section provides a detailed description of how `draw tree` and friends draw the tree.

First, here's the default course of events. `draw tree` is called from style `draw tree stage` in the context of the formal root node. It does not draw the tree directly, but rather produces Ti*k*Z code that actually does the drawing. The tree-drawing instructions are enclosed in a `tikzpicture` environment and come in three parts: the (non-phantom) nodes are drawn first, followed by edges between the drawn nodes and finally the custom Ti*k*Z code (of all, including phantom nodes). Each of those is drawn for the entire (sub)tree of the current node, in recursive, depth-first parent-first first-child-first order.

Most parts of the tree drawing procedure are customizable. Zooming in from the invocation of `draw tree` to the keys that produce the drawing code, the customization options are as follows.

There are two ways the invocation of `draw tree` can differ from the default. First, `draw tree` can be called within the context of any node. As a first approximation, that node will become the root of the tree that is being drawn; for the whole truth, see `draw tree method`. Second, `draw tree` can be called not only at `draw tree stage`, but any time after the nodes to be drawn have been typeset (see `typeset nodes stage`) and their absolute coordinates (`x` and `y`) computed (see `compute xy stage`).

`begin draw/.code`=⟨toks: TEX code⟩ `\begin {tikzpicture}`

`end draw/.code`=⟨toks: TEX code⟩ `\end {tikzpicture}`

The code produced by `draw tree` is put in the environment specified by `begin draw` and `end draw`. Thus, it is this environment, normally a `tikzpicture`, that does the actual drawing.

A common use of these keys might be to enclose the `tikzpicture` environment in a `center` environment, thereby automatically centering all trees; or, to provide the Ti*k*Z code to execute at the beginning and/or end of the picture.

Note that `begin draw` and `end draw` are *not* node options: they are `\pgfkeys`' code-storing keys [2, §55.4.3–4].

Repeating from (§3.4.1), there are two variants of `draw tree`, which differ in how they use the node boxes created by `typeset nodes`: `draw tree` includes them using `\box`, so they are gone; `draw tree'` uses `\copy`, so they are preserved. Next, setting `draw tree box` will cause the tree to be drawn in the given TEX box.

style `draw tree method`

This is the heart of the tree-drawing procedure: it determines which parts of the tree are drawn and in what order. What this style does by default was already described above, but is actually best seen from the definition itself:

```
draw tree method/.style={
  for nodewalk={
    draw tree nodes processing order/.try,
    draw tree processing order/.retry,
    processing order/.lastretry
  }{draw tree node},
  for nodewalk={
    draw tree edges processing order/.try,
    draw tree processing order/.retry,
    processing order/.lastretry
  }{draw tree edge},
  for nodewalk={
    draw tree tikz processing order/.try,
    draw tree processing order/.retry,
    processing order/.lastretry
  }{draw tree tikz}
},
```

This style may be modified by the user, but it is and should be invoked only within `draw tree`, by the package: *do not execute this style directly!*

The nodewalks occurring in the default definition of this style are, with the exception of `processing order`, not used anywhere else in the package.

nodewalk style `draw tree nodes processing order`

nodewalk style `draw tree edges processing order`

nodewalk style `draw tree tikz processing order`

For each of these nodewalk styles the following holds. If it is defined, it determines which nodes / edges / pieces of `tikz` code are drawn and in which order. If any of these styles is not defined, its function is taken over by `draw tree processing order`. By default, none of them are defined.

nodewalk style `draw tree processing order`

If this nodewalk is defined, it functions as a fallback for node-, edge- and tikz-code-specific nodewalks. If it is not defined (the default situation), it has its own fallback: `processing order` (which defaults to `tree`).

`draw tree node`

`draw tree node'`

Draws the current node at location specified by `x` and `y`. The `'` variant draws the node even if it's `phantom`.

These keys should only be used only within the definition of `draw tree method`.

conditional `if node drawn`=⟨nodewalk⟩⟨true keylist⟩⟨false keylist⟩

Execute ⟨true keylist⟩ if the node at the end of ⟨nodewalk⟩ was already drawn in the current invocation of `draw tree`; otherwise, execute ⟨false keylist⟩.

`draw tree edge`

`draw tree edge'`

Draws the edge from the current node to its parent, using the information in `edge path` and `edge`.

The variant without `'` variant tries to be smart: it draws the edge only if both the current node and its parent have been drawn in the current invocation of `draw tree`. (This prevents drawing the edge from the root node and edges from or to phantom nodes.) The `'` variant is dumb.

These keys should only be used only within the definition of `draw tree method`.

style `draw tree tikz` `draw tree tikz'`

`draw tree tikz'`

Executes the custom code stored in option `tikz` of the current node.

By default, both keys execute the code without performing any checks. Specifically, `tikz` code of phantom nodes is executed. To change this behaviour easily, the user can redefine `draw tree tikz`, which is a style; probably, the definition will employ `draw tree tikz'`. For example, to execute `tikz` code only if the node is not `phantom`, write

```
draw tree tikz/.style={if phantom={draw tree tikz'}{}}
```

These keys should only be used only within the definition of `draw tree method`.

3.5 Node keys

FOREST is mostly controlled using PGF's key management utility `pgfkeys` [2, §55]. Most of the keys can be given next to the content in the bracket representation of a tree (§3.3): we call these *node keys*. Some keys, notably *nodewalk steps* (§3.8), must be used as arguments of specific commands.

Most node keys perform some operation on the *current node*. When the keylist given after the content of a node is processed, the current node is set to that node. However, the current node can be temporarily changed, for example by spatial propagators (§3.5.1) or, more genarally, nodewalks (§3.8).

The most common function that node keys perform is to set or modify an *option* of the current node (§3.6), usually to determine the appearance or position of the node and its edge (§3.7), but there are also several kinds of more exotic keys like spatial (§3.5.1) propagators, which temporarily change the current node, temporal (§3.4.2) propagators, which delay the processing of the keylist until some other stage in the workflow, keys that dynamically create and move nodes (§3.11), keys that control the way FOREST processes the tree (§3.4.1) etc. Finally, users can also define their own keys, either by defining `pgfkeys` styles[18] [2, §55.4.4] or using FOREST's option declaration mechanism (§3.6.3).

→ The style definitions and option declarations given among the other keys in the bracket specification are local to the current tree (but note that FOREST's keylist processing, including temporal and spatial propagation, introduces no groups). To define globally accessible styles and options (well, they are always local to the current TEX group), use macro `\forestset` outside the `forest` environment, e.g. in the preamble of the document. (Although `\forestset`⟨keylist⟩ is currently equivalent to `\pgfkeys{/forest,`⟨keylist⟩`}`, don't rely on this as it will change in some (near) future version of the package, as there is a plan to introduce namespaces . . .)

By default, unknown keys are assumed to be Ti*k*Z keys and are forwarded to `node options`. This behaviour can be changed using `unknown to`.

The following subsections list the node keys which are not described elsewhere (see above): spatial propagators (§3.5.1) and general-purpose node keys, i.e. those which don't deal with tree formatting (§3.5.2).

3.5.1 Spatial propagators

Spatial propagators pass the given ⟨keylist⟩ to other node(s) in the tree.

Spatial propagation does not change the current node: after visiting the nodes the keys are propagated to, a spatial propagator (silently, using a so-called fake step) returns to the origin of the embedded nodewalk.

FOREST provides many spatial propagators. Almost all of them are built from long-form nodewalk steps using prefix `for`. This is why the list below is so short: it only documents this prefix and the exceptions. For the list of nodewalk steps, see §3.8, in particular §3.8.2 for single-step keys and §3.8.3 for multi-step keys.

propagator `for` ⟨step⟩=⟨arg$_1$⟩ . . . ⟨arg$_n$⟩⟨keylist: every-step⟩

propagator `for nodewalk`=⟨nodewalk⟩⟨keylist: every-step⟩

propagator `for Nodewalk`=⟨keylist: config⟩⟨nodewalk⟩⟨keylist: every-step⟩

Walks the (single- or multi-step) ⟨step⟩ from the current node and executes the given ⟨keylist⟩ at every visited node. The current node remains unchanged.

⟨step⟩ must be a long-form nodewalk step. If it has any arguments, they (⟨arg$_1$⟩ . . . ⟨arg$_n$⟩) should be given before every-step ⟨keylist⟩, with two exceptions: embedded nodewalk steps (`Nodewalk` and `nodewalk`) already require the ⟨keylist: every-step⟩ argument, so it should be omitted, as it makes no sense to provide the every-step keylist twice.

Examples:

- `for parent={l sep+=3mm}`
- `for n=2{circle,draw}`
- `for nodewalk={uu2}{blue}`
- `for tree={s sep+=1em}`

[18] Styles are a feature of the `pgfkeys` package. They are named keylists, whose usage ranges from mere abbreviations through templates to devices implementing recursion. To define a style, use PGF's handler `.style` [2, §55.4.4]: ⟨style name⟩`/.style=`⟨keylist⟩.

Here's the big list of all spatial propagators built with prefix `for`: `for -level`, `for -level'`, `for ancestors`, `for branch`, `for branch'`, `for c-commanded`, `for c-commanders`, `for children`, `for children reversed`, `for current`, `for current and ancestors`, `for current and following nodes`, `for current and following siblings`, `for current and following siblings reversed`, `for current and preceding nodes`, `for current and preceding siblings`, `for current and preceding siblings reversed`, `for current and siblings`, `for current and siblings reversed`, `for descendants`, `for descendants breadth-first`, `for descendants breadth-first reversed`, `for descendants children-first`, `for descendants children-first reversed`, `for descendants reversed`, `for filter`, `for first`, `for first leaf`, `for first leaf'`, `for following nodes`, `for following siblings`, `for following siblings reversed`, `for group`, `for id`, `for last`, `for last dynamic node`, `for last leaf`, `for last leaf'`, `for leaves`, `for level`, `for level reversed`, `for level reversed<`, `for , for level<`, `for , for load`, `for max`, `for maxs`, `for min`, `for mins`, `for n`, `for n'`, `for name`, `for next`, `for next leaf`, `for next node`, `for next on tier`, `for Nodewalk`, `for nodewalk`, `for nodewalk'`, `for origin`, `for parent`, `for preceding nodes`, `for preceding siblings`, `for preceding siblings reversed`, `for previous`, `for previous leaf`, `for previous node`, `for previous on tier`, `for relative level`, `for relative level reversed`, `for relative level reversed<`, `for , for relative level<`, `for , for reverse`, `for root`, `for root'`, `for save`, `for save append`, `for save prepend`, `for sibling`, `for siblings`, `for siblings reversed`, `for sort`, `for sort'`, `for to tier`, `for tree`, `for tree breadth-first`, `for tree breadth-first reversed`, `for tree children-first`, `for tree children-first reversed`, `for tree reversed`, `for unique`, `for walk and reverse`, `for walk and save`, `for walk and save append`, `for walk and save prepend`, `for walk and sort`, `for walk and sort'`. For details on nodewalk steps, see §3.8.

propagator `for tree'`=⟨keylist 1⟩⟨keylist 2⟩ A "combination" of `for tree children-first` and `for tree`.

Passes the keylists to the current node and its the descendants. At each node, the ⟨keylist 1⟩ is processed first; then, children are processed recursively; finally, ⟨keylist 2⟩ is processed.

For an example, see the definition of `draw brackets` from `linguistics`.

propagator `for 1`, ..., `for 9`=⟨keylist⟩

propagator `for -1`, ..., `for -9`=⟨keylist⟩

Although `for` normally cannot precede short forms of steps, an exception is made for `1`, ..., `9`. (These keys will work even if the short steps are redefined.)

`for` n passes the ⟨keylist⟩ to the nth child of the current node. `for -`n starts counting at the last child.

`Nodewalk`=⟨keylist: config⟩⟨nodewalk⟩⟨keylist: every-step⟩

Configures and executes the ⟨nodewalk⟩. This key is a nodekey-space copy of nodewalk step `Nodewalk`.

→ Use this key carefully as it can change the current node!

→ The envisioned purpose of this key is to change the current node within the every-step keylist of (an outer) nodewalk, where only node keys are accepted. The config defaults (independent every-step, shared history) are set to facilitate that purpose. But it can also be used as a simple node key, of course.

`node walk`=⟨node walk⟩ **Deprecated!!!** Requires `compat=1.0-nodewalk`. Please use `for nodewalk` in new code. From the old documentation:

> This is the most general way to use a ⟨node walk⟩.
>
> Before starting the ⟨node walk⟩, key `node walk/before walk` is processed. Then, the ⟨step⟩s composing the ⟨node walk⟩ are processed: making a step (normally) changes the current node. After every step, key `node walk/every step` is processed. After the walk, key `node walk/after walk` is processed.
>
> `node walk/before walk`, `node walk/every step` and `node walk/after walk` are processed with `/forest` as the default path: thus, FOREST's node keys can be used normally inside their definitions.

→ Node walks can be tail-recursive, i.e. you can call another node walk from `node walk/after walk` — embedding another node walk in `node walk/before walk` or `node walk/every step` will probably fail, because the three node walk styles are not saved and restored (a node walk doesn't create a TEX group).

→ every step and after walk can be redefined even during the walk. Obviously, redefining before walk during the walk has no effect (in the current walk).

3.5.2 Various

style afterthought=⟨toks⟩ Provides the afterthought explicitely.

This key is normally not used by the end-user, but rather called by the bracket parser. By default, this key is a style defined by `afterthought/.style={tikz+={#1}}`: afterthoughts are interpreted as (cumulative) Ti*k*Z code. If you'd like to use afterthoughts for some other purpose, redefine this style — this will take effect even if you do it in the tree preamble.

autoforward=⟨option⟩⟨keylist⟩, autoforward register=⟨register⟩⟨keylist⟩

autoforward'=⟨option⟩⟨keylist⟩, autoforward register'=⟨register⟩⟨keylist⟩

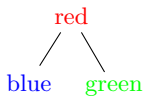
Whenever the value of an autoforwarded option or register is given or changed (via an augmented assignment), ⟨option⟩=⟨new value⟩ or ⟨register⟩=⟨new value⟩ is appended to ⟨keylist⟩. This can be used to "intercept and remember" Ti*k*Z options, like anchor and rotate.

The `autoforward'` variant keeps only a single instance of ⟨option⟩ in ⟨keylist⟩.

If you ever need to use the non-forwarded version of the key, prefix it with word `autoforwarded`, e.g. `autoforwarded rotate`. Autoforwarding is limited to the current TEX group.

Autoforward=⟨option⟩⟨style definition⟩, Autoforward register=⟨register⟩⟨style definition⟩

This is a more generic variant of autoforwarding. After the value of an option or register autoforwarded with this key is changed, the style defined by ⟨style definition⟩ is called with the new option/register value as its argument.




```
\forestset{Autoforward={content}{node options={#1}}}
\begin{forest}
  [red[blue][green]]
\end{forest}
```
(55)

unautoforward=⟨option or register⟩ Undoes the autoforwarding of the option or register made by any of the autoforwarding keys.

content to=⟨key⟩ When parsing the bracket representation of the tree, store the given content using ⟨key⟩=⟨content⟩.

copy command key=⟨pgfkey: source⟩⟨pgfkey: destination⟩

Copies the pgf key in a way that `.add code` and `.add style` handlers still work.

register default preamble=⟨keylist⟩ {}

register preamble=⟨keylist⟩

These registers hold the content of the default preamble and the preamble of the current tree.

`preamble` is set by the bracket parser. Set `default preamble` outside the forest environment using \forestset.

As `default preamble` and `preamble` are not styles but keylist registers, the `#` characters do not need to be doubled: you can freely copy and paste your keylists between the node options of the root node, the preamble and the default preamble. The only difference will be the order of execution: first default preamble, then preamble, and finally the root node's options.

save and restore register=⟨register⟩⟨keylist⟩

Restores the current value of ⟨register⟩ after executing the ⟨keylist⟩.

split=⟨toks⟩⟨separator⟩⟨keylist⟩

split option=⟨option⟩⟨separator⟩⟨keylist⟩

split register=⟨register⟩⟨separator⟩⟨keylist⟩

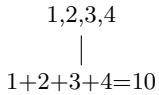
Split ⟨toks⟩ or the value of ⟨option⟩ or ⟨register⟩ at occurrences of ⟨separator⟩ (which must be a single token), and process the keys in ⟨keylist⟩ with the pieces of the split token list as arguments, in the order given.

⟨option⟩ can be either a simple ⟨option name⟩ or a ⟨relative node name⟩`.`⟨option name⟩.

The difference in the number of split values and given keys is handled gracefully. If there is not enough values, the superfluous keys are not processed; if there are too many values, the last key is called repeatedly.

The keys in ⟨keylist⟩ can be any valid keys, including augmented assignments, non-current option assignments, even TeX or user-defined styles. Actually, as `split` works by simply appending `={`⟨current value⟩`}` to the relevant given key, it is possible for the key to be a (sub)keylist ending in a simple, non-valued key, like shown below.

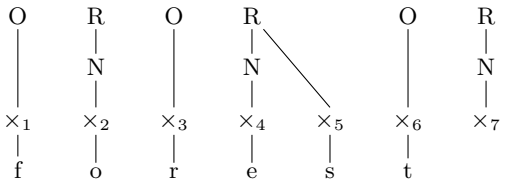
→ Pay attention to `%` characters around the subkeylist. In order for it to actually function as a sublist, its braces should be stripped, but this can only happen if no spaces surround it.




```
\begin{forest}                                                    (56)
  [
    {1,2,3,4}
    [,delay={
      split option=
        {!parent.content}
        {,}
        {
          content',%
          {content+={+},content+}%
        },
        tempcounta'/.process={0+n}{content},
        content+={=},
        content+/.register=tempcounta,
      }
    ]
  ]
\end{forest}
```

TeX=⟨toks: TEX code⟩ The given code is executed immediately.

This can be used for e.g. enumerating nodes:




```
\newcount\xcount                                                  (57)
\begin{forest} GP1,
  delay={TeX={\xcount=0},
    where tier={x}{TeX={\advance\xcount1},
       content/.expanded={##1$_{\the\xcount}$}}{}}
  [
    [O[x[f]]]
    [R[N[x[o]]]]
    [O[x[r]]]
    [R[N[x[e]]][x[s]]]
    [O[x[t]]]
    [R[N[x]]]
  ]
\end{forest}
```

TeX'=⟨toks: TEX code⟩ This key is a combination of keys TeX and TeX'': the given code is both executed and externalized.

TeX''=⟨toks: TEX code⟩ The given code is externalized, i.e. it will be executed when the externalized images are loaded.

The image-loading and `TeX'(')` produced code are intertwined.

typeout=⟨toks⟩ A FOREST version of LATEX macro `\typeout`. Useful for debugging, trust me on this one.

unknown to=⟨key⟩ Forward unknown keys to ⟨key⟩. `node options`

→ Do *not* use handler `.unknown` to deal with unknown keys, as it is used internally by FOREST, and is set up to make it possible to set options of non-current nodes (see §3.6.1).

`unknown key error`=⟨keyval⟩ Produces an error.

Write `unknown to`=`unknown key error` to produce an error when a key unknown to Forest is used.

3.6 Options and registers

Forest introduces two types of data storage: *node options* (or just *options* for short) and *registers*.

Options store data related to particular nodes. Each node has its own set of option values, i.e. the value of an option at some node is independent of its value at other nodes: in particular, setting an option of a node does *not* set this option for the node's descendants. Register values are not associated to nodes.

Note that option and register keys share the same "namespace" (`pgfkeys` path and `pgfmath` function names) so it is not possible to have an option and a register of the same name!

3.6.1 Setting

The simplest way to set the value of an option or a register is to use the key of the same name.

assignment ⟨option⟩=⟨value⟩ Sets the value of ⟨option⟩ of the current node to ⟨value⟩.

Note that option types ⟨keylist⟩ and ⟨autowrapped toks⟩ redefine this basic key.

assignment ⟨register⟩=⟨value⟩ Sets the value of ⟨register⟩ to ⟨value⟩.

Note that register types ⟨keylist⟩ and ⟨autowrapped toks⟩ redefine this basic key.

Options can also be set for the non-current node:

assignment ⟨relative node name⟩.⟨option⟩=⟨value⟩

Sets the value of ⟨option⟩ of the node specified by ⟨relative node name⟩ to ⟨value⟩.

Notes: (i) ⟨value⟩ *is evaluated in the context of the current node.* (ii) In general, the resolution of ⟨relative node name⟩ depends on the current node; see §3.15. (iii) ⟨option⟩ can also be an "augmented assignment operator" (see below) or, indeed, any node key.

Additional keys for setting and modifying the value of an option or a register exist, depending on its data type. Informally, you can think of these keys as *augmented operators* known from various programming languages.

type ⟨toks⟩ contains TeX's ⟨balanced text⟩ [1, 275].

A toks ⟨option⟩ additionally defines the following keys:

augmented assignment ⟨option⟩+=⟨toks⟩ appends the given ⟨toks⟩ to the current value of the option.

augmented assignment +⟨option⟩=⟨toks⟩ prepends the given ⟨toks⟩ to the current value of the option.

type ⟨autowrapped toks⟩ is a subtype of ⟨toks⟩ and contains TeX's ⟨balanced text⟩ [1, 275].

⟨option⟩=⟨toks⟩ of an autowrapped ⟨option⟩ is redefined to ⟨option⟩`/.wrap value`=⟨toks⟩ of a normal ⟨toks⟩ option.

Keyvals ⟨option⟩+=⟨toks⟩ and +⟨option⟩=⟨toks⟩ are redefined to ⟨option⟩+`/.wrap value`=⟨toks⟩ and +⟨option⟩`/.wrap value`=⟨toks⟩, respectively. The normal toks behaviour can be accessed via keys ⟨option⟩', ⟨option⟩+', and +⟨option⟩'.

type ⟨keylist⟩ is a subtype of ⟨toks⟩ and contains a comma-separated list of ⟨key⟩[=⟨value⟩] pairs.

Augmented assignment operators ⟨option⟩+ and +⟨option⟩ automatically insert a comma before/after the appended/prepended material.

Augmented assignment operator ⟨option⟩-=⟨keylist⟩ deletes the keys from keylist ⟨option⟩. ⟨keylist⟩ specifies which keys to delete. If a key is given no value, all occurrences of that key will be deleted. If a key is given a value, only occurrences with that value will be deleted. To delete occurrences without value, use special value `\forestnovalue`. (Note: if you include a key in ⟨keylist⟩ more than once, only the last occurrence counts.)

⟨option⟩=⟨keylist⟩ of a keylist option is redefined to ⟨option⟩+=⟨keylist⟩. In other words, keylists behave additively by default. The rationale is that one usually wants to add keys to a keylist. The usual, non-additive behaviour can be accessed by ⟨option⟩'=⟨keylist⟩.

Manipulating the keylist option using augmented assignments might have the side-effect of adding an empty key to the list.

type ⟨dimen⟩ contains a dimension.

The value given to a dimension option is automatically evaluated by `pgfmath`. In other words, ⟨option⟩=⟨value⟩ is implicitly understood as ⟨option⟩`/.pgfmath=`⟨value⟩.

For a ⟨dimen⟩ option ⟨option⟩, the following additional keys ("augmented assignments") are defined:

augmented assignment ⟨option⟩`+=`⟨value⟩ is equivalent to ⟨option⟩=⟨option⟩`()+`⟨value⟩

augmented assignment ⟨option⟩`-=`⟨value⟩ is equivalent to ⟨option⟩=⟨option⟩`()-`⟨value⟩

augmented assignment ⟨option⟩`*=`⟨value⟩ is equivalent to ⟨option⟩=⟨option⟩`()*`⟨value⟩

augmented assignment ⟨option⟩`:=`⟨value⟩ is equivalent to ⟨option⟩=⟨option⟩`()/`⟨value⟩

The evaluation of ⟨pgfmath⟩ can be quite slow. There are two tricks to speed things up *if* the ⟨pgfmath⟩ expression is simple, i.e. just a TEX ⟨dimen⟩:

1. `pgfmath` evaluation of simple values can be sped up by prepending `+` to the value [2, §62.1];
2. use the key ⟨option⟩`'=`⟨value⟩ to invoke a normal TEX assignment.

The two above-mentioned speed-up tricks work for the augmented assignments as well. The keys for the second, TEX-only trick are: `'+`, `'-`, `'*`, `':` — note that for the latter two, the value should be an integer.

type ⟨count⟩ contains an integer.

The additional keys and their behaviour are the same as for the ⟨dimen⟩ options.

type ⟨boolean⟩ contains 0 (false) or 1 (true).

In the general case, the value given to a ⟨boolean⟩ option is automatically parsed by `pgfmath` (just as for ⟨count⟩ and ⟨dimen⟩): if the computed value is non-zero, 1 is stored; otherwise, 0 is stored. Note that `pgfmath` recognizes constants `true` and `false`, so it is possible to write ⟨option⟩`=true` and ⟨option⟩`=false`.

If key ⟨option⟩ is given no argument, pgfmath evaluation does not apply and a true value is set. To quickly set a false value, use key `not` ⟨option⟩ (with no arguments).

3.6.2 Reading

Option and register values can be accessed using the four macros listed below, handlers `.option` and `.register` (§3.12) and `pgfmath` functions (3.18).

macro `\forestoption{`⟨option⟩`}`

macro `\foresteoption{`⟨option⟩`}`

macro `\forestregister{`⟨register⟩`}`

macro `\foresteregister{`⟨register⟩`}`

These macros expand to the value of the given option or register. Note that `\forestoption` and `\foresteoption` expand to the value of the given option of the *current node*; to access option values of a non-current node, use `pgfmath` functions.

In the context of `\edef`, `\forestoption` and `\forestregister` expand precisely to the token list of the option value, while `\foresteoption` and `\foresteregister` fully expand the value.

→ These macros can be useful in TEX code introduced by `TeX` or PGF's handler `.expanded` [2, §55.4.6].

3.6.3 Declaring

Using the following keys, users can also declare their own options and registers. The new options and registers will behave exactly like the predefined ones.

Note that the declaration of an option must provide a default value, while the declaration of a register must not do that (registers are initialized to the empty string, `0pt` or `0`, as appropriate for the type). The default value of an option will be assigned to any newly created nodes; the existing nodes are not affected.

`declare toks=`⟨option name⟩⟨default value⟩ Declares a ⟨toks⟩ option.

`declare autowrapped toks=`⟨option name⟩⟨default value⟩ Declares an ⟨autowrapped toks⟩ option.

declare keylist=⟨option name⟩⟨default value⟩ Declares a ⟨keylist⟩ option.

declare dimen=⟨option name⟩⟨default value⟩ Declares a ⟨dimen⟩ option. The default value is processed by ⟨forestmath⟩.

declare count=⟨option name⟩⟨default value⟩ Declares a ⟨count⟩ option. The default value is processed by ⟨forestmath⟩.

declare boolean=⟨option name⟩⟨default value⟩ Declares a ⟨boolean⟩ option. The default value is processed by ⟨forestmath⟩.

declare toks register=⟨register name⟩ Declares a ⟨toks⟩ register.

declare autowrapped toks register=⟨register name⟩ Declares an ⟨autowrapped toks⟩ register.

declare keylist register=⟨register name⟩ Declares a ⟨keylist⟩ register.

declare dimen register=⟨register name⟩ Declares a ⟨dimen⟩ register.

declare count register=⟨register name⟩ Declares a ⟨count⟩ register.

declare boolean register=⟨register name⟩ Declares a ⟨boolean⟩ register.

Several scratch registers are predefined:

register temptoksa, temptoksb, temptoksc, temptoksd Predefined ⟨toks⟩ registers.

register tempkeylista, tempkeylistb, tempkeylistc, tempkeylistd Predefined ⟨keylist⟩ registers.

register tempdima, tempdimb, tempdimc, tempdimd, tempdimx, tempdimy, tempdiml, tempdims, tempdimxa, tempdimya, tempdimla, tempdimsa, tempdimxb, tempdimyb, tempdimlb, tempdimsb Predefined ⟨dimen⟩ registers.

register tempcounta, tempcountb, tempcountc, tempcountd Predefined ⟨count⟩ registers.

register tempboola, tempboolb, tempboolc, tempboold Predefined ⟨boolean⟩ registers.

3.7 Formatting the tree

3.7.1 Node appearance

The following options apply at stage typeset nodes. Changing them afterwards has no effect in the normal course of events.

option align=left | center | right | ⟨toks: tabular header⟩ {}

Creates a left/center/right-aligned multiline node, or a tabular node. In the content option, the lines of the node should separated by \\ and the columns (if any) by &, as usual.

The vertical alignment of the multiline/tabular node can be specified by option base.

special value	actual value
left	@{}l@{}
center	@{}c@{}
right	@{}r@{}

top base
right aligned

left aligned
bottom base

```
\begin{forest} l sep+=2ex
  [special value&actual value\\\hline
    \indexdef{value of=align>left}&||\texttt{@\{\}l@\{\}}\\
    \indexdef{value of=align>center}&||\texttt{@\{\}c@\{\}}\\
    \indexdef{value of=align>right}&||\texttt{@\{\}r@\{\}}\\
    ,align=ll,draw
    [top base\\right aligned, align=right,base=top]
    [left aligned\\bottom base, align=left,base=bottom]
  ]
\end{forest}
```
(58)

Internally, setting this option has two effects:

1. The option value (a tabular environment header specification) is set. The special values left, center and right invoke styles setting the actual header to the value shown in the above example.

→ If you know that the align was set with a special value, you can easily check the value using if in align.

2. Option content format is set to the following value:

```
\noexpand\begin{tabular}[\forestoption{base}]{\forestoption{align}}%
  \forestoption{content}%
\noexpand\end{tabular}%
```

As you can see, it is this value that determines that options base, align and content specify the vertical alignment, header and content of the table.

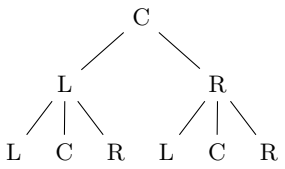
option base=⟨toks: vertical alignment⟩ t

This option controls the vertical alignment of multiline (and in general, tabular) nodes created with align. Its value becomes the optional argument to the tabular environment. Thus, sensible values are t (the top line of the table will be the baseline) and b (the bottom line of the table will be the baseline). Note that this will only have effect if the node is anchored on a baseline, like in the default case of anchor=base.

For readability, you can use top and bottom instead of t and b. (top and bottom are still stored as t and b.)

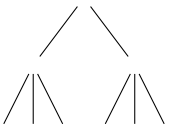
option content=⟨autowrapped toks⟩ The content of the node. {}

Normally, the value of option content is given implicitely by virtue of the special (initial) position of content in the bracket representation (see §3.3). However, the option also be set explicitely, as any other option.




```
\begin{forest}                                          (59)
  delay={for tree={
      if n=1{content=L}
           {if n'=1{content=R}
                   {content=C}}}}
  [[[][]][][[][][]]]
\end{forest}
```

Note that the execution of the content option should usually be delayed: otherwise, the implicitely given content (in the example below, the empty string) will override the explicitely given content.



```
\begin{forest}                                          (60)
  for tree={
      if n=1{content=L}
           {if n'=1{content=R}
                   {content=C}}}
  [[[][]][][[][][]]]
\end{forest}
```

option content format=⟨toks⟩ \forestoption {content}

When typesetting the node under the default conditions (see option node format), the value of this option is passed to the Ti*k*Z node operation as its ⟨text⟩ argument [2, §16.2]. The default value of the option simply puts the content in the node.

This is a fairly low level option, but sometimes you might still want to change its value. If you do so, take care of what is expanded when. Most importantly, if you use a formatting command such as \textbf in the default setting of node format, be sure to precede it with \noexpand. For details, read the documentation of option node format and macros \forestoption and \foresteoption; for an example, see option align.

math content Changes content format so that the content of the node will be typeset in a math environment.

plain content Resets content format to the default value.

option node format=⟨toks⟩ \noexpand\node(\forestoption{name})
[\forestoption{node options}]{\foresteoption{content format}};

The node is typeset by executing the expansion of this option's value in a tikzpicture environment.

Important: the value of this option is first expanded using `\edef` and only then executed. Note that in its default value, `content format` is fully expanded using `\foresteoption`: this is necessary for complex content formats, such as `tabular` environments.

This is a low level option. Ideally, there should be no need to change its value. If you do, note that the Ti*k*Z node you create should be named using the value of option `name`; otherwise, parent–child edges can't be drawn, see option `edge path`.

`node format'`=⟨toks⟩

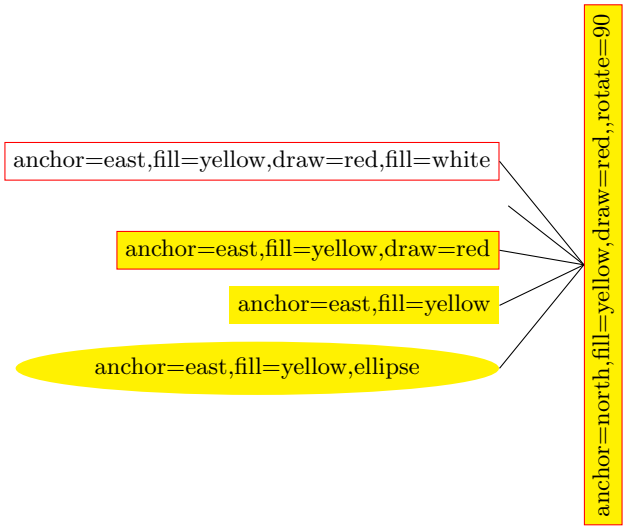
Sets `node format`, automatically wrapping the given ⟨toks⟩ by `\noexpand\node(\forestoption{name})` and `;`. Only the node options and content must therefore be given.

option `node options`=⟨keylist⟩ `anchor=base`

When the node is being typeset under the default conditions (see option `node format`), the content of this option is passed to Ti*k*Z as options to the Ti*k*Z `node` operation [2, §16].

This option is rarely manipulated manually: almost all options unknown to Forest are automatically appended to `node options`. Exceptions are (i) `label` and `pin`, which require special attention in order to work; and (ii) `anchor`, which is saved in order to retain the information about the selected anchor.

(61)

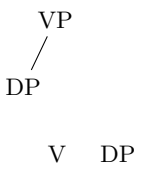



```
\begin{forest}
  for descendants={anchor=east,child anchor=east},
  grow=west,anchor=north,parent anchor=north,
  l sep=1cm,
  for tree={fill=yellow},where={n()<=3}{draw=red}{},
  delay={for tree={content/.pgfmath=node_options}}
  [root,rotate=90,
    [,fill=white]
    [,node options']
    []
    []
    [,node options={ellipse}]
  ]
\end{forest}
```

option `phantom`=⟨boolean⟩ `false`

A phantom node and its surrounding edges are taken into account when packing, but not drawn. (This option applies in stage `draw tree`.)

(62)




```
\begin{forest}
  [VP[DP][V',phantom[V][DP]]]
\end{forest}
```

3.7.2 Node position

Most of the following options apply at stage `pack`. Changing them afterwards has no effect in the normal course of events. (Options `l`, `s`, `x`, `y` and `anchor` are exceptions; see their documentation for details).

option `anchor`=⟨toks: Forest anchor⟩ `base`

While this option is saved by Forest, it is essentially an option of Ti*k*Z's `\node` command [see 2, §16.5.1]. Forest `autoforward`s it to keylist option `node options`, which is passed on to Ti*k*Z's `\node` command when the node is typeset. (Option `anchor` thus normally applies in stage `typeset nodes`.)

In the Ti*k*Z code, you can refer to the node's anchor using Forest's anchor `anchor`; this anchor is sometimes also called the node anchor in this documentation, to distinguish it clearly from parent and child anchors.

⟨toks: FOREST anchor⟩ can be any Ti*k*Z anchor. Additionally, FOREST defines several tree hierarchy related anchors; for details, see §3.17.

The effect of setting the node anchor is twofold:

- during packing, the anchors of all siblings are `l`-aligned;
- some `calign` methods use node anchors (of the parent and/or certain children) to `s`-align the block of children to the parent.

option `calign=child|child edge|midpoint|edge midpoint|fixed angles|fixed edge angles` `center`
`first|last|center`.

The packing algorithm positions the children so that they don't overlap, effectively computing the minimal distances between the node anchors of the children. This option (`calign` stands for child alignment) specifies how the children are positioned with respect to the parent (while respecting the above-mentioned minimal distances).

The child alignment methods refer to the primary and the secondary child, and to the primary and the secondary angle. These are set using the keys described just after `calign`.

`calign=child` s-aligns the node anchors of the parent and the primary child.

`calign=child edge` s-aligns the parent anchor of the parent and the child anchor of the primary child.

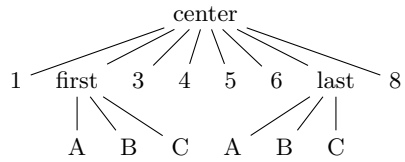
`calign=first` is an abbreviation for `calign=child,calign child=1`.

`calign=last` is an abbreviation for `calign=child,calign child=-1`.

`calign=midpoint` s-aligns the parent's node anchor and the midpoint between the primary and the secondary child's node anchor.

`calign=edge midpoint` s-aligns the parent's parent anchor and the midpoint between the primary and the secondary child's child anchor.

`calign=center` is an abbreviation for
`calign=midpoint, calign primary child=1, calign secondary child=-1`.



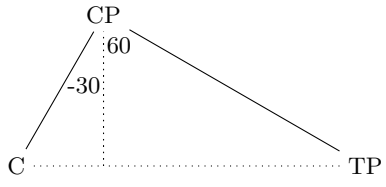

```
\begin{forest}                                                    (63)
  [center,calign=center[1]
    [first,calign=first[A][B][C]][3][4][5][6]
    [last,calign=last[A][B][C]][8]]
\end{forest}
```

`calign=fixed angles`: The angle between the direction of growth at the current node (specified by option `grow`) and the line through the node anchors of the parent and the primary/secondary child will equal the primary/secondary angle.

To achieve this, the block of children might be spread or further distanced from the parent.

`calign=fixed edge angles`: The angle between the direction of growth at the current node (specified by option `grow`) and the line through the parent's parent anchor and the primary/secondary child's child anchor will equal the primary/secondary angle.

To achieve this, the block of children might be spread or further distanced from the parent.




```
\begin{forest}                                                    (64)
  calign=fixed edge angles,
  calign primary angle=-30,calign secondary angle=60,
  for tree={l=2cm}
  [CP[C][TP]]
  \draw[dotted] (!1) -| coordinate(p) () (!2) -| ();
  \path ()--(p) node[pos=0.4,left,inner sep=1pt]{-30};
  \path ()--(p) node[pos=0.1,right,inner sep=1pt]{60};
\end{forest}
```

`calign child`=⟨count⟩ is an abbreviation for `calign primary child`=⟨count⟩.

option `calign primary child`=⟨count⟩ Sets the primary child. (See `calign`.) 1

⟨count⟩ is the child's sequence number. Negative numbers start counting at the last child.

option `calign secondary child`=⟨count⟩ Sets the secondary child. (See `calign`.) `-1`

⟨count⟩ is the child's sequence number. Negative numbers start counting at the last child.

`calign angle`=⟨count⟩ is an abbreviation for: `calign primary angle`=-⟨count⟩, `calign secondary angle`=⟨count⟩.

option `calign primary angle`=⟨count⟩ Sets the primary angle. (See `calign`.) `-35`

option `calign secondary angle`=⟨count⟩ Sets the secondary angle. (See `calign`.) `35`

`calign with current` s-aligns the node anchors of the current node and its parent. This key is an abbreviation for:

```
for parent/.wrap pgfmath arg={calign=child,calign primary child=##1}{n}.
```

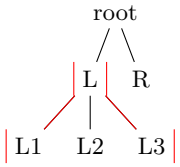
`calign with current edge` s-aligns the child anchor of the current node and the parent anchor of its parent. This key is an abbreviation for:

```
for parent/.wrap pgfmath arg={calign=child edge,calign primary child=##1}{n}
```

option `fit`=`tight`|`rectangle`|`band` `tight`

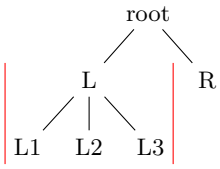
This option sets the type of the (s-)boundary that will be computed for the subtree rooted in the node, thereby determining how it will be packed into the subtree rooted in the node's parent. There are three choices:[19]

- `fit=tight`: an exact boundary of the node's subtree is computed, resulting in a compactly packed tree. Below, the boundary of subtree L is drawn.



(65)
```
\begin{forest}
  delay={for tree={name/.pgfmath=content}}
  [root
    [L,fit=tight, % default
       show boundary
      [L1][L2][L3]]
    [R]
  ]
\end{forest}
```

- `fit=rectangle`: puts the node's subtree in a rectangle and effectively packs this rectangle; the resulting tree will usually be wider.

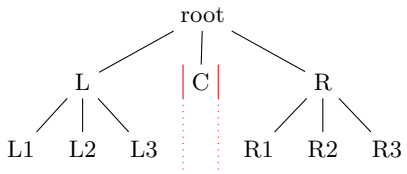


(66)
```
\begin{forest}
  delay={for tree={name/.pgfmath=content}}
  [root
    [L,fit=rectangle,
       show boundary
      [L1][L2][L3]]
    [R]
  ]
\end{forest}
```

- `fit=band`: puts the node's subtree in a rectangle of "infinite depth": the space under the node and its descendants will be kept clear.

[19] Below is the definition of style `show boundary`. The `use path` trick is adjusted from TEX Stackexchange question Calling a previously named path in tikz.

```
\makeatletter\tikzset{use path/.code={\tikz@addmode{\pgfsyssoftpath@setcurrentpath#1}
  \appto\tikz@preactions{\let\tikz@actions@path#1}}}\makeatother
\forestset{show boundary/.style={
  before drawing tree={get min s tree boundary=\minboundary, get max s tree boundary=\maxboundary},
  tikz+={\draw[red,use path=\minboundary]; \draw[red,use path=\maxboundary];}}}
```




```
\begin{forest}                                                    (67)
  delay={for tree={name/.pgfmath=content}}
  [root
    [L[L1][L2][L3]]
    [C,fit=band]
    [R[R1][R2][R3]]
  ]
  \draw[thin,red]
    (C.south west)--(C.north west)
    (C.north east)--(C.south east);
  \draw[thin,red,dotted]
    (C.south west)--+(0,-1)
    (C.south east)--+(0,-1);
\end{forest}
```

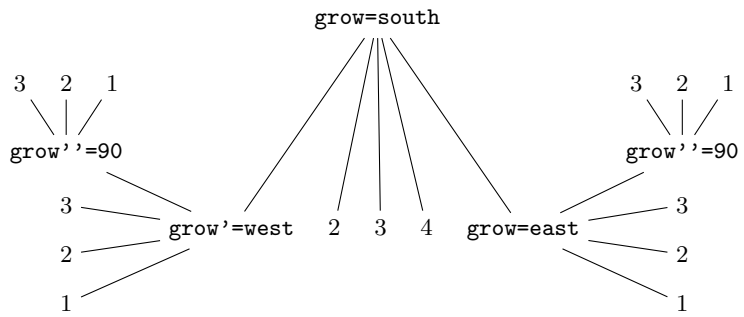
option grow=⟨count⟩, grow'=⟨count⟩, grow''=⟨count⟩ 270

The direction of the tree's growth at the node.

The growth direction is understood as in Ti*k*Z's tree library [2, §18.5.2] when using the default growth method: the (node anchor's of the) children of the node are placed on a line orthogonal to the current direction of growth. (The final result might be different, however, if `l` is changed after packing or if some child undergoes tier alignment.)

This option is essentially numeric (`pgfmath` function `grow` will always return an integer), but there are some twists. The growth direction can be specified either numerically or as a compass direction (`east`, `north east`, ...). Furthermore, like in Ti*k*Z, setting the growth direction using key `grow` additionally sets the value of option `reversed` to `false`, while setting it with `grow'` sets it to `true`; to change the growth direction without influencing `reversed`, use key `grow''`.

Between stages `pack` and `compute xy`, the value of `grow` should not be changed.




```
\begin{forest}                                                    (68)
  delay={where in content={grow}{
      for current/.pgfmath=content,
      content=\texttt{#1}
    }{}
  }
  [{grow=south}
    [{grow'=west}[1][2][3]
      [{grow''=90}[1][2][3]]]
    [2][3][4]
    [{grow=east}[1][2][3]
      [{grow''=90}[1][2][3]]]]
\end{forest}
```

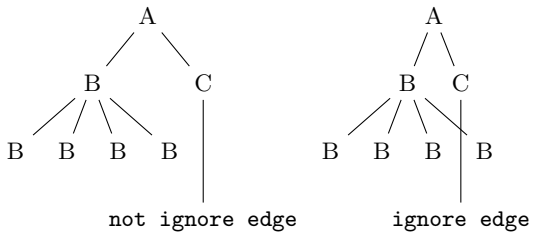
option ignore=⟨boolean⟩ false

If this option is set, the packing mechanism ignores the node, i.e. it pretends that the node has no boundary. Note: this only applies to the node, not to the tree.

Maybe someone will even find this option useful for some reason ...

option ignore edge=⟨boolean⟩ false

If this option is set, the packing mechanism ignores the edge from the node to the parent, i.e. nodes and other edges can overlap it. (See §6.2 for some problematic situations.)




```
\begin{forest}                                                    (69)
  [A[B[B][B][B][B]][C
    [\texttt{not ignore edge},l*=2]]]
\end{forest}
\begin{forest}
  [A[B[B][B][B][B]][C
    [\texttt{ignore edge},l*=2,ignore edge]]]
\end{forest}
```

option l=⟨dimen⟩ The l-position of the node, in the parent's ls-coordinate system. (The origin of a node's ls-coordinate system is at its (node) anchor. The l-axis points in the direction of the tree growth at the node, which is given by option grow. The s-axis is orthogonal to the l-axis; the positive side is in the counter-clockwise direction from l axis.)

The initial value of l is set from the standard node. By default, it equals:

$$\texttt{l sep} + 2 \cdot \texttt{outer ysep} + \text{total height(standard node)}$$

The value of l can be changed at any point, with different effects.

- The value of l at the beginning of stage pack determines the minimal l-distance between the anchors of the node and its parent. Thus, changing l before packing will influence this process. (During packing, l can be increased due to parent's l sep, tier alignment, or calign methods fixed angles and fixed edge angles.
- Changing l after packing but before stage compute xy will result in a manual adjustment of the computed position. (The augmented assignment operators can be useful here.)
- Changing l after the absolute positions have been computed has no effect in the normal course of events.

option l sep=⟨dimen⟩ The minimal l-distance between the node and its descendants.

This option determines the l-distance between the *boundaries* of the node and its descendants, not node anchors. The final effect is that there will be a l sep wide band, in the l-dimension, between the node and all its descendants.

The initial value of l sep is set from the standard node and equals

$$\text{height(strut)} + \texttt{inner ysep}$$

Note that despite the similar name, the semantics of l sep and s sep are quite different.

option reversed=⟨boolean⟩ false

If false, the children are positioned around the node in the counter-clockwise direction; if true, in the clockwise direction. See also grow.

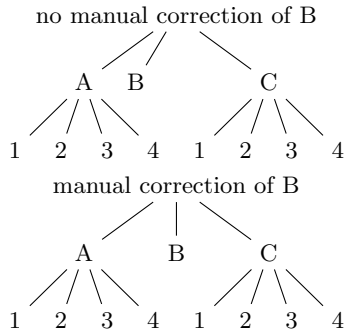
option rotate=⟨count⟩ 0

This option is saved and autoforwarded to Ti*k*Z's \node command via node options.

option s=⟨dimen⟩ The s-position of the node, in the parent's ls-coordinate system. (The origin of a node's ls-coordinate system is at its (node) anchor. The l-axis points in the direction of the tree growth at the node, which is given by option grow. The s-axis is orthogonal to the l-axis; the positive side is in the counter-clockwise direction from l axis.)

The value of s is computed in stage pack stage and used in stage compute xy stage, so it only makes sense to (inspect and) change it in before computing xy and during packing (before packing node and after packing node). *Any value given before packing is overridden, and changing the value after computing xy has no effect.*

For example, consider the manual correction below. By default, B is closer to A than C because packing proceeds from the first to the last child — the position of B would be the same if there was no C. Adjusting s at the right moment, it is easy to center B between A and C.




```
\begin{forest}
  [no manual correction of B
    [A[1][2][3][4]]
    [B]
    [C[1][2][3][4]]
  ]
\end{forest}
```

(70)

```
 \begin{forest}
  [manual correction of B
    [A[1][2][3][4]]
    [B,before computing xy={s/.average={s}{siblings}}]
    [C[1][2][3][4]]
  ]
\end{forest}
```

option **s sep**=⟨dimen⟩

The subtrees rooted in the node's children will be kept at least `s sep` apart in the s-dimension. Note that `s sep` is about the minimal distance between node *boundaries*, not node anchors.

The initial value of `s sep` is set from the standard node and equals 2 · `inner xsep`.

Note that despite the similar name, the semantics of `s sep` and `l sep` are quite different.

option **tier**=⟨toks⟩ {}

Setting this option to something non-empty "puts a node on a tier." All the nodes on the same tier are aligned in the l-dimension.

Tier alignment across changes in growth direction is impossible. In the case of incompatible options, FOREST will yield an error.

Tier alignment also does not work well with `calign=fixed angles` and `calign=fixed edge angles`, because these child alignment methods may change the l-position of the children. When this might happen, FOREST will yield a warning.

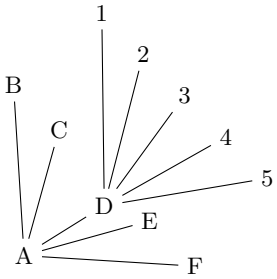
option **x**=⟨dimen⟩

option **y**=⟨dimen⟩

`x` and `y` are the coordinates of the node in the "normal" (paper) coordinate system, relative to the root of the tree that is being drawn. So, essentially, they are absolute coordinates.

The values of `x` and `y` are computed in stage `compute xy`. It only makes sense to inspect and change them (for manual adjustments) afterwards (normally, in the `before drawing tree` hook, see §3.4.1.) `x` and `y` of the (formal) root node are exceptions, as they are not changed in stage `compute xy`.

(71)




```
\begin{forest}
  for tree={grow'=45,l=1.5cm}
  [A[B][C][D,before drawing tree={y-=4mm}[1][2][3][4][5]][E][F]]
\end{forest}
```

3.7.3 Edges

These options determine the shape and position of the edge from a node to its parent. They apply at stage `draw tree`.

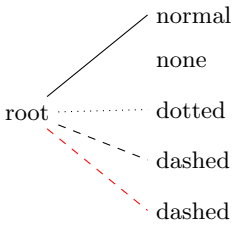
option **child anchor**=⟨toks: FOREST anchor⟩ See `parent anchor`. {}

option **edge**=⟨keylist⟩ `draw`

When `edge path` has its default value, the value of this option is passed as options to the TikZ `\path` expression used to draw the edge between the node and its parent.

Also see key `no edge`.

(72)

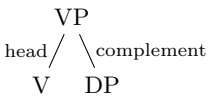



```
\begin{forest} for tree={grow'=0,l=2cm,anchor=west,child anchor=west},
  [root
    [normal]
    [none,no edge]
    [dotted,edge=dotted]
    [dashed,edge=dashed]
    [dashed,edge={dashed,red}]
  ]
\end{forest}
```

option **edge label**=⟨toks: TikZ code⟩ {}

When `edge path` has its default value, the value of this option is used at the end of the edge path specification to typeset a node (or nodes) along the edge.

The packing mechanism is not sensitive to edge labels.




```
\begin{forest}                                                          (73)
  [VP
    [V,edge label={node[midway,left,font=\scriptsize]{head}}]
    [DP,edge label={node[midway,right,font=\scriptsize]{complement}}]
  ]
\end{forest}
```

option edge path=⟨toks: Ti*k*Z code⟩ `\noexpand\path[\forestoption{edge}] (!u.parent anchor)--(.child anchor)\forestoption{edge label};`

This option contains the code that draws the edge from the node to its parent. By default, it creates a path consisting of a single line segment between the node's child anchor and its parent's parent anchor. Options given by edge are passed to the path; by default, the path is simply drawn. Contents of edge label are used to potentially place a node (or nodes) along the edge.

When specifying the edge path, the values of options edge and edge label can be used. Furthermore, two anchors, parent anchor and child anchor, are defined, to facilitate access to options parent anchor and child anchor from the Ti*k*Z code.

The node positioning algorithm is sensitive to edges, i.e. it will avoid a node overlapping an edge or two edges overlapping. However, the positioning algorithm always behaves as if the `edge path` had the default value — *changing the* ***edge path*** *does not influence the packing!* Sorry. (Parent–child edges can be ignored, however: see option ignore edge.)

edge path'=⟨toks: Ti*k*Z code⟩

Sets edge path, automatically wrapping the given path by `\noexpand\path[\forestoption{edge}]` and `\forestoption{edge label};`.

option parent anchor=⟨toks: Forest anchor⟩ (Information also applies to option child anchor.) `{}`

Forest defines anchors parent anchor and child anchor (which work only for Forest and not also Ti*k*Z nodes, of course) to facilitate reference to the desired endpoints of child–parent edges. Whenever one of these anchors is invoked, it looks up the value of the `parent anchor` or `child anchor` of the node named in the coordinate specification, and forwards the request to the (Ti*k*Z) anchor given as the value.

The intended use of the two anchors is chiefly in edge path specification, but they can used in any Ti*k*Z code.


```
\begin{forest}                                                          (74)
  for tree={parent anchor=south,child anchor=north}
  [VP[V][DP]]
  \path[fill=red] (.parent anchor) circle[radius=2pt];
  \path[fill=blue] (!1.child anchor) circle[radius=2pt]
                   (!2.child anchor) circle[radius=2pt];
\end{forest}
```

The empty value (which is the default) is interpreted as in Ti*k*Z: as an edge to the appropriate border point. See also §3.17 for a list of additional anchors defined by Forest.

no edge Clears the edge options (edge'={}) and sets ignore edge.

3.7.4 Information about node

The values of these options provide various information about the tree and its nodes.

alias=⟨toks⟩
alias'=⟨toks⟩ Sets the alias for the node's name.

Unlike name, `alias` is *not* an option: you cannot e.g. query it's value via a `pgfmath` expression.

If the given alias clashes with an existing node name, `alias` will yield an error, while `alias'` will silently rename the node with this name to its default value (`node@`⟨id⟩).

Aliases can be used as the ⟨forest node name⟩ part of a relative node name and as the argument to the name step of a node walk. The latter includes the usage as the argument of the for name propagator.

Technically speaking, Forest alias is *not* a Ti*k*Z alias! However, you can still use it as a "node name" in Ti*k*Z coordinates, since Forest hacks Ti*k*Z's implicit node coordinate system to accept relative node names; see §3.16.

readonly option `id`=⟨count⟩ The internal id of the node.

readonly option `level`=⟨count⟩ The hierarchical level of the node. The root is on level 0.

readonly option `max x`=⟨dimen⟩

readonly option `max y`=⟨dimen⟩

readonly option `min x`=⟨dimen⟩

readonly option `min y`=⟨dimen⟩ Measures of the node, in the shape's coordinate system [see 2, §16.2,§48,§75] shifted so that the node anchor is at the origin.

In `pgfmath` expressions, these options are accessible as `max_x`, `max_y`, `min_x` and `min_y`.

readonly option `n`=⟨count⟩ The child's sequence number in the list of its parent's children.

The enumeration starts with 1. For a geometric root, `n` equals 0.

readonly option `n'`=⟨count⟩ Like `n`, but starts counting at the last child.

In `pgfmath` expressions, this option is accessible as `n_`.

option `name`=⟨toks⟩ `node@`⟨id⟩

`name'`=⟨toks⟩ Sets the name of the node.

The expansion of ⟨toks⟩ becomes the ⟨forest node name⟩ of the node. The Ti*k*Z node created from the Forest node will get the name specified by this option.

Node names must be unique. If a node with the given name already exists, `name` will yield an error, while `name'` will silently rename the node with this name to its default (`node@`⟨id⟩) value. Use an empty argument to reset the node's name to its default value.

readonly option `n children`=⟨count⟩ The number of children of the node.

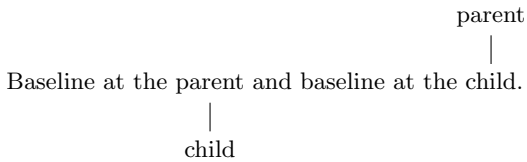
In `pgfmath` expressions, this option is accessible as `n_children`.

3.7.5 Various

`baseline` The node's anchor becomes the baseline of the whole tree [cf. 2, §69.3.1].

In plain language, when the tree is inserted in your (normal TEX) text, it will be vertically aligned to the anchor of the current node.

Behind the scenes, this style sets the alias of the current node to `forest@baseline@node`.




```
{\tikzexternaldisable                                    (75)
Baseline at the
\begin{forest}
  [parent,baseline,use as bounding box'
    [child]]
\end{forest}
and baseline at the
\begin{forest}
  [parent
    [child,baseline,use as bounding box']]
\end{forest}.}
```

tikz key `/tikz/fit to`=⟨nodewalk⟩ Fits the Ti*k*Z node to the nodes in the given ⟨nodewalk⟩.

This key should be used like `/tikz/fit` of the Ti*k*Z's fitting library [see 2, §34]: as an option to Ti*k*Z*'s* `node` operation, the obvious restriction being that `fit to` must be used in the context of some Forest node. For an example, see footnote 8.

This key works by calling `/tikz/fit` and providing it with the the coordinates of the subtree's boundary.

The ⟨nodewalk⟩ inherits its history from the outer nodewalk (if there is one). Its every-step keylist is empty.

`get min s tree boundary`=⟨cs⟩

`get max s tree boundary`=⟨cs⟩

Puts the boundary computed during the packing process into the given ⟨cs⟩. The boundary is in the form of PGF path. The `min` and `max` versions give the two sides of the node. For an example, see how the boundaries in the discussion of `fit` were drawn.

option `label`=⟨toks: Ti*k*Z node⟩ The current node is labelled by a Ti*k*Z node.

The label is specified as a Ti*k*Z option `label` [2, §16.10]. Technically, the value of this option is passed to Ti*k*Z's as a late option [2, §16.14]. (This is so because FOREST must first typeset the nodes separately to measure them (stage `typeset nodes`); the preconstructed nodes are inserted in the big picture later, at stage `draw tree`.) Another option with the same technicality is `pin`.

option `pin`=⟨toks: Ti*k*Z node⟩ The current node gets a pin, see [2, §16.10]. The technical details are the same as for `label`.

`use as bounding box` The current node's box is used as a bounding box for the whole tree.

`use as bounding box'` Like `use as bounding box`, but subtracts the (current) inner and outer sep from the node's box. For an example, see `baseline`.

option `tikz`=⟨toks: Ti*k*Z code⟩ "Decorations." `{}`

The code given as the value of this option will be included in the `tikzpicture` environment used to draw the tree. By default, the code is included after all nodes of the tree have been drawn, so it can refer to any node of the tree (furthermore, relative node names can be used to refer to nodes of the tree, see §3.15) and the code given to various nodes is appended in a depth-first, parent-first fashion. See §3.4.3 for details and customization.

By default, bracket parser's afterthoughts feed the value of this option. See `afterthought`.

3.8 Nodewalks

A *nodewalk* is a sequence of *steps* describing a path through the tree. Most steps are defined relative to the current node, for example `parent` steps to the parent of the current node, and `n=2` steps to the second child of the current node, where "to make a step" means to change the current node. Thus, nodewalk `parent, parent, n=2` describes the path which first steps to the parent of the *origin* node, then to its grandparent and finally to the second child of the origin's grandparent.

The origin of the nodewalk depends on how the nodewalk is invoked. When used after the ! in a relative node name (§3.15), the origin is the node with the name given before !; when invoked by a spatial propagator such as `for nodewalk` (§3.5.1), the origin is the current node; when invoked within another (outer) nodewalk, the origin is the current node of the outer nodewalk.

Formally, a ⟨nodewalk⟩ is a list of `pgfkeys` key–value pairs. Steps in a nodewalk are thus separated by commas. However, FOREST also recognizes *short-form* steps, whose names consist of a single character and which do not need to be separated by a comma. For example, nodewalk `parent, parent, n=2` can be concisely written as `uu2`. Long and short forms can be mixed freely, like this: `next, uu2, previous`.

Besides nodewalk keys, a ⟨nodewalk⟩ can also contain node keys (or even Ti*k*Z keys).[20] These keys do their usual function, but within the context of the current node of the nodewalk: `parent, s=2em, parent, text=red` sets the parent's `s` to 2em and the grandparent's text color to red. It is worth noting that node keys include `TeX`, which makes it possible to execute any TEX code while nodewalking.

Some steps target a single node, like above-mentioned `parent` and `n`. Others, called multi-steps, describe mini-walks themselves: for example `children` visits each child of the node in turn, and `tree` visits each of the node's descendants (including the node itself). The path of many steps is determined by the geometric relations of the tree, or the value of some option. However, there are also keys for embedding nodewalks (`nodewalk`, `branch`, etc.), saving and loading nodewalks, sorting them, or even re-walking the history of

[20] The precise algorithm for keyname resulotion in nodewalks is as follows.

- First, FOREST searches for the given ⟨keyname⟩ in the `/forest/nodewalk` path. If found (a long-form step or a nodewalk style), it is executed.
- Next, it is checked whether ⟨keyname⟩ is a sequence of short-form steps; is so, they are executed.
- Otherwise, ⟨key⟩ is executed in the `/forest` path. This includes both FOREST's and Ti*k*Z's keys. The latter are usually forwarded to Ti*k*Z via `node options`.

There are some clashes between node key and nodewalk step names. For example, `l` is both a ⟨dimen⟩ option and a short form of the step to the last child. According to the rules above, the nodewalk step will take precedence in case of a clash. Use nodewalk key `options` to execute a clashing node key.

steps made (like in a web browser).[21] Finally, if all this is not enough, you can define your own steps, see §3.8.8.

Each nodewalk has an associated *every-step keylist*: a keylist of node keys[22] which get executed after each step of the nodewalk. The every-step keylist of the current nodewalk is contained in register `every step` and can be changed at any point during the nodewalk. Its value at the start of the nodewalk depends on how the nodewalk was invoked. In most cases (e.g. `nodewalk` or prefix `for`-based spatial propagators), it is given explicitely as an argument to the key that executes the nodewalk. However, see `Nodewalk` option `every step` for information on how the every-step keylist of an embedded nodewalk can interact with the every-step keylist of its parent nodewalk.

Each nodewalk step can be either *real* or *fake*. Fake steps only change the current node. Real steps also trigger execution of the every-step keylist and update of history. Fake steps are sometimes useful as a "computational tool". They can be introduced explicitely using `fake`; some other keys (like several history nodewalk keys, §3.8.5) introduce fake steps implicitely.

In some cases, the nodewalk might step "out of the tree". (Imagine using `parent` at the root of the tree, or `n=42` at a node with less that 42 children.) Our official term will be that the nodewalk stepped on an *invalid node*; what happens formally is that the current node is changed to the node with `id=0`. Normally, such an event raises an error. However, the full story is told by `on invalid`.

Nodewalks can be hard to follow, especially when designing styles. FOREST does its best to help. First, it logs the nodewalk stack in case of error. Second, if package option `debug=nodewalks` is given, it logs every step made.

3.8.1 Invoking (embedded) nodewalks

There are many ways to invoke a nodewalk. For example, several keys, like `/tikz/fit to`, and aggregate functions (§3.14) expect a ⟨nodewalk⟩ argument. This section lists keys which can be used to explicitely invoke a nodewalk.

The keys in this section can be used not only as node keys (in fact, not all of them can be used so), but also as nodewalk keys. The latter fact means that they can be used to introduce embedded nodewalks, which (can) have its own every-step keylist, history and on-invalid mode; for details on how these properties of outer and embedded nodewalk can interact, see `Nodewalk`. There is no limit to the depth of nodewalks embedding (nodewalk within nodewalk within nodewalk ...).

An embedded nodewalk functions as a single, fake step of the outer nodewalk. Specifically, this means that, while stepping through the embedded nodewalk, the every-step keylist of the outer nodewalk is not executed. Furthermore, by default, modifying the every-step keylist of the inner walk (by manipulating register `every step`) does not influence the outer nodewalk (but see option `every step`).

An embedded nodewalk does not count as a (real, every-step keyslist invoking) step of the outer nodewalk. After it is finished, there are two options with respect to the new current node of the outer nodewalk,[23] depending on whether the embedded nodewalk was invoked using a variant of the key with or without the `for` prefix (all keys in this section have the `for` variant).

- For keys *without* the `for` prefix, the current node of the outer nodewalk changes, *via a fake step*, to the final node visited by the embedded nodewalk. This holds even if the final node was reached as a fake step and even if it is invalid (`id=0`). The fake step in the outer nodewalk cannot be made real, not even by `real`: if you want to execute the every-step keylist of the outer nodewalk at the finishing node of the embedded nodewalk, follow the latter by step `current`.

- For keys *with* the `for` prefix, the current node of the outer nodewalk remains unchanged. For this reason, the `for`-prefixed keys are available as node keys (we call them spatial propagators, §3.5.1), while the steps without this prefix are generally not, with the sole exception of `Nodewalk`, which I advise to use carefully.

[21] Note that nesting operation (§3.8.4) and history (§3.8.5) steps, or embedding nodewalks under these steps doesn't work, for most combinations, as many of them internally manipulate nodewalk history.

[22] When executing the `every step` keylist, FOREST switches into the `/forest` path, which makes it impossible to directly include a nodewalk into the every-step keylist. The reason is performance. Every time a `/forest/nodewalk` key is not found, the short-form nodewalk recognition algorithm is executed, and this algorithm is slow. As `every step` is used a lot (it is for example used every invocation of every spatial propagator) and the keys in `every step` are usually node options from `/forest` path, FOREST would spend way too much time checking if a given node option is actually a short-form nodewalk.

If you need to execute nodewalk keys within the every-step keylist, use node key `Nodewalk`.

[23] Even the outermost explicitly invoked nodewalks actually have the outer nodewalk. It is "static" in the sense that no real step is ever made in it, but it has all the nodewalk properties — the current node, `every step` keylist register, `history` and `on invalid` mode (error) — which can interact with the embedded nodewalk.

All steps described in this section can be prefixed by `for`. All of them, with or without this prefix, are available as nodewalk keys. The list of keys from this section which are available as node keys: `Nodewalk`, `for Nodewalk`, `for nodewalk`; you will most often want to use the latter.

step `Nodewalk`=⟨keylist: config⟩⟨nodewalk⟩⟨keylist: every-step⟩

Walks an ⟨nodewalk⟩ starting at the current node.

This is the most generic form of embedding a nodewalk. Unlike other keys described in this subsection, it can also be used as a node key even without the `for` prefix, but take care as it will, in general, change the current node.

The ⟨config⟩ argument serves to specify the interaction between the outer and embedded nodewalk. It can contain the following keys:

Nodewalk option `every step`=`{independent|inherited|shared}` `independent`

Nodewalk option `history`=`{independent|inherited|shared}` `shared`

The following table shows what happens to the every-step keylist and history depending on the value of `every step` and `history`, respectively. State B is ⟨every-step⟩ for every step and empty for history.

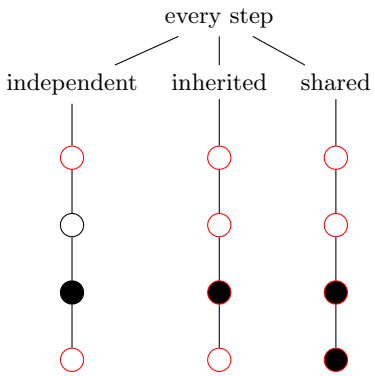
	`independent`	`inherited`	`shared`
state of the outer nodewalk	A	A	A
initial state of the inner nodewalk	B	A	A
...			
final state of the inner nodewalk	C	C	C
state of the outer nodewalk	A	A	C

As shown in the table above, argument ⟨every-step⟩ is used to initialize the embedded nodewalk's every-step keylist when it is independent of the outer nodewalk. In other cases, this argument is ignored (use `{}`).

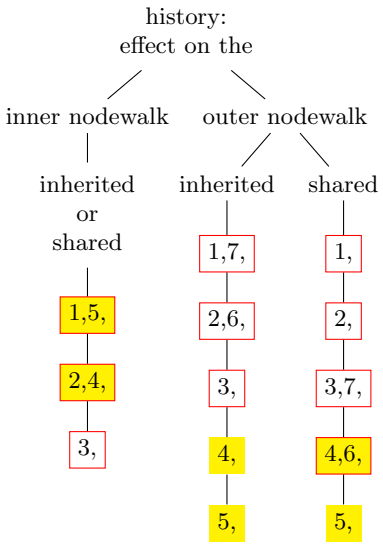
Nodewalk option `on invalid`=`{error|fake|error in real|last valid|inherited}` `inherited`

Like `on invalid`, but local to this nodewalk. The additional alternative `inherited` (which is the default) means to retain the current value, regardless of how it was set (by an outer nodewalk, explicit `on invalid`, or the package default, `error`).

→ Use `Nodewalk` if you need to execute nodewalk keys within the every-step keylist.




```
\begin{forest}                                                                    (76)
  for 2=calign with current, for children={for descendants={circle,draw}}
  [every step,
    [independent,for nodewalk={
        1,Nodewalk={every step=independent}{1,every step=fill,1}{},1
      }{draw=red},
      [[[[]]]]]
    [inherited,for nodewalk={
        1,Nodewalk={every step=inherited}{1,every step=fill,1}{},1
      }{draw=red},
      [[[[]]]]]
    [shared,for nodewalk={
        1,Nodewalk={every step=shared}{1,every step=fill,1}{},1
      }{draw=red},
      [[[[]]]]]]
\end{forest}
```




```
\begin{forest}                                                              (77)
  mark/.style={tempcounta+=1,content+/.register=tempcounta,content+={,}},
  [history:\\effect on the,align=center
    [inner nodewalk
      % uncommenting this would result in an error:
      % [independent, delay={for nodewalk={
      %     tempcounta=0,111,
      %     Nodewalk={history=independent}{walk back=2}{mark,fill=yellow}
      %   }{mark,draw=red}},
      %   [[[]]]]
      [inherited\\or\\shared, align=center,delay={for nodewalk={
            tempcounta=0,111,
            Nodewalk={history=inherited}{walk back=2}{mark,fill=yellow}
          }{mark,draw=red}},
        [[[]]]]]
    [outer nodewalk
      [inherited,delay={for nodewalk={
          tempcounta=0,111,
          Nodewalk={history=inherited}{11}{mark,fill=yellow},
          walk back=2
        }{mark,draw=red}},
        [[[[[]]]]]]
      [shared,delay={for nodewalk={
          tempcounta=0,111,
          Nodewalk={history=shared}{11}{mark,fill=yellow},
          walk back=2
        }{mark,draw=red}},
        [[[[[]]]]]]]]
\end{forest}
```

step **nodewalk**=⟨nodewalk⟩⟨keylist: every-step⟩

This key is a shorthand for

```
Nodewalk={every step=independent,history=independent,on
    invalid=inherited}⟨nodewalk⟩⟨keylist: every-step⟩
```

→ `for nodewalk` is the most common way to explicitely invoke a nodewalk from a node keylist (the keylist immediately following the content of the node).

step **nodewalk'**=⟨nodewalk⟩

This key is a shorthand for

```
Nodewalk={every step=inherited,history=independent,on
    invalid=inherited}⟨nodewalk⟩{}
```

→ Using this key, it is easy to "*temporarily* change" the `every step` keylist of a nodewalk.

→ Using `for nodewalk'` is probably the easiest way to make a "trip" within a nodewalk, i.e. walk some steps but return to their origin afterwards.

→ This key (with or without the `for` prefix) is not available as a node key — it would make little sense there, as it has no every-step keylist argument.

3.8.2 Single-step keys

Single-step nodewalk keys visit a single node. The behaviour in the situation when the target node does not exist is determined by `on invalid`.

For each single-step key, spatial propagator `for` ⟨step⟩ is also defined. `for` ⟨step⟩=⟨keylist⟩ is equivalent to `for nodewalk={`⟨step⟩`}{`⟨keylist⟩`}`. If the step takes an argument, then its `for` ⟨step⟩ propagator takes two and the argument of the step precedes the ⟨keylist⟩. See also §3.5.1.

Linear order below means the order of nodes in the bracket representation, i.e. depth-first parent-first first-child-first.

step `current` an "empty" step: the current node remains the same[24]

step `first` the first child

step `first leaf`, `first leaf'` the first leaf (terminal node) of the current node's descendants (`first leaf`) or subtree (`first leaf'`), in the linear order

step `id=`⟨id⟩ the node with the given id; this step does not depend on the current node

step `last` the last child

step `last dynamic node` the last non-integrated (created/removed/replaced) node; see §3.11

step `last leaf`, `last leaf'` the last leaf (terminal node) of the current node's descendants (`last leaf`) or subtree (`last leaf'`), in the linear order

step `n=`n the nth child; counting starts at 1[25]

step `n'=`n the nth child, starting the count from the last child

step `name=`⟨name⟩ the node with the given name or alias; this step does not depend on the current node

step `next` the next sibling

step `next leaf` the next node (in the linear order) which is a leaf (the current node need not be a leaf)

step `next node` the next node of the entire tree, in the linear order

step `next on tier=`⟨tier⟩ the next node (in the linear order) on the given tier; if no tier is given, assume the tier of the current node

step `origin` the starting node of the nodewalk; note that the starting point does not automatically count as a step: if you want to step on it, use this key (or `current`, at the beginning of the nodewalk)

step `parent` the parent

step `previous` the previous sibling

step `previous leaf` the previous node (in the linear order) which is a leaf (the current node need not be a leaf)

step `previous node` the previous node of the entire tree, in the linear order

step `previous on tier=`⟨tier⟩ the previous node (in the linear order) on the given tier; if no tier is given, assume the tier of the current node

step `root` the root node, i.e. the ancestor of the current node without the parent; note that this key *does* depend on the current node

step `root'` the formal root node (see `set root` in §3.11); this key does not depend on the current node

step `sibling` the sibling

(don't use if the parent doesn't have exactly two children . . .)

step `to tier=`⟨tier⟩ the first ancestor of the current node (or the node itself) on the given ⟨tier⟩

[24] While it might at first sight seem stupid to have an empty step, this is not the case. For example, using propagator `for current` derived from this step, one can process a ⟨keylist⟩ constructed using `.wrap (`n`) pgfmath arg(s)` or `.wrap value`.

[25] Note that `n` *without* an argument is a short form of `next`.

3.8.3 Multi-step keys

Multi-step keys visit several nodes, in general. If a multi-step key visits no nodes, the current node remains unchanged.

For each multi-step key, spatial propagator `for` ⟨step⟩ is also defined, see §3.5.1.

Many of the keys below have a `reversed` variant. Those keys reverse the order of *children*. Note that in general, this differs from operation key `reverse`, which reverses the order of the entire embedded nodewalk.

Linear order below means the order of nodes in the bracket representation, i.e. depth-first parent-first first-child-first.

step `children`, `children reversed`

Visit all the children of the current node.

step `tree`, `tree reversed`

step `tree children-first`, `tree children-first reversed`

step `tree breadth-first`, `tree breadth-first reversed`

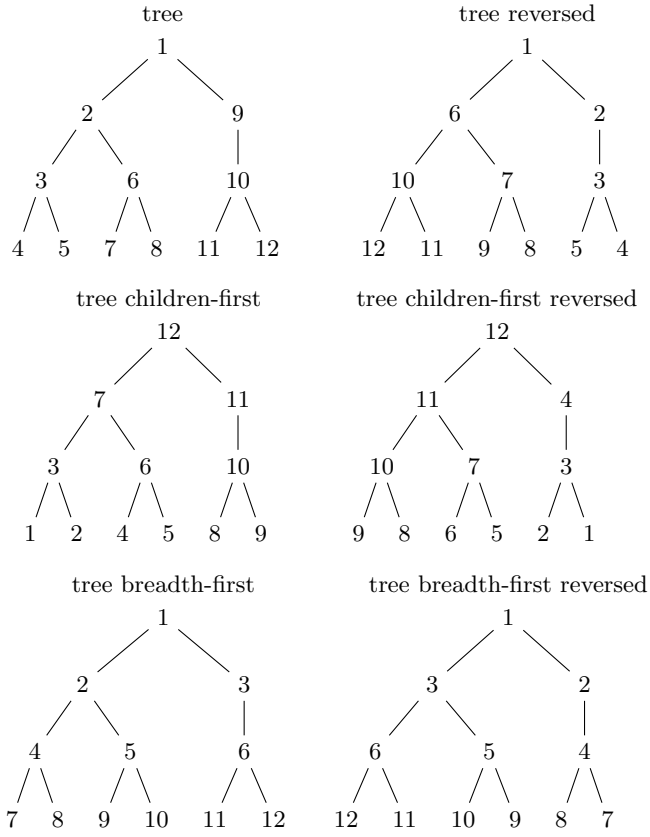
Visit the current node and all its descendants.

The above keys differ in the order the nodes are visited. The basic key, `tree`, traverses the nodes in the depth-first, parent-first first-child-first order, i.e. the order in which they are given in the bracket representation: so it visits the parent before its children and it visits the children from the first to the last.

`reversed` variants reverse the order of *children*, visiting them from the last to the first (from the viewpoint of the bracket representation).

`children-first` variants visit the children before the parent.

`breadth-first` variants behave like `level` steps below: they first visit level `0` nodes, then level `1` nodes etc.



(78)

```
\forestset{
  enumerate/.style={
    tempcounta=1,
    for #1={
      content/.pgfmath=tempcounta,
      tempcounta+=1
    }
  }
}
\newcommand\enumtree[1]{%
  \begin{forest}
    [#1,l sep=0,for n=1{
      l=0,no edge,delay={enumerate=#1}}
      [[[[][]][[][]]][[[][]]]]
    ]
  \end{forest}
}
\renewcommand\arraystretch{2}
\begin{tabular}{cc}
  \enumtree{tree}&
  \enumtree{tree reversed}\\
  \enumtree{tree children-first}&
  \enumtree{tree children-first reversed}\\
  \enumtree{tree breadth-first}&
  \enumtree{tree breadth-first reversed}
\end{tabular}
```

step `descendants`, `descendants reversed`

step `descendants children-first`, `descendants children-first reversed`

step `descendants breadth-first`, `descendants breadth-first reversed`

Visit all the descendants of the current node.

Like the `tree` keys, but the current node is not visited.

step `relative level<` , `relative level` , `relative level>` =⟨count⟩
step `relative level reversed<`, `relative level reversed`, `relative level reversed>`=⟨count⟩
step `level<` , `level` , `level>` =⟨count⟩
step `level reversed<`, `level reversed`, `level reversed>`=⟨count⟩

Visits the nodes in the subtree of the current node whose level (depth) is less than *or equal to*, equal to, or greater than *or equal to* the given level.

The `relative` variants consider the level as relative to the current node: relative level of the current node is 0; relative level of its children is 1, of its grandchildren 2, etc. The absolute variants consider the depth with respect to the (geometric) root, i.e. as returned by node option `level`.

The nodes are traversed in the breadth-first order. The `reversed` variants reverse the order of the children within each level, but the levels are still traversed from the highest to the deepest.

step `leaves`

Visits all the leaves in the current node's subtree.

step `-level`=⟨count⟩
step `-level'`=⟨count⟩

Visits all the nodes ⟨count⟩ levels above the leaves in the current node's subtree.

step `preceding siblings` `following siblings`
step `current and preceding siblings` `current and following siblings`
step `preceding siblings reversed` `following siblings reversed`
step `current and preceding siblings reversed` `current and following siblings reversed`
step `siblings`
step `current and` `siblings`
step `siblings reversed`
step `current and` `siblings reversed`

Visit preceding, following or all siblings; visit the current node as well or not; visit in normal or reversed order.

step `ancestors`
step `current and ancestors`

Visit the ancestors of the current node, starting from the parent/current node, ending at the root node.

step `preceding nodes` `following nodes`
step `current and preceding nodes` `current and following nodes`

Visit all preceding or following nodes of the entire tree, in the linear order; visit the current node as well or not.

3.8.4 Operations

Generally speaking, nodewalk operations take an *input nodewalk* and transform it into an *output nodewalk*, while possibly also having side effects.

The most important categorization of operations is in terms of the input nodewalk:

- "Normal" keys execute the input nodewalk "invisibly", i.e. with a every-step keylist that is initially empty. However, even such an "invisible" nodewalk might not always be completely without effect. For example, the effects of any node keys contained in the input nodewalk or modifications of its (initially empty) every-step keylist will be felt.
- Most of the operation keys have the `walk and ...` variant, where input given nodewalk is meant to be "visible": it is walked directly in the context of the invoking nodewalk (specifically, with its every-step keylist in effect).

- Some operation keys have the `... in nodewalk` variant, which operates on the portion of the current nodewalk that was already walked.
- `load` has no input nodewalk.

All operation keys except `... in nodewalk` variants can be prefixed by `for` to create a spatial propagator (§3.5.1).

The output nodewalk is always walked in the context of the invoking nodewalk. However, note that, as mentioned above, in the case of `walk and ...` variants, that context can be changed during the execution of the input nodewalk.

Trivia: `save` is the only operation with no output nodewalk and also the only operation with a "side effect" (of saving the nodewalk, obviously).

For some operations (`filter` and `branch`), the every-step keylist contains instructions on how collect the relevant information. While you can safely append and prepend to `every step` keylist of their input nodewalk, you should not completely rewrite it. If you want the operations to actually work, of course.

step `group=`⟨nodewalk⟩

Treat ⟨nodewalk⟩ as a single step of the (outer) nodewalk, i.e. the outer every-step keylist is executed only at the end of the embedded nodewalk. The embedded ⟨nodewalk⟩ inherits history from the outer nodewalk. Using this key is equivalent to writing

`Nodewalk={every step=independent,history=inherited}`⟨nodewalk⟩`{}, current`

step `reverse=`⟨nodewalk⟩

step `walk and reverse=`⟨nodewalk⟩

Visits the nodes of the given ⟨nodewalk⟩ in the reversed order.

step `unique=`⟨nodewalk⟩

Walks the ⟨nodewalk⟩, but visits each node at most once.

step `filter=`⟨nodewalk⟩⟨forestmath: condition⟩

Visit the nodes of the given ⟨nodewalk⟩ for which the given ⟨condition⟩ is true.

→ You can safely append and prepend to `every step` keylist during the input ⟨nodewalk⟩, but you should not completely rewrite it.

step `branch={`⟨nodewalk$_1$⟩`,...,`⟨nodewalk$_n$⟩`}`

step `branch'={`⟨nodewalk$_1$⟩`,...,`⟨nodewalk$_n$⟩`}`

Visit the nodes in a "cartesian product" of any number of nodewalks, where a cartesian product is defined as a nodewalk where at every step of ⟨nodewalk$_i$⟩ ($1 \leq i < n$), ⟨nodewalk$_{i+1}$⟩ is executed.

The `branch` variant visits only the nodes visited by the innermost nodewalk, ⟨nodewalk$_n$⟩. The `branch'` variant visits the nodes visited by all the nodewalks of the product, ⟨nodewalk$_1$⟩ ... ⟨nodewalk$_n$⟩.

For an example of each, see `c-commanded` and `c-commanders` from the `linguistics` library.

→ You can safely append and prepend to `every step` keylists during the input ⟨nodewalk⟩s, but you should not completely rewrite them.

step `save=`⟨toks: name⟩⟨nodewalk⟩

step `walk and save=`⟨toks: name⟩⟨nodewalk⟩

Saves the given ⟨nodewalk⟩ under the given name.

step `save append=`⟨toks: name⟩⟨nodewalk⟩

step `save prepend=`⟨toks: name⟩⟨nodewalk⟩

step `walk and save append=`⟨toks: name⟩⟨nodewalk⟩

step `walk and save prepend=`⟨toks: name⟩⟨nodewalk⟩

Appends/prepends the given ⟨nodewalk⟩ to nodewalk ⟨name⟩.

step load=⟨toks: name⟩ Walks the nodewalk saved under the given name.

Note that it is node ids that are saved: loading a named nodewalk with in a context of a different current node, or even with a tree whose geometry has changed (see §3.11) will still visit exactly the nodes that were visited when the nodewalk was saved.

step sort=⟨nodewalk⟩

step sort'=⟨nodewalk⟩

step walk and sort=⟨nodewalk⟩

step walk and sort'=⟨nodewalk⟩

Walks the nodes of the nodewalk in the order specified by the last invokation of sort by. The sort variants sort in the ascending order, the sort' variants in the descending order. The walk and sort variants first visit the nodes in the order specified by the given ⟨nodewalk⟩.

sort by={⟨forestmath⟩,...,⟨forestmath⟩}

Sets the sorting order used by all keys comparing nodes: sort, min and max key families in the nodewalk namespace, and the sort key family in the option namespace (dynamic tree).

For each node being ordered, an "n-dimensional coordinate" is computed by evaluating the given list of pgfmath expressions in the context of that node.[26] Nodes are then ordered by the usual sort order for multi-dimensional arrays: the first item is the most important, the second item is the second most important, etc.

Simply put, if you want to sort first by the number of children and then by content, say sort by={n children, content}.

In the simplest case, the given ⟨forestmath⟩ expressions are simply node options. However, as any pgfmath expression is allowed in the sort key, you can easily sort by the product of the content of the current node and the content of its first child: sort by={content()*content("!1").

To sort alphabetically, one must use the argument processor (§3.13) to specify the sort order. In particular, the key must be marked as text using t. The first example below shows a simple alphabetical sort by content; the second sorts the filenames by suffix first (in the ascending order) and then by the basename (in the descending order, see -).

(79)

example.aux
example.log
example.pdf
example.tex
thesis.aux
thesis.log
thesis.pdf
thesis.tex
thesis.toc

```
\begin{forest}
  [,phantom,grow'=0,for children={anchor=west,child anchor=west},s sep=0,
    delay={sort by=>0+t{content},sort}
    [example.tex][example.pdf][example.log][example.aux]
    [thesis.tex][thesis.pdf][thesis.log][thesis.aux][thesis.toc]
  ]
\end{forest}
```

(80)

thesis.aux
example.aux
thesis.log
example.log
thesis.pdf
example.pdf
thesis.tex
example.tex
thesis.toc

```
\begin{forest}
  declare toks={basename}{},
  declare toks={extension}{},
  [,phantom,grow'=0,for children={anchor=east},s sep=0,
    delay={
      for children={split option={content}{.}{basename,extension}},
      sort by={>0+t{extension},>0+t-{basename}},
      sort,
    }
    [example.tex][example.pdf][example.log][example.aux]
    [thesis.tex][thesis.pdf][thesis.log][thesis.aux][thesis.toc]
  ]
\end{forest}
```

step min=⟨nodewalk⟩, max=⟨nodewalk⟩

step walk and min=⟨nodewalk⟩, walk and max=⟨nodewalk⟩

[26] Don't worry, lazy evaluation is used.

step `mins`=⟨nodewalk⟩, `maxs`=⟨nodewalk⟩

step `walk and mins`=⟨nodewalk⟩, `walk and maxs`=⟨nodewalk⟩

Visit the node(s) in the given ⟨nodewalk⟩ with a minimum/maximum value with respect to the sort order previously specified by `sort by`.

Variants `mins`/`maxs` visit all the nodes that with the minimum/maximum value of the sorting key; variants `min`/`max` visit only the first such node (first in the order specified by the given nodewalk).

step `min in nodewalk`, `max in nodewalk`

step `mins in nodewalk`, `maxs in nodewalk`

step `min in nodewalk'`, `max in nodewalk'`

These keys search for the minimum/maximum among the nodes that were already visited in the current nodewalk.

Keys `mins in nodewalk` and `maxs in nodewalk` visits all nodes that reach the minimum/maximum, while keys `min in nodewalk` and `max in nodewalk` variants visit only the first such node.

Keys `min in nodewalk'` and `max in nodewalk'` visit the first minimal/maximal node by moving back in the history, see `back`.

3.8.5 History

FOREST keeps track of nodes visited in a nodewalk and makes it possible to revisit them, in a fashion similar to clicking the back and forward button in a web browser.

These keys cannot be prefixed by `for`.

step `back`=⟨count: n⟩

step `jump back`=⟨count: n⟩

step `walk back`=⟨count: n⟩

Move n steps back in the history. In the `back` variant, all steps are fake; in the `jump back` variant, the final step is real; and in the `walk back` variant, all steps are real.

Note that as the origin is not a part of the history, these keys will *not* step there (unless `current` was the first step of your nodewalk). (Use `origin` to move to the origin of the nodewalk.)

step `forward`=⟨count: n⟩

step `jump forward`=⟨count: n⟩

step `walk forward`=⟨count: n⟩

Move n steps forward in the history. In the `forward` variant, all steps are fake; in the `jump forward` variant, the final step is real; and in the `walk forward` variant, all steps are real.

`save history`=⟨toks: back name⟩⟨toks: forward name⟩

Saves the backwards and forwards history under the given names. (Load them using `load`.) The backwards history is saved in the reverse order of how it was walked, i.e. outward from the perspective of the current position in the nodewalk.

3.8.6 Miscellaneous

The following nodewalk keys are not steps. Rather, they influence the behaviour of nodewalk steps in various ways. The keys in this section having ⟨nodewalk⟩ arguments do not start a new nodewalk in the sense of §3.8.1; the given nodewalk steps rather become a part of the current nodewalk.

register `every step`=⟨keylist⟩ Contains the every-step keylist of the current nodewalk.

nodewalk key `fake`=⟨nodewalk⟩

nodewalk key `real`=⟨nodewalk⟩

The ⟨nodewalk⟩ embedded under `fake` consists of "fake" steps: while the current node is changed, every-step keylist is not executed and the history is not updated.

Note that these keys do not introduce an embedded nodewalk. The given ⟨nodewalk⟩ will not have its own history and every-step keylist.

`real` undoes the effect of `fake`, but cannot make real the implicitely fake steps, such as the return to the origin in spatial propagators like `for nodewalk`. `fake` and `real` can be nested without limit.

step `last valid`

step `last valid'`

If the current node is valid, these keys do nothing. If the current node of the nodewalk is invalid (i.e. its `id` is 0), they step to the last valid visited node. If there was no such node, they step to the origin of the nodewalk.

The variant *without* `'` makes a fake step. More precisely, it behaves as if both `fake` and `on invalid=fake` are in effect.

nodewalk key `on invalid={error|fake|step}`⟨nodewalk⟩

This key determines what should happen if a nodewalk step landed on the invalid node, i.e. the node with `id=0`.

There is a moment within the step when the current node is changed but the step itself is not yet really done, is "still fake", i.e. the history is not yet updated and the every-step keylist is not yet executed. If the new current node is invalid, this key determines what should happen next.

`on invalid={error}`⟨nodewalk⟩ produces an error;

`on invalid={fake}`⟨nodewalk⟩ does nothing: history is not updated and the every-step keylist is not executed, thus making the step essentialy fake;

`on invalid={error if real}`⟨nodewalk⟩ produces an error unless `fake` is in effect.

`on invalid={last valid}`⟨nodewalk⟩ returns to the last valid node, by making a fake step, like `last valid`.

Loops with the implicit `id=0` condition (§3.10) automatically switch to `on invalid=fake` mode.

See also `Nodewalk` option `on invalid`.

nodewalk key `options`=⟨keylist: node keys⟩

Execute the given node options in the context of the current node.

There is not much need to use this key, as any keys that are not (long) steps or sequences of short steps are automatically used as FOREST node options any way, but there are still usage cases, for example whenever the names of node options and (long) steps are the same, or in a style that wants to ensure there is no overlap.

nodewalk key `strip fake steps`=⟨nodewalk⟩

If ⟨nodewalk⟩ ends with fake steps, return to the last node current before those steps were made. For details, see `define long step`.

3.8.7 Short-form steps

All short forms of steps are one token long. When using them, there is no need to separate them by commas. Here's the list of predefined short steps and their corresponding long-form steps.

short step `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` the first, . . . , ninth child — `n=1,...,9`

short step `l` the last child — `last`

short step `u` the parent (up) — `parent`

short step `p` the previous sibling — `previous`

short step `n` the next sibling — `next`

short step `s` the sibling — `sibling`

short step `P` the previous leaf — `previous leaf`

short step `N` the next leaf — `next leaf`

short step `F` the first leaf — `first leaf`

short step `L` the last leaf — `last leaf`

short step `>` the next node on the current tier — `next on tier`

short step `<` the previous node on the current tier — `previous on tier`

short step `c` the current node — `current`

short step `o` the origin — `origin`

short step `r` the root node — `root`

short step `R` the formal root node — `root'`

short step `b` back one fake step in history — `back=1`

short step `f` forward one fake step in history — `forward=1`

short step `v` last valid node in the current nodewalk, fake version — `last valid`

short step `*`⟨count: n⟩⟨keylist⟩ repeat keylist *n* times — `repeat=`⟨count: n⟩⟨keylist⟩

{⟨keylist⟩} put keylist in a group — `group=`⟨keylist⟩

3.8.8 Defining steps

You can define your own steps, both long and short, or even redefine predefined steps. Note, though, that it is not advisable to redefine long steps, as their definitions are interdependent; redefining short steps is always ok, however, as they are never used in the code of the package.

`define long step=`⟨name⟩⟨options⟩⟨nodewalk⟩

Define a long-form step named ⟨name⟩ as equivalent to ⟨nodewalk⟩. ⟨options⟩ control the exact behaviour or the defined step.

`n args=`⟨number⟩ `0`

`make for=`⟨boolean⟩ `true`

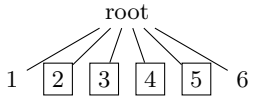
Should we make a `for` prefix for the step?

`strip fake steps=`⟨boolean⟩ `true`

Imagine that ⟨nodewalk⟩ ends with fake steps. Now, fake steps are usually just a computational tool, so we normally wouldn't want the current node after the walk to be one of them. As far as the outer world is concerned, we want the node to end at the last real step. However, simply appending `last valid` to our style will not work. Imagine that the nodewalk results in no steps. In this case, we'd want to remain at the origin of our empty nodewalk. However, imagine further that the (outer) step just before the empty nodewalk was fake. Then `last valid` will not step to the correct node: instead of staying at the origin, it will go to the node that the last real step prior to our nodewalk stepped to. In case there was no such real step, we'd even step to the invalid node (normally getting an error).

Defining the step using `strip fake steps` ensures the correct behaviour described above. Set `strip fake steps=false` only when the fake steps at the end of the nodewalk are important to you.

→ See also nodewalk key `strip fake steps`.




```
\forestset{                                                    (81)
  define long step={children from to}{n args=2}{
    if={#1>#2}{}{n=#1,while={n()<#2}{next}}
  }
}
\begin{forest}
  for children from to={2}{5}{draw}
  [root[1][2][3][4][5][6]]
\end{forest}
```

`define short step`=⟨token: short step⟩⟨n args⟩⟨nodewalk⟩

Define short step taking n arguments as the given ⟨nodewalk⟩. Refer to the arguments in the usual way, via `#1`, ...

To (re)define braces, `{}`, write `define short step={group}{1}{...}`.

handler `.nodewalk style`=⟨nodewalk⟩

⟨nodewalk key⟩`/.nodewalk style`=⟨nodewalk⟩ is a shorthand for
`for nodewalk={`⟨nodewalk key⟩`/.style=`⟨nodewalk⟩`}{}`.

3.9 Conditionals

All conditionals take arguments ⟨true keylist⟩ and ⟨false keylist⟩. The interpretation of the keys in these keylists depends on the environment the conditional appears in. If it is a part of a nodewalk specification, the keys are taken to be nodewalk keys (§3.8), otherwise node keys (§3.5).

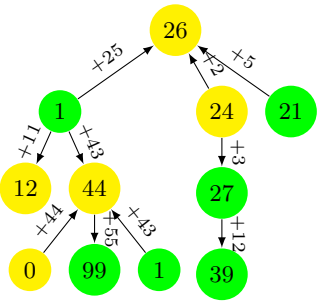
All the conditionals can be nested safely.

conditional `if`=⟨forestmath: condition⟩⟨true keylist⟩⟨false keylist⟩

If ⟨forestmath: condition⟩ evaluates to `true` (non-zero), ⟨true keylist⟩ is processed (in the context of the current node); otherwise, ⟨false keylist⟩ is processed.

For a detailed description of `pgfmath` expressions, see [2, part VI]. (In short: write the usual mathematical expressions.)

In the following example, `if` is used to orient the arrows from the smaller number to the greater, and to color the odd and even numbers differently. (Style `random tree` is defined in the front page example.)




```
\pgfmathsetseed{314159}                                          (82)
\begin{forest}
  before typesetting nodes={
    for descendants={
      if={content()>content("!u")}{edge=->}{
        if={content()<content("!u")}{edge=<-}{}},
      edge label/.wrap pgfmath arg=
        {node[midway,above,sloped,font=\scriptsize]{+#1}}
        {int(abs(content()-content("!u")))}
    },
    for tree={circle,if={mod(content(),2)==0}
                        {fill=yellow}{fill=green}}
  }
  [,random tree={3}{3}{100}]
\end{forest}
```

conditional `if` ⟨option⟩=⟨value⟩⟨true keylist⟩⟨false keylist⟩

This simple conditional is defined for every ⟨option⟩ (except boolean options, see below): if ⟨value⟩ equals the value of the option at the current node, ⟨true keylist⟩ is executed; otherwise, ⟨false keylist⟩.

conditional `if` ⟨boolean option⟩=⟨true keylist⟩⟨false keylist⟩

Execute ⟨true keylist⟩ if ⟨boolean option⟩ is true; otherwise, execute ⟨false keylist⟩.

conditional `if in` ⟨toks option⟩=⟨toks⟩⟨true keylist⟩⟨false keylist⟩

Checks if ⟨toks⟩ occurs in the option value; if it does, ⟨true keylist⟩ are executed, otherwise ⟨false keylist⟩.

This conditional is defined only for ⟨toks⟩ options, see §3.6.1.

conditional `if` ⟨dimen option⟩`>`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `if` ⟨dimen option⟩`<`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `if` ⟨count option⟩`>`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `if` ⟨count option⟩`<`=⟨value⟩⟨true keylist⟩⟨false keylist⟩

If the current value of the dimen/count option is greater/less than or equal to ⟨value⟩, execute ⟨true keylist⟩; else, execute ⟨false keylist⟩.

conditional `if nodewalk valid`=⟨keylist: test nodewalk⟩⟨true keylist⟩⟨false keylist⟩

If the test nodewalk finished on a valid node, ⟨true keylist⟩ is processed (in the context of the current node); otherwise, ⟨false keylist⟩ is processed.

conditional `if nodewalk empty`=⟨keylist: test nodewalk⟩⟨true keylist⟩⟨false keylist⟩

If the test nodewalk contains no (real) steps, ⟨true keylist⟩ is processed (in the context of the current node); otherwise, ⟨false keylist⟩ is processed.

conditional `if current nodewalk empty`=⟨true keylist⟩⟨false keylist⟩

If the current nodewalk contains no (real) steps, ⟨true keylist⟩ is processed (in the context of the current node); otherwise, ⟨false keylist⟩ is processed.

conditional `if in saved nodewalk`=⟨nodewalk⟩⟨toks: nodewalk name⟩⟨true keylist⟩⟨false keylist⟩

If the node at the end of ⟨nodewalk⟩ occurs in the saved nodewalk, ⟨true keylist⟩ is processed (in the context of the current node); otherwise, ⟨false keylist⟩ is processed.

propagator `if have delayed`=⟨true keylist⟩⟨false keylist⟩ If any options were delayed in the current cycle (more precisely, up to the point of the execution of this key), process ⟨true keylist⟩, otherwise process ⟨false keylist⟩. (`delay n` will trigger "true" for the intermediate cycles.)

This key assumes that the processing order of the innermost invocation of `process keylist` or `process keylist'` is given by `processing order`. If this is not the case, explicitely supply the processing order using `if have delayed'`.

propagator `if have delayed'`=⟨nodewalk⟩⟨true keylist⟩⟨false keylist⟩ Like `if have delayed`, but assume the processing order given by ⟨nodewalk⟩.

The following keys are shortcuts: they execute their corresponding `if ...` conditional for every node in the subtree of the current node (including the node itself). In other words:

`where` ...⟨arg$_1$⟩...⟨arg$_n$⟩`/.style={for tree={if` ...=⟨arg$_1$⟩...⟨arg$_n$⟩`}}`

→ Except in special circumstances, you probably don't want to embed keys from the `where` family within a `for tree`, as this results in two nested loops. It is more usual to use an `if` family key there. For an example where using `where` actually does the wrong thing, see question Smaller roofs for forest on TEX Stackexchange.

propagator `where`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `where` ⟨option⟩=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `where` ⟨boolean option⟩=⟨true keylist⟩⟨false keylist⟩
conditional `where in` ⟨toks option⟩=⟨toks⟩⟨true keylist⟩⟨false keylist⟩
conditional `where` ⟨dimen option⟩`>`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `where` ⟨dimen option⟩`>`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `where` ⟨count option⟩`>`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
conditional `where` ⟨count option⟩`>`=⟨value⟩⟨true keylist⟩⟨false keylist⟩
step `where nodewalk valid`=⟨toks: nodewalk name⟩⟨true keylist⟩⟨false keylist⟩
step `where nodewalk empty`=⟨toks: nodewalk name⟩⟨true keylist⟩⟨false keylist⟩
step `where in saved nodewalk`=⟨nodewalk⟩⟨toks: nodewalk name⟩⟨true keylist⟩⟨false keylist⟩

3.10 Loops

All loops take a ⟨keylist⟩ argument. The interpretation of the keys in these keylists depends on the environment the loop appears in. If it is a part of a nodewalk specification, the keys are taken to be nodewalk keys (§3.8), otherwise node keys (§3.5).

All loops can be nested safely.

loop `repeat`=⟨number⟩⟨keylist⟩

The ⟨keylist⟩ is processed ⟨number⟩ times.

The ⟨number⟩ expression is evaluated using `pgfmath`.

loop `while`=⟨forestmath: condition⟩⟨keylist⟩

loop `do while`=⟨forestmath: condition⟩⟨keylist⟩

loop `until`=⟨forestmath: condition⟩⟨keylist⟩

loop `do until`=⟨forestmath: condition⟩⟨keylist⟩

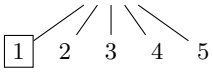
`while` loops cycle while the condition is true, `until` loops terminate when the condition becomes true.

The `do` variants check the condition after processing the ⟨keylist⟩; thus, the keylist is executed at least once. The variants without the `do` prefix check the condition before processing the ⟨keylist⟩, which might therefore not be processed at all.

When ⟨forestmath: condition⟩ is an empty string, condition `valid` is implicitely used, and ⟨keylist⟩ is implicitely embedded in `on invalid=fake`. Thus, the `while` loops will cycle until they "walk out of the tree", and `until` loops will cycle until they "walk into the tree."

→ If a loop "walks out of the tree", you can get it back in using `last valid` or `strip fake steps`.

tried: 15, 20, 10, 14, 20, 13, 1,




```
\pgfmathsetseed{1234}
\begin{forest}
  try/.style={root',content+={#1,\ },n=#1},
  delay={
    for nodewalk={do until={}{try/.pgfmath={random(1,20)}},draw}{},
  },
  [tried:\ [1][2][3][4][5]]
\end{forest}
```
(83)

loop `while nodewalk valid`=⟨nodewalk⟩⟨keylist⟩

loop `do while nodewalk valid`=⟨nodewalk⟩⟨keylist⟩

loop `until nodewalk valid`=⟨nodewalk⟩⟨keylist⟩

loop `do until nodewalk valid`=⟨nodewalk⟩⟨keylist⟩

loop `while nodewalk empty`=⟨nodewalk⟩⟨keylist⟩

loop `do while nodewalk empty`=⟨nodewalk⟩⟨keylist⟩

loop `until nodewalk empty`=⟨nodewalk⟩⟨keylist⟩

loop `do until nodewalk empty`=⟨nodewalk⟩⟨keylist⟩

⟨nodewalk⟩ is embedded within `on invalid=fake`.

`break`=*n* Break out of the loop. 0

The loop is only exited after all the keys in the current cycle have been processed.

The optional argument n ($n \geq 0$) specifies which level of embedding to break from; the default is to break out of the innermost loop.

pgfmath function `forestloopcount`([*n*]) 0

How many times has the loop repeated until now?

The optional argument n ($n \geq 0$) specifies the level of embedding to provide information for; the default is to count the repetitions of the current, most deeply embedded loop.

3.11 Dynamic tree

The following keys can be used to change the geometry of the tree by creating new nodes and integrating them into the tree, moving and copying nodes around the tree, and removing nodes from the tree.

The ⟨node⟩ that will be (re)integrated into the tree can be specified in the following ways:

⟨empty⟩: uses the last non-integrated (i.e. created/removed/replaced) node.

→ This node can also be referred to using nodewalk step `last dynamic node`.

→ The list of all such nodes is automatically saved in named nodewalk `dynamic nodes`, to be `load`ed when needed.

⟨node⟩: a new node is created using the given bracket representation (the node may contain children, i.e. a tree may be specified), and used as the argument to the key.

The bracket representation must be enclosed in brackets, which will usually be enclosed in braces to prevent them being parsed while parsing the "host tree."

→ Unlike the bracket representation in a `forest` environment, the bracket representation of a dynamically created node *must* start with `[`. Specifically, it cannot begin with a preamble or the action character.

⟨relative node name⟩: the node ⟨relative node name⟩ resolves to will be used.

A dynamic tree operation is made in two steps:

- If the argument is given by a ⟨node⟩ argument, the new node is created immediately, i.e. while the dynamic tree key is being processed. Any options of the new node are implicitely `delay`ed.
- The requested changes in the tree structure are actually made between the cycles of keylist processing.

→ Such a two-stage approach is employed because changing the tree structure during the dynamic tree key processing would lead to an unmanageable order of keylist processing.

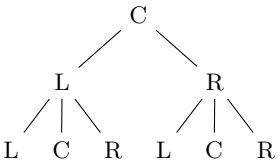
→ A consequence of this approach is that nested dynamic tree keys take several cycles to complete. Therefore, be careful when using `delay` and dynamic tree keys simultaneously: in such a case, it is often safer to use `before typesetting nodes` instead of `delay`, see example (84), and it is also possible to define additional stages, see §3.4.

→ Examples: title page (in style `random tree`) and (100) (in style `xlist`).

Here is the list of dynamic tree keys:

dynamic tree `append`=⟨empty⟩ | `[`⟨node⟩`]` | ⟨relative node name⟩

The specified node becomes the new final child of the current node. If the specified node had a parent, it is first removed from its old position.




```
\begin{forest}                                          (84)
  before typesetting nodes={for tree={
    if n=1{content=L}
          {if n'=1{content=R}
                   {content=C}}}}
  [,repeat=2{append={[
    ,repeat=3{append={[]}}
  ]}}]
\end{forest}
```

dynamic tree `create`=`[`⟨node⟩`]`

Create a new node. The new node becomes the last node.

dynamic tree `create'`=`[`⟨node⟩`]`

Create a new node and process its given options immediately. The new node becomes the last node.

dynamic tree `insert after`=⟨empty⟩ | `[`⟨node⟩`]` | ⟨relative node name⟩

The specified node becomes the new following sibling of the current node. If the specified node had a parent, it is first removed from its old position.

dynamic tree `insert before`=⟨empty⟩ | `[`⟨node⟩`]` | ⟨relative node name⟩

The specified node becomes the new previous sibling of the current node. If the specified node had a parent, it is first removed from its old position.

dynamic tree `prepend`=⟨empty⟩ | `[`⟨node⟩`]` | ⟨relative node name⟩

The specified node becomes the new first child of the current node. If the specified node had a parent, it is first removed from its old position.

dynamic tree `remove`

The current node is removed from the tree and becomes the last node.

The node itself is not deleted: it is just not integrated in the tree anymore. Removing the root node has no effect.

dynamic tree `replace by`=⟨empty⟩ | `[`⟨node⟩`]` | ⟨relative node name⟩

The current node is replaced by the specified node. The current node becomes the last node.

If the specified node is a new node containing a dynamic tree key, it can refer to the replaced node by the ⟨empty⟩ specification. This works even if multiple replacements are made.

If `replace by` is used on the root node, the "replacement" becomes the root node (`set root` is used).

If given an existing node, most of the above keys *move* this node (and its subtree, of course). Below are the versions of these operations which rather *copy* the node: either the whole subtree (`'`) or just the node itself (`''`).

dynamic tree `append'`, `insert after'`, `insert before'`, `prepend'`, `replace by'`

Same as versions without `'` (also the same arguments), but it is the copy of the specified node and its subtree that is integrated in the new place.

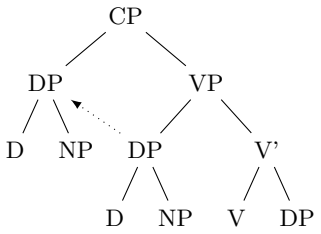
dynamic tree `append''`, `insert after''`, `insert before''`, `prepend''`, `replace by''`

Same as versions without `''` (also the same arguments), but it is the copy of the specified node (without its subtree) that is integrated in the new place.

→ You might want to `delay` the processing of the copying operations, giving the original nodes the chance to process their keys first!

dynamic tree `copy name template`=⟨empty⟩ | ⟨macro definition⟩ ⟨empty⟩

Defines a template for constructing the `name` of the copy from the name of the original. ⟨macro definition⟩ should be either empty (then, the `name` is constructed from the `id`, as usual), or an expandable macro taking one argument (the name of the original).



(85)
```
\begin{forest}
  copy name template={copy of #1}
  [CP,delay={prepend'=subject}
    [VP[DP,name=subject[D][NP]][V'[V][DP]]]]
  \draw[->,dotted] (subject)--(copy of subject);
\end{forest}
```

dynamic tree `set root`=⟨empty⟩ | `[`⟨node⟩`]` | ⟨relative node name⟩

The specified node becomes the new *formal* root of the tree.

Note: If the specified node has a parent, it is *not* removed from it. The node becomes the root only in the sense that the default implementation of stage-processing will consider it a root, and thus typeset/pack/draw the (sub)tree rooted in this root. The processing of keys such as `for parent` and `for root` is not affected: `for root` finds the real, geometric root of the current node. To access the formal root, use nodewalk step `root'`, or the corresponding propagator `for root'`.

dynamic tree `sort`, `sort'` Sort the children of the current node, using the currently active sort key specified in `sort by` (see §3.8.4). `sort` sorts in ascending and `sort'` in descending order.

3.12 Handlers

Handlers are a powerful mechanism of `pgfkeys`, documented in [3, §82.3.5]. Handlers defined by Forest perform a computation and invoke the handled key with its result. The simple handlers are documented in this section: for `.process`, see §3.13; for aggregate function handlers, see §3.14.

handler `.option`=⟨option⟩

The result is the value of ⟨option⟩ at the current node.

handler `.register`=⟨register⟩

The result is the value of ⟨register⟩.

handler `.pgfmath`=⟨pgfmath expression⟩

The result is the evaluation of ⟨pgfmath expression⟩ in the context of the current node.

→ If you only need to access an option or register value, using `.option` or `.register` is much faster than using `.pgfmath`.

handler `.wrap value`=⟨macro definition⟩

The result is the (single) expansion of the given ⟨macro definition⟩. The defined macro takes one parameter. The current value of the handled option will be passed as that parameter.

handler `.wrap` *n* `pgfmath args=`⟨macro definition⟩⟨arg 1⟩...⟨arg *n*⟩

The result is the (single) expansion of the given ⟨macro definition⟩. The defined macro takes n parameters, where $n \in \{2, \ldots, 8\}$. Expressions ⟨arg 1⟩ to ⟨arg n⟩ are evaluated using `pgfmath` and passed as arguments to the defined macro.

handler `.wrap pgfmath arg=`⟨macro definition⟩⟨arg⟩

Like `.wrap` *n* `pgfmath args` would work for $n = 1$.

3.13 Argument processor

For a gentle(r) introduction to the argument processor, see §2.6.

The argument processor takes a sequence of instructions and an arbitrary number of arguments and transforms the given arguments according to the instructions. There are two ways to invoke the argument processor:

handler `.process=`⟨instructions⟩⟨arg 1⟩...⟨arg *n*⟩

The result of the computation is passed on to the handled key as a sequence of arguments. Any number of arguments can be returned.[27]

⟨forestmath⟩ = ⟨pgfmath⟩ | `>`⟨instructions⟩⟨arg 1⟩...⟨arg *n*⟩

In words, a ⟨forestmath⟩ expression is either a ⟨pgfmath⟩ expression or an argument processor expression prefixed by `>`.

In other words, FOREST accepts an argument processor expression anywhere[28] it accepts a ⟨pgfmath⟩ expression. To indicate that we're providing an argument processor expression, we prefix it with `>`.

The result of argument processor's computation should be a single item.

The syntax of argument processor is a cross between `expl3`'s function argument specification and a Turing machine, spiced with a bit of reversed Polish notation. ;-)

Think of ⟨instructions⟩ as a program and ⟨arg 1⟩...⟨arg *n*⟩ as the data that this program operates on.

If you're familiar with Turing machines: like a Turing machine, the argument processor has a notion of a head; unlike a Turing machine, the argument processor head is positioned not over some argument, but between two arguments. If you're not familiar with Turing machines: imagine the arguments as items on a tape and the argument processor as a head that is always located between some two items. As the head is between two arguments, we can talk about the arguments on the left and the arguments on the right.

In general, an instruction will take some items from the left and some from the right (deleting them from the tape), perform some computation and insert the result on the tape, some result items to the left and some to the right. However, most instructions simply take an item from the right, do something with it, and put the (single-item) result to the left; in effect, the head is moved one item to the right. At the beggining, all the arguments are always on the right, so the general idea is that the program will walk through the given arguments, processing them in order.

Descriptions of individual instructions, given below, contain (at the right edge of the page) the argument specification, which tells us about the number of input and output items and the movement of the head. The input and output are separated by an arrow (⟶), and the green eye (👁) signifies the position of the head with respect to the (input or output) items.

For example, instruction `0`, which converts an option name into the option value, exemplifies the most common scenario: it takes one argument from the right and puts the result to the left (in other words, the head moves one item to the right). Wrapping instrucion `w` is more complicated. Given instruction `w`*n*, the argument processor takes one argument from the right (the wrapping macro body) and n items from the left (which become the arguments of the wrapping macro), and puts the resulting item to the left. Comparisons and boolean operations are the instructions resembling the reverse Polish notation: they take the arguments from the left and put the result to the left, as well. Finally, it is worth mentioning instructions `noop` and `+`, which simply move the head to the right and left, respectively; given that the usual movement of the head is to the right, `+` can be thought of as a process-the-argument-again instruction.

Before we finally list the available instructions, some notes:

- ⟨Instructions⟩ may be given in braces or not. If not, everything until the first opening brace is considered to be an instruction.

[27] For backward compatibility, `.process` is also available as `.process args`.

[28] The only exceptions to the above rule are handler `.pgfmath` and argument processor instruction `P`.

- An argument item ($\langle \text{arg}_k \rangle$) is a standard TEX macro argument: either a token or a braced token list. (The obvious exception: $\langle \text{arg}_1 \rangle$ needs to be braced if it follows braceless instructions.)
- Spaces in ⟨instructions⟩ and between arguments are ignored. Format your `.process` as you wish!
- Instructions followed by [*n*] below take an optional numeric modifier.
  - The modifier should be given within the instruction string itself, immediately following the instruction. In particular, no spaces are allowed there. (Sorry for the little white lie above.) The number should not be enclosed in braces, even if it is more than one digit long.
  - This modifier is always optional: its default value varies from instruction to instruction. (Providing 0 means to use the default.)
  - Unless noted otherwise, the optional numerical argument *n* instruct the argument processor to repeat the previous instruction *n* times (by default, 1). For example, `O3` is equivalent to `OOO`.

process instruction `_`[*n*] no-op ⟨arg⟩ ⟶ ⟨arg⟩

The argument is not processed, but simply skipped over. In other words, this instruction only moves the head one item to the right. (This is like `expl3`'s argument specifier `n`.)

n means repetition.

When the end of the instructions is reached, any remaining arguments on the right are processed using this no-op instruction.

process instruction `o`[*n*] expand *o*nce ⟨arg⟩ ⟶ ⟨result⟩

⟨arg⟩ is expanded once. (This is like `expl3`'s argument specifier `o`).

The operation is repeated *n* times (default, one) without moving the head between the repetition. For example, `o3` expands the argument three times (and then moves the head right).

process instruction `x` fully e*x*pand ⟨arg⟩ ⟶ ⟨result⟩

⟨arg⟩ is fully expanded using `\edef`. (This is like `expl3`'s argument specifier `x`.)

process instruction `O`[*n*] *o*ption ⟨option⟩ ⟶ ⟨result⟩

⟨option⟩=⟨option name⟩|⟨relative node name⟩`.`⟨option name⟩

In the former case, ⟨result⟩ is the value of option at the current node, in the latter, the value of option at the node referred to by ⟨relative node name⟩.

n means repetition.

process instruction `R`[*n*] *r*egister ⟨register⟩ ⟶ ⟨result⟩

⟨result⟩ is the value of register ⟨register⟩.

n means repetition.

process instruction `P`[*n*] *p*gfmath ⟨pgfmath expr⟩ ⟶ ⟨result⟩

⟨result⟩ is the result of evaluating ⟨pgfmath expr⟩ using `\pgfmathparse`.

n means repetition.

Combining `P` and `w`, `.process` is capable of anything `.wrap` *n* `pgfmath args` can do. Even better, as we can combine pgfmath and non-pgfmath methods, computations that use `.process` can be (much!) faster. Study the following examples to see how less and less pgfmath is used to achieve the same result — but note that such extreme antipgfmathism probably only makes sense for style/package developers in computations that get performed many times.

(a) $6 * 7 = 42$

(b) $6 * 7 = 42$

(c) $6 * 7 = 42$

(d) $6 * 7 = 42$

```
\begin{forest}                                                                    (86)
  [,grow'=east, where level=1{}{phantom,ignore,ignore edge}
    [(a),delay={content/.wrap 4 pgfmath args={#1 $#2*#3=#4$}
         {content}{content("!1")}{content("!2")}{int(content("!1")*content("!2"))}}
      [6][7]]
    [(b),delay={content/.process={O OOP w4}
      {content}
      {!1.content}{!2.content}{int(content("!1")*content("!2"))}
      {#1 $#2*#3=#4$}}
      [6][7]]
    [(c),delay={content/.process={O OO W2+P w4}
      {content}
      {!1.content}{!2.content}{int(#1*#2)}
      {#1 $#2*#3=#4$}}
      [6][7]]
    [(d),delay={content/.process={O OO W2+n w4}
      {content}
      {!1.content}{!2.content}{#1*#2}
      {#1 $#2*#3=#4$}}
      [6][7]]
  ]
\end{forest}
```

process instruction n[n] $n$umexpr ⟨numexpr⟩ ⟶ ⟨result⟩

⟨result⟩ is the result of evaluating ⟨dimexpr⟩ using eTEX's `\number\numexpr`.

n means repetition.

process instruction d[n] $d$imexpr ⟨dimexpr⟩ ⟶ ⟨result⟩

⟨result⟩ is the result of evaluating ⟨dimexpr⟩ using eTEX's `\the\dimexpr`.

n means repetition.

process instruction +[n] chain instructions ⟨arg⟩ ⟶ ⟨arg⟩

This action allows one to "process the same argument more than once". It does not process the current argument (in fact, there need not be any current argument), but rather moves the last result back in the argument queue. In other words, our machine's head moves one step left. You can also imagine it as an inverse of noop.

n means repetition.

The value of my option `fit` is `tight`.
Yes it is!

```
\begin{forest}                                                                    (87)
  test/.style n args={3}{align=center,
    content={The value of my option \texttt{#1} is \texttt{#2}.\\#3}}
  [fit,delay={test/.process={O O+O}{content}{content}{Yes it is!}}]
\end{forest}
```

process instruction w[n] (consuming) $w$rap ⟨arg$_1$⟩…⟨arg$_n$⟩ ⟨macro body⟩ ⟶ ⟨result⟩

process instruction W[n] (non-consuming) $w$rap ⟨arg$_1$⟩…⟨arg$_n$⟩ ⟨macro body⟩ ⟶ ⟨arg$_1$⟩…⟨arg$_n$⟩⟨result⟩

Defines a temporary macro with n undelimited arguments using the ⟨macro body⟩ given on the right and expands it (once). The arguments given to the temporary macro are taken from the left: ⟨arg$_1$⟩…⟨arg$_n$⟩. The result of the expansion is stored as ⟨result⟩ to the right.

With `w`, ⟨arg$_1$⟩…⟨arg$_n$⟩ are "consumed", i.e. they are removed from the result list on the left. `W` keeps ⟨arg$_1$⟩…⟨arg$_n$⟩ in the result list.

Default n is 1. (Specifying $n > 9$ raises an error.)

process instruction &[n] boolean "and"

process instruction |[n] boolean "or" ⟨arg$_1$⟩⟨arg$_2$⟩ ⟶ ⟨result⟩

⟨result⟩ is a boolean conjunction/disjunction of n arguments. The arguments are taken from the left. They should be numbers (positive integers): `0` means false, any other number means true. The ⟨result⟩ is always `0` or `1`.

Default n is 2.

process instruction `!` boolean "not" ⟨arg⟩ 👁 ⟶ ⟨result⟩ 👁

⟨result⟩ is a boolean negation of the argument. The argument is taken from the left. It should be a number (positive integer): `0` means false, any other number means true. The ⟨result⟩ is always `0` or `1`.

process instruction `?` conditional (if … then… else) ⟨condition⟩ 👁 ⟨true arg⟩⟨false arg⟩ ⟶ 👁 ⟨result⟩

⟨result⟩ is ⟨true arg⟩ is ⟨condition⟩ is true (non-zero), otherwise ⟨false arg⟩.

The condition is taken from the left. The true and false arguments are expected on the right, where the winner is left as well.

process instruction `=` comparison: $\langle arg_1 \rangle = \langle arg_2 \rangle$?

process instruction `<` comparison: $\langle arg_1 \rangle < \langle arg_2 \rangle$?

process instruction `>` comparison: $\langle arg_1 \rangle > \langle arg_2 \rangle$? $\langle arg_1 \rangle\langle arg_2 \rangle$ 👁 ⟶ ⟨result⟩ 👁

Compare $\langle arg_1 \rangle$ and $\langle arg_2 \rangle$, returning `1` (true) if $\langle arg_1 \rangle$ is equal to / less than / greater than $\langle arg_2 \rangle$, `0` (false) otherwise.

The arguments are taken from the left. They can be either numbers, dimensions, text or token lists. Both arguments are expected to be of the same type. The type of comparison is determined by the type of the result returned by the last instruction. `O`/`R` look up the type of option/register to determine the type (booleans are numbers and keylists are toks). Text type must be marked explicitely using `t`.

Comparison is carried out using `\ifnum` for numbers, `\ifdim` for dimensions (this includes unitless decimals returned by pgfmath) and `\pdfstrcmp`[29] for text — for these three types, all three comparison operators are supported. For generic token lists, only `=` makes sense and is carried out using `\ifx`.

In the following example, (a) performs lexicographical comparison because we have marked `21` as text; (b) and (c) perform numeric comparison: in (b), the type is automatically determined from the type of register `tempcounta`, in (c) `21` is marked manually using `n`.

100 > 21? (a) no, (b) yes, (c) yes

```
\forestset{                                                                    (88)
  tempcounta=100,
  TeX/.process={Rw1}{tempcounta}{$#1>21$?\ },
  TeX={(a)\ }, if={>{Rt>}{tempcounta}{21}}{TeX=yes}{TeX=no},TeX={,\ },
  TeX={(b)\ }, if={>{_R<}{21}{tempcounta}}{TeX=yes}{TeX=no},TeX={,\ },
  TeX={(c)\ }, if={>{Rn>}{tempcounta}{21}}{TeX=yes}{TeX=no},
}
```

process instruction `t` mark as *t*ext 👁 ⟨arg⟩ ⟶ ⟨arg⟩ 👁

The result is not changed, only its type is changed to text. This is relevant only for comparisons — both argument processor's comparisons `=`, `>` and `<` and sort keys (see `sort by`).

process instruction `c` to lowercase 👁 ⟨arg⟩ ⟶ ⟨result⟩ 👁

process instruction `C` to uppercase 👁 ⟨arg⟩ ⟶ ⟨result⟩ 👁

process instruction `-` toggle ascending/descending order (negate) ⟨arg⟩ 👁 ⟶ ⟨result⟩ 👁

If the argument is of the text type, its sorting order is changed from ascending to descending or vice versa.

For any numerical argument type (number, dimension, unitless dimension), the argument is actually negated, which obviously has the same effect on sorting.

For generic type arguments, this operation is a no-op.

process instruction `u` *u*ngroup 👁 ⟨arg⟩ ⟶ 👁 $\langle item_1 \rangle \dots \langle item_n \rangle$

As every TEX undelimited macro argument, ⟨arg⟩ is a list of tokens or braced token lists. This instruction puts those items back to the right as "separate arguments".

process instruction `s`[*n*] (consuming) *s*ave $\langle arg_1 \rangle \dots \langle arg_n \rangle$ 👁 ⟶ 👁

[29] `\pdfstrcmp` expands its arguments.

process instruction S[n] (non-consuming) *s*ave $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ 👁 ⟶ $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ 👁

Saves the last n arguments from the left into a "special place".

With `s`, $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ are "consumed", i.e. they are removed from the result list on the left. `S` keeps $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ in the result list.

Default n is 1.

process instruction l[n] (consuming) *l*oad 👁 ⟶ $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ 👁

process instruction L[n] (non-consuming) *l*oad 👁 ⟶ $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ 👁

Loads n arguments from the "special place" to the left.

With `l`, $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ are "consumed", i.e. they are removed from the special place. `S` keeps $\langle \text{arg}_1 \rangle \dots \langle \text{arg}_n \rangle$ in the special place.

The default n is 0 and indicates that the entire special place should be loaded.

process instruction r *r*everse (key)list 👁 ⟨list⟩ ⟶ ⟨result⟩ 👁

⟨list⟩ should be a comma-separated list (*not* a name of a keylist option or register). ⟨result⟩ contains the same elements in the reverse order.

3.14 Aggregate functions

Aggregate functions walk a nodewalk and use the information found in the visited nodes to calculate something.

All aggregate functions are available both as key handlers and `pgfmath` functions.

aggregate .count=⟨nodewalk⟩, aggregate_count("⟨nodewalk⟩")

Store the number of nodes visited in the nodewalk into the handled option.

aggregate .sum=⟨forestmath⟩⟨nodewalk⟩, aggregate_sum("⟨forestmath⟩","⟨nodewalk⟩")
aggregate .average=⟨forestmath⟩⟨nodewalk⟩, aggregate_average("⟨forestmath⟩","⟨nodewalk⟩")
aggregate .product=⟨forestmath⟩⟨nodewalk⟩, aggregate_product("⟨forestmath⟩","⟨nodewalk⟩")
aggregate .min=⟨forestmath⟩⟨nodewalk⟩, aggregate_min("⟨forestmath⟩","⟨nodewalk⟩")
aggregate .max=⟨forestmath⟩⟨nodewalk⟩, aggregate_max("⟨forestmath⟩","⟨nodewalk⟩")

Calculate the value of the given ⟨forestmath⟩ expression at each visited node. Store the sum / average / product / minimum / maximum of these values into the handled option (handlers) or return it (pgfmath functions).

aggregate .aggregate=⟨forestmath: start value⟩⟨forestmath: every step⟩⟨forestmath: after walk⟩⟨nodewalk⟩
pgfmath function aggregate("⟨forestmath: start value⟩","⟨forestmath: every step⟩","⟨forestmath: after walk⟩","⟨nodewalk⟩")

The generic aggregate function. First, register `aggregate result` is set to ⟨forestmath: start value⟩. Then, the given nodewalk is walked. After each step of the ⟨nodewalk⟩, ⟨forestmath: every step⟩ expression is evaluated in the context of the new current node and stored into `aggregate result`. After the walk, the current node is reset to the origin. ⟨forestmath: after walk⟩ expression is then evaluated in its context and stored into `aggregate result` as the final result.

Use `aggregate result` and `aggregate n` in the ⟨forestmath⟩ expressions to refer to the current result value and step number.

register aggregate n=⟨count⟩ the current step number

In the every-step expression of an aggregate function, refers to the (real) step number in the aggregate's ⟨nodewalk⟩. In the after-walk expression, refers to the total number of (real) steps made.

register aggregate result=⟨toks⟩ the current value of the result

This register is where the intermediate results are stored.

register aggregate value=⟨toks⟩ the value of the expression at the current node

This only applies to special aggregates like .sum, not to the generic .aggregate.

aggregate postparse=none|int|print|macro

Roughly speaking, how should the result be formatted? For details, see [3, §89]. Applies only to `pgfmath` versions of aggregate functions, i.e. not to the ' variants.

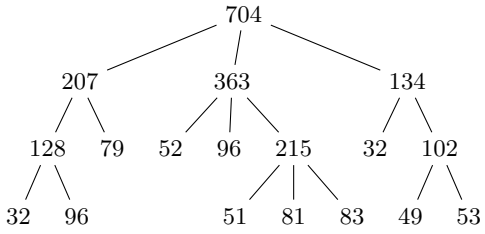
`none` No formatting.

`int` The result is an integer.

`print` Use `pgf`'s number printing extension, see [3, §93].

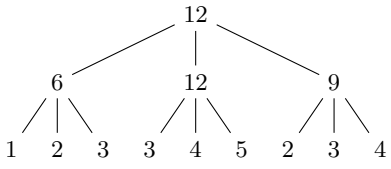
`macro` Use a custom macro. Specify the macro using `aggregate postparse macro`=⟨cs⟩.

Example 1. Randomly generate the content of leaves. The content of a parent is the sum of its children's content. Note how we use `tree children-first` to proceed in a bottom-up fashion.



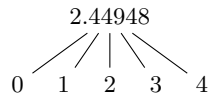

```
\begin{forest}                                                    (89)
  delay={
    aggregate postparse=int,
    for tree children-first={
      if n children=0
        {content/.pgfmath={random(0,100)}}
        {content/.sum={content}{children}}
    }
  }
  [[[[][]][][[][][[][][]]][[][[][]]]]]
\end{forest}
```

Example 2: nested aggregate functions. We are given the black numbers. The inner aggregate, the sum of children, is applied at every blue node. (See how we actually display the blue numbers by storing `aggregate value` to `content`.) The outer aggregate stores the maximum blue number into the red root.




```
\begin{forest}                                                    (90)
  delay={
    aggregate postparse=int,
    content/.max=%
      {aggregate_sum("content","children")}%
      {every step={content/.register=aggregate value},children}%
  } [ [[1][2][3]] [[3][4][5]] [[2][3][4]] ]
\end{forest}
```

Example 3: calculate root mean square of children using the generic `.aggregate` handler.




```
\begin{forest}                                                    (91)
  delay={
    content/.aggregate=
      {0}{aggregate_result()+content()^2}{sqrt(aggregate_result/aggregate_n)}
      {children}
  }
  [[0][1][2][3][4]]
\end{forest}
```

3.15 Relative node names

⟨relative node name⟩=[⟨forest node name⟩][!⟨nodewalk⟩]

⟨relative node name⟩ refers to the FOREST node at the end of the ⟨nodewalk⟩ starting at node named ⟨forest node name⟩. If ⟨forest node name⟩ is omitted, the walk starts at the current node. If ⟨nodewalk⟩ is omitted, the "walk" ends at the start node. (Thus, an empty ⟨relative node name⟩ refers to the current node.)

The ⟨nodewalk⟩ inherits its history from the outer nodewalk (if there is one). Its every-step keylist is empty.

Relative node names can be used in the following contexts:

- FOREST's `pgfmath` option functions (§3.18) take a relative node name as their argument, e.g. `content("!u")` and `content("!parent")` refer to the content of the parent node.
- An option of a non-current node can be set by ⟨relative node name⟩.⟨option name⟩=⟨value⟩, see §3.6.1.
- The `forest` coordinate system, both explicit and implicit; see §3.16.

3.16 The `forest` coordinate system

Unless package options `tikzcshack` is set to `false`, Ti*k*Z's implicit node coordinate system [2, §13.2.3] is hacked to accept relative node names.[30].

The explicit `forest` coordinate system is called simply `forest` and used like this: `(forest cs:`⟨forest cs spec⟩`)`; see [2, §13.2.5]. ⟨forest cs spec⟩ is a keylist; the following keys are accepted.

forest cs `name`=⟨node name⟩ The node with the given name becomes the current node. The resulting point is its (node) anchor.

forest cs `id`=⟨node id⟩ The node with the given name becomes the current node. The resulting point is its (node) anchor.

forest cs `go`=⟨nodewalk⟩ Walk the given nodewalk, starting at the current node. The node at the end of the walk becomes the current node. The resulting point is its (node) anchor. The embedded ⟨nodewalk⟩ inherits history from the outer nodewalk.

forest cs `anchor`=⟨anchor⟩ The resulting point is the given anchor of the current node.

forest cs `l`=⟨dimen⟩

forest cs `s`=⟨dimen⟩ Specify the `l` and `s` coordinate of the resulting point.

The coordinate system is the node's ls-coordinate system: its origin is at its (node) anchor; the l-axis points in the direction of the tree growth at the node, which is given by option `grow`; the s-axis is orthogonal to the l-axis; the positive side is in the counter-clockwise direction from `l` axis.

The resulting point is computed only after both `l` and `s` were given.

Any other key is interpreted as a ⟨relative node name⟩[.⟨anchor⟩].

3.17 Anchors

Forest defines several anchors which can be used with any Ti*k*Z node belonging to a Forest tree (manually added Ti*k*Z nodes are thus excluded).

anchor `parent anchor`

anchor `child anchor`

anchor `anchor`

These anchors point to coordinates determined by node options `parent anchor`, `child anchor` and `anchor`.

anchor `parent`, `parent'`, `-parent`, `-parent'`

anchor `parent first`, `parent first'`, `-parent first`, `-parent first'`

anchor `first`, `first'`

anchor `children first`, `children first'`, `-children first`, `-children first'`

anchor `children`, `children'`, `-children`, `-children'`

anchor `children last`, `children last'`, `-children last`, `-children last'`

anchor `last`, `last'`

anchor `parent last`, `parent`, `-parent last`,`-parent last'`

Growth direction based anchors.

Ti*k*Z's "compass anchors" `east`, `north` etc. resolve to coordinates on the border of the node facing east, north etc. (for the shapes that define these anchors). The above Forest's anchors are similar in that they also resolve to coordinates on the border of the node. However, the "cardinal directions" are determined by the `grow`th direction of the tree in the node and its parent:

- anchor `parent` faces the parent node (or, in case of the root, where the parent would be);
- anchor `children` faces the children (or, in case of a node without children, where the children would be);
- anchor `first` faces the first child (or ... you get it, right?);

[30] Actually, the hack can be switched on and off on the fly, using `\iforesttikzcshack`.

- anchor `last` faces the last child (or ... you know!).

Combinations like `children first` work like combinations of compass directions, e.g. `north west`, but note that

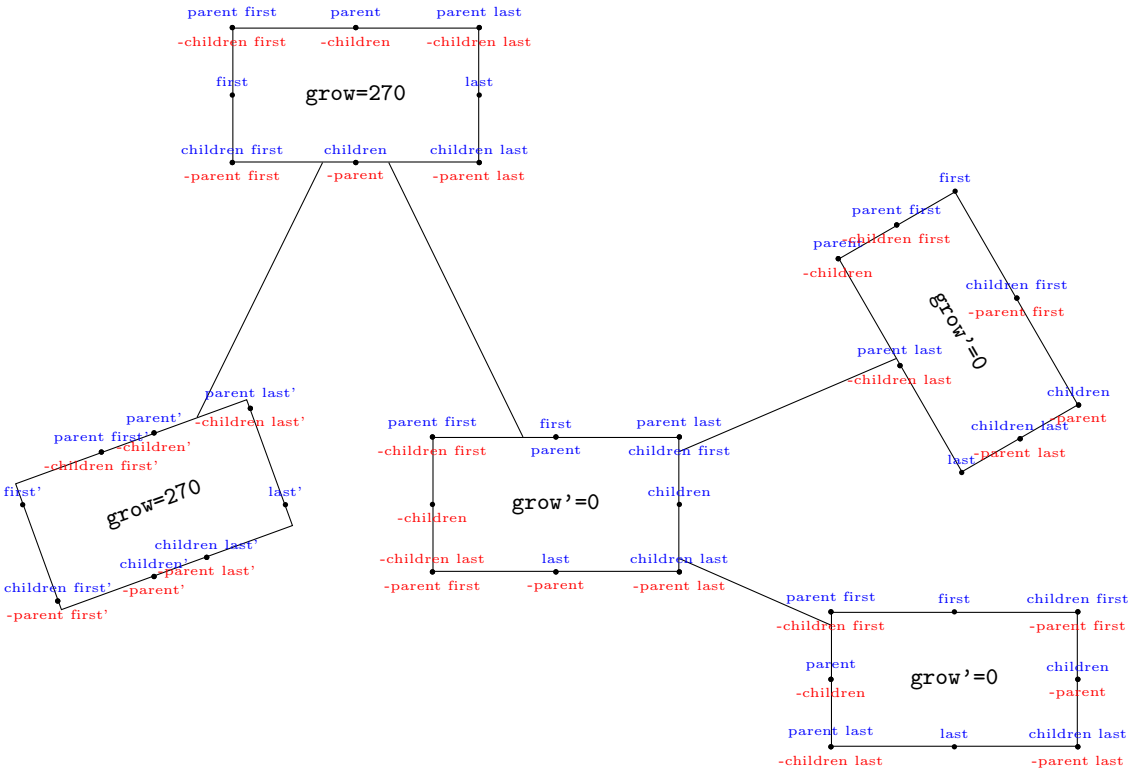
- when `first` and `last` are combined with `parent` into `parent first` and `parent last`, they refer to the first and last child of the parent node, i.e. siblings of the current node.

While `first` and `last` always point in opposite directions, `parent` and `children` do not do so if the `grow`th direction of the tree changes in the node, i.e. if the node's `grow` differs from it's parent's `grow`. Thus in general, it is useful to have anchors `-parent` and `-children`, which point in the opposite directions as `parent` and `children`, respectively, and their combinations with `first` and `last`.

The `'` variants refer precisely to the point where the cardinal growth direction intersects the border. Variants without `'` snap to the closest compass anchor of the node.

These anchors work as expected even if the node is `rotate`d, or if the children are `reversed`.

For simple examples, see definitions of `sn edges` and `roof`; for more involved examples, see the `edges` library.



(92)

```
\def\redorblue#1{\expandafter\redorbluei#1\END}%
\def\redorbluei#1#2\END{\expandafter\ifx#1-red\else blue\fi}%
\forestset{
  draw anchors/.style n args=3{% #1=above, #2=below, #3='-variant of anchor?
    tikz={
      \foreach \a in {first,last,parent first,parent last,children,children last,#1}
        {\fill[](.\a#3)circle[radius=1pt] node[above,font=\tiny,color=\redorblue\a]{\a#3};}
      \foreach \a in {-parent first,-parent,-parent last,-children,-children first,#2}
        {\fill[](.\a#3)circle[radius=1pt] node[below,font=\tiny,color=\redorblue\a]{\a#3};}
    }
  },
  draw anchors/.default={parent,children first}{-children last}{},
}
\begin{forest}
  for tree={
    minimum width=10em, minimum height=13ex, s sep+=5em,
    draw, draw anchors,
    font=\tt, delay={content/.process=OOw2{grow}{reversed}{grow\ifnum#2=1'\fi=#1}}
  }
  [
    [,rotate=20,draw anchors={parent,children first}{-children last}{'}]
    [,for tree={grow'=0}, l sep+=5em, draw anchors={-children last}{parent,children first}{}
      [,rotate=-60]
      []
    ]
  ]
\end{forest}
```

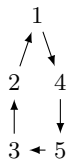
3.18 Additional pgfmath functions

For every option and register, FOREST defines a `pgfmath` function with the same name, with the proviso that the name might be mangled in order to conform to `pgfmath`'s naming rules. Specifically, all non-alphanumeric characters in the option/register name and the initial number, if the name starts with one, are replaced by an underscore `_` in the `pgfmath` function name.

Pgfmath functions corresponding to options take one argument, a ⟨relative node name⟩ (see §3.15) expression, making it possible to refer to option values of non-current nodes. The ⟨relative node name⟩ expression must be enclosed in double quotes in order to prevent pgfmath evaluation: for example, to refer to the content of the parent, write `content("!u")`. To refer to the option of the current node, use empty parentheses: `content()`.[31]

If the ⟨relative node name⟩ resolves to the invalid node, these functions will an return empty token list (for ⟨toks⟩ options), 0pt (for ⟨dimen⟩ options) or 0 (for ⟨count⟩ options).

Note that the nodewalk in the relative node name inherits its history from the outer nodewalk (if there is one), so strange but useful constructions like the following are possible.




```
\begin{forest}                                                    (93)
  for tree={no edge},
  before typesetting nodes={
    for nodewalk={
      c,
      every step={
        tikz/.wrap pgfmath arg=
          {\draw[<-] ()--(#1);}
          {name("!b")}
        },
      21{up1},ancestors
    }{}
  },
  [1[2[3]][4[5]]]
\end{forest}
```

Boolean function `valid` returns true if the node's `id`$\neq 0$, i.e. if the node is a real, valid node; see §3.5.1 and §3.8. Boolean function `invalid` is a negation of `valid`.

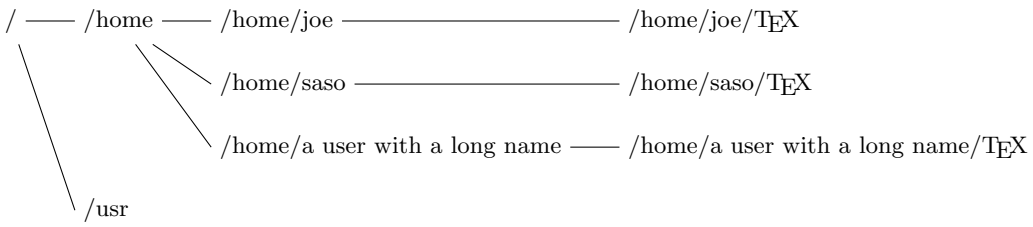
[31] In most cases, the parentheses are optional, so `content` is ok. A known case where this doesn't work is preceding an operator: `l+1cm` will fail.

pgfmath function `min_l`=(⟨nodewalk: node⟩,⟨nodewalk: context node⟩)
pgfmath function `min_s`=(⟨nodewalk: node⟩,⟨nodewalk: context node⟩)
pgfmath function `max_l`=(⟨nodewalk: node⟩,⟨nodewalk: context node⟩)
pgfmath function `max_s`=(⟨nodewalk: node⟩,⟨nodewalk: context node⟩)

These functions return the minimum/maximum value of `l`/`s` of node at the end of ⟨nodewalk: node⟩ in the context (i.e. growth direction) of node at the end of ⟨nodewalk: context node⟩.

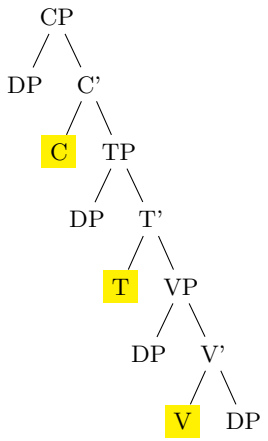
Three string functions are also added to `pgfmath`: `strequal` tests the equality of its two arguments; `instr` tests if the first string is a substring of the second one; `strcat` joins an arbitrary number of strings.

Some random notes on `pgfmath`: (i) `&&`, `||` and `!` are boolean "and", "or" and "not", respectively. (ii) The equality operator (for numbers and dimensions) is `==`, *not* `=`. And some examples:



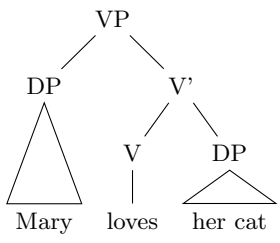
(94)

```
\begin{forest}
  for tree={grow'=0,calign=first,l=0,l sep=2em,child anchor=west,anchor=base
    west,fit=band,tier/.pgfmath=level()},
  fullpath/.style={if n=0{}{content/.wrap 2
      pgfmath args={##1/##2}{content("!u")}{content()}}},
  delay={for tree=fullpath,content=/},
  before typesetting nodes={for tree={content=\strut#1}}
  [
    [home
      [joe
        [\TeX]]
      [saso
        [\TeX]]
      [a user with a long name
        [\TeX]]]
    [usr]]
\end{forest}
```



(95)

```
% mark non-phrasal terminal nodes
\begin{forest}
  delay={for tree={if=
    {!instr("P",content) && n_children==0}
    {fill=yellow}
    {}
  }}
  [CP[DP][C'[C][TP[DP][T'[T][VP[DP][V'[V][DP]]]]]]]
\end{forest}
```



(96)

```
% roof terminal phrases
\useforestlibrary{linguistics}
% ...
\begin{forest}
  delay={where n children=0{tier=word,
    if={instr("P",content("!u"))}{roof}{}
  }{}},
  [VP[DP[Mary]][V'[V[loves]][DP[her cat]]]]
\end{forest}
```

3.19 Standard node

macro `\forestStandardNode`⟨node⟩⟨environment fingerprint⟩⟨calibration procedure⟩⟨exported options⟩

This macro defines the current *standard node*. The standard node declares some options as *exported*. When a new node is created, the values of the exported options are initialized from the standard node. At the beginning of every `forest` environment, it is checked whether the *environment fingerprint* of the standard node has changed. If it did, the standard node is *calibrated*, adjusting the values of exported options. The *raison d'etre* for such a system is given in §2.4.1.

In ⟨node⟩, the standard node's content and possibly other options are specified, using the usual bracket representation. The ⟨node⟩, however, *must not contain children*. The default: `[dj]`.

The ⟨environment fingerprint⟩ must be an expandable macro definition. It's expansion should change whenever the calibration is necessary.

⟨calibration procedure⟩ is a keylist (processed in the `/forest` path) which calculates the values of exported options.

⟨exported options⟩ is a comma-separated list of exported options.

This is how the default standard node is created:

```
\forestStandardNode[dj]
  {%
    \forestOve{\csname forest@id@of@standard node\endcsname}{content},%
    \the\ht\strutbox,\the\pgflinewidth,%
    \pgfkeysvalueof{/pgf/inner ysep},\pgfkeysvalueof{/pgf/outer ysep},%
    \pgfkeysvalueof{/pgf/inner xsep},\pgfkeysvalueof{/pgf/outer xsep}%
  }
  {
    l sep={\the\ht\strutbox+\pgfkeysvalueof{/pgf/inner ysep}},
    l={l_sep()+abs(max_y()-min_y())+2*\pgfkeysvalueof{/pgf/outer ysep}},
    s sep={2*\pgfkeysvalueof{/pgf/inner xsep}}
  }
  {l sep,l,s sep}
```

3.20 Externalization

Externalized tree pictures are compiled only once. The result of the compilation is saved into a separate `.pdf` file and reused on subsequent compilations of the document. If the code of the tree (or the context, see below) is changed, the tree is automatically recompiled.

Externalization is enabled by:

```
\usepackage[external]{forest}
\tikzexternalize
```

Both lines are necessary. Ti*k*Z's externalization library is automatically loaded if necessary.

`external/optimize` Parallels `/tikz/external/optimize`: if `true` (the default), the processing of non-current trees is skipped during the embedded compilation.

`external/context` If the expansion of the macro stored in this option changes, the tree is recompiled.

`external/depends on macro`=⟨cs⟩ Adds the definition of macro ⟨cs⟩ to `external/context`. Thus, if the definition of ⟨cs⟩ is changed, the tree will be recompiled.

FOREST respects or is compatible with several (not all) keys and commands of Ti*k*Z's externalization library. In particular, the following keys and commands might be useful; see [2, §32].

- `/tikz/external/remake next`
- `/tikz/external/prefix`
- `/tikz/external/system call`
- `\tikzexternalize`
- `\tikzexternalenable`

- `\tikzexternaldisable`

FOREST does not disturbe the externalization of non-FOREST pictures. (At least it shouldn't ...)

The main auxiliary file for externalization has suffix `.for`. The externalized pictures have suffices `-forest-`n (their prefix can be set by `/tikz/external/prefix`, e.g. to a subdirectory). Information on all trees that were ever externalized in the document (even if they were changed or deleted) is kept. If you need a "clean" `.for` file, delete it and recompile. Deleting `-forest-`n`.pdf` will result in recompilation of a specific tree.

Using `draw tree` and `draw tree'` multiple times *is* compatible with externalization, as is drawing the tree in the box (see `draw tree box`). If you are trying to externalize a `forest` environment which utilizes `TeX` to produce a visible effect, you will probably need to use `TeX'` and/or `TeX''`.

4 Libraries

This chapter contains not only the reference of commands found in libraries and some examples of their usage, but also their definitions. This is done in the hope that these definitions, being mostly styles, will be useful as examples of the core features of the package. I even managed to comment them a bit ...

Disclaimer. At least in the initial stages of a library's development, the function and interface of macros and keys defined in a library might change without backwards compatibility support! Though I'll try to keep this from happening ...

```
\RequirePackage{forest}
```

4.1 linguistics

```
\ProvidesForestLibrary{linguistics}[2015/11/14 v0.1]
```

Defaults:

```
\forestset{
  libraries/linguistics/defaults/.style={
    default preamble={
```

Edges of the children will "meet" under the node:

```
      sn edges,
```

The root of the tree will be aligned with the text ... or, more commonly, the example number.

```
      baseline,
```

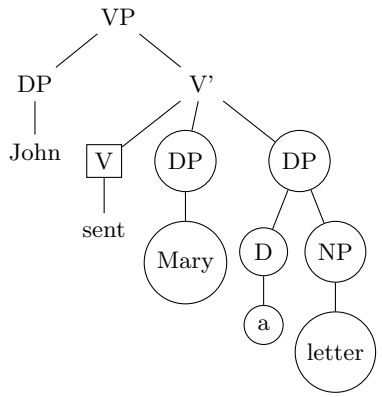
Enable (centered) multi-line nodes.

```
      for tree={align=center},
    },
  },
}
```

There's no linguistics without c-command[32] ...

step `c-commanded` Visit all the nodes c-commanded by the current node.

step `c-commanders` Visit all the c-commanders of the current node, starting from the closest.



(97)

```
\begin{forest}
  [VP
    [DP[John]]
    [V'
      [V, draw, for c-commanded={draw,circle}
        [sent]
      ]
      [DP[Mary]]
      [DP[D[a]][NP[letter]]]
    ]
  ]
\end{forest}
```

[32] The definition of c-command is as follows: a node c-commands its siblings and their descendants.

See how `branch'` is used to define `c-commanded`, and how `while nodewalk valid` and `fake` are combined in the definition of `c-commanders`.

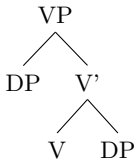
```
\forestset{
  define long step={c-commanded}{style}{branch'={siblings,descendants}},
  define long step={c-commanders}{style}{while nodewalk valid={parent}{siblings,fake=parent}},
}
```

`c-commanders` could also be defined using `branch`:

```
branch={current and ancestors, siblings}
```

sn edges

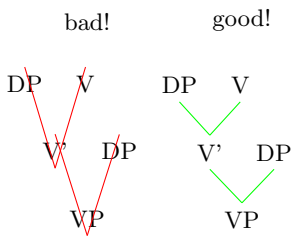
In linguistics, most people want the parent-child edge to go from the *s*outh of the parent to the *n*orth of the child. This is achieved by this (badly named) style, which makes the entire (sub)tree have such edges.




```
\begin{forest}
  sn edges
  [VP
    [DP]
    [V'
      [V]
      [DP]
    ]
  ]
\end{forest}
```
(98)

```
\forestset{
  sn edges/.style={
    for tree={
      parent anchor=children, child anchor=parent
    }
  },
}
```

A note on implementation. Despite its name, this style does not refer to the `south` and `north` anchor of the parent and the child node directly. If it did so, it would only work for trees with standard linguistic `grow=-90`. So we rather use FOREST's growth direction based anchors: `children` always faces the children and `parent` always faces the parent, so the edge will always be between them, and the normal, upward growing trees will look good as well.




```
\begin{forest}
  [bad![VP,no edge, for tree={grow=90, edge=red},
    for tree={parent anchor=south, child anchor=north} % bad
  [DP][V'[V][DP]]]]
\end{forest}
\begin{forest}
  [good![VP, no edge, for tree={grow=90, edge=green},
            sn edges                                  % good!
  [DP][V'[V][DP]]]]
\end{forest}
```
(99)

`roof` Makes the edge to parent a triangular roof.

```
\forestset{
  roof/.style={edge path'={%
      (.parent first)--(!u.children)--(.parent last)--cycle
    }
  },
}
```

nice empty nodes

We often need empty nodes: tree (100a) shows how they look like by default: ugly.

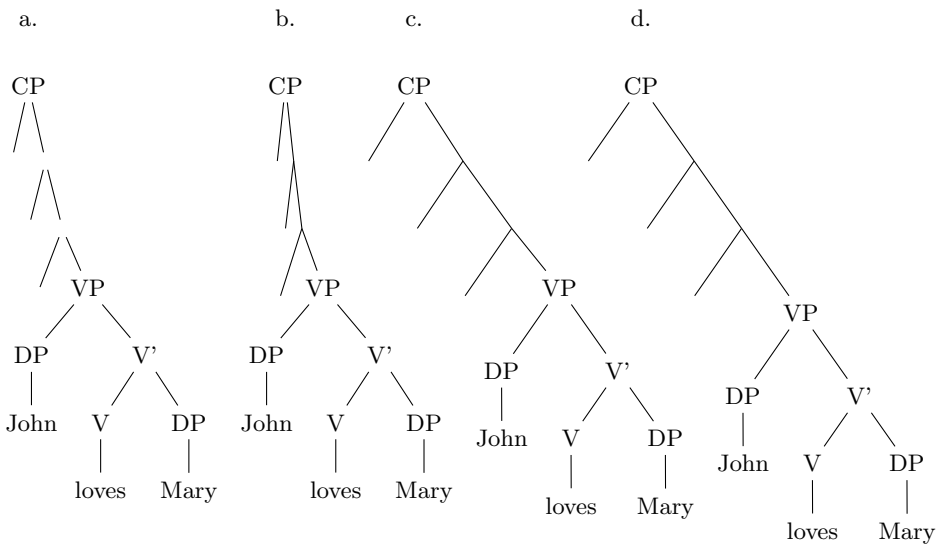
First, we don't want the gaps: we change the shape of empty nodes to coordinate. We get tree (100b).

Second, the empty nodes seem too close to the other (especially empty) nodes (this is a result of a small default `s sep`). We could use a greater `s sep`, but a better solution seems to be to use `calign=fixed angles`. The result is shown in (100c).

However, at the transitions from empty to non-empty nodes, tree (100c) seems to zigzag (although the base points of the spine nodes are perfectly in line), and the edge to the empty node left to VP seems too long (it reaches to the level of VP's base, while we'd prefer it to stop at the same level as the edge to VP itself). The first problem is solved by substituting `fixed angles` for `fixed edge angles`; the second one, by anchoring siblings of empty nodes at north. Voil, (100d)!

```
\forestset{
  xlist/.style={
    phantom,
    for children={no edge,replace by={[,append,
        delay={content/.wrap pgfmath arg={\csname @alph\endcsname{##1}.}{n()+#1}}
        ]}}
  },
  xlist/.default=0
}
\begin{forest}
  [,xlist,
  for tree={after packing node={s+=0.1pt}}, % hack!!!
    [CP,                                                          %(a)
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
    [CP, delay={where content={}{shape=coordinate}{}}             %(b)
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
    [CP, for tree={calign=fixed angles},                          %(c)
         delay={where content={}{shape=coordinate}{}}
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
    [CP, nice empty nodes                                         %(d)
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
  ]
\end{forest}
```

(100)




```
\forestset{
  nice empty nodes/.style={
    for tree={calign=fixed edge angles},
    delay={where content={}{shape=coordinate,for parent={
          for children={anchor=north}}}{}}
  },
}
```

`draw brackets` Outputs the bracket representation of the tree.

`draw brackets compact`

`draw brackets wide` These keys control whether the brackets have extra spaces around them (`wide`) or not (`compact`).

```
    \providecommand\text[1]{\mbox{\scriptsize#1}}
    \forestset{
      draw brackets compact/.code={\let\drawbracketsspace\relax},
      draw brackets wide/.code={\let\drawbracketsspace\space},
      draw brackets/.style={
```

There's stuff to do both before (output the opening bracket and the content) and after (output the closing bracket) processing the children, so we use `for tree'`.

```
        for tree'={
          TeX={[%
```

Complication: `content format` must be expanded in advance, to correctly process tabular environments implicitely loaded by `align=center`, which is the default in this library. (Not that one would want a multiline output in the bracket representation, but it's better than crashing.)

```
            \edef\forestdrawbracketscontentformat{\foresteoption{content format}}%
          },
          if n children=0{
            TeX={\drawbracketsspace\forestdrawbracketscontentformat\drawbracketsspace}
          }{
            TeX={\textsubscript{\text{\forestdrawbracketscontentformat}}\drawbracketsspace}
          },
        }{
          TeX={]\drawbracketsspace},
        }
      },
      draw brackets wide
    }
```

4.1.1 GP1

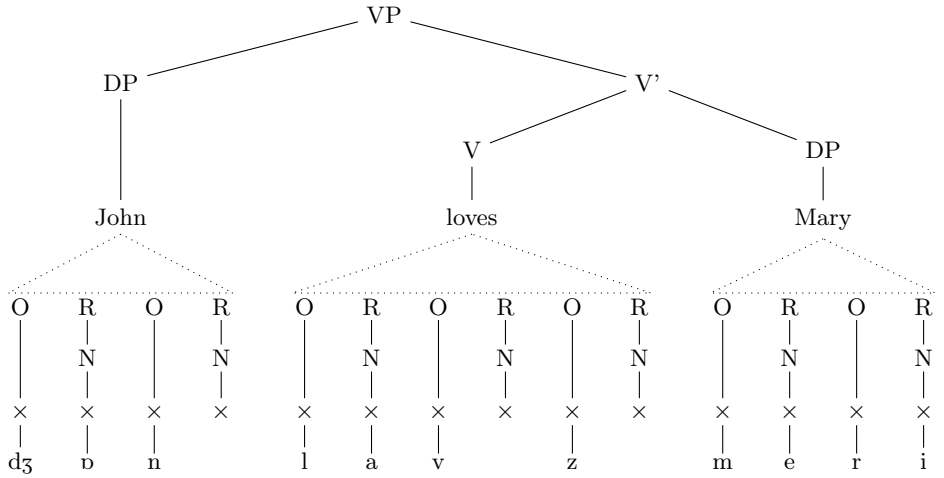
`GP1`

For Government Phonology (v1) representations. Here, the big trick is to evenly space ×s by having a large enough `outer xsep` (adjustable), and then, before drawing (timing control option `before drawing tree`), setting `outer xsep` back to 0pt. The last step is important, otherwise the arrows between ×s won't draw!

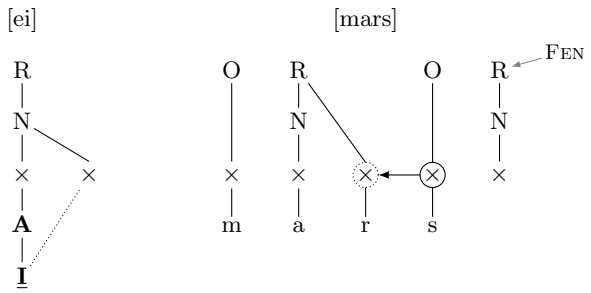
An example of an "embedded" `GP1` style:

```
\begin{forest}
  myGP1/.style={
    GP1,
    delay={where tier={x}{
        for children={content=\textipa{##1}}}{}},
    tikz={\draw[dotted](.south)--
          (!1.north west)--(!l.north east)--cycle;},
    for children={l+=5mm,no edge}
  }
  [VP[DP[John,tier=word,myGP1
          [O[x[dZ]]]
          [R[N[x[6]]]]
          [O[x[n]]]
          [R[N[x]]]
  ]][V'[V[loves,tier=word,myGP1
           [O[x[l]]]
           [R[N[x[a]]]]
           [O[x[v]]]
           [R[N[x]]]
           [O[x[z]]]
           [R[N[x]]]
  ]][DP[Mary,tier=word,myGP1
         [O[x[m]]]
         [R[N[x[e]]]]
         [O[x[r]]]
         [R[N[x[i]]]]
  ]]]]
\end{forest}%
```

(101)



And an example of annotations.




```
\begin{forest}[,phantom,s sep=1cm
  [{[ei]}, GP1
    [R[N[x[A,el[I,head,associate=N]]][x]]]
  ]
  [{[mars]}, GP1
    [O[x[m]]]
    [R[N[x[a]]][x,encircle,densely dotted[r]]]
    [O[x,encircle,govern=<[s]]]
    [R,fen[N[x]]]
  ]
]\end{forest}
```

(102)

```
\newbox\standardnodestrutbox
\setbox\standardnodestrutbox=\hbox to 0pt{\phantom{\forestOve{standard node}{content}}}
\def\standardnodestrut{\copy\standardnodestrutbox}
\forestset{
  GP1/.style 2 args={
```

```
      for n={1}{baseline},
      s sep=0pt, l sep=0pt,
      for descendants={
        l sep=0pt, l={#1},
        anchor=base,calign=first,child anchor=north,
        inner xsep=1pt,inner ysep=2pt,outer sep=0pt,s sep=0pt,
      },
      delay={
        if content={}{phantom}{for children={no edge}},
        for tree={
          if content={O}{tier=OR}{},
          if content={R}{tier=OR}{},
          if content={N}{tier=N}{},
          if content={x}{
            tier=x,content={$\times$},outer xsep={#2},
            for tree={calign=center},
            for descendants={content format={\noexpand\standardnodestrut\forestoption{content}}},
            before drawing tree={outer xsep=0pt,delay={typeset node}},
            s sep=4pt
          }{},
        },
      },
      before drawing tree={where content={}{parent anchor=center,child anchor=center}{}},
    },
    GP1/.default={5ex}{8.0pt},
    associate/.style={%
      tikz+={\draw[densely dotted](!)--(!#1);}},
    spread/.style={
      before drawing tree={tikz+={\draw[dotted](!)--(!#1);}}},
    govern/.style={
      before drawing tree={tikz+={\draw[->](!)--(!#1);}}},
    p-govern/.style={
      before drawing tree={tikz+={\draw[->](.north) to[out=150,in=30] (!#1.north);}}},
    no p-govern/.style={
      before drawing tree={tikz+={\draw[->,loosely dashed](.north) to[out=150,in=30] (!#1.north);}}},
    encircle/.style={before drawing tree={circle,draw,inner sep=0pt}},
    fen/.style={pin={[font=\footnotesize,inner sep=1pt,pin edge=<-]10:\textsc{Fen}}},
    el/.style={content=\textsc{\textbf{##1}}},
    head/.style={content=\textsc{\textbf{\underline{##1}}}}
  }
```

4.2 edges

```
\ProvidesForestLibrary{edges}[2016/12/05 v0.1.1]
```

forked edge'

Sets a forked edge to the current node. Arbitrary growth direction and node rotation are supported.

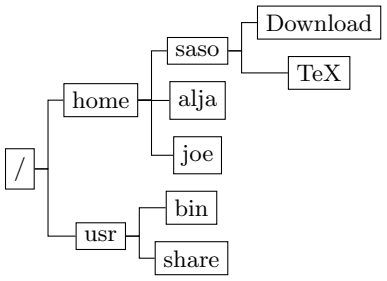
forked edge

Like forked edge', but it also sets parent anchor and child anchor to the likely values of children and parent, respectively.

forked edges=⟨nodewalk⟩ tree

Invokes forked edge for all nodes in the ⟨nodewalk⟩, by default the entire (sub)tree rooted in the current node.

option fork sep The l-distance between the parent anchor and the fork.




```
\begin{forest}                                                    (103)
  for tree={grow'=0,draw},
  forked edges,
  [/
    [home
      [saso
        [Download]
        [TeX]
      ]
      [alja]
      [joe]
    ]
    [usr
      [bin]
      [share]
    ]
  ]
\end{forest}
```

See how growth direction based anchors `children` and `parent` are used in the definition below to easily take care of arbitrary `grow` and `rotate`.

```
\forestset{
  declare dimen={fork sep}{0.5em},
  forked edge'/.style={
    edge={rotate/.option=!parent.grow},
    edge path'={(!u.parent anchor) -- ++(\forestoption{fork sep},0) |- (.child anchor)},
  },
  forked edge/.style={
    on invalid={fake}{!parent.parent anchor=children},
    child anchor=parent,
    forked edge',
  },
  forked edges/.style={for nodewalk={#1}{forked edge}},
  forked edges/.default=tree,
}
```

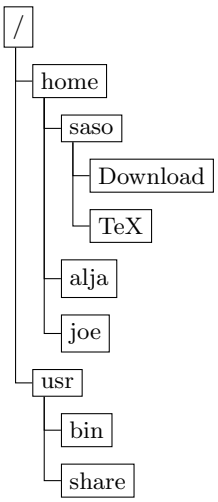
`folder` The children of the node are drawn like folders.

All growth directions are supported (well, cardinal directions work perfectly; the others await the sensitivity of packing to `edge path`), as well as node rotation and `reversed` order of children.

The outlook of the folder can be influenced by setting standard FOREST's options `l sep` and `s sep` any time before packing, or `l` and `s` after packing. Setting `l` and `s` before packing will have no influence on the layout of the tree.

register `folder indent`=⟨dimen⟩ `.45em`

Specifies the shift of the parent's side of the edge in the `l`-direction.



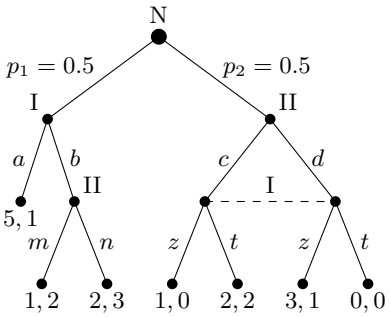

```
\begin{forest}                                                    (104)
  for tree={grow'=0,folder,draw}
  [/
    [home
      [saso
        [Download]
        [TeX]
      ]
      [alja]
      [joe]
    ]
    [usr
      [bin]
      [share]
    ]
  ]
\end{forest}
```

```
\forestset{
  declare dimen register=folder indent,
  folder indent=.45em,
  folder/.style={
    parent anchor=-children last,
    anchor=parent first,
    calign=child,
    calign primary child=1,
    for children={
      child anchor=parent,
      anchor=parent first,
      edge={rotate/.option=!parent.grow},
      edge path'/.expanded={
        ([xshift=\forestregister{folder indent}]!u.parent anchor) |- (.child anchor)
      },
    },
    after packing node={
      if n children=0{}{
        tempdiml=l_sep()-l("!1"),
        tempdims={-abs(max_s("","")-min_s("",""))-s_sep()},
        for children={
          l+=tempdiml,
          s+=tempdims()*(reversed()-0.5)*2,
        },
      },
    },
  }
}
```

5 Gallery

5.1 Decision tree

The following example was inspired by a question on TEX Stackexchange: How to change the level distance in tikz-qtree for one level only?. The question is about `tikz-qtree`: how to adjust the level distance for the first level only, in order to avoid first-level labels crossing the parent–child edge. While this example solves the problem (by manually shifting the offending labels; see `elo` below), it does more: the preamble is setup so that inputing the tree is very easy.



(105)

```
\forestset{
  declare toks={elo}{}, % Edge Label Options
  anchors/.style={anchor=#1,child anchor=#1,parent anchor=#1},
  dot/.style={tikz+={\fill (.child anchor) circle[radius=#1];}},
  dot/.default=2pt,
  decision edge label/.style n args=3{
    edge label/.expanded={node[midway,auto=#1,anchor=#2,\forestoption{elo}]{\strut$\unexpanded{#3}$}}
  },
  decision/.style={if n=1
    {decision edge label={left}{east}{#1}}
    {decision edge label={right}{west}{#1}}
  },
  decision tree/.style={
    for tree={
      s sep=0.5em,l=8ex,
      if n children=0{anchors=north}{
        if n=1{anchors=south east}{anchors=south west}},
      math content,
    },
    anchors=south, outer sep=2pt,
    dot=3pt,for descendants=dot,
    delay={for descendants={split option={content}{;}{content,decision}}},
  }
}
\begin{forest} decision tree
  [N,plain content
    [I;{p_1=0.5},plain content,elo={yshift=4pt}
      [{5,1};a]
      [II;b,plain content
        [{1,2};m]
        [{2,3};n]
      ]
    ]
    [II;{p_2=0.5},plain content,elo={yshift=4pt}
      [;c
        [{1,0};z]
        [{2,2};t]
      ]
      [;d
        [{3,1};z]
        [{0,0};t]
      ]
    ] {\draw[dashed](!1.anchor)--(!2.anchor) node[pos=0.5,above]{I};}
  ]
\end{forest}
```

5.2 forest-index

The indexing system used to document the FOREST package uses the package itself quite heavily. While this might be a bit surprising at first sight, as indexing draws no trees, the indexing package illustrates the usage of some of the more exotic features and usage-cases of the FOREST package, which is why its source is included in this documentation.[33]

This package has three main functions:

- It is possible to index subentries using a *short form* of their index key, i.e. without referring to their ancestor entries. For example, instead of writing `\index{option>content}` one can simply write `\index{content}`. (Obviously, the subentry must "content" be defined as belonging to entry "option" first. This is done using `\indexdef{option>content}`.) This works for all keys which are a subentry of a single entry.

- All subentries are automatically entered as main entries as well, with a qualificator of which entry they belong to. So, `\index{option>content}` produces two index entries: entry "option" with subentry

[33] Indexing with this package makes the compilation very slow, so I cannot whole-heartedly recommend it, but I still hope that it will make a useful example.

"content" and entry "content option". This works for an arbitrary number of subentry levels.

- Entries can be given options that format the appearance of the entry and/or its descendants in both text and index. (Entries that format the appearance of their descendants are called categories below.)
- If `hyperref` package is loaded, the following hyperlinks are created besides the standard ones linking the page numbers in index to text: (i) entries in text link to the definition in text, (ii) definitions in the text link to the index entry, (iii) categories in index are cross-linked.

The Forest package mainly enters the picture with respect to the entry formatting. A simple (narrow) tree is built containing an entry and all its ancestors. Formatting instructions are then processed using Forest's option processing mechanisms.

Finally, note that this package might change without retaining backwards compatibility, and that changes of this package will not be entered into the changelog.

Identification.

```
\ProvidesPackage{forest-index}
\RequirePackage{forest}
```

Remember the original LaTeX's `\index` command.

```
\let\forestindex@LaTeX@index\index
```

The user interface macros

`\index` `\indexdef` `\indexex` `\indexitem` `\indexset`

`\index` is the general purpose macro. `\indexdef` and `\indexex` are shorthands for indexing definitions and examples. `\indexitem` is a combination of `\indexdef` and the `\item` of the `lstdoc` package. It automatically indexes the command being documented. `\indexset` neither typesets or indexes the entry, but only configures it; it is usually used to configure categories. All these macros parse their arguments using `xparse`. The arguments, listed in the reverse order:

- The final argument, which is the only mandatory argument, is a comma-separated list of index keys.
- The boolean switch `>` just before the mandatory argument signals that the keys are given in the full form. Otherwise, keys without a level separator are considered short.
- Indexing options are given by the `[optional ]` argument.
- The first (optional) argument of:
  - `\indexitem`: specifies the default value of the command.
  - `\index`: is used to provide "early" options.

Among the options of these commands, three keylists are of special importance: `index key format`, `index form format` and `print format`. These hold instructions on how to format the index key, the form of the entry in the index and the form of the entry in the main text. They work by modifying the contents of an ⟨autowrapped toks⟩ register `result`.

An example: how macros are indexed in this documentation. Style `macro` defined below does everything needed to format a macro name: it detokenizes the given name (in case the name contains some funny characters), prefixes the backslash, wraps in in the typewriter font, adds color and hyperlink (the final two styles are defined in below this package). Note the usage of `\protect`: it is needed because we want to use these styles to format entries not just in the main next, but also in the index.

```
\forestset{
  detokenize/.style={result=\protect\detokenize{##1}},
  tt/.style={result=\protect\texttt{##1}},
  macro/.style={detokenize, +result={\char\escapechar}, tt, print in color, hyper},
}
```

Then, we configure the main level entry "macro": the child of this entry will be formatted (both in index and in the main text) using the previously defined style.

```
\indexset
  [for first={format=macro}]
  >{macro}
```

Usage is then simple: we write `\indexex{macro>forestoption}` (or simply `\indexex{forestoption}` to get \forestoption.

```
\DeclareDocumentCommand\indexdef{O{} t> m}{%
  \IfBooleanTF{#2}
    {\let\forestindex@resolvekey\forestindex@resolvekey@long}
    {\let\forestindex@resolvekey\forestindex@resolvekey@shortorlong}%
  \forestindex@index{definition}{#1}{#3}}
\DeclareDocumentCommand\indexex{O{} t> m}{%
  \IfBooleanTF{#2}
    {\let\forestindex@resolvekey\forestindex@resolvekey@long}
    {\let\forestindex@resolvekey\forestindex@resolvekey@shortorlong}%
  \forestindex@index{example}{#1}{#3}}
% \DeclareDocumentCommand\indexitem{D(){} O{} t> m}{%
%   \IfBooleanTF{#3}
%     {\let\forestindex@resolvekey\forestindex@resolvekey@long}
%     {\let\forestindex@resolvekey\forestindex@resolvekey@shortorlong}%
%   \forestindex@index{definition}{default={#1},print format=item,#2}{#4}}
\DeclareDocumentCommand\indexitem{D(){} O{} t> m}{%
  \let\forestindex@resolvekey\forestindex@resolvekey@long
  \forestindex@index{definition}{default={#1},#2,print format+=item}{#4}}
\DeclareDocumentCommand\indexset{O{} t> m}{%
  \IfBooleanTF{#2}
    {\let\forestindex@resolvekey\forestindex@resolvekey@long}
    {\let\forestindex@resolvekey\forestindex@resolvekey@shortorlong}%
  \forestindex@index{not print,not index,definition}{set={#1}}{#3}}
\DeclareDocumentCommand\index{D(){} O{} t> m}{%
  \IfBooleanTF{#3}
    {\let\forestindex@resolvekey\forestindex@resolvekey@long}
    {\let\forestindex@resolvekey\forestindex@resolvekey@shortorlong}%
  \forestindex@index{#1}{#2}{#4}%
}
```

All UI macros call this macro.

#1 early option keylist

#2 late option keylist

#3 a comma-sep list of forest index key (full or short form). A key can be given an argument using `key=argument` syntax. How the argument is used is up to the user. For example, the "environment" entry of the FOREST documentation uses it to typeset the contents of the environment:

```
\indexitem{environment>forest={[\texttt{(}\meta{config}\texttt{)}]\meta{tree}}}
```

```
\def\forestindex@index#1#2#3{%
```

Partition the index keylist into single keys. And put it all in a group: the persistent stuff is saved globally.

```
  {\forcsvlist{\forestindex@forkey{#1}{#2}}{#3}}%
}
\def\forestindex@forkey#1#2#3{%
```

Short-key resolution. The result is stored into `\forestindex@fullkey`.

```
  \forestindex@resolvekey{#3}%
```

Manipulate arguments a bit, so that we can use our quick-and-dirty one-arg memoization.

```
  %\forestset{@index/.process={__o}{#1}{#2}{\forestindex@fullkey}}
  \edef\forest@marshal{%
    \noexpand\forestindex@index@{%
      {\unexpanded{#1}}%
      {\unexpanded{#2}}%
      {\expandonce{\forestindex@fullkey}}%
    }%
  }\forest@marshal
}
```

Call the central processing command, style `@index`. See how `.process` is used to expand (once) the last argument.

```
\def\forestindex@index@#1{\forestset{@index/.process={__o}#1}}
\forestset{
```

Declarations

Should we print and/or index the entry? For example, `\index[not print]{...}` will index silently (as LaTeX's `\index` command does).

```
declare boolean register=print,
declare boolean register=index,
declare boolean register=short, short,
```

Options `name`, `content`, `key` and `argument` hold info about the current entry. We need to declare only the latter two, the former two we steal from Forest.

```
declare toks={key}{},
declare toks={argument}{},
```

These options will hold first the initial, and then the calculated values of the index key, index form and the form in text. When (late) options are executed, these options are initialized to the value of option `key`; it is safe to modify them at this point. Afterwards, they will be further processed by keylists `index key format`, `index form format` and `print format`, respectively.

```
declare toks={index key}{},
declare toks={index form}{},
declare toks={print form}{},
```

The customization of entries' appearance is done by specifying the following three keylists. The keylists work by modifying register `result`.

```
declare keylist={index key format}{},
declare keylist={index format}{},
declare keylist={print format}{},
declare autowrapped toks register=result,
```

Some shorthands.

```
format'/.style={print format'={#1}, index format'={#1}},
format/.style={print format={#1}, index format={#1}},
format+/.style={print format+={#1}, index format+={#1}},
+format/.style={+print format={#1}, +index format={#1}},
form/.style={print form={#1},index form={#1}},
form+/.style={print form+={#1},index form+={#1}},
+form/.style={+print form={#1},+index form={#1}},
```

Entry types are normal (default), definition, example. Only definitions are special, as their options are automatically saved.

```
declare toks register=index entry type,
definition/.style={index entry type=definition},
normal/.style={index entry type=normal},
example/.style={index entry type=example},
normal,
```

This option is used internally to store the hyper ids.

```
declare toks={index@hypertarget}{},
every index begin/.style={},
every index end/.style={},
```

Some formatting tools need to know whether we're typesetting text or index: this info is stored in the `stage` register.

declare toks register

```
declare toks register=stage,
```

The central processing command

#1 early option keylist (these are only used to define category "@unknown" at the end of this package)
#2 late option keylist
#3 index key (full form)

```
@index/.style n args={3}{
```

Set the defaults.

```
  print, index, index entry type=normal, set'={},
```

Create the tree structure: `[entry[subentry[subsubentry...]]]`. Three options of every node created:

- `key` contains the key of the (sub)entry
- `name` contains the full path to the (sub)entry

- `arguments` contains the arguments given to the (sub)entry's key
- `content` contains the full key, with arguments for all but the most deeply embedded subentry

`for nodewalk` is used because `create@subentry@node` walks down the created tree. At `if n=0` below, we're thus positioned at the lowest node.

nodewalk

```
79    for nodewalk={
```

The components of the full key are separated using `split`, with different keys being executed for the first component and the rest.

```
80      split={#3}{>}{create@main@entry@node, create@subentry@node},
```

Remove the argument from the most deeply embedded subentry.

```
81      if n=0{
82        content/.option=key,
83      }{
84        content/.process={OOw2} {!parent.content} {key} {##1>##2},
85      }
86    }{},
87    for root'={
```

Don't memoize if the key is of an unknown category.

```
88      if strequal/.process={0}{!root.name}{@unknown}{TeX=\global\forest@memoizing@ok@false}{},
```

Option `print form` is what will be typeset in the text. Option `index key` is the key that will be used for sorting the index. Option `index form` is what will be typeset in the index. All these are initialized to the `key`. See how `.option` is used to assign an option value to another option.

```
89      for tree={
90        print form/.option=key,
91        index key/.option=key,
92        index form/.option=key,
93      },
```

Below, `on invalid` is set to `fake` at four points. This is so we won't get in trouble when `\indexsetting` the categories: when the category formatting code will try to step into the child, it will fail as the child does not exist when `\indexset` is called for the category; but we ignore the failure.

first leaf'

Go to the the most deeply embedded subentry.

```
94      for first leaf'={
```

Execute every index options and the given early options.

```
95        on invalid={fake}{
96          every index begin,
97          #1,
98        },
```

Ancestors are walked in the `reverse` order (top down). At every node, the saved configuration is executed (if it exists).

```
99        for reverse={current and ancestors}{on invalid={fake}{@@index/\forestoption{name}/.try}},
```

We don't execute the saved configuration for definitions, as definitions are where the configuration is set.

```
100       if index entry type={definition}{}{%
101         on invalid={fake}{@@index/\forestoption{name}/.try},
102       },
```

Execute late (well, normal) options. See the discussion about early options above.

```
103       on invalid={fake}{
104         #2,
105         every index end
106       },
```

Remember the given config for the rest of the document.

```
107       if set={}{}{save@session},
```

If we're at a definition, save the config into the auxiliary file.

```
108       if index entry type={definition}{save@foridx}{},
109     },
110     stage={},
```

Create hyperlink targets of the form `.entry.subentry.subsubentry...`.

FOREST points: (i) the generic conditional `if`, (ii) handler `.process`,

```
    if index={
      index@hypertarget/.expanded={.\forestoption{index key}},
      for descendants={
        index@hypertarget/.process= {OOw2}
            {!parent.index@hypertarget} {index key} {##1.##2},
      },
    }{},
```

Index.

```
    if index={
      begingroup,
      stage=index,
```

For each (sub)entry, format the `index key` using the instructions in `index key format`.

```
      for tree={
        result/.option=index key,
        process keylist'={index key format}{current},
        index key/.register=result,
      },
```

For each (sub)entry, format the `index form` using the instructions in `index form format`.

```
      for tree={
        result/.option=index form,
        process keylist'={index format}{current},
        index form/.register=result,
      },
```

Create an index entry for all nodes where `index form` is non-empty.

```
      where index form={}{}{
```

All the ancestor nodes with an non-empty `index form` will be appended (in script size, as a hyperlink) to the `index form` of the current node.

```
        if n=0{
          temptoksb={},
        }{
          temptoksc={},
          for ancestors={
            if index form={}{}{
              temptoksb+/.expanded={\forestregister{temptoksc}%
                \noexpand\protect\noexpand\hyperlinknocolor{%
                  \forestoption{index@hypertarget}}{\forestoption{index form}}},
              temptoksc={,\space},
            },
          },
          if temptoksb={}{}{
            +temptoksb={\protect\space\begingroup\protect\scriptsize},
            temptoksb+={\endgroup},
          },
        },
        temptoksa={},
        result'={},
        if n children=0{tempboola}{not tempboola},
        where index form={}{}{
```

Create the hypertarget that the definitions in text and other index entries will point to.

```
          temptoksd/.expanded={\noexpand\protect\noexpand\hypertarget{%
                \forestoption{index@hypertarget}}{}},
```

Add the (inner) current node to the index entry of the (outer) current node.

```
          result+/.expanded={%
            \forestregister{temptoksa}%
            \forestoption{index key}%
            =\forestoption{index form}%
            \forestregister{temptoksd}%
            \forestregister{temptoksb}%
```

```
      },
      temptoksa={>},
      temptoksb={},
    },
```

Do the actual indexing.

```
    result+/.expanded={|indexpagenumber\forestregister{index entry type}},
    TeX and memoize/.expanded={\noexpand\forestindex@LaTeX@index{\forestregister{result}}},
  },
  endgroup
}{},
if print={
  begingroup,
  stage=print,
```

For each (sub)entry, format the `print form` using the instructions in `print form format`.

```
  for tree={
    result/.option=print form,
    process keylist'={print format}{current},
    print form/.register=result,
  },
```

Typeset the entry in the text.

```
  for first leaf'={TeX and memoize/.expanded={\forestoption{print form}}},
  endgroup,
}{},
}
},
```

Create the main entry node and set to be the root.

```
create@main@entry@node/.style={% #1 = subentry
  set root={[]},
  do dynamics, for root'={process delayed=tree},
  root',
  setup@entry@node={#1}
},
```

Create a subentry node and move into it.

```
create@subentry@node/.style={
  append={[]},
  do dynamics, for root'={process delayed=tree},
  n=1,
  setup@entry@node={#1}
},
```

Parse #1 into `key` and `argument`, and assign `name` and `content`.

```
setup@entry@node/.style={
  options={
    split={#1}{=}{key,argument},
    if n=0{
      name'/.option=key,
      content={#1},
    }{
      name'/.process={OOw2} {!parent.name} {key} {##1>##2},
      content/.process={Ow1} {!parent.content} {##1>#1},
    },
  }
},
}
```

Saving and loading the options

```
\forestset{
```

This register holds whatever we need to remember.

```
  declare keylist register=set,
```

Autoforward register

Besides storing the keylist in the register, also immediately execute it..

```
  Autoforward register={set}{##1},
```
Remember things by saving them in a global style.
```
  save@session/.style={@@index/\forestoption{name}/.global style/.register=set},
```
Save thinks to the auxiliary file.
```
  save@foridx/.style={
```
Don't save entries of unknown category.
```
    if strequal/.process={0}{!root.name}{@unknown}{}{
```
Don't save if nothing is set.
```
      if set={}{}{
        TeX and memoize/.expanded={%
          \noexpand\immediate\noexpand\write\noexpand\forestindex@out{%
            \noexpand\string\noexpand\indexloadsettings\noexpand\unexpanded{{\forestoption{name}}{\forest
          }%
        },
      },
    },
```
Save the full form of the key in the auxiliary file. Obviously, do it only for subentries. The full form contains whatever arguments were given to the non-last component.
```
    if key/.process={0}{content} {} {%
      if short={
        TeX and memoize/.expanded={%
          \noexpand\immediate\noexpand\write\noexpand\forestindex@out{%
            \noexpand\string\noexpand\indexdefineshortkey\noexpand\unexpanded{{\forestoption{key}}{\fores
          }%
        }{}%
      }
    }
  }
}
```
Load settings from the auxiliary file into the global style. Warn if anything was configured more than once (e.g. by `\indexdef`ing the same key twice).
```
\def\indexloadsettings#1#2{%
  \pgfkeysifdefined{/forest/@@index/#1/.@cmd}{%
    \forestindex@loadsettings@warning{#1}%
  }{}%
  % #s in #2 are doubled; the following \def removes one layer of doubling
  \def\forest@temp{#2}%
  \forestset{@@index/#1/.global style/.expand once=\forest@temp}%
}
\def\forestindex@loadsettings@warning#1{%
  \PackageWarning{forest-index}{Forest index key "#1" was configured more than once!
    I'm using the last configuration.}%
}
```
Load the full form of a short key from the auxiliary file. Out of kindness for the end user, remember all the full keys corresponding to a short key: this will make a more informative warning below.
```
\def\indexdefineshortkey#1#2{%
  \def\forestindex@temp@short{#1}%
  \def\forestindex@temp@long{#2}%
  \ifx\forestindex@temp@short\forestindex@temp@long
  \else
    \ifcsdef{index@long@#1}{%
      \global\cslet{index@long@#1}\relax
      \csgappto{index@alllong@#1}{,#2}%
    }{%
      \global\csgdef{index@long@#1}{#2}%
      \global\csgdef{index@alllong@#1}{#2}%
    }%
  \fi
}
```
Short key resolution

Nothing to do for a long key.

```
\def\forestindex@resolvekey@long#1{\def\forestindex@fullkey{#1}}
```

Decide whether a key is short or long based on the absence or presence of the level separator `>`.

```
\def\forestindex@resolvekey@shortorlong#1{%
  \pgfutil@in@>{#1}%
  \ifpgfutil@in@
    \expandafter\def\expandafter\forestindex@fullkey
  \else
    \expandafter\forestindex@resolvekey@short
  \fi
  {#1}%
}
```

Before resolving the short key, we need to split the user input into the key and the argument. The latter is then appended to the full key (which can, in principle, contain arguments for other components as well).

```
\def\forestindex@resolvekey@short#1{%
  \forestset{split={#1}{=}{index@resolveshortkey@key,index@resolveshortkey@arg}}%
}
\forestset{
  index@resolveshortkey@key/.code={%
    \ifcsvoid{index@long@#1}{%
      \forestindex@resolveshortkey@warning{#1}%
      \def\forestindex@fullkey{@unknown>#1}%
    }{%
      \letcs\forestindex@fullkey{index@long@#1}%
    }%
  },
  index@resolveshortkey@arg/.code={%
    \appto\forestindex@fullkey{=#1}%
  },
}
\def\forestindex@resolveshortkey@warning#1{%
  \PackageWarning{forest-index}{Cannot resolve short forest index key "#1".
    These are the definitions I found (from the previous run):
    "\csuse{index@alllong@#1}"}%
}
```

Formatting styles

Define default colors for index entry types and provide a style that typesets the entry in text (but not index) in the desired color.

```
\forestset{
  normal color/.initial=blue,
  definition color/.initial=red,
  example color/.initial=darkgreen,
  print in color/.style={if stage={print}{result/.expanded=\noexpand\protect\noexpand\textcolor{%
        \pgfkeysvalueof{/forest/#1 color}}{\unexpanded{##1}}}{}},
  print in color/.default=\forestregister{index entry type},
```

Use this style in `... format` keylists if you want the index entries to be hyperlinks to the definition, and the definition to be a hyperlink to the index.

```
  hyper/.style={
    if stage={index}{}{
      if index entry type={definition}{
        result/.expanded={\noexpand\hypertarget{\forestoption{name}}%
          {\noexpand\hyperlink{\forestoption{index@hypertarget}}{\forestregister{result}}}}
      }{
        result/.expanded=\noexpand\hyperlink{\forestoption{name}}{\forestregister{result}}
      }
    }
  },
}
```

Color page numbers in the index, with or without `hyperref` package.

```
\ifdef\hyperpage{%
  \newcommand\indexpagenumbernormal[1]{{%
```

```
      \hypersetup{linkcolor=\pgfkeysvalueof{/forest/normal color}}\hyperpage{#1}}}
  \newcommand\indexpagenumberdefinition[1]{{%
      \hypersetup{linkcolor=\pgfkeysvalueof{/forest/definition color}}\hyperpage{#1}}}
  \newcommand\indexpagenumberexample[1]{{%
      \hypersetup{linkcolor=\pgfkeysvalueof{/forest/example color}}\hyperpage{#1}}}
}{
  \newcommand\indexpagenumbernormal[1]{%
    \textcolor{\pgfkeysvalueof{/forest/normal color}}{#1}}
  \newcommand\indexpagenumberdefinition[1]{%
    \textcolor{\pgfkeysvalueof{/forest/definition color}}{#1}}
  \newcommand\indexpagenumberexample[1]{%
    \textcolor{\pgfkeysvalueof{/forest/example color}}{#1}}
}
```

Provide dummy `\hyper...` commands if `hyperref` is not loaded.

```
\providecommand\hyperlink[2]{#2}
\providecommand\hypertarget[2]{#2}
\providecommand\hypersetup[1]{}
```

This is used by entry qualifiers: we want them to be hyperlinks, but black.

```
\newcommand\hyperlinknocolor[2]{{\hypersetup{linkcolor=black}\hyperlink{#1}{#2}}}
```

Use style `item` to have the index entry (in text) function as the `\item` of a `lstdoc`'s `syntax` environment.

```
\forestset{
  declare toks register=default,
  default={},
  item/.style={
    result/.process= {_RORw4}
      {} {default} {!parent.print form} {result}
      {\item[,##2,##3]{##4}},
  },
}
```

Utilities

We will need a global version of several `pgfkeys` commands.

```
\pgfkeys{/handlers/.global style/.code=\pgfkeys{\pgfkeyscurrentpath/.global code=\pgfkeysalso{#1}}}
\pgfkeysdef{/handlers/.global code}{\pgfkeysglobaldef{\pgfkeyscurrentpath}{#1}}
\long\def\pgfkeysglobaldef#1#2{%
  \long\def\pgfkeys@temp##1\pgfeov{#2}%
  \pgfkeysgloballet{#1/.@cmd}{\pgfkeys@temp}%
  \pgfkeysglobalsetvalue{#1/.@body}{#2}%
}
\def\pgfkeysgloballet#1#2{%
  \expandafter\global\expandafter\let\csname pgfk@#1\endcsname#2%
}
\long\def\pgfkeysglobalsetvalue#1#2{%
  \pgfkeys@temptoks{#2}\expandafter\xdef\csname pgfk@#1\endcsname{\the\pgfkeys@temptoks}%
}
\forestset{
  % unlike pgfmath function strequal, |if strequal| does not expand the compared args!
  if strequal/.code n args={4}{\ifstrequal{#1}{#2}{\pgfkeysalso{#3}}{\pgfkeysalso{#4}}},
}
```

Begin and end group, FOREST-style:

```
\forestset{
  begingroup/.code={\begingroup},
  endgroup/.code={\endgroup},
}
```

5.2.1 Memoize

Quick and dirty memoization. Single argument macros only. Does not support nesting.

```
\newtoks\forest@memo@key
\newtoks\forest@memo
\newif\ifforest@memoizing@now@
```

```
\newif\ifforest@memoizing@ok@
\newif\ifforest@execandmemoize@
\def\forest@memoize#1{% #1 = \cs
  \cslet{forest@memo@orig@\string#1}#1%
  \def#1##1{%
    \ifforest@memoizing@now@
      \forest@globalsaveandrestoreifcs{forest@execandmemoize@}{%
        \global\forest@execandmemoize@false
        \csname forest@memo@orig@\string#1\endcsname{##1}%
      }%
    \else
      \expandafter\global\expandafter\forest@memo@key\expandafter{\detokenize{forest@memo@#1{##1}}}%
      \ifcsname\the\forest@memo@key\endcsname
        \@escapeifif{\csname\the\forest@memo@key\endcsname}%
      \else
        \@escapeifif{%
          \global\forest@memo{}%
          \global\forest@memoizing@ok@true
          \global\forest@memoizing@now@true
          \global\forest@execandmemoize@true
          \csname forest@memo@orig@\string#1\endcsname{##1}%
          \global\forest@execandmemoize@false
          \global\forest@memoizing@now@false
          \ifforest@memoizing@ok@
            \csxdef{\the\forest@memo@key}{\the\forest@memo}%
            \immediate\write\forest@memo@out{%
              \noexpand\forest@memo@load{\the\forest@memo@key}{\the\forest@memo}%
            }%
          \fi
        }%
      \fi
    \fi
  }%
}
\def\forest@memo@load#1#2{%
```

The following two `\def`s remove one level of hash-doubling from the arguments, introduced by `\write`.

```
  \def\forest@temp@key{#1}%
  \def\forest@temp@value{#2}%
  \csxdef{\detokenize\expandafter{\forest@temp@key}}{\expandonce\forest@temp@value}%
  \immediate\write\forest@memo@out{%
    \noexpand\forest@memo@load{\detokenize\expandafter{\forest@temp@key}}{\detokenize\expandafter{\forest@
  }%
}
\forestset{
  TeX and memoize/.code={\forest@execandmemoize{#1}},
}
\def\forest@execandmemoize#1{%
  \ifforest@execandmemoize@
    \let\forest@memo@next\forest@execandmemoize@
  \else
    \let\forest@memo@next\@gobble
  \fi
  \forest@memo@next{#1}%
  #1%
}
\def\forest@execandmemoize@#1{%
  \gapptotoks\forest@memo{#1}%
}
\def\forest@memo@filename{\jobname.memo}
\newwrite\forest@memo@out
\immediate\openout\forest@memo@out=\forest@memo@filename.tmp
\IfFileExists{\forest@memo@filename}{%
  \input\forest@memo@filename\relax
}{}%
```

```
\AtEndDocument{%
  \immediate\closeout\forest@memo@out
  \forest@file@copy{\forest@memo@filename.tmp}{\forest@memo@filename}%
}
```

Commenting the following line turns off memoization.

```
\forest@memoize\forestindex@index@
```

Initialize

Declare category "@unknown".

```
\index(not print,not index)[%
    set={
      index key=unknown,
      form={\textbf{unknown!!}},
      for first={format={result/.expanded=\noexpand\textbf{\forestregister{result}??}}}
    },
  ]>{@unknown}
```

Load the auxiliary file made in the previous compilation, and open it for writing to save data from this compilation.

```
\def\forestindex@filename{\jobname.foridx}
\IfFileExists{\forestindex@filename}{%
  \input\forestindex@filename\relax
}{}%
\newwrite\forestindex@out
\immediate\openout\forestindex@out=\forestindex@filename.tmp
\AtEndDocument{%
  \immediate\closeout\forestindex@out
  \forest@file@copy{\forestindex@filename.tmp}{\forestindex@filename}%
}
\endinput
```

6 Past, present and future

Roadmap What's planned for future releases?

- filling up the libraries
- faster externalization
- custom-edge aware packing algorithm and a more flexible (successor of) `calign`
- support for specialized `forest` environments, including:
  - selectable input parser,
  - namespaces (different function, different options),
  - better support for different output types.

  In short, everything you need to make FOREST your favourite spreadsheet! ;-)
- code cleanup and extraction of sub-packages possibly useful to other package writers

6.1 Changelog

First of all, the list of all `compat` key values for backward compatibility, and their groupings. Remember, compat values that reside in styles with suffix `-most` are harmless: they will not disrupt the new functionality of the package. But take care when using stuff which only resides in `-all` styles.

```
most/.style={1.0-most},
all/.style={1.0-all},
none/.style={},
1.0-most/.style={
  1.0-triangle,1.0-linear,1.0-nodewalk,1.0-ancestors,
  1.0-fittotree,1.0-for,1.0-forall,
```

```
    2.0.2-most,
  },
  1.0-all/.style={
    1.0-most,
    1.0-forstep,1.0-rotate,1.0-stages,1.0-name,
    2.0.2-all,
  },
  2.0.2-most/.style={
    2.0-most,
  },
  2.0.2-all/.style={
    2.0.2-delayn,2.0.2-wrapnpgfmathargs,
    2.0-all,
  },
  2.0-most/.style={},
  2.0-all/.style={
    2.0-most,
    2.0-delayn,
    2.0-edges,
  },
  2.0-edges/.style={2.0-anchors,2.0-forkededge,2.0-folder},
```

6.1.1 v2.1

v2.1 (2016/12/05)

Backward incompatible changes (with a `compat` key):

in -all `compat=2.0-edges` This compat key groups the three changes listed below: the final two depend on the first, so you will probably want to revert them all or none.

in -all `compat=2.0-anchors` This is really a bugfix. Growth direction based anchors `parent`, `parent first` and `parent last` were not facing to the direction of the parent if the growth direction of the tree changed at the node.

in -all `compat=2.0-forkededge`

in -all `compat=2.0-folder` Update the code of keys `forked edge` (and friends) and `folder` from the `edges` library to reflect the above bugfix.

in -all `compat=2.0-delayn` Fixing yet another bug in `delay n`! The number of cycles was reevaluated at each cycle. Now it is computed immediately, and fixed. Use this key to revert to the old behaviour.

Performance:

- Substantially enhance the argument processor (§3.13), including the ability to use it as a drop-in replacement for `pgfmath`.
- Internally, avoid using `\pgfmathparse` and friends whereever possible.
- Implement a fast set of macros to determine if a `pgfmath` expression is just a ⟨count⟩ or ⟨dimen⟩ expression.
- Optimize `split option` and `split register`.

Minor improvements:

- Allow ⟨relative node name⟩s in `.option`.
- Make aggregate functions (§3.14) nestable and implement their `pgfmath` versions.
- Implement `if` ⟨dimen option⟩`>`, `if` ⟨dimen option⟩`<`, `if` ⟨count option⟩`>`, `if` ⟨count option⟩`<`, `where` ⟨dimen option⟩`>`, `where` ⟨dimen option⟩`>`, `where` ⟨count option⟩`>` and `where` ⟨count option⟩`>`,
- Implement `if current nodewalk empty`.
- Implement nodewalk steps `leaves`, `-level` and `-level'`.
- Implement nodewalk operation `unique`.
- Implement `on invalid` values `error if real` and `last valid`, remove value `step` (no `compat` key, as it was broken and useless).

- Implement '-' anchors (`-parent` etc.).
- Implement `save and restore register`.
- Implement `.nodewalk style`.
- Implement `forestloopcount`.
- Allow multiple occurrences of package option `compat`.

Bugfixes:

- Fix a bug in externalization (`\forest@file@copy` set `\endlinechar` to `-1`, which caused problems for several packages, e.g. `biblatex`).
- Fix a bug in `delay n`: the number of cycles was reevaluated at each cycle.
- Fix a bug in `fixed edge angles`.
- Fix `compat` key values `silent`, `1.0-forstep` and `1.0-stages`.
- Fix invocations of spatial propagators `for nodewalk` and `for Nodewalk` and `Nodewalk`.
- Fix invocations of `for group`, `for next on tier` and `for previous on tier`.
- Fix behaviour of `for next on tier`, `for previous on tier` and `for to tier` on arrival to the invalid node.
- Fix problems with interaction between `folder` and `forked edges`.

6.1.2 v2.0

v2.0.3 (2016/04/03)

Backward incompatible changes (with a `compat` key):

in -all `compat=2.0.2-delayn`

in -all `compat=2.0.2-wrapnpgfmathargs`

This is really a bugfix: keys `delay n` and `.wrap` n `pgfmath args` (for $n \geq 2$) were introducing two layers of hash doubling. Now this confusing behaviour is gone, but as finding the correct number of hashes is always a tough job, `compat` keys are provided.

Improvements:

- Rework `draw tree edge` so that by default, an edge is drawn only it both its node and its node's parent are drawn. And yes, implement `if node drawn`.
- Implement circularity detection in dynamic node operations.
- Implement debug categories and debugging of dynamic node operations.
- Declare some further `tempdim...` registers.
- Make option `id` accessible via `\forestoption`.

Bugfixes:

- Execute `tikz` code for all (including phantom) nodes. (The feature of ignoring phantom nodes was introduced in v2.0.2, but turns out it was a bad idea: for example, having a phantom root with some tikz code is not uncommon.)
- Keys `label` and `pin` now *append* to option `tikz`, as makes sense.
- Fix nodewalk steps `filter` and `branch` so that they can be embedded under nodewalk operations. (Uh, and recategorize them as operations themselves.)
- Execute `before packing node` even when the node has no children.
- `level<={0}{...}` now works as expected.
- Re-setting the node name to the same value doesn't yield an error anymore.
- Don't add the separator when adding the first element to a keylist option or register.
- Copy externalization files in TeX (don't rely on `\write18`).
- Consistently store dimen options and registers with `pt`s of catcode other.
- Properly initialize readonly count options (`n`, `n'`, `n children` and `level`).

- Fix some typos.

v2.0.2 (2016/03/04)

Backward incompatible changes:

- The semantics of the parenthesized optional argument to `forest` environment and `\Forest` macro has changed. The argument was introduced in v2.0.0: if present, it redefined `stages` style for the current environment/macro. This argument is now generalized to allow further (pre-`stages`) customization in future versions of the package. To temporarily redefine `stages`, write `(stages={...})`.

New functionality:

- Key `last dynamic node` and named nodewalk `dynamic nodes`.
- An optional argument to `\useforestlibrary` to pass package options to libraries.
- Handler `.nodewalk style`.
- Keys `draw tree node'`, `draw tree edge'` and `draw tree tikz'`.

Bugfixes:

- Fixed `replace by` when applied to the root node.
- Registers are now initialized to an empty string, 0pt, or 0.
- Packing doesn't destroy the current pgfpath anymore.
- `\forestStandardNode` now uses `name'`.
- `draw tree edge` now respects `phantom`.

v2.0.1 (2016/02/20)

New functionality:

- `current and siblings`, `current and siblings reversed`
- Add `*` argument to `\useforestlibrary`.

Bugfixes:

- Correctly mangle option/register names to pgfmath names (§3.18).
- Refer to parent (not node) anchor in `calign=edge midpoint`.
- Accept key `history` in `Nodewalk` config.

v2.0.0 (2016/01/30) [34]

Backwards incompatible changes (*without* a `compat` key — sorry!):

- The unintended and undocumented way to specify defaults using `\forestset{.style={...}}` (see question Making a certain tree style the default for forest at TEX SE) does not work anymore. (Actually, it has never truly worked, and that's why it has not `compat` key.) Use `default preamble`.
- Renamed augmented assignment operator ⟨option⟩`-` for prepending to ⟨toks⟩ and ⟨keylist⟩ options `+`⟨option⟩. A new ⟨option⟩`-` is defined for keylist options and means "delete key from keylist."
- Short nodewalk steps are not simply styles anymore: use `define short step` to define them.

Backwards incompatible changes with a `compat` key:

in -all `compat=1.0-stages`

Processing of `given options`, which is now exposed, and the new keylists `default preamble` and `preamble` is now included at the start of the default `stages` style. When changing `stages`, the instruction to process these keylists must now be given explicitely.

[34] The year of the release date in the package was wrong ... 2015.

in -all compat=1.0-forstep

In v1.0, a spatial propagator `for` ⟨step⟩ could never fail. This turned out to be difficult to debug. In this version, when a propagator steps "out of the tree", an error is raised by default. Check out `on invalid` to learn how to simulate the old behaviour without using this compatibility key.

in -all compat=1.0-rotate

This version of the package introduces option `rotate` and `autoforward`s it to `node options`. This is needed to handle the new FOREST anchors (§3.17). However, in some rare cases (like the tree on the title page of this manual) it can lead to a discrepancy between the versions, as the time when the value given to `rotate` is processed is different. `1.0-rotate` removes option `rotate`.

in -all compat=1.0-name

Documentation of v1.0 requested that node names be unique, but this was not enforced by the package, sometimes leading to errors. v2.0 enforces node name uniqueness. If this causes problems, use this compatibility key. In most cases using `name'` instead of `name` should fix the problem without using compatibility mode.

These keys have been renamed:

old	new	`compat` key (all but the last are in `-most`)
`node walk`	`for nodewalk`[35]	`1.0-nodewalk`
`for`	`for group`	`1.0-for`
`for all next`	`for following siblings`	`1.0-forall`
`for all previous`	`for preceding siblings`	`1.0-forall`
`for ancestors'`	`for current and ancestors`	`1.0-ancestors`
(`for`) `linear next`	(`for`) `next node`	`1.0-linear`
(`for`) `linear previous`	(`for`) `previous node`	`1.0-linear`
`triangle`	`roof` (library `linguistics`)	`1.0-triangle`
`/tikz/fit to tree`	`/tikz/fit to=tree`[36]	`1.0-fittotree`
`begin forest`, `end forest`	none (use `stages`)	`1.0-stages`
`end forest`, `end forest`	none (use `stages`)	`1.0-stages`

Good news:

- Added temporal propagators `before packing node` and `after packing node`.
- *Much* improved nodewalks, see §3.8 and §3.5.1.
- Implemented looping mechanisms and more conditionals, see §3.9.
- Implemented library support and started filling up the libraries:
  - `linguistics`: `sn edges`, `nice empty nodes`, `draw brackets`, `c-commanded` and `c-commanders`
  - `edges`: `forked edges` and `folder`
- Implemented aggregate functions, see §3.14.
- Added key `default preamble`.
- Implemented anchors `parent`, `children`, `first`, `last`, etc.
- Added key `split` and friends.
- Implemented sorting of children, see §3.11.
- Introduced registers, see §3.6.
- Implemented handlers `.option`, `.register` and `.process args`.
- Implemented several friends to `process keylist`, introduced `processing order`s and `draw tree method`.
- Added the optional argument `(`⟨stages⟩`)` to the `forest` environment and `\Forest` macro.
- Implemented `autoforward`ing.
- Implemented flexible handling of unknown keys using `unknown to`.
- Implemented `pgfmath` functions `min_l`, `max_l`, `min_s`, `max_s`.
- Implemented augmented assignment operator ⟨keylist option⟩`-` for removing keys from keylists.

[35] Nodewalks are much improved in v2.0, so some syntax and keys are different than in v1.0!
[36] The v1.0 key `/tikz/fit to tree` also set `inner sep=0`; the v2.0 key `/tikz/fit to` does not do that.

- Implemented a generalized `/tikz/fit to` key.
- Implemented a very slow FOREST-based indexing system (used to index this documentation) and included it in the gallery (§5.2).
- Added some minor keys: `edge path'`, `node format'`, `create'` and `plain content`.
- Added some developer keys: `copy command key`, `typeout`.

Bugfixes:

- In computation of numeric tree-structure info, when called for a non-root node.
- TikZ's externalization internals (signature of `\ tikzexternal@externalizefig@systemcall@uptodatech` have changed: keep up to date, though only formally.
- `delay` was not behaving additively.
- `name`, `alias` and `baseline` didn't work properly when setting them for a non-current node.
- Augmented assignments for count options were leaking '.0pt'.
- `create` didn't work properly in some cases.
- `triangle` (now `roof` in `linguistics`) didn't use `cycle` in the edge path

6.1.3 v1.0

v1.0.10 (2015/07/22)

- Bugfix: a left-over debugging `\typeout` command was interfering with a `forest` within `tabular`, see this question on TeX.SE.
- A somewhat changed versioning scheme ...

v1.09 (2015/07/15)

- Bugfix: child alignment was not done in nodes with a single child, see this question on TeX.SE.

v1.08 (2015/07/10)

- Fix externalization (compatibility with new `tikz` features).

v1.07 (2015/05/29)

- Require package `elocalloc` for local boxes, which were previously defined by package `etex`.

v1.06 (2015/05/04)

- Load `etex` package: since v2.1a, `etoolbox` doesn't do it anymore.

v1.05 (2014/03/07)

- Fix the node boundary code for rounded rectangle. (Patch contributed by Paul Gaborit.)

v1.04 (2013/10/17)

- Fixed an externalization bug.

v1.03 (2013/01/28)

- Bugfix: options of dynamically created nodes didn't get processed.
- Bugfix: the bracket parser was losing spaces before opening braces.
- Bugfix: a family of utility macros dealing with affixing token lists was not expanding content correctly.
- Added style `math content`.
- Replace key `tikz preamble` with more general `begin draw` and `end draw`.
- Add keys `begin forest` and `end forest`.

v1.02 (2013/01/20)

- Reworked style `stages`: it's easier to modify the processing flow now.
- Individual stages must now be explicitely called in the context of some (usually root) node.
- Added `delay n` and `if have delayed`.
- Added (experimental) `pack'`.
- Added reference to the style repository.

v1.01 (2012/11/14)

- Compatibility with the `standalone` package: temporarily disable the effect of `standalone`'s package option `tikz` while typesetting nodes.
- Require at least the [2010/08/21] (v2.0) release of package `etoolbox`.
- Require version [2010/10/13] (v2.10, rcs-revision 1.76) of PGF/Ti*k*Z. Future compatibility: adjust to the change of the "not yet positioned" node name (2.10 `@` → 2.10-csv `PGFINTERNAL`).
- Add this changelog.

v1.0 (2012/10/31) First public version

6.2 Known bugs

If you find a bug (there are bound to be some . . .), please contact me at saso.zivanovic@guest.arnes.si.

System requirements

This package requires LaTeX and eTeX. If you use something else: sorry.

The requirement for LaTeX might be dropped in the future, when I get some time and energy for a code-cleanup (read: to remedy the consequences of my bad programming practices and general disorganization).

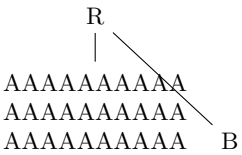
The requirement for eTeX will probably stay. If nothing else, Forest is heavy on boxes: every node requires its own . . . and consequently, I have freely used eTeX constructs in the code . . .

pgf internals

Forest relies on some details of pgf implementation, like the name of the "not yet positioned" nodes. Thus, a new bug might appear with the development of pgf. If you notice one, please let me know.

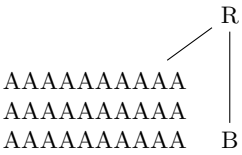
Edges cutting through sibling nodes

In the following example, the R–B edge crosses the AAA node, although `ignore edge` is set to the default `false`.



(106)
```
\begin{forest}
  calign=first
  [R[AAAAAAAAAA\\AAAAAAAAAA\\AAAAAAAAAA,align=center,base=bottom][B]]
\end{forest}
```

This happens because s-distances between the adjacent children are computed before child alignment (which is obviously the correct order in the general case), but child alignment non-linearly influences the edges. Observe that the with a different value of `calign`, the problem does not arise.

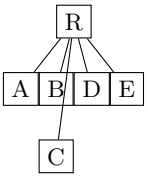


(107)
```
\begin{forest}
  calign=last
  [R[AAAAAAAAAA\\AAAAAAAAAA\\AAAAAAAAAA,align=center,base=bottom][B]]
\end{forest}
```

While it would be possible to fix the situation after child alignment (at least for some child alignment methods), I have decided against that, since the distances between siblings would soon become too large. If the AAA node in the example above was large enough, B could easily be pushed off the paper. The bottomline is, please use manual adjustment to fix such situations.

Orphans

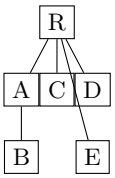
If the `l` coordinates of adjacent children are too different (as a result of manual adjustment or tier alignment), the packing algorithm might have nothing so say about the desired distance between them: in this sense, node C below is an "orphan."



(108)
```
\begin{forest}
  for tree={s sep=0,draw},
  [R[A][B][C,l*=2][D][E]]
\end{forest}
```

To prevent orphans from ending up just anywhere, I have decided to vertically align them with their preceding sibling — although I'm not certain that's really the best solution. In other words, you can rely that the sequence of s-coordinates of siblings is non-decreasing.

The decision also influences a similar situation illustrated below. The packing algorithm puts node E immediately next to B (i.e. under C): however, the monotonicity-retaining mechanism then vertically aligns it with its preceding sibling, D.




```
\begin{forest}
  for tree={s sep=0,draw},
  [R[A[B,tier=bottom]][C][D][E,tier=bottom]]
\end{forest}
```

(109)

Obviously, both examples also create the situation of an edge crossing some sibling node(s). Again, I don't think anything sensible can be done about this, in general.

6.3 Acknowledgements

This package has turned out to be much more successful and widespread than I could have ever imagined and I want to thank all the users for the trust. Many of you have also contributed to the package in some way: by providing comments and ideas, sending patches, reporting bugs and so on. To you, I'm doubly grateful! I will not even try to list you all here, as the list is getting too long for me to maintain, but I do want to mention one person, a member of the friendly community at the excellent and indispensable TeX – LaTeX Stack Exchange and the author of the very first FOREST-based package, Prooftrees: without `cfr`'s uncountable questions, answers, bug reports and ideas, FOREST would be a much poorer package indeed.

References

[1] Donald E. Knuth. *The TeXbook*. Addison-Wesley, 1996.

[2] Till Tantau. *TikZ & PGF, Manual for Version 2.10*, 2007.

[3] Till Tantau. *TikZ & PGF, Manual for Version 3.0.0*, 2013.

Color legend: definition, example, other. If an entry belongs to a library, the library name is given in parenthesis. All page numbers are hyperlinks, and definitions in text are hyperlinked to this index.

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y