

# FOREST: a PGF/TikZ-based package for drawing linguistic trees

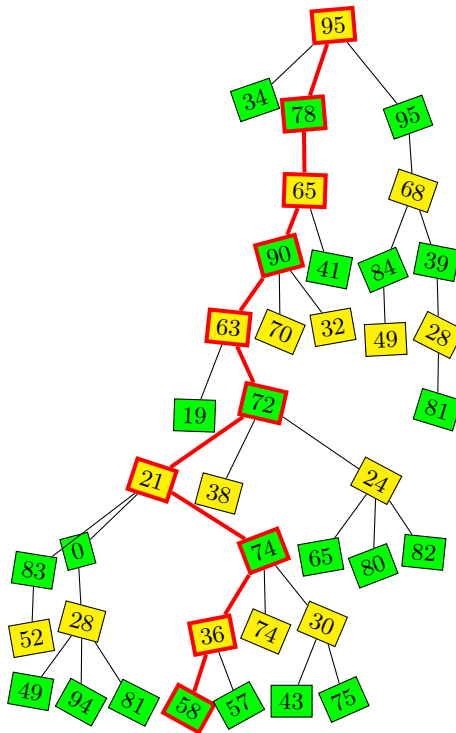
v1.0

Sašo Živanović\*

October 31, 2012

## Abstract

FOREST is a PGF/TikZ-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) the possibility to decorate the tree using the full power of PGF/TikZ; (v) 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}]}}{}}},
  for deepest/.style={before drawing tree={
    alias=deepest,
    where={y(<y("deepest"))}{alias=deepest}{},
    for name={deepest}{#1}},
    colorone/.style={fill=yellow,for children=colortwo}, colortwo/.style={fill=green,for children=colorone},
    important/.style={draw=red,line width=1.5pt,edge={red,line width=1.5pt,draw}},
    before typesetting nodes={colorone, for tree={draw,s sep=2pt,rotate={int(30*rand)},l+={5*rand}}},
    for deepest={for ancestors'={important,typeset node}}
    [,random tree={9}{3}{100}]
  }
\end{forest}
```

---

\*e-mail: [saso.zivanovic@guest.arnes.si](mailto:saso.zivanovic@guest.arnes.si); web: <http://spj.ff.uni-lj.si/zivanovic/>

# Contents

<b>I</b>	<b>User's Guide</b>	<b>4</b>
<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Tutorial</b>	<b>4</b>
2.1	Basic usage . . . . .	4
2.2	Options . . . . .	6
2.3	Decorating the tree . . . . .	8
2.4	Node positioning . . . . .	10
2.4.1	The defaults, or the hairy details of vertical alignment . . . . .	16
2.5	Advanced option setting . . . . .	18
2.6	Externalization . . . . .	20
2.7	Expansion control in the bracket parser . . . . .	21
<b>3</b>	<b>Reference</b>	<b>22</b>
3.1	Environments . . . . .	22
3.2	The bracket representation . . . . .	22
3.3	Options and keys . . . . .	23
3.3.1	Node appearance . . . . .	25
3.3.2	Node position . . . . .	27
3.3.3	Edges . . . . .	32
3.3.4	Readonly . . . . .	34
3.3.5	Miscellaneous . . . . .	34
3.3.6	Propagators . . . . .	36
3.3.7	Stages . . . . .	38
3.3.8	Dynamic tree . . . . .	39
3.4	Handlers . . . . .	41
3.5	Relative node names . . . . .	42
3.5.1	Node walk . . . . .	42
3.5.2	The <code>forest</code> coordinate system . . . . .	44
3.6	New <code>pgfmath</code> functions . . . . .	44
3.7	Standard node . . . . .	45
3.8	Externalization . . . . .	46
3.9	Package options . . . . .	47
<b>4</b>	<b>Gallery</b>	<b>47</b>
4.1	Styles . . . . .	47
4.2	Examples . . . . .	51
<b>5</b>	<b>Known bugs</b>	<b>53</b>
<b>II</b>	<b>Implementation</b>	<b>54</b>
<b>6</b>	<b>Patches</b>	<b>54</b>
<b>7</b>	<b>Utilities</b>	<b>58</b>
7.1	Sorting . . . . .	60
<b>8</b>	<b>The bracket representation parser</b>	<b>64</b>
8.1	The user interface macros . . . . .	64
8.2	Parsing . . . . .	65
8.3	The tree-structure interface . . . . .	69

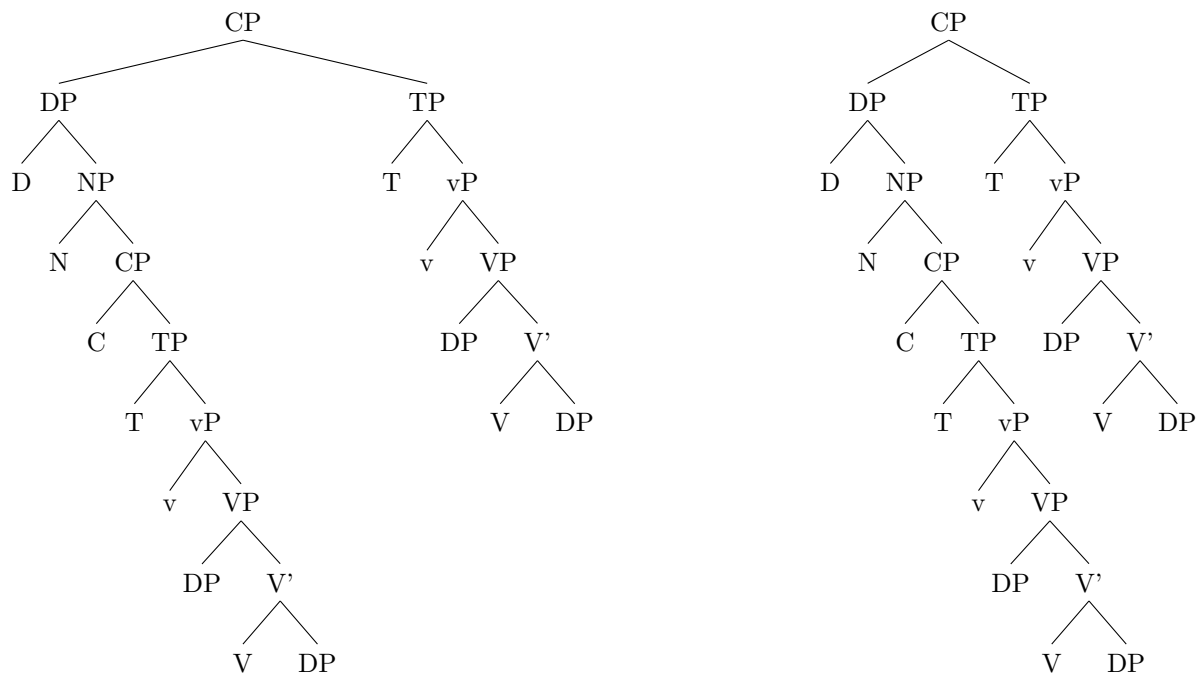
<b>9</b>	<b>Nodes</b>	<b>70</b>
9.1	Option setting and retrieval . . . . .	70
9.2	Tree structure . . . . .	72
9.3	Node walk . . . . .	77
9.4	Node options . . . . .	81
9.4.1	Option-declaration mechanism . . . . .	81
9.4.2	Declaring options . . . . .	87
9.4.3	Option propagation . . . . .	90
9.4.4	<code>pgfmath</code> extensions . . . . .	91
9.5	Dynamic tree . . . . .	92
<b>10</b>	<b>Stages</b>	<b>93</b>
10.1	Typesetting nodes . . . . .	95
10.2	Packing . . . . .	97
10.2.1	Tiers . . . . .	106
10.2.2	Node boundary . . . . .	110
10.3	Compute absolute positions . . . . .	115
10.4	Drawing the tree . . . . .	115
<b>11</b>	<b>Geometry</b>	<b>118</b>
11.1	Projections . . . . .	118
11.2	Break path . . . . .	121
11.3	Get tight edge of path . . . . .	123
11.4	Get rectangle/band edge . . . . .	129
11.5	Distance between paths . . . . .	130
11.6	Utilities . . . . .	133
<b>12</b>	<b>The outer UI</b>	<b>134</b>
12.1	Package options . . . . .	134
12.2	Externalization . . . . .	134
12.3	The <code>forest</code> environment . . . . .	136
12.4	Standard node . . . . .	138
12.5	<code>ls</code> coordinate system . . . . .	140
	<b>References</b>	<b>141</b>

# Part I

## User's Guide

### 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/TikZ, 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/TikZ that FOREST is now, I believe, the most flexible tree typesetting package for L<sup>A</sup>T<sub>E</sub>X you can get.

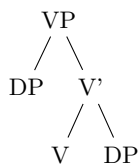
After all the advertising, a disclaimer. Although the present version is definitely usable (and has been already used), the package and its documentation are still under development: comments, criticism, suggestions and code are all very welcome!

### 2 Tutorial

This short tutorial progresses from basic through useful to obscure ...

#### 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.



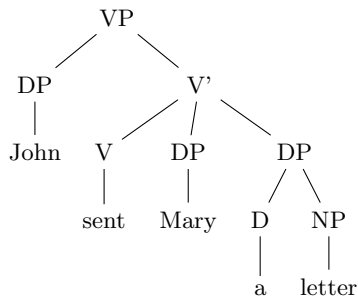
```

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

```

(2)

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.



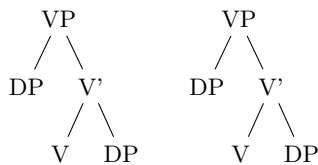
```

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

```

(3)

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



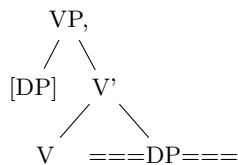
```

\begin{forest}
[VP[DP][V'[V][DP]]]
\end{forest}
\quad
\begin{forest}[VP
[DP][V'[V][DP]]
]\end{forest}

```

(4)

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: comma , and equality sign =.



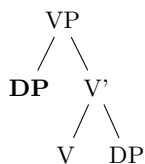
```

\begin{forest}
[V{P,}]
[{{[DP]}}]
[V'
[V]
[{{===DP===}}]]
]\end{forest}

```

(5)

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



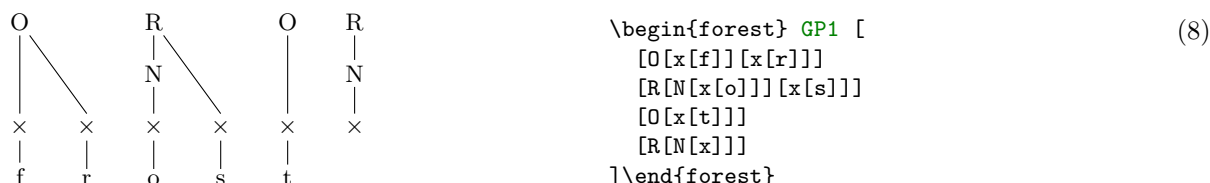
```

\begin{forest}
[VP
[{{\textbf{DP}}}]
[V'
[V]
[DP]]]
]\end{forest}

```

(6)

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 §4 to easily typeset (Government Phonology) phonological representations. The style is applied simply by writing its name before the first (opening) bracket of the tree.

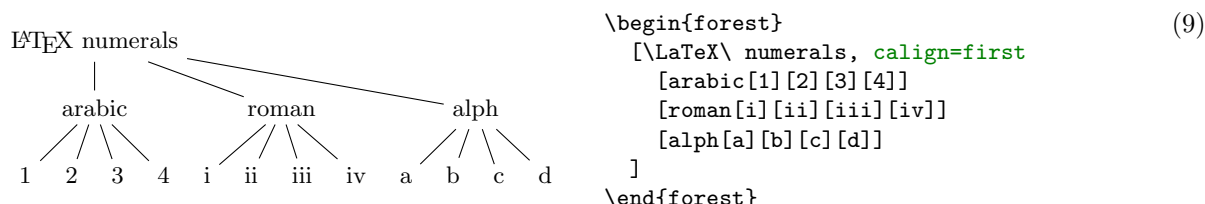


Of course, someone needs to develop the style — you, me, your local T<sub>E</sub>Xnician ... Fortunately, designing styles is not very difficult once you know your FOREST options. If you write one, please contribute!

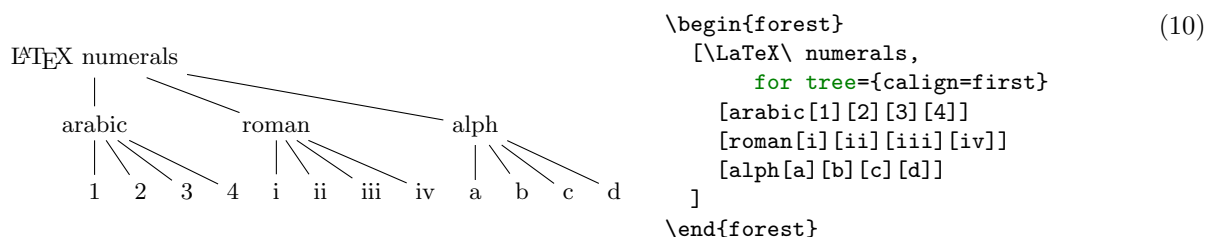
## 2.2 Options

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 *child-alignment*) 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>`.)



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 options, called *propagators*. (We'll call the options that actually change some property of the node *node options*.) What we need above is the `for tree` propagator. Observe:

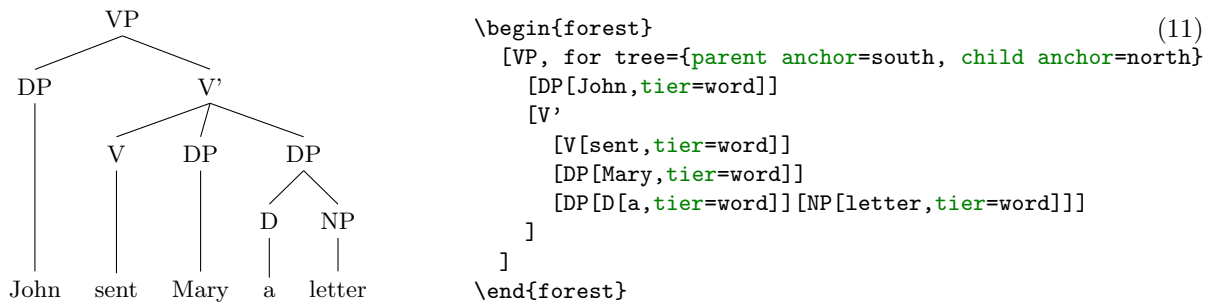


The value of propagator `for tree` is the option string that we want to process. This option string is propagated to all the nodes in the subtree<sup>1</sup> rooted in the current node (i.e. the node where `for tree` was given), including the node itself. (Propagator `for descendants` is just like `for tree`, only that it excludes the node itself. There are many other `for ...` propagators; for the complete list, see sections 3.3.6 and 3.5.1.)

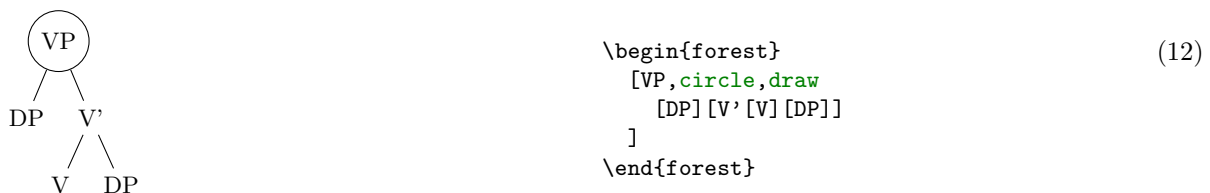
Some other useful options are `parent anchor`, `child anchor` and `tier`. The `parent anchor` and `child anchor` options tell where the parent's and child's endpoint of the edge between them should be, respectively: usually, the value is either empty (meaning a smartly determined border point [see ? , §16.11]; this is the default) or a compass direction [see ? , §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 `x` in example (8) 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!)

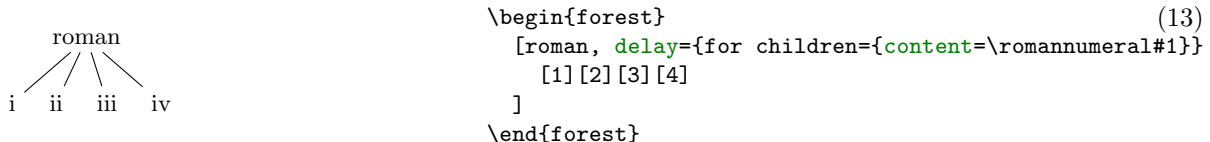
<sup>1</sup>It might be more precise to call this option `for subtree ...` but this name at least saves some typing.



Before discussing the variety of FOREST's options, it is worth mentioning that FOREST's node accepts all options [?, see §16] that TikZ's node does — mostly, it just passes them on to TikZ. For example, you can easily encircle a node like this:<sup>2</sup>



Let's have another look at example (8). You will note that the skeletal positions were input by typing `xs`, while the result looks like this:  $\times$  (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 (10) a bit fancier: we'll write the input in the arabic numbers and have L<sup>A</sup>T<sub>E</sub>X convert it to the other formats. We'll start with the easiest case of roman numerals: to get them, we can use the (plain) T<sub>E</sub>X 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.<sup>3</sup>



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. The processing proceeds through the tree in a depth-first, parent-first fashion (first the parent is processed, and then its children, recursively). The option string of a node is processed linearly, in the order they were given. (Option `content` is specified implicitly 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. Option `delay` is true to its name. It delays the processing of its option string 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 `delays` as deep as you need.

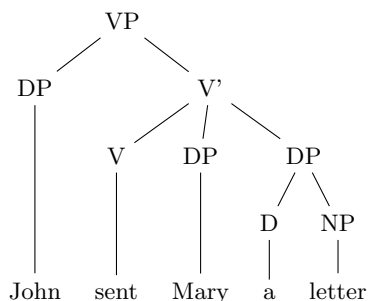
Unlike `for ...` options we have met before, option `delay` is not a spatial, but a temporal propagator. Several other temporal propagators options exist, see §3.3.7.

We are now ready to learn about simple conditionals. Every node option has the corresponding `if ...` and `where ...` keys. `if <option>=<value><true options><false options>` checks whether the value of

<sup>2</sup>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 [?, §16].

<sup>3</sup>This mechanism is called *wrapping*. `content` is the only option where wrapping works implicitly (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 §3.4.

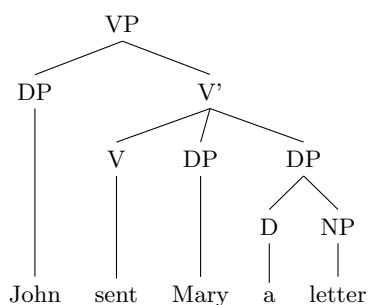
$\langle option \rangle$  equals  $\langle value \rangle$ . If so,  $\langle true options \rangle$  are processed, otherwise  $\langle false options \rangle$ . The `where ...` keys are the same, but do this for the every node in the subtree; informally speaking, `where` = `for tree` + `if`. To see this in action, consider the rewrite of the `tier` example (11) 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`.<sup>4</sup>



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

(14)

Finally, let's talk about styles. Styles are simply collections of options. (They are not actually defined in the `FOREST` package, but rather inherited from `pgfkeys`.) If you often want to have non-default parent/child anchors, say south/north as in example (11), you would save some typing by defining a style. Styles are defined using PGF's handler `.style`. (In the example below, style `ns edges` is first defined and then used.)



```
\begin{forest}
  \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}
```

(15)

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}}}
}
```

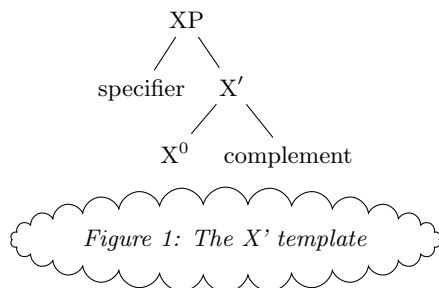
You might have noticed that the last two examples contain options (actually, keys) even before the first opening bracket, contradicting what was said at the beginning of this section. This is mainly just syntactic sugar (it can separate the design and the content): such preamble keys behave as if they 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.

## 2.3 Decorating the tree

The tree can be decorated (think movement arrows) with arbitrary TikZ code.

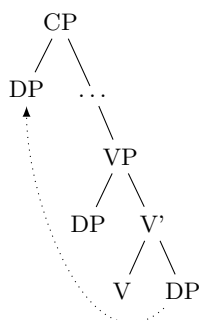
<sup>4</sup>We could omit the braces around 0 because it is a single character. If we were hunting for nodes with 42 children, we'd have to write `where n children={42}....`





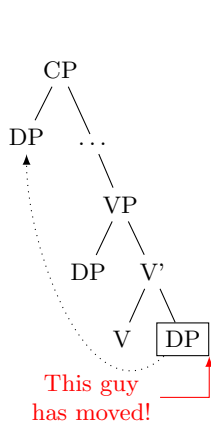
```
\begin{forest}
[XP
[specifier
[X$'$
[X$^0$]
[complement]
]
]
\node at (current bounding box.south)
[below=1ex,draw,cloud,aspect=6,cloud puffs=30]
{\emph{Figure 1: The X' template}};
\end{forest}
```

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 TikZ name using the `name` option, and then use this name in TikZ code as any other (TikZ) node name.



```
\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 TikZ code after the tree specification, i.e. after the closing bracket of the root node. In fact, you can put TikZ 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 ? , §16.11]: below, `(.south east)` and `()`.



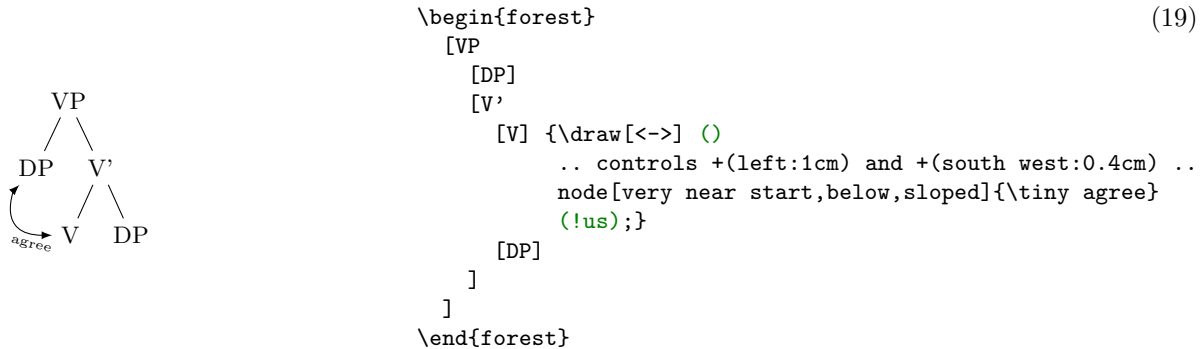
```
\begin{forest}
[CP
[DP,name=spec CP]
[\dots
[,phantom]
[VP
[DP]
[V'
[V]
[DP,draw] {
\draw[->,dotted] () to[out=south west,in=south] (spec CP);
\draw[<-,red] (.south east)--++(0em,-4ex)--++(-2em,0pt)
node[anchor=east,align=center]{This guy\\has moved!};
}
]]]]
\end{forest}
```

Important: the TikZ 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.)

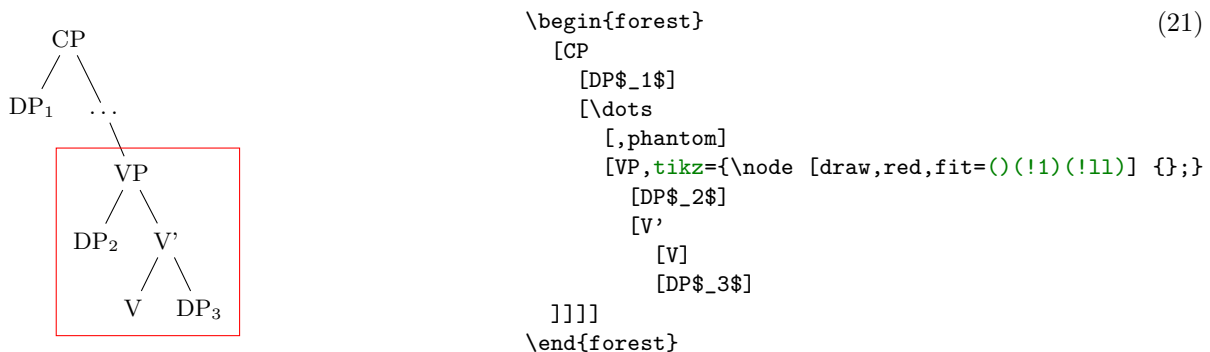
Finally, the most powerful tool in the node reference toolbox: *relative nodes*. It is possible to refer to other nodes which stand in some (most often geometrical) relation to the current node. To do this, follow the node's name with a `!` and a *node walk* specification.

A node walk is a concise<sup>5</sup> 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 3.5.1.

To see the node walk in action, consider the following examples. In the first example, the agree arrow connects the V node, specified simply as `()`, since the TikZ code follows `[V]`, and the DP node, which is described as “a sister of V’s parent”: `!us = up + sibling`.



The second example uses TikZ’s fitting library to compute the smallest rectangle containing node VP, its first child (DP<sub>2</sub>) and its last grandchild (DP<sub>3</sub>). The example also illustrates that the TikZ code can be specified via the “normal” option syntax, i.e. as a value to option `tikz`.<sup>6</sup>

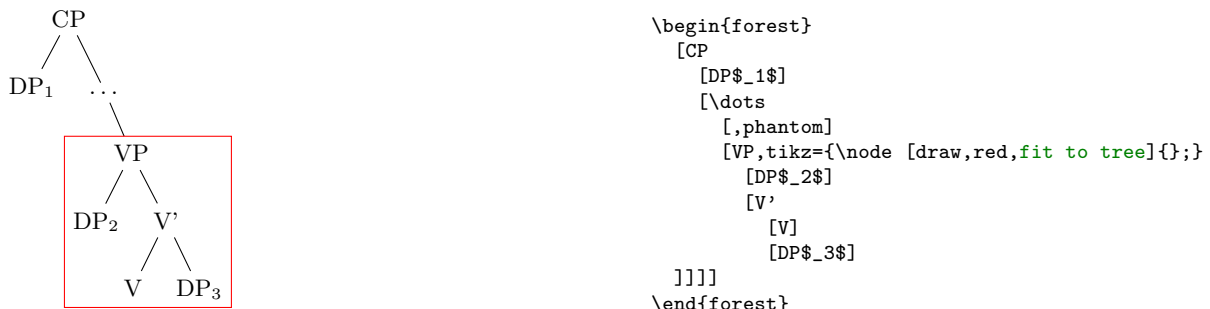


## 2.4 Node positioning

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>`.<sup>7</sup> The system thus computes and stores the positions of children using a coordinate

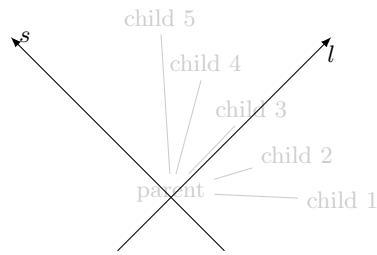
<sup>5</sup>Actually, FOREST distinguishes two kinds of steps in node walks: long and short steps. This section introduces only short steps. See §3.5.1.

<sup>6</sup>Actually, there’s a simpler way to do this: use `fit to tree`!



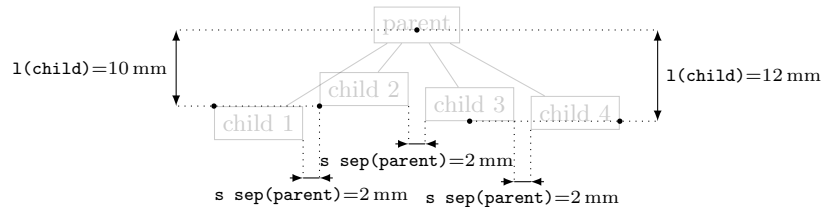
<sup>7</sup>The direction can be specified either in degrees (following the standard mathematical convention that 0 degrees is to the right, and that degrees increase counter-clockwise) or by the compass directions: `east`, `north`, `east`, `north`, etc.

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 *level*, s for *sibling*. The example shows the ls-coordinate system for a node with `grow=45`.



```
\begin{forest} background tree
  [parent, grow=45
    [child 1][child 2][child 3][child 4][child 5]
  ]
  \draw[,>](-135:1cm)--(45:3cm) node[below]{$l$};
  \draw[,>](-45:1cm)--(135:3cm) 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. 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 [?, §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.<sup>8</sup>



<sup>8</sup>Here are the definitions 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][#####]{\measurexorydistance{#2}{#3}{#4}{#5}{x}{-}{(5pt,0)}{#1}}
\newcommand\measureydistance[5][#####]{\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}
```

```

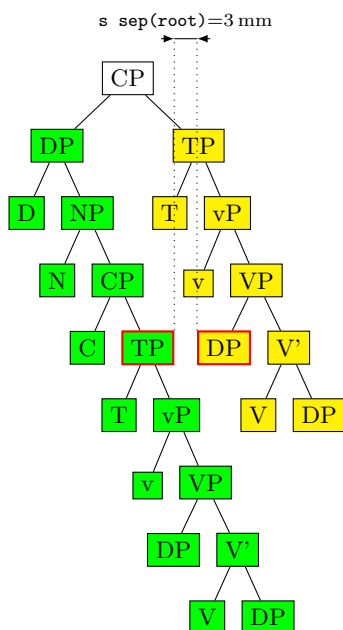
\begin{forest} background tree,
  for tree={draw,tikz={\fill[](.anchor)circle[radius=1pt];}}
  [parent
    [child 1, l=10mm, anchor=north west]
    [child 2, l=10mm, anchor=south west]
    [child 3, l=12mm, anchor=south]
    [child 4, l=12mm, anchor=base east]
  ]
  \measureydistance[\texttt{l(child)}=#1]{(!2.anchor)}{(!.anchor)}{(!1.anchor)+(-5mm,0)}{left}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

```

(24)

Positioning the children in the s-dimension is the job and *raison d'être* 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!

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`.



```

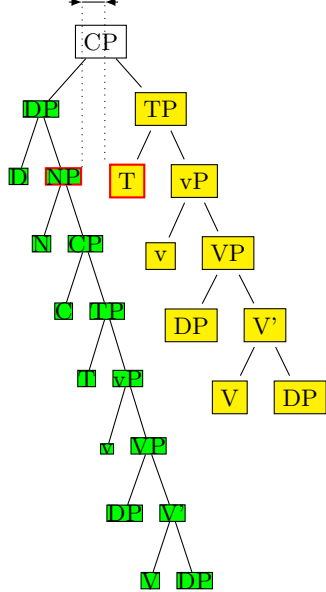
\begin{forest}
  important/.style={name=#1,draw={red,thick}}
  [CP, s sep=3mm, for tree=draw
    [DP, for tree={fill=green}
      [D] [NP[N] [CP[C] [TP,important=left
        [T] [vP[v] [VP[DP] [V' [V] [DP]]]]]]]]
      [TP,for tree={fill=yellow}
        [T] [vP[v] [VP[DP,important=right] [V' [V] [DP]]]]]]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

```

(25)

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 TikZ's `inner sep` and `outer sep` [?, §16.2.2], as shown below:

s sep(root)=3 mm



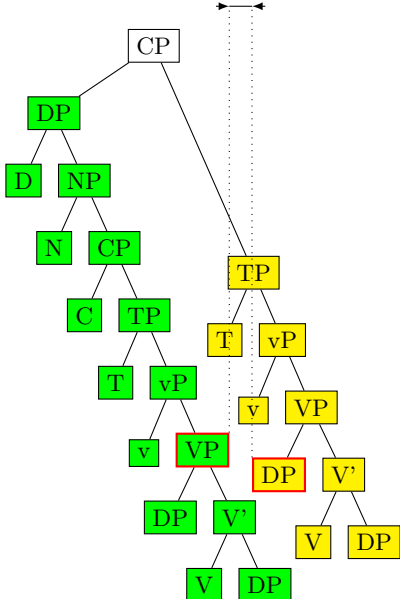
```
\begin{forest}
important/.style={name=#1,draw={red,thick}}
[CP, s sep=3mm, for tree=draw
  [DP, for tree={fill=green,inner sep=0}
    [D] [NP,important=left[N] [CP[C] [TP[T] [vP[v]
      [VP[DP] [V' [V] [DP]]]]]]]]
    [TP,for tree={fill=yellow,outer sep=2pt}
      [T,important=right] [vP[v] [VP[DP]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

(26)

(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 3cm: and the yellow submarine sinks diagonally ... Now, the closest nodes are the higher yellow DP and the green VP.

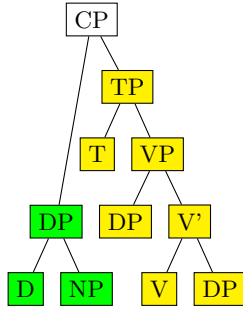
s sep(root)=3 mm



```
\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}
```

(27)

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.)

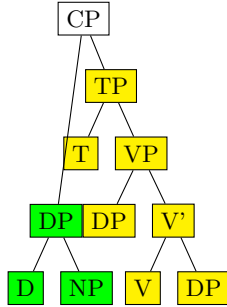


```
\begin{forest}
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D] [NP]]
    [TP,for tree={fill=yellow}
      [T] [VP [DP] [V' [V] [DP]]]]
  ]
\end{forest}
```

(28)

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

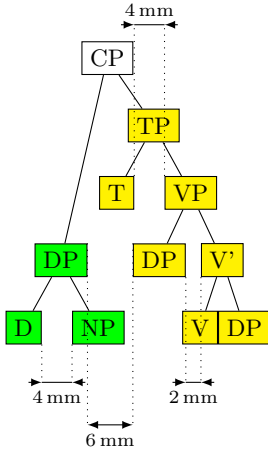
Let's now play with option `s sep`.



```
\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}
```

(29)

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 2mm at level 2, to 4mm at level 1 and to 6mm at level 0.



```
\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}
```

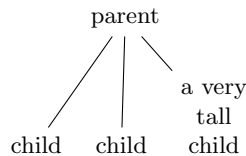
(30)

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

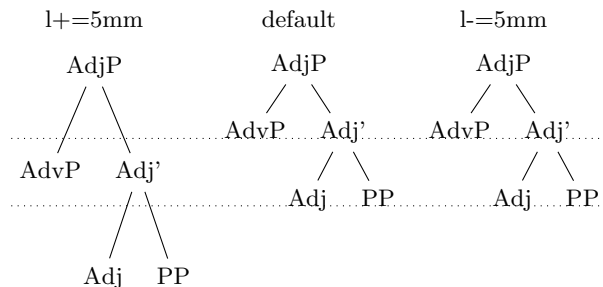
As you have probably noticed, this example also demonstrated that we can compute the value of an option using an (arbitrarily complex) formula. This is thanks to PGF's module `pgfmath`. FOREST provides an interface to `pgfmath` by defining `pgfmath` functions for every node option, and some other

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 §3.6.

The final separation parameter is `1 sep`. It determines the minimal separation of a node from its descendants. If the value of `1` is too small, then *all* the children (and thus their subtrees) are pushed away from the parent (by increasing their `1s`), so that the distance between the node's and each child's subtree boundary is at least `1 sep`. The initial `1` 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 §3.3.1). Bottom vertical alignment is achieved by setting `base=bottom`; the default, unlike in `TikZ`, is `base=top`).


$$\begin{array}{l} \begin{array}{l} \text{\texttt{\textbackslashbegin{forest}}} \\ \text{\texttt{[parent}} \\ \text{\texttt{[child}} \\ \text{\texttt{[child}} \\ \text{\texttt{[a very\\tall\\child, align=center, base=bottom}} \\ \text{\texttt{]}} \\ \text{\texttt{\textbackslashend{forest}}} \end{array} \end{array} \quad (31)$$

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.



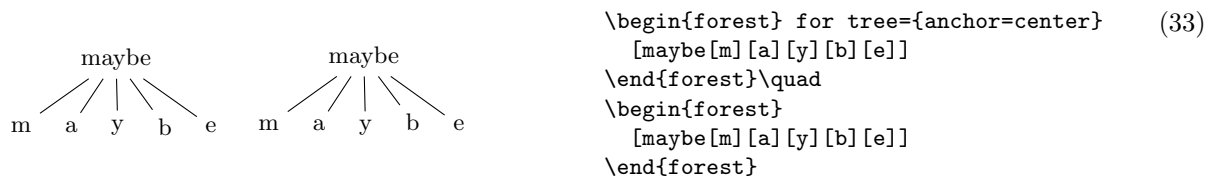
```
\begin{forest}
[,phantom,for children={l sep=lex,fit=band,
  for=1{edge'=',l=0},baseline}
  [{l+=5mm},for descendants/.pgfmath=content
    [AdjP[AdvP][Adj'[Adj][PP]]]]
[default
  [AdjP[AdvP][Adj'[Adj][PP]]]]
[{l-=5mm},for descendants/.pgfmath=content
  [AdjP[AdvP][Adj'[Adj][PP]]]]
]
\path (current 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{4cm}
\raisebox{0pt}[height]{0pt}{\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)]]]]]]]]]]]]
]
\end{forest}}
```

[illegible]

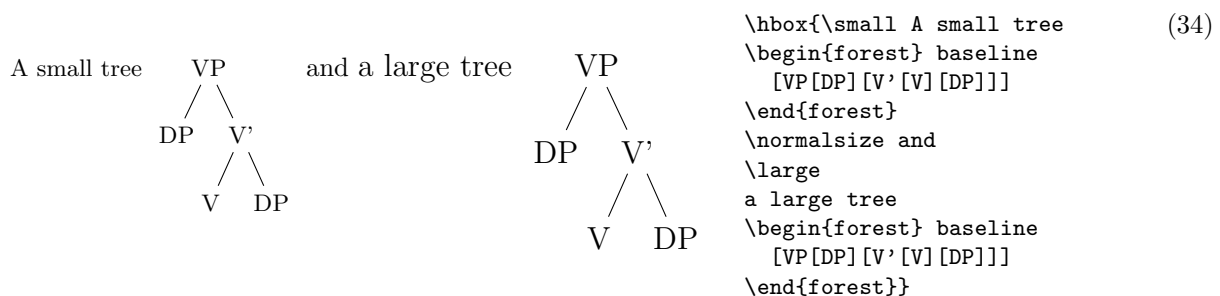
### 2.4.1 The defaults, or the hairy details of vertical alignment

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: `TikZ`'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 [?, §16.5.1]. The difference is particularly clear for a “phonological” representation:



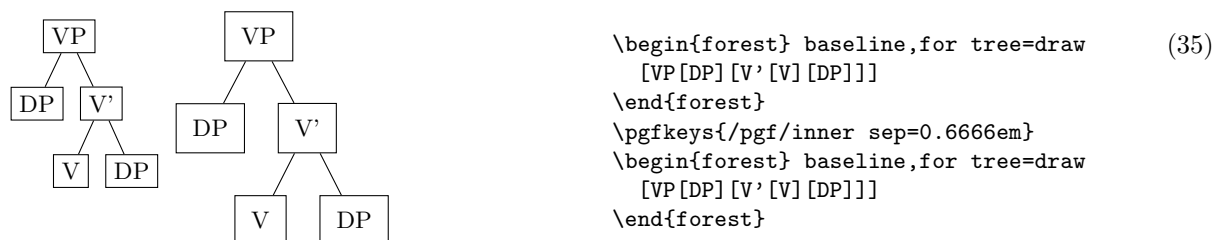
The following example shows that the vertical distance between nodes depends on the current font size.



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).

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

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

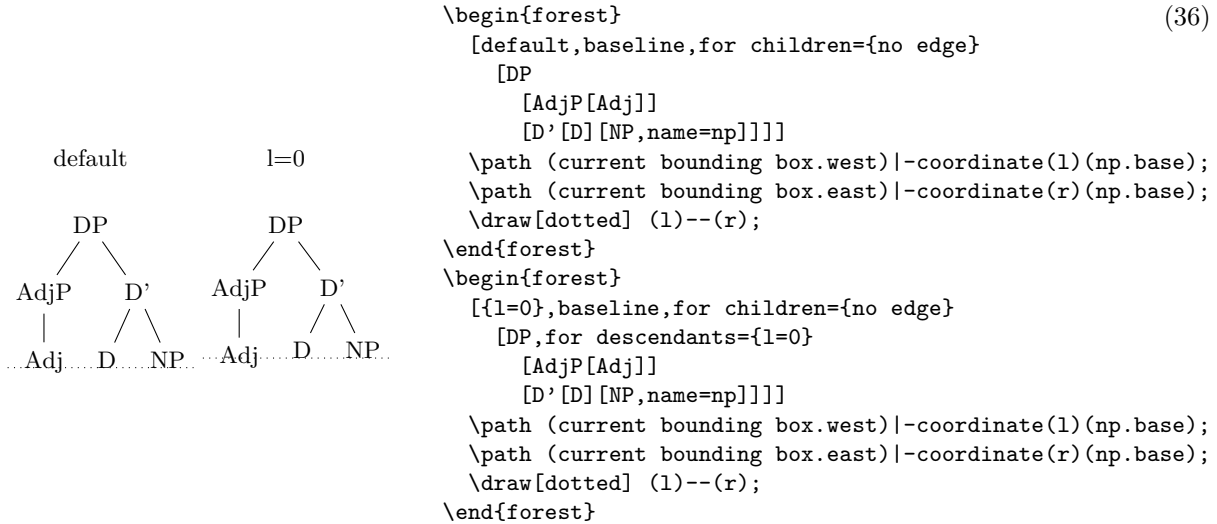


Now a hairy detail: the formula for the default `l`.

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

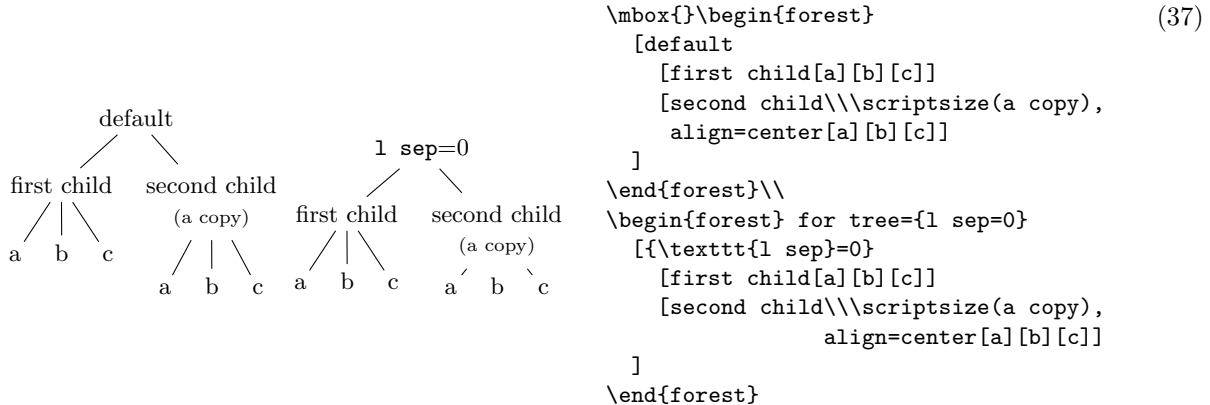
To understand what this is all about we must first explain why it is necessary to set the default `l` at all? Wouldn't it be enough to simply set `l sep` (leaving `l` at 0)? The problem is that not all letters have the same height and depth. A tree where the vertical position of the nodes would be controlled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





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 AdjP, 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 (32).

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 ...



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 T<sub>E</sub>X’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 (32), 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 3.7.

## 2.5 Advanced option setting

We have already seen that the value of options can be manipulated: in (13) we have converted numeric content from arabic into roman numerals using the *wrapping* mechanism `content=\romannumeral#1`; in (28), we have tripled the value of 1 by saying `1*=3`. In this section, we will learn about the mechanisms for setting and referring to option values offered by FOREST.

One other way to access an option value is using macro `\forestoption`. The macro takes a single argument: an option name. (For details, see §3.3.) In the following example, the node’s child sequence number is appended to the existing content. (This is therefore also an example of wrapping.)

$$\begin{array}{cccccc}
 c_1 & o_2 & u_3 & n_4 & t_5 \\
 \end{array}$$

```

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

```

(38)

However, only options of the current node can be accessed using `\forestoption`. To access option values of other nodes, FOREST’s extensions to the PGF’s mathematical library `pgfmath`, documented in [?, part VI], must be used. 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 (TikZ) option `rotate` is a full-fledged `pgfmath` expression yielding an integer in the range from  $-30$  to  $30$ . Similarly, `l+` adds a random float in the  $[-5, 5]$  range to the current value of `l`.

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

```

(39)

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).

```

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

```

(40)

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 §3.5) argument.<sup>9</sup> In the next example, we rotate the node based on the content of its parent.

```

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

```

(41)

<sup>9</sup>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.)

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.

Handlers `.wrap pgfmath arg` and `.wrap n pgfmath args` (for  $n = 2, \dots, 8$ ) 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 with the given macro. Below, this is used to include the number of parent's children in the index.

```
\begin{forest} [,phantom,delay={for descendants={
                                content/.wrap 3 pgfmath args=
                                {#1$_{#2/#3}$}{content}{n}{n_children("!u")}}]
  c1/5  o2/5  u3/5  n4/5  t5/5
  [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 (9) 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`). (Also, the numbers are not specified in content anymore: we simply read the sequence number `n`. And, to save some horizontal space for the code, each child of the root is pushed further down.)

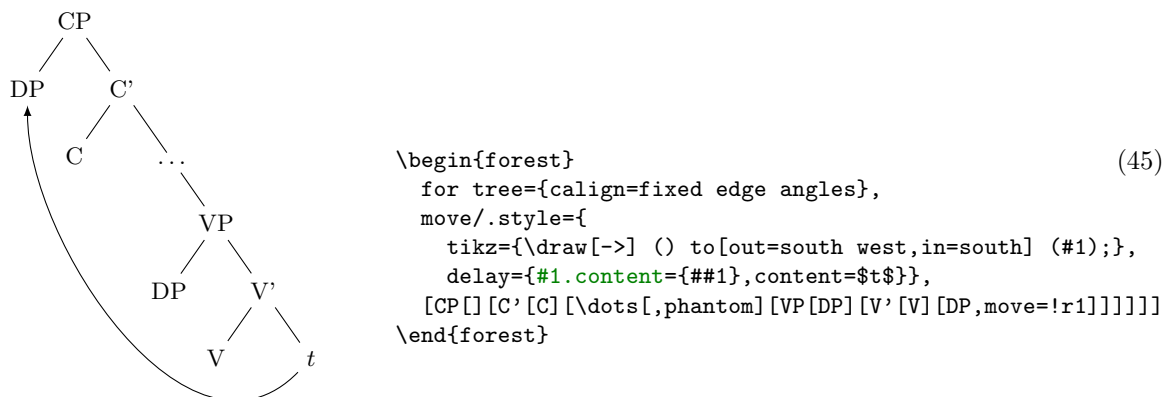
```
\begin{forest}
  delay={where level={2}{content/.wrap 2 pgfmath args=
    {\csname @#1\endcsname{#2}}{content("!u")}{n}}},
  for children={l*=n},
  [\LaTeX numerals,
   [arabic[] [] [] []]
   [roman[] [] [] []]
   [alph[] [] [] []]
  ]
\end{forest}
```

The final way to use `pgfmath` expressions in FOREST: `if` clauses. In section 2.2, we have seen that every option has a corresponding `if ...` (and `where ...`) option. However, these are just a matter of convenience. The full power resides in the general `if` option, which takes three arguments: `if=<condition><true options><>false options>`, where `<condition>` can be any `pgfmath` expression (non-zero means true, zero means false). (Once again, option `where` is an abbreviation for `for tree={if=...}`.) In the following example, `if` option is used to orient the arrows from the smaller number to the greater, and to color the odd and even numbers differently.

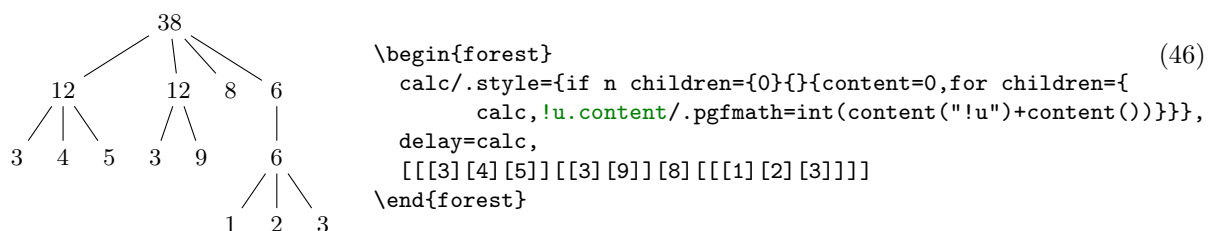
```
\pgfmathsetseed{314159}
\begin{forest}
  before typesetting nodes={
    for descendants={
      if={content()>content("!u")}{edge=->}{
        if={content()<content("!u")}{edge=<-}{}},
      edge label/.wrap pgfmath arg=
        {node[midway,above,sloped,font=\scriptsize]{+#1}}
        {int(abs(content()-content("!u")))}
    },
    for tree={circle,if={mod(content(),2)==0}
      {fill=yellow}{fill=green}}
  }
  [,random tree={3}{3}{100}]
\end{forest}
```

This exhausts the ways of using `pgfmath` in forest. 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 (`pgfmath` or `wrap`) 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 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`, i.e. the given node walk, while **##1** is the original option value (in this case, `content`).



In the following example, the content of the branching nodes is computed by `FOREST`: a branching node is a sum of its children. Besides the use of the relative node setting, this example notably uses a recursive style: for each child of the node, style `calc` first applies itself to the child and then adds the result to the node; obviously, recursion is made to stop at terminal nodes.



## 2.6 Externalization

`FOREST` can be quite slow, due to the slowness of both `PGF/TikZ` 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 lengthy tree-typesetting all over again.

`FOREST`'s externalization mechanism is built on top of `TikZ`'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:<sup>10</sup>

```

\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  $\text{\TeX}$  group.

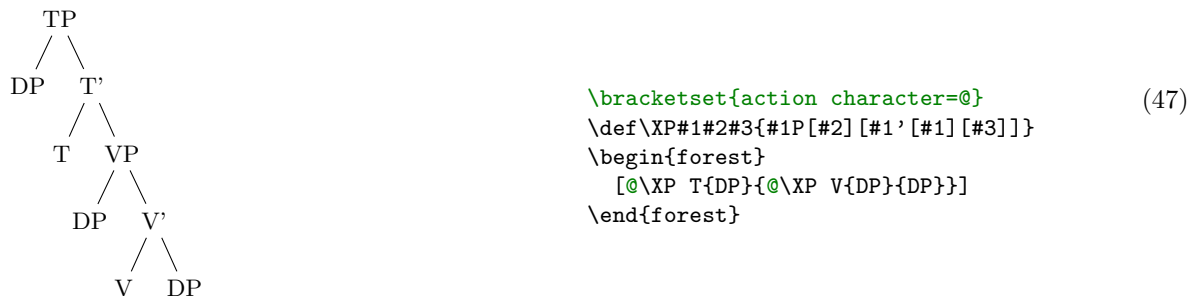
`TikZ`'s externalization library promises a `\label` inside the externalized graphics to work out-of-box, while `\ref` inside the externalized graphics should work only if the externalization is run manually or by `make` [?, §32.4.1]. A bit surprisingly perhaps, the situation is roughly reversed in `FOREST`. `\ref` inside the externalized graphics will work out-of-box. `\label` inside the externalized graphics will not work at

<sup>10</sup>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.

all. Sorry. (The reason is that FOREST prepares the node content in advance, before merging it in the whole tree, which is when *TikZ*'s externalization is used.)

## 2.7 Expansion control in the bracket parser

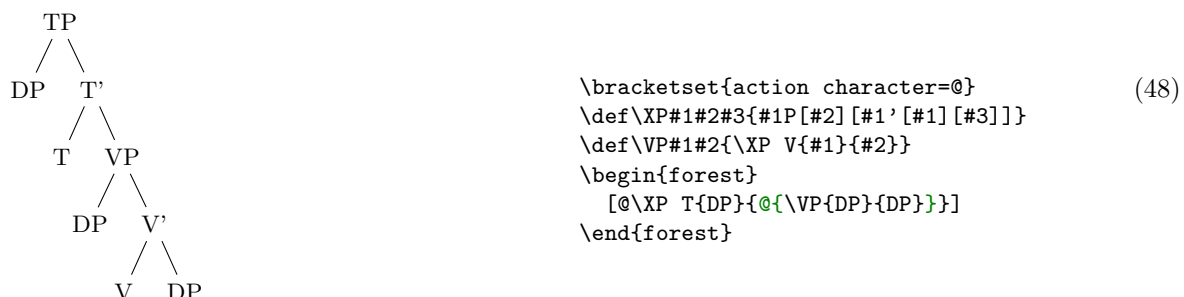
By default, macros in the bracket encoding of a tree are not expanded until nodes are being drawn — this way, node specification can contain formatting instructions, as illustrated in section 2.1. However, sometimes it is useful to expand macros while parsing the bracket representation, for example to define tree templates such as the X-bar template, familiar to generative grammarians:<sup>11</sup>



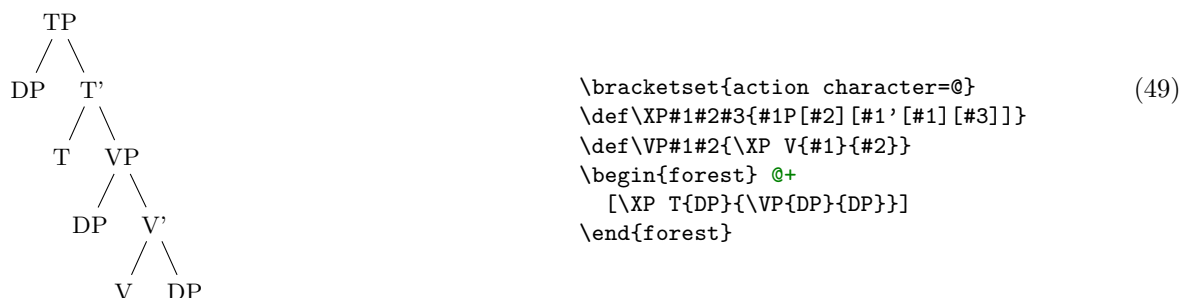
In the above example, the `\XP` macro is preceded by the *action character* `@`: as the result, the token following the action character was expanded before the parsing proceeded.

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.



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.



<sup>11</sup>Honestly, dynamic node creation might be a better way to do this; see §3.3.8.

All the action commands discussed above were dealing only with T<sub>E</sub>X’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:

$  \begin{array}{cccccc}  \times_1 & \times_2 & \times_3 & \times_4 & \times_5 & \times_6 \\    &   &   &   &   &   \\  f & o & r & e & s & t  \end{array}  $	<pre> \bracketset{action character=@} \newcount\xcount \def\x#1{@@\advance\xcount1 \edef\xtemp{[<math>\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} </math></pre>
---	---

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 stuck in front of the expansion.

## 3 Reference

### 3.1 Environments

*environment* `\begin{forest}` $\langle tree \rangle$ `\end{forest}`

`\Forest` $[*]$  $\{\langle tree \rangle\}$

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. (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.)

### 3.2 The bracket representation

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

$$\begin{aligned}
 \langle tree \rangle &= [\langle preamble \rangle] \langle node \rangle \\
 \langle node \rangle &= [ [\langle content \rangle] [ , \langle keylist \rangle ] ] \langle afterthought \rangle \\
 \langle preamble \rangle &= \langle keylist \rangle \\
 \langle keylist \rangle &= \langle key-value \rangle [ , \langle keylist \rangle ] \\
 \langle key-value \rangle &= \langle key \rangle | \langle key \rangle = \langle value \rangle \\
 \langle children \rangle &= \langle node \rangle [ \langle children \rangle ]
 \end{aligned}$$

The actual input might be different, though, since expansion may have occurred during the input reading. Expansion control sequences of FOREST’s bracket parser are shown below.

$\langle action\ character \rangle -$	no-expansion mode (default): nothing is expanded
$\langle action\ character \rangle +$	expansion mode: everything is fully expanded
$\langle action\ character \rangle \langle token \rangle$	expand $\langle token \rangle$
$\langle action\ character \rangle \langle T_{E}X-group \rangle$	fully expand $\langle T_{E}X-group \rangle$
$\langle action\ character \rangle \langle action\ character \rangle$	yield control; upon finishing its job, user’s code should call <code>\bracketResume</code>

**Customization** To customize the bracket parser, call `\bracketset{keylist}`, where the keys can be the following.

```
opening bracket= $\langle character \rangle$  [
closing bracket= $\langle character \rangle$  ]
action character= $\langle character \rangle$  none
```

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

**new node**= $\langle preamble \rangle \langle node\ specification \rangle \langle csname \rangle$ . 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  $\langle csname \rangle$ .

**set afterthought**= $\langle afterthought \rangle \langle node\ id \rangle$ . Required semantics: store the afterthought in the node with given id.

### 3.3 Options and keys

The position and outlook of nodes is controlled by *options*. Many options can be set for a node. *Each node's options are set independently of other nodes*: in particular, setting an option of a node does *not* set this option for the node's descendants.

Options are set using PGF's key management utility `pgfkeys` [?, §55]. In the bracket representation of a tree (see §3.2), each node can be given a  $\langle keylist \rangle$ . After parsing the representation of the tree, the keylists of the nodes are processed (recursively, in a depth-first, parent-first fashion). The preamble is processed first, in the context of the root node.<sup>12</sup>

The node whose keylist is being processed is the *current node*. During the processing of the keylist, the current node can temporarily change. This mainly happens when propagators (§3.3.6) are being processed.

Options can be set in various ways, depending on the option type (the types are listed below). The most straightforward way is to use the key with the same name as the option:

$\langle option \rangle = \langle value \rangle$  Sets the value of  $\langle option \rangle$  of the current node to  $\langle value \rangle$ .

Notes: (i) Obviously, this does not work for read-only options. (ii) Some option types override this behaviour.

It is also possible to set a non-current option:

$\langle relative\ node\ name \rangle . \langle option \rangle = \langle value \rangle$  Sets the value of  $\langle option \rangle$  of the node specified by  $\langle relative\ node\ name \rangle$  to  $\langle value \rangle$ .

Notes: (i)  $\langle value \rangle$  is evaluated in the context of the current node. (ii) In general, the resolution of  $\langle relative\ node\ name \rangle$  depends on the current node; see §3.5. (iii)  $\langle option \rangle$  can also be an “augmented operator” (see below) or an additional option-setting key defined for a specific option.

The option values can be not only set, but also read.

- Using macros `\forestoption{option}` and `\foresteoption{option}`, options of the current node can be accessed in T<sub>E</sub>X code. (“T<sub>E</sub>X code” includes  $\langle value \rangle$  expressions!).

In the context of `\edef` or PGF's handler `.expanded` [?, §55.4.6], `\forestoption` expands precisely to the token list of the option value, while `\foresteoption` allows the option value to be expanded as well.

- Using `pgfmath` functions defined by FOREST, options of both current and non-current nodes can be accessed. For details, see §3.6.

<sup>12</sup>The value of a key (if it is given) is interpreted as one or more arguments to the key command. If there is only one argument, the situation is simple: the whole value is the argument. When the key takes more than one argument, each argument should be enclosed in braces, unless, as usual in T<sub>E</sub>X, the argument is a single token. (The pairs of braces can be separated by whitespace.) An argument should also be enclosed in braces if it contains a special character: a comma `,`, an equal sign `=` or a bracket `[]`.



We continue with listing of all keys defined for every option. The set of defined keys and their meanings depends on the option type. Option types and the type-specific keys can be found in the list below. Common to all types are two simple conditionals, `if <option>` and `where <option>`, which are defined for every `<option>`; for details, see §3.3.6.

*type* `<toks>` contains T<sub>E</sub>X's `<balanced text>` [?, 275].

A toks `<option>` additionally defines the following keys:

`<option>+=<toks>` appends the given `<toks>` to the current value of the option.

`<option>-=<toks>` prepends the given `<toks>` to the current value of the option.

`if in <option>=<toks><true keylist><false keylist>` checks if `<toks>` occurs in the option value; if it does, `<true keylist>` are executed, otherwise `<false keylist>`.

`where in <option>=<toks><true keylist><false keylist>` is a style equivalent to `for tree={if in <option>=<toks><true keylist><false keylist>}`: for every node in the subtree rooted in the current node, `if in <option>` is executed in the context of that node.

*type* `<autowrapped toks>` is a subtype of `<toks>` and contains T<sub>E</sub>X's `<balanced text>` [?, 275].

`<option>=<toks>` of an autowrapped `<option>` is equivalent to `<option>/.wrap value=<toks>` of a normal `<toks>` option.

Keyvals `<option>+=<toks>` and `<option>-=<toks>` are equivalent 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 operators `<option>+` and `<option>-` automatically insert a comma before/after the appended/prepended material.

`<option>=<keylist>` of a keylist option is equivalent 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>`.

*type* `<dimen>` contains a dimension.

The value given to a dimension option is automatically evaluated by pgfmath. In other words:

`<option>=<pgfmath>` is an implicit `<option>/.pgfmath=<pgfmath>`.

For a `<dimen>` option `<option>`, the following additional keys (“augmented assignments”) are defined:

- `<option>+=<value>` is equivalent to `<option>=<option>()+<value>`
- `<option>-=<value>` is equivalent to `<option>=<option>()-<value>`
- `<option>*=<value>` is equivalent to `<option>=<option>()*<value>`
- `<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 T<sub>E</sub>X `<dimen>`:

1. pgfmath evaluation of simple values can be sped up by prepending `+` to the value [?, §62.1];
2. use the key `<option>'=<value>` to invoke a normal T<sub>E</sub>X assignment.

The two above-mentioned speed-up tricks work for the augmented assignments as well. The keys for the second, T<sub>E</sub>X-only trick are: `<option>'+`, `<option>'-`, `<option>'*` and `<option>:'` — 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).

The following subsections are a complete reference to the part of the user interface residing in the **pgfkeys**’ path */forest*. In plain language, they list all the options known to **FOREST**. More precisely, however, not only options are listed, but also other keys, such as propagators, conditionals, etc.

Before listing the keys, it is worth mentioning that users can also define their own keys. The easiest way to do this is by using *styles*. 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** [?, §55.4.4]: *style name*/**.style**=*keylist*.

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

**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.

**declare count**=*option name*<*default value*> Declares a *count* option.

**declare boolean**=*option name*<*default value*> Declares a *boolean* option.

The style definitions and option declarations given among the other keys in the bracket specification are local to the current tree. To define globally accessible styles and options (well, definitions are always local to the current  $\text{\TeX}$  group), use macro **\forestset** outside the **forest** environment:<sup>13</sup>

**\forestset**{*keylist*}

Execute *keylist* with the default path set to */forest*.

→ Usually, no current node is set when this macro is called. Thus, executing node options 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.

### 3.3.1 Node appearance

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

option **align**=*left*,**aspect**=*align* | *center*,**aspect**=*align* | *right*,**aspect**=*align* | *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 be 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
<b>left</b>	@{}l@{}
<b>center</b>	@{}c@{}
<b>right</b>	@{}r@{}

top base  
right aligned

left aligned  
bottom base

```
\begin{forest} 1 sep+=2ex
[special value&actual value\\hline
\rkeyname{left,aspect=align}&||\texttt{@\{\\}l@{\}}\\
\rkeyname{center,aspect=align}&||\texttt{@\{\\}c@{\}}\\
\rkeyname{right,aspect=align}&||\texttt{@\{\\}r@{\}}\\
,align=ll,draw
[top base\\right aligned, align=right,base=top]
[left aligned\\bottom base, align=left,base=bottom]
]
\end{forest}
```

<sup>13</sup>**\forestset**{*keylist*} is equivalent to **\pgfkeys**{*/forest*,*keylist*}.

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`= $\langle toks: vertical alignment \rangle$

`t`

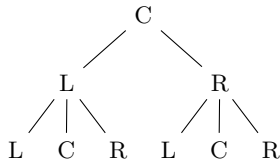
This option controls the vertical alignment of multiline (and in general, `tabular`) nodes created with `align`. Its value becomes the optional argument to the `tabular` environment. Thus, sensible values are `t` (the top line of the table will be the baseline) and `b` (the bottom line of the table will 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`= $\langle autowrapped toks \rangle$  The content of the node.

`{}`

Normally, the value of option `content` is given implicitly by virtue of the special (initial) position of content in the bracket representation (see §3.2). However, the option also be set explicitly, as any other option.



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

(52)

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



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

(53)

option `content format`= $\langle toks \rangle$

`\forestoption{content}`

When typesetting the node under the default conditions (see option `node format`), the value of this option is passed to the TikZ `node` operation as its  $\langle text \rangle$  argument [?, §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. For details, read the documentation of option `node format` and macros `\forestoption` and `\foresteoption`; for an example, see option `align`.

*option* **node format**= $\langle toks \rangle$  **\noexpand\node**  

$$[\backslash\forestoption\{node\ options\}, anchor=\backslash\forestoption\{anchor\}]$$

$$(\backslash\forestoption\{name\})\{\backslash\forestoption\{content\ format\}\};$$

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

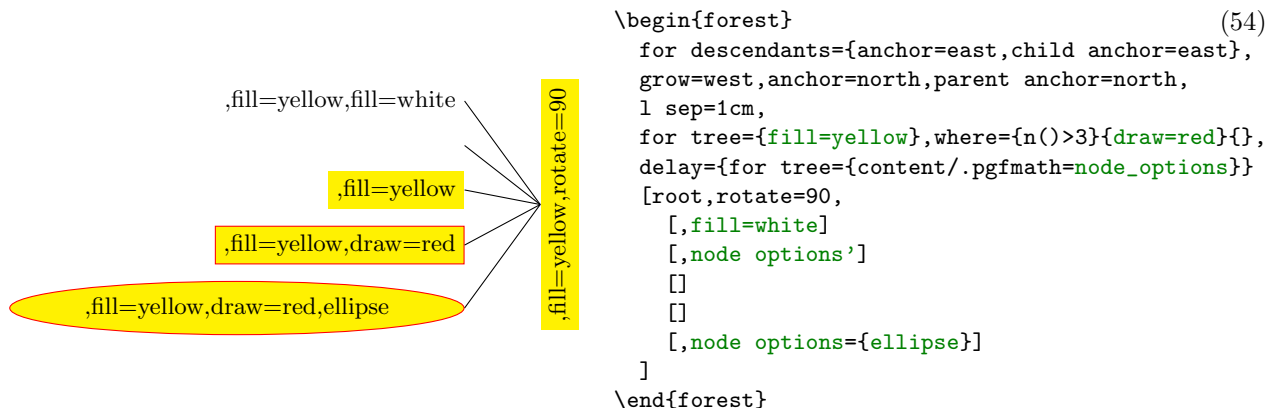
Important: the value of this option is first expanded using `\edef` and only then executed. Note that in its default value, **content format** is fully expanded using `\forestoption`: this is necessary for complex content formats, such as `tabular` environments.

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

*option* **node options**= $\langle keylist \rangle$  **{}**

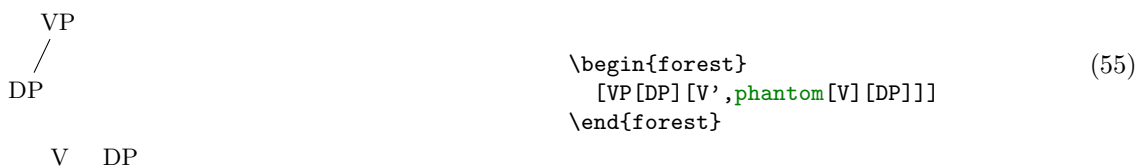
When the node is being typeset under the default conditions (see option **node format**), the content of this option is passed to TikZ as options to the TikZ `node` operation [? , §16].

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.



*option* **phantom**= $\langle boolean \rangle$  **false**

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



### 3.3.2 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**= $\langle toks: TikZ\ anchor\ name \rangle$  **base**

This is essentially a TikZ option [see ? , §16.5.1] — it is passed to TikZ as a node option when the node is typeset (this option thus applies in stage **typeset nodes**) — but it is also saved by FOREST.

The effect of this option is only observable when a node has a sibling: the anchors of all siblings are s-aligned (if their `ls` have not been modified after packing).

In the TikZ code, you can refer to the node's anchor using the generic anchor `anchor`.

*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`.

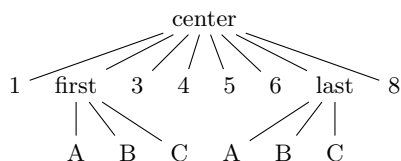
`calign=last` is an abbreviation for `calign=child,calign child=-1`.

`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`.



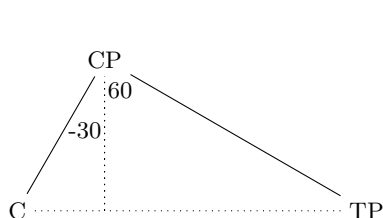
```
\begin{forest}
[center,calign=center[1]
[first,calign=first[A] [B] [C]] [3] [4] [5] [6]
[last,calign=last[A] [B] [C]] [8]]
\end{forest}
```

`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}
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**= $\langle count \rangle$  Sets the secondary child. (See **calign**.) -1

$\langle count \rangle$  is the child's sequence number. Negative numbers start counting at the last child.

**calign angle**= $\langle count \rangle$  is an abbreviation for **calign primary angle**= $-\langle count \rangle$ , **calign secondary angle**= $\langle count \rangle$ .

option **calign primary angle**= $\langle count \rangle$  Sets the primary angle. (See **calign**.) -35

option **calign secondary angle**= $\langle count \rangle$  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}.

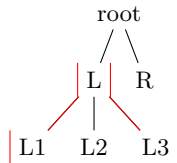
**calign with current edge** s-aligns the child anchor of the current node and the parent anchor of its parent. This key is an abbreviation for:

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

option **fit**=tight|rectangle|band **tight**

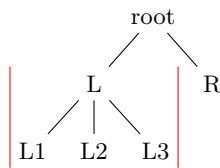
This option sets the type of the (s-)boundary that will be computed for the subtree rooted in the node, thereby determining how it will be packed into the subtree rooted in the node's parent. There are three choices:<sup>14</sup>

- **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.



```
\begin{forest} (59)
  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.

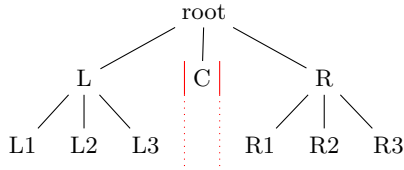


```
\begin{forest} (60)
  delay={for tree={name/.pgfmath=content}}
  [root
    [L,fit=rectangle,
      show boundary
      [L1] [L2] [L3]]
    [R]
  ]
\end{forest}
```

- **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.

<sup>14</sup>Below is the definition of style **show boundary**. The use path trick is adjusted from T<sub>E</sub>X 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];}}
```



```
(61)
\begin{forest}
  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}
```

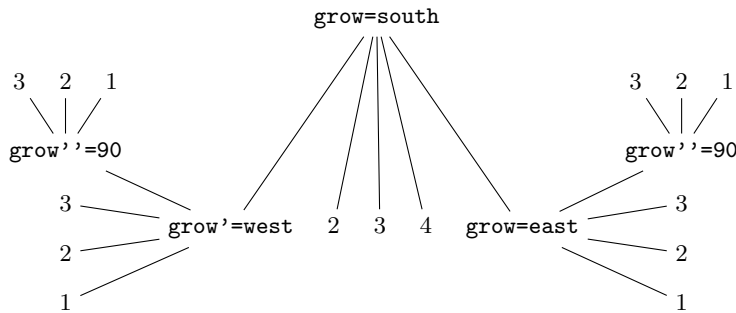
option **grow**= $\langle count \rangle$  The direction of the tree's growth at the node.

270

The growth direction is understood as in TikZ's tree library [?, §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 TikZ, 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.



```
(62)
\begin{forest}
  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**= $\langle boolean \rangle$

false

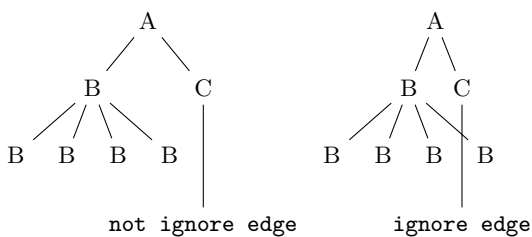
If this option is set, the packing mechanism ignores the node, i.e. it pretends that the node has no boundary. Note: this only applies to the node, not to the tree.

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

option **ignore edge**= $\langle boolean \rangle$

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 §5 for some problematic situations.)



```
(63)
\begin{forest}
  [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**= $\langle \text{dimen} \rangle$  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:

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

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** method **fixed (edge) angles**.)
- Changing **l** after packing but before stage **compute xy** will result in a manual adjustment of the computed position. (The augmented 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**= $\langle \text{dimen} \rangle$  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

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

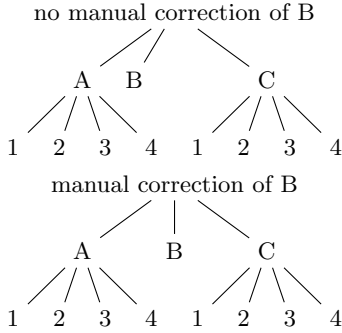
Note that despite the similar name, the semantics of **l sep** and **s sep** are quite different.

*option* **reversed**= $\langle \text{boolean} \rangle$  **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* **s**= $\langle \text{dimen} \rangle$  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 by the packing mechanism. Any value given before packing is overridden. In short, it only makes sense to (inspect and) change this option after stage **pack**, which can be useful for manual corrections, like below. (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.) Changing the value of **s** after stage **compute xy** has no effect.



```

\begin{minipage}{.5\linewidth}
\begin{forest}
  [no manual correction of B
    [A[1][2][3][4]]
    [B]
    [C[1][2][3][4]]
  ]
\end{forest}

\begin{forest}
  [manual correction of B
    [A[1][2][3][4]]
    [B,before computing xy={s=(s("!p")+s("!n"))/2}]
    [C[1][2][3][4]]
  ]
\end{forest}
\end{minipage}

```

(64)

*option* **s sep**= $\langle \text{dimen} \rangle$

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  $2 \cdot \text{inner xsep}$ .

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

*option* **tier**= $\langle \text{toks} \rangle$  { }

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.

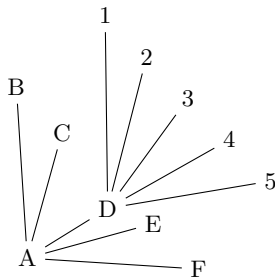
Tier alignment also does not work well with **calign=fixed (edge) angles**, because these child alignment methods may change the l-position of the children. When this might happen, FOREST will yield a warning.

*option* **x**= $\langle \text{dimen} \rangle$

*option* **y**= $\langle \text{dimen} \rangle$

**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 §3.3.7.)



```

\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}

```

(65)

### 3.3.3 Edges

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

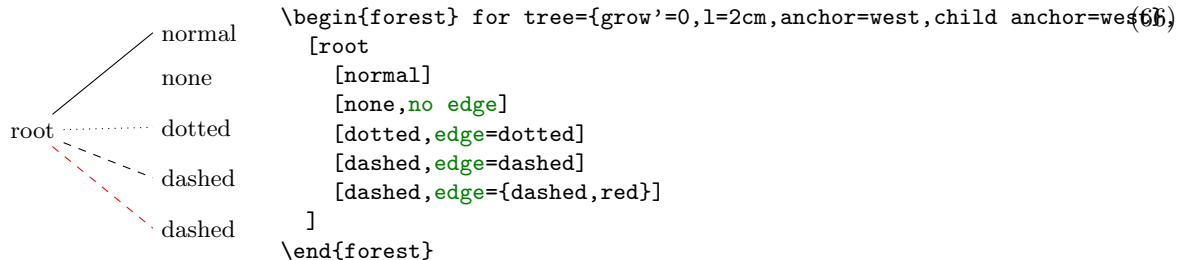


option **child anchor**=*(toks)* See **parent anchor**. {}

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.

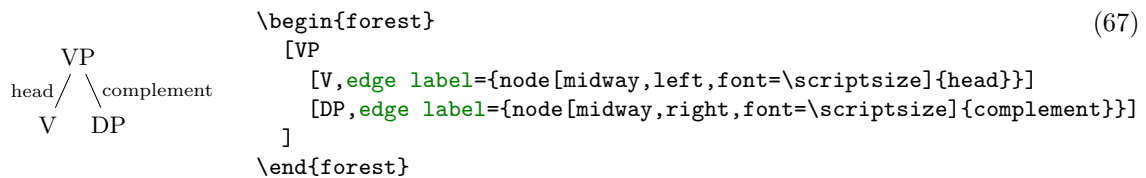
Also see key **no edge**.



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.



option **edge path**=*(toks: TikZ 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 setting this option, the values of options **edge** and **edge label** can be used in the edge path specification to include the values of options **edge** and **edge node**. Furthermore, two generic anchors, **parent anchor** and **child anchor**, are defined, to facilitate access to options **parent anchor** and **child anchor** from the TikZ 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**.)

option **parent anchor**=*(toks: TikZ anchor)* (Information also applies to option **child anchor**.) {}

FOREST defines generic anchors **parent anchor** and **child anchor** (which work only for FOREST and not also TikZ 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 (TikZ) anchor given as the value.

The indented use of the two anchors is chiefly in **edge path** specification, but they can be used in any TikZ code.



```

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

```

(68)

The empty value (which is the default) is interpreted as in TikZ: as an edge to the appropriate border point.

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

**triangle** Makes the edge to parent a triangular roof. Works only for south-growing trees. Works by changing the value of **edge path**.

### 3.3.4 Readonly

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

*option* **id**= $\langle count \rangle$ ) The internal id of the node.

*option* **level**= $\langle count \rangle$  The hierarchical level of the node. The root is on level 0.

*option* **max x**= $\langle dimen \rangle$

*option* **max y**= $\langle dimen \rangle$

*option* **min x**= $\langle dimen \rangle$

*option* **min y**= $\langle dimen \rangle$  Measures of the node, in the shape's coordinate system [see ? , §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**.

*option* **n**= $\langle count \rangle$  The child's sequence number in the list of its parent's children.

The enumeration starts with 1. For the root node, **n** equals 0.

*option* **n'**= $\langle count \rangle$  Like **n**, but starts counting at the last child.

In **pgfmath** expressions, this option is accessible as **n\_**.

*option* **n children**= $\langle count \rangle$  The number of children of the node.

In **pgfmath** expressions, this option is accessible as **n\_children**.

### 3.3.5 Miscellaneous

**afterthought**= $\langle toks \rangle$  Provides the afterthought explicitly.

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) TikZ code. If you'd like to use afterthoughts for some other purpose, redefine the key — this will take effect even if you do it in the tree preamble.

**alias**= $\langle toks \rangle$  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.

Aliases can be used as the  $\langle forest\ node\ name \rangle$  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 TikZ alias! However, you can still use it as a “node name” in TikZ coordinates, since FOREST hacks TikZ's implicit node coordinate system to accept relative node names; see §3.5.2.

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

In plain language, when the tree is inserted in your (normal  $\text{\TeX}$ ) 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`.

<p style="text-align: center;">parent</p> <p style="text-align: center;"> </p> <p>Baseline at the parent and baseline at the child.</p> <p style="text-align: center;"> </p> <p style="text-align: center;">child</p>	<pre style="margin: 0;">{\tikzexternaldisable 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}.}</pre>
---	---

(69)

**fit to tree** Fits the *TikZ* node to the current node's subtree.

This key should be used as an option to *TikZ*'s `node` operation, in the context of some *FOREST* node; see the example in footnote 6.

**get min s tree boundary**= $\langle cs \rangle$

**get max s tree boundary**= $\langle cs \rangle$

Puts the boundary computed during the packing process into the given  $\langle cs \rangle$ . 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.

**label**= $\langle toks: \text{\textit{TikZ node}} \rangle$  The current node is labelled by a *TikZ* node.

The label is specified as a *TikZ* option `label` [? , §16.10]. Technically, the value of this option is passed to *TikZ*'s as a late option [? , §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* **name**= $\langle toks \rangle$  Sets the name of the node. **node**@ $\langle id \rangle$

The expansion of  $\langle toks \rangle$  becomes the  $\langle forest\ node\ name \rangle$  of the node. Node names must be unique. The *TikZ* node created from the *FOREST* node will get the name specified by this option.

**node walk**= $\langle node\ walk \rangle$  This key is the most general way to use a  $\langle node\ walk \rangle$ .

Before starting the  $\langle node\ walk \rangle$ , key `node walk/before walk` is processed. Then, the  $\langle step \rangle$ s composing the  $\langle node\ walk \rangle$  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 options and keys described in §3.3 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  $\text{\TeX}$  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).

**pin**= $\langle toks: \text{\textit{TikZ node}} \rangle$  The current node gets a pin, see [? , §16.10].

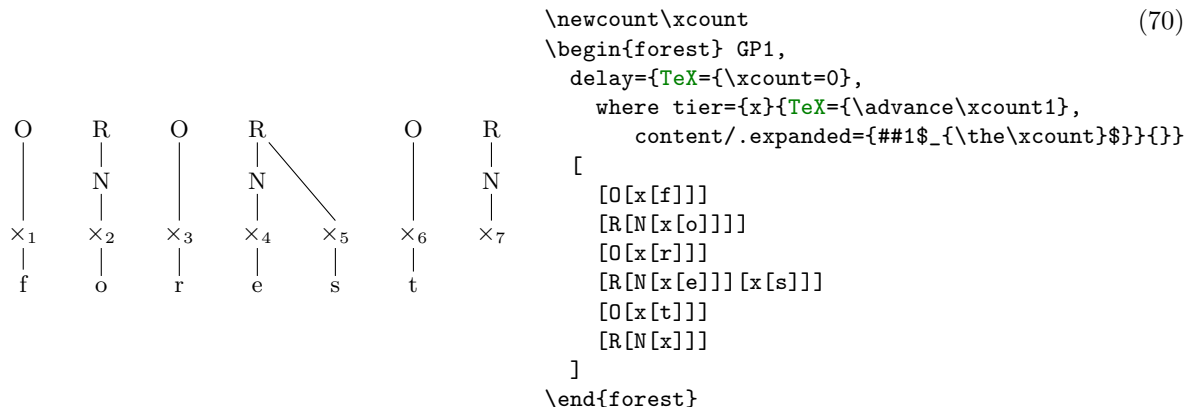
The technical details are the same as for `label`.

**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](#).

**TeX**= $\langle$ *toks: T<sub>E</sub>X code* $\rangle$  The given code is executed immediately.

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



**TeX’**= $\langle$ *toks: T<sub>E</sub>X code* $\rangle$  This key is a combination of keys **TeX** and **TeX’**: the given code is both executed and externalized.

**TeX’’**= $\langle$ *toks: T<sub>E</sub>X code* $\rangle$  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.

*option* **tikz**= $\langle$ *toks: TikZ code* $\rangle$  “Decorations.” { }

The code given as the value of this option will be included in the **tikzpicture** environment used to draw the tree. The code given to various nodes is appended in a depth-first, parent-first fashion. 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 §3.5.

By default, bracket parser’s afterthoughts feed the value of this option. See [afterthought](#).

*option* **tikz preamble**= $\langle$ *toks: TikZ code* $\rangle$  { }

If the current node is the root of the tree that is being drawn (see stage **draw tree**), the code given to this option is prepended to the generated code.

### 3.3.6 Propagators

Propagators pass the given  $\langle$ *keylist* $\rangle$  to other node(s), delay their processing, or cause them to be processed only under certain conditions.

A propagator can never fail — i.e. if you use **for next** on the last child of some node, no error will arise: the  $\langle$ *keylist* $\rangle$  will simply not be passed to any node. (The generic node walk propagator **for** is an exception. While it will not fail if the final node of the walk does not exist (is null), its node walk can fail when trying to walk away from the null node.)

**Spatial propagators** pass the given  $\langle$ *keylist* $\rangle$  to other node(s) in the tree. (**for** and **for**  $\langle$ *step* $\rangle$  always pass the  $\langle$ *keylist* $\rangle$  to a single node.)

*propagator* **for**= $\langle$ *node walk* $\rangle$  $\langle$ *keylist* $\rangle$  Processes  $\langle$ *keylist* $\rangle$  in the context of the final node in the  $\langle$ *node walk* $\rangle$  starting at the current node.

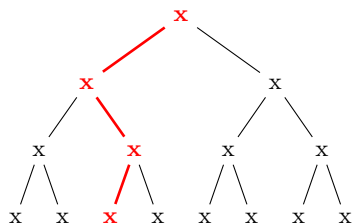
*key prefix* **for**  $\langle step \rangle = \langle keylist \rangle$  Walks a single-step node-walk  $\langle step \rangle$  from the current node and passes the given  $\langle keylist \rangle$  to the final (i.e. second) node.

$\langle step \rangle$  must be a long node walk step; see §3.5.1. **for**  $\langle step \rangle = \langle keylist \rangle$  is equivalent to **for**  $= \langle step \rangle keylist$ .

Examples: **for**  $parent = \{1\} sep += 3mm$ , **for**  $n = 2 \{circle, draw\}$ .

*propagator* **for** **ancestors**  $= \langle keylist \rangle$

*propagator* **for** **ancestors'**  $= \langle keylist \rangle$  Passes the  $\langle keylist \rangle$  to itself, too.



```
\pgfkeys{/forest,
  inptr/.style={%
    red,delay={content={\textbf{##1}}},
    edge={draw,line width=1pt,red}},
  ptr/.style={for ancestors'=inptr}
}
\begin{forest}
  [x
    [x[x[x][x]][x[x,ptr][x]]]
    [x[x[x][x]][x[x][x]]]
  ]
\end{forest}
```

(71)

*propagator* **for** **all next**  $= \langle keylist \rangle$  Passes the  $\langle keylist \rangle$  to all the following siblings.

*propagator* **for** **all previous**  $= \langle keylist \rangle$  Passes the  $\langle keylist \rangle$  to all the preceding siblings.

*propagator* **for** **children**  $= \langle keylist \rangle$

*propagator* **for** **descendants**  $= \langle keylist \rangle$

*propagator* **for** **tree**  $= \langle keylist \rangle$

Passes the key to the current node and its the descendants.

This key should really be named **for subtree** ...

**Conditionals** For all conditionals, both the true and the false keylist are obligatory! Either keylist can be empty, however — but don't omit the braces!

*propagator* **if**  $= \langle pgfmath\ condition \rangle \langle true\ keylist \rangle \langle false\ keylist \rangle$

If  $\langle pgfmath\ condition \rangle$  evaluates to **true** (non-zero),  $\langle true\ keylist \rangle$  is processed (in the context of the current node); otherwise,  $\langle false\ keylist \rangle$  is processed.

For a detailed description of **pgfmath** expressions, see [? , part VI]. (In short: write the usual mathematical expressions.)

*key prefix* **if**  $\langle option \rangle = \langle value \rangle \langle true\ keylist \rangle \langle false\ keylist \rangle$

A simple conditional is defined for every  $\langle option \rangle$ : if  $\langle value \rangle$  equals the value of the option at the current node,  $\langle true\ keylist \rangle$  is executed; otherwise,  $\langle false\ keylist \rangle$ .

*propagator* **where**  $= \langle value \rangle \langle true\ keylist \rangle \langle false\ keylist \rangle$

Executes conditional **if** for every node in the current subtree.

*key prefix* **where**  $\langle option \rangle = \langle value \rangle \langle true\ keylist \rangle \langle false\ keylist \rangle$

Executes simple conditional **if**  $\langle option \rangle$  for every node in the current subtree.

*key prefix* **if in**  $\langle option \rangle = \langle toks \rangle \langle true\ keylist \rangle \langle false\ keylist \rangle$

Checks if  $\langle toks \rangle$  occurs in the option value; if it does,  $\langle true\ keylist \rangle$  are executed, otherwise  $\langle false\ keylist \rangle$ .

This conditional is defined only for  $\langle toks \rangle$  options, see §3.3.

*key prefix* **where in**  $\langle toks \ option \rangle = \langle toks \rangle \langle true \ keylist \rangle \langle false \ keylist \rangle$

A style equivalent to **for tree=if in**  $\langle option \rangle = \langle toks \rangle \langle true \ keylist \rangle \langle false \ keylist \rangle$ : for every node in the subtree rooted in the current node, **if in**  $\langle option \rangle$  is executed in the context of that node.

This conditional is defined only for  $\langle toks \rangle$  options, see §3.3.

**Temporal propagators** There are two kinds of temporal propagators. The **before** ... propagators defer the processing of the given keys to a hook just before some stage in the computation. The **delay** propagator is “internal” to the current hook (the first hook, the given options, is implicit): the keys in a hook are processed cyclically, and **delay** delays the processing of the given options until the next cycle. All these keys can be nested without limit. For details, see §3.3.7.

*propagator* **delay**= $\langle keylist \rangle$  Defers the processing of the  $\langle keylist \rangle$  until the next cycle.

*propagator* **before typesetting nodes**= $\langle keylist \rangle$  Defers the processing of the  $\langle keylist \rangle$  to until just before the nodes are typeset.

*propagator* **before packing**= $\langle keylist \rangle$  Defers the processing of the  $\langle keylist \rangle$  to until just before the nodes are packed.

*propagator* **before computing xy**= $\langle keylist \rangle$  Defers the processing of the  $\langle keylist \rangle$  to until just before the absolute positions of the nodes are computed.

*propagator* **before drawing tree**= $\langle keylist \rangle$  Defers the processing of the  $\langle keylist \rangle$  to until just before the tree is drawn.

### Other propagators

**repeat**= $\langle number \rangle \langle keylist \rangle$  The  $\langle keylist \rangle$  is processed  $\langle number \rangle$  times.

The  $\langle number \rangle$  expression is evaluated using **pgfmath**. Propagator **repeat** also works in node walks.

### 3.3.7 Stages

FOREST does its job in several steps. The normal course of events is the following:

1. parsing the bracket representation of the tree (and storing it in a data structure)
2. processing the given options (including the preamble, which is processed first, in the context of the root node)
3. processing style **stages**, which by default contains the following keys:

*style* **before typesetting nodes hook** {**process keylist=before typesetting nodes**}

*stage* **typeset nodes**

Obviously, the nodes must be measured before they can be packed. Thus, each node is typeset in its own **tikzpicture** environment, saved in a box and its measures are taken.

*style* **after typesetting nodes hook** {}

*style* **before packing hook** {**process keylist=before packing**}

*stage* **pack**

The relative positions of the nodes are computed so that they don’t overlap. That’s difficult. The result: option **s** is set for all nodes. (Sometimes, the value of **l** is adjusted as well.)

*style* **after packing hook** {}

*style* **before computing xy hook** {**process keylist=before computing xy**}

*stage* **compute xy**

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.

```

style after computing xy hook {}
style before drawing tree hook {process keylist=before drawing tree}
stage draw tree
  All the nodes are poured in a single tikzpicture, using the boxes from stage typeset nodes
  and positions from stage compute xy. They are followed by parent–child edges, custom code
  and details.
style after drawing tree hook {}

```

All the user input comes into the system in steps 1 (the hierarchical structure) and 2 (the content). The package does its work in step 3, which consists of four *stages*, where the work is really done, and hooks before and after each stage, which provide a way to fine-tune the workflow.

By default, the `after ...` hooks do nothing, and the `before ...` hooks process the deferred keylists. For a simple example, see example (65): 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 `x`s uniformly spaced; before drawing the tree, the `outer xsep` is set to `0pt` to make the arrows look better.

Style `stages` and the hook styles can be redefined. Obviously, a style must be redefined before it is processed, so its safest to do so either outside the `forest` environment using macro `\forestset`, or in stage 2 among the non-deferred keys. The following keys might be useful in a redefinition. (`typeset nodes`, `pack` and `compute xy` are not included in the list, since they were already fully described above.)

*stage* `draw tree'` Like `draw tree`, but the node boxes (usually computed in stage `typeset nodes`) are included in the picture using `\copy`, not `\box`, thereby preserving them.

Using `draw tree` and `draw tree'` multiple times *is* compatible with externalization.

`draw tree box=[ $\langle T_{\text{E}}X \text{ box} \rangle$ ]` The picture drawn by the subsequent invocations of `draw tree` and `draw tree'` is put into  $\langle T_{\text{E}}X \text{ box} \rangle$ . If the argument is omitted, the pictures are typeset normally (the default).

Using this key *is* compatible with externalization.

*stage* `typeset nodes'` Like `typeset nodes`, but the node box's content is not overwritten if it already exists.

`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 `x`s is implemented in the `GP1` style.

`process keylist= $\langle \text{keylist option name} \rangle$`  Processes the keylist saved in option  $\langle \text{keylist option name} \rangle$  for all the nodes in the *whole* tree.

This key is not sensitive to the current node: it processes the keylists for the whole tree. The calls of this key should *not* be nested.

Keylist-processing proceeds in cycles. In a given cycle, the value of option  $\langle \text{keylist option name} \rangle$  is processed for every node, in a recursive (parent-first, depth-first) fashion. During a cycle, keys may be *delayed* using key `delay`. (Keys of the dynamically created nodes are automatically delayed.) Keys delayed in a cycle are processed in the next cycle. The number of cycles is unlimited. When no keys are delayed in a cycle, the processing of a hook is finished.

### 3.3.8 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:

$\langle empty \rangle$ : uses the last (non-integrated, i.e. created/removed/replaced) node.

$\langle node \rangle$ : 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.

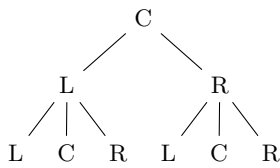
The bracket representation must be enclosed in brackets, which will usually be enclosed in braces to prevent them being parsed while parsing the “host tree.”

$\langle relative\ node\ name \rangle$ : the node  $\langle relative\ node\ name \rangle$  resolves to will be used.

Here is the list of dynamic tree keys:

*dynamic tree* **append**= $\langle empty \rangle$  | [ $\langle node \rangle$ ] |  $\langle relative\ node\ name \rangle$

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}
  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}

```

(72)

*dynamic tree* **create**= $\langle node \rangle$

Create a new node. The new node becomes the last node.

*dynamic tree* **insert after**= $\langle empty \rangle$  | [ $\langle node \rangle$ ] |  $\langle relative\ node\ name \rangle$

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**= $\langle empty \rangle$  | [ $\langle node \rangle$ ] |  $\langle relative\ node\ name \rangle$

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**= $\langle empty \rangle$  | [ $\langle node \rangle$ ] |  $\langle relative\ node\ name \rangle$

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**= $\langle empty \rangle$  | [ $\langle node \rangle$ ] |  $\langle relative\ node\ name \rangle$

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  $\langle empty \rangle$  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).



dynamic tree **set root**

The current node becomes the new *formal* root of the tree.

Note: If the current 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 node walk step **root'**, or the corresponding propagator **for root'**.

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.

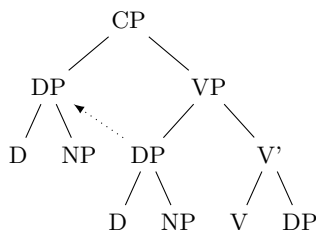
dynamic tree **append''**, **insert after''**, **insert before''**, **prepend''**, **replace by''**

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.

dynamic tree **copy name template**= $\langle empty \rangle$  |  $\langle macro\ definition \rangle$   $\langle empty \rangle$

Defines a template for constructing the **name** of the copy from the name of the original.  $\langle macro\ definition \rangle$  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).

→ You might want to **delay** the processing of the copying operations, giving the original nodes the chance to process their keys first!



```

\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}

```

(73)

A dynamic tree operation is made in two steps:

- If the argument is given by a  $\langle node \rangle$  argument, the new node is created immediately, i.e. while the dynamic tree key is being processed. Any options of the new node are implicitly **delayed**.
- 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 (72).

→ Further examples: title page (in style **random tree**), (80).

### 3.4 Handlers

handler **.pgfmath**= $\langle pgfmath\ expression \rangle$

The result is the evaluation of  $\langle pgfmath\ expression \rangle$  in the context of the current node.

handler **.wrap value**= $\langle macro\ definition \rangle$

The result is the (single) expansion of the given  $\langle macro\ definition \rangle$ . The defined macro takes one parameter. The current value of the handled option will be passed as that parameter.

handler **.wrap** *n* **pgfmath** **args**= $\langle macro\ definition \rangle \langle arg\ 1 \rangle \dots \langle arg\ n \rangle$

The result is the (single) expansion of the given  $\langle macro\ definition \rangle$ . The defined macro takes  $n$  parameters, where  $n \in \{2, \dots, 8\}$ . Expressions  $\langle arg\ 1 \rangle$  to  $\langle arg\ n \rangle$  are evaluated using **pgfmath** and passed as arguments to the defined macro.

handler **.wrap** **pgfmath** **arg**= $\langle macro\ definition \rangle \langle arg \rangle$

Like **.wrap** *n* **pgfmath** **args** for  $n = 1$ .

### 3.5 Relative node names

$\langle relative\ node\ name \rangle = [(\langle forest\ node\ name \rangle)][!\langle node\ walk \rangle]$

$\langle relative\ node\ name \rangle$  refers to the FOREST node at the end of the  $\langle node\ walk \rangle$  starting at node named  $\langle forest\ node\ name \rangle$ . If  $\langle forest\ node\ name \rangle$  is omitted, the walk starts at the current node. If  $\langle node\ walk \rangle$  is omitted, the “walk” ends at the start node. (Thus, an empty  $\langle relative\ node\ name \rangle$  refers to the current node.)

Relative node names can be used in the following contexts:

- FOREST’s **pgfmath** option functions (§3.6) 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  $\langle relative\ node\ name \rangle.\langle option\ name \rangle = \langle value \rangle$ , see §3.3.
- The **forest** coordinate system, both explicit and implicit; see §3.5.2.

#### 3.5.1 Node walk

A  $\langle node\ walk \rangle$  is a sequence of  $\langle step \rangle$ s describing a path through the tree. The primary use of node walks is in relative node names. However, they can also be used in a “standalone” way, using key **node walk**; see §3.3.5.

Steps are keys in the **/forest/node walk** path. (FOREST always sets this path as default when a node walk is to be used, so step keynames can be used.) Formally, a  $\langle node\ walk \rangle$  is thus a keylist, and steps must be separated by commas. There is a twist, however. Some steps also have *short* names, which consist of a single character. The comma between two adjacent short steps can be omitted. Examples:

- **parent, parent, n=2** or **uu2**: the grandparent’s second child (of the current node)
- **first leaf, uu**: the grandparent of the first leaf (of the current node)

The list of long steps:

$\langle step \rangle$  **current** an “empty” step: the current node remains the same<sup>15</sup>

$\langle step \rangle$  **first** the primary child

$\langle step \rangle$  **first leaf** the first leaf (terminal node)

$\langle step \rangle$  **group**= $\langle node\ walk \rangle$  treat the given  $\langle node\ walk \rangle$  as a single step

$\langle step \rangle$  **last** the last child

$\langle step \rangle$  **last leaf** the last leaf

$\langle step \rangle$  **id**= $\langle id \rangle$  the node with the given *id*

$\langle step \rangle$  **linear next** the next node, in the processing order

<sup>15</sup>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  $\langle keylist \rangle$  constructed using **.wrap** (*n*) **pgfmath** **arg(s)** or **.wrap value**.

$\langle \text{step} \rangle$  **linear previous** the previous node, in the processing order  
 $\langle \text{step} \rangle$  **n**= $n$  the  $n$ th child; counting starts at 1 (not 0)  
 $\langle \text{step} \rangle$  **n'**= $n$  the  $n$ th child, starting the count from the last child  
 $\langle \text{step} \rangle$  **name** the node with the given name  
 $\langle \text{step} \rangle$  **next** the next sibling  
 $\langle \text{step} \rangle$  **next leaf** the next leaf  
     (the current node need not be a leaf)  
 $\langle \text{step} \rangle$  **next on tier** the next node on the same tier as the current node  
 $\langle \text{step} \rangle$  **node walk**= $\langle \text{node walk} \rangle$  embed the given  $\langle \text{node walk} \rangle$   
     (the **node walk/before walk** and **node walk/after walk** are processed)  
 $\langle \text{step} \rangle$  **parent** the parent  
 $\langle \text{step} \rangle$  **previous** the previous sibling  
 $\langle \text{step} \rangle$  **previous leaf** the previous leaf  
     (the current node need not be a leaf)  
 $\langle \text{step} \rangle$  **previous on tier** the next node on the same tier as the current node  
**repeat**= $n \langle \text{node walk} \rangle$  repeat the given  $\langle \text{node walk} \rangle$   $n$  times  
     (each step in every repetition counts as a step)  
 $\langle \text{step} \rangle$  **root** the root node  
 $\langle \text{step} \rangle$  **root'** the formal root node (see **set root** in §3.3.8)  
 $\langle \text{step} \rangle$  **sibling** the sibling  
     (don't use if the parent doesn't have exactly two children ...)  
 $\langle \text{step} \rangle$  **to tier**= $\langle \text{tier} \rangle$  the first ancestor of the current node on the given  $\langle \text{tier} \rangle$   
 $\langle \text{step} \rangle$  **trip**= $\langle \text{node walk} \rangle$  after walking the embedded  $\langle \text{node walk} \rangle$ , return to the current node; the return does not count as a step

For each long  $\langle \text{step} \rangle$  except **node walk**, **group**, **trip** and **repeat**, propagator **for**  $\langle \text{step} \rangle$  is also defined. Each such propagator takes a  $\langle \text{keylist} \rangle$  argument. If the step takes an argument, then so does its propagator; this argument precedes the  $\langle \text{keylist} \rangle$ . See also §3.3.6.

Short steps are single-character keys in the **/forest/node walk** path. They are defined as styles resolving to long steps, e.g. **1/.style={n=1}**. The list of predefined short steps follows.

$\langle \text{short step} \rangle$  **1, 2, 3, 4, 5, 6, 7, 8, 9** the first, ..., ninth child  
 $\langle \text{short step} \rangle$  **l** the last child  
 $\langle \text{short step} \rangle$  **u** the parent (up)  
 $\langle \text{short step} \rangle$  **p** the previous sibling  
 $\langle \text{short step} \rangle$  **n** the next sibling  
 $\langle \text{short step} \rangle$  **s** the sibling  
 $\langle \text{short step} \rangle$  **P** the previous leaf  
 $\langle \text{short step} \rangle$  **N** the next leaf

*<short step>* **F** the first leaf

*<short step>* **L** the last leaf

*<short step>* **>** the next node on the current tier

*<short step>* **<** the previous node on the current tier

*<short step>* **c** the current node

*<short step>* **r** the root node

→ You can define your own short steps, or even redefine predefined short steps!

### 3.5.2 The forest coordinate system

Unless package options **tikzcshack** is set to **false**, TikZ's implicit node coordinate system [?, §13.2.3] is hacked to accept relative node names.<sup>16</sup>

The explicit **forest** coordinate system is called simply **forest** and used like this: (**forest cs:***<forest cs spec>*); see [?, §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 became the current node. The resulting point is its (node) anchor.

*forest cs* **id**=*<node id>* The node with the given name became the current node. The resulting point is its (node) anchor.

*forest cs* **go**=*<node walk>* Walk the given node walk, starting at the current node. The node at the end of the walk becomes the current node. The resulting point is its (node) anchor.

*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>*].

## 3.6 New pgfmath functions

For every option, FOREST defines a pgfmath function with the same name, with the proviso that all non-alphanumeric characters in the option name are replaced by an underscore **\_** in the pgfmath function name.

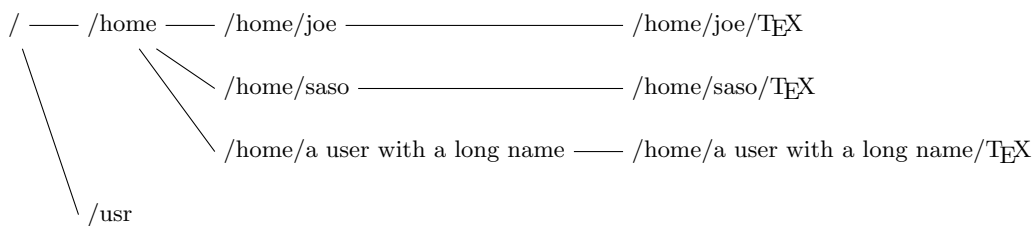
Pgfmath functions corresponding to options take one argument, a *<relative node name>* (see §3.5) expression, making it possible to refer to option values of non-current nodes. The *<relative node name>* expression must be enclosed in double quotes in order to prevent pgfmath evaluation: for example, to refer to the content of the parent, write **content**("!u"). To refer to the option of the current node, use empty parentheses: **content**().<sup>17</sup>

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.

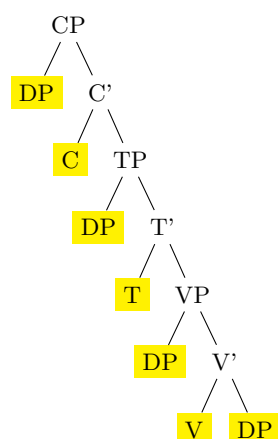
Some random notes on pgfmath: (i) **&&**, **||** and **!** are boolean “and”, “or” and “not”, respectively. (ii) The equality operator (for numbers and dimensions) is **==**, *not* **=**. And some examples:

<sup>16</sup>Actually, the hack can be switched on and off on the fly, using **\ifforesttikzcshack**.

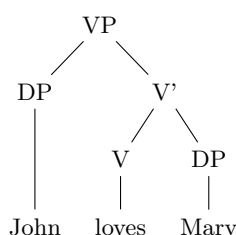
<sup>17</sup>In most cases, the parentheses are optional, so **content** is ok. A known case where this doesn't work is preceding an operator: **1+1cm** will fail.



```
\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}
```



```
\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}
```



```
\begin{forest}
  where n children=0{tier=word,
    if={instr("!P",content("!u"))}{no edge,
      tikz={\draw (!.north west)--
        (!.north east)--(!u.south)--cycle;
      }}{}}
[VP[DP[John]][V'[V[loves]][DP[Mary]]]]
\end{forest}
```

### 3.7 Standard node

**\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'être* for such a system is given in §2.4.1.

In  $\langle node \rangle$ , the standard node's content and possibly other options are specified, using the usual bracket representation. The  $\langle node \rangle$ , however, *must not contain children*. The default: [dj].

The  $\langle environment fingerprint \rangle$  must be an expandable macro definition. It's expansion should change whenever the calibration is necessary.

$\langle calibration procedure \rangle$  is a keylist (processed in the /forest path) which calculates the values of exported options.

$\langle exported options \rangle$  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}
```

### 3.8 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. TikZ'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**= $\langle cs \rangle$  Adds the definition of macro  $\langle cs \rangle$  to external/context. Thus, if the definition of  $\langle cs \rangle$  is changed, the tree will be recompiled.

FOREST respects or is compatible with several (not all) keys and commands of TikZ's externalization library. In particular, the following keys and commands might be useful; see [?, §32].

- /tikz/external/remake next
- /tikz/external/prefix
- /tikz/external/system call
- \tikzexternalize
- \tikzexternalenable
- \tikzexternaldisable

FOREST does not disturb 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.

Externalizing a `forest` environment with custom `stages` will only work if the only externally visible products are the trees produced by `draw tree` and `draw tree'` (with or without the optional argument).

### 3.9 Package options

<i>package option</i>	<code>external=true false</code>	<code>false</code>
	Enable/disable externalization, see §3.8.	
<i>package option</i>	<code>tikzcshack=true false</code>	<code>true</code>
	Enable/disable the hack into TikZ's implicate coordinate syntax hacked, see §3.5.	
<i>package option</i>	<code>tikzinstallkeys=true false</code>	<code>true</code>
	Install certain keys into the <code>/tikz</code> path. Currently: <code>fit to tree</code> .	

## 4 Gallery

### 4.1 Styles

**GP1** For Government Phonology (v1) representations. Here, the big trick is to evenly space `xs` 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 `xs` won't draw!

```

\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={\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}}}}
}

```

An example of an “embedded” GP1 style:

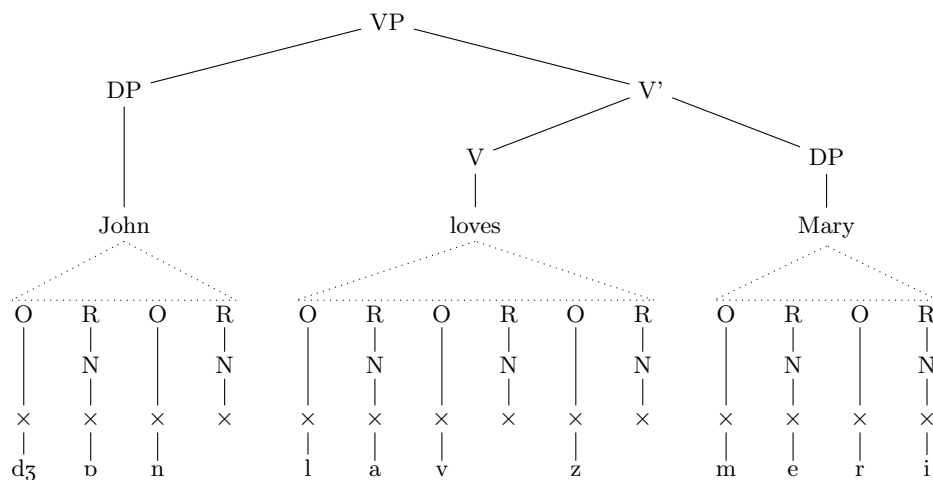


(77)

```

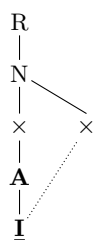
\begin{forest}
myGP1/.style={
  GP1,
  delay={where tier={x}{
    for children={content=\textipa{##1}}{}}{
    tikz={\draw[dotted](.south)--
      (!1.north west)--(!1.north east)--cycle;},
    for children={l+=5mm,no edge}
  }
}
[VP[DP[John,tier=word,myGP1
  [O[x[dZ]]]
  [R[N[x[6]]]]
  [O[x[n]]]
  [R[N[x]]]
]] [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}%

```

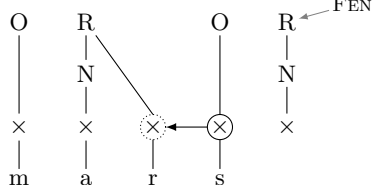


And an example of annotations.

[ei]



[mars]



```

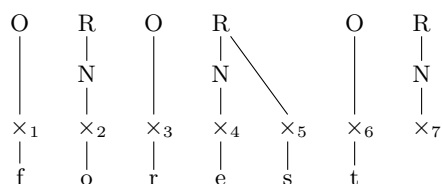
\begin{forest}[,phantom,s sep=1cm
[{{ei}}, GP1
[R[N[x[A,el[I,head,associate=N]]][x]]]
]
[{{mars}}, GP1
[O[x[m]]]
[R[N[x[a]]][x,encircle,densely dotted[r]]]
[O[x,encircle,govern=<[s]]]
[R,fen[N[x]]]
]
]\end{forest}

```

(78)

**rlap and llap** The FOREST versions of TeX’s `\rlap` and `\llap`: the “content” added by these styles will influence neither the packing algorithm nor the anchor positions.

```
\forestset{
  llap/.style={tikz+={
    \edef\forest@temp{\noexpand\node[\forestoption{node options},
      anchor=base east,at=(.base east)]}
    \forest@temp{#1\phantom{\forestoption{content format}}};
  }},
  rlap/.style={tikz+={
    \edef\forest@temp{\noexpand\node[\forestoption{node options},
      anchor=base west,at=(.base west)]}
    \forest@temp{\phantom{\forestoption{content format}}#1};
  }}
}
\newcount\xcount
\begin{forest} GP1,
  delay={
    TeX={\xcount=0},
    where tier={x}{TeX={\advance\xcount1},rlap/.expanded={$_{\the\xcount}$}}{}
  }
  [
    [O[x[f]]]
    [R[N[x[o]]]]
    [O[x[r]]]
    [R[N[x[e]]][x[s]]]
    [O[x[t]]]
    [R[N[x]]]
  ]
\end{forest}
```



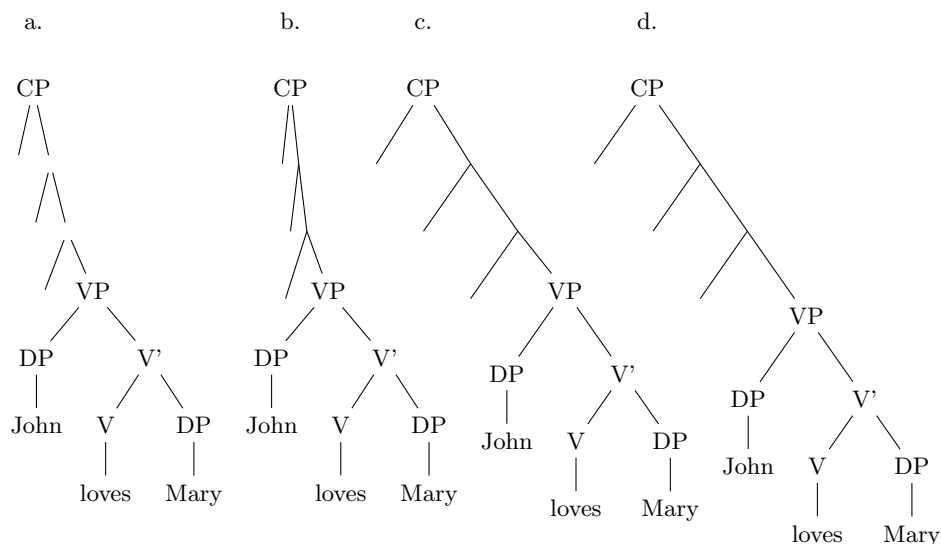
**xlist** This style makes it easy to put “separate” trees in a picture and enumerate them. For an example, see the `nice empty nodes` style.

```
\makeatletter
\forestset{
  xlist/.style={
    phantom,
    for children={no edge,replace by={[,append,
      delay={content/.wrap pgfmath arg={\@alph{##1}.}{n()+#1}}
    ]}}
  },
  xlist/.default=0
}
\makeatother
```

**nice empty nodes** We often need empty nodes: tree (a) 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 (b). 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=node angle`. The result is shown in (c).

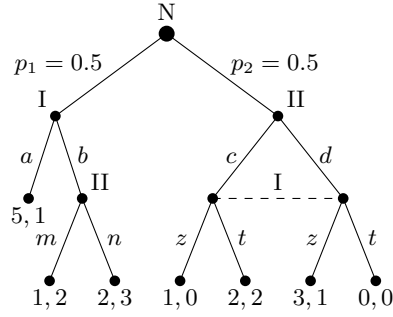
However, at the transitions from empty to non-empty nodes, tree (d) above seems to zigzag (although the base points of the spine nodes are perfectly in line), and the edge to the empty node left to VP seems too long (it reaches to the level of VP's base, while we'd prefer it to stop at the same level as the edge to VP itself). The first problem is solved by substituting `node angle` for `edge angle`; the second one, by anchoring siblings of empty nodes at north.

```
\forestset{
  nice empty nodes/.style={
    for tree={calign=fixed edge angles},
    delay={where content={}{shape=coordinate,for parent={for children={anchor=north}}{}}
  }}
\begin{forest}
  [,xlist
    [CP,
      [ [ [ [ [VP[DP[John]] [V' [V[loves]] [DP[Mary]]]]]]]]
    ]
    [CP, delay={where content={}{shape=coordinate}{}}
      [ [ [ [ [VP[DP[John]] [V' [V[loves]] [DP[Mary]]]]]]]]
    ]
    [CP, for tree={calign=fixed angles},
      delay={where content={}{shape=coordinate}{}}
      [ [ [ [ [VP[DP[John]] [V' [V[loves]] [DP[Mary]]]]]]]]
    ]
    [CP, nice empty nodes
      [ [ [ [ [VP[DP[John]] [V' [V[loves]] [DP[Mary]]]]]]]]
    ]
  ]
\end{forest}
```



## 4.2 Examples

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 inputting the tree is very easy.



(82)

```

\def\getfirst#1;#2\endget{#1}
\def\getsecond#1;#2\endget{#2}
\forestset{declare toks={elo}{}} % edge label options
\begin{forest}
  anchors/.style={anchor=#1,child anchor=#1,parent anchor=#1},
  for tree={
    s sep=0.5em,l=8ex,
    if n children=0{anchors=north}{
      if n=1{anchors=south east}{anchors=south west}},
    content format={\$\forestoption{content}$}
  },
  anchors=south, outer sep=2pt,
  nomath/.style={content format=\forestoption{content}},
  dot/.style={tikz+={\fill (.child anchor) circle[radius=#1];}},
  dot/.default=2pt,
  dot=3pt,for descendants=dot,
  decision edge label/.style n args=3{
    edge label/.expanded={node[midway,auto=#1,anchor=#2,\forestoption{elo}]{\strut$#3$}}
  },
  decision/.style={if n=1
    {decision edge label={left}{east}{#1}}
    {decision edge label={right}{west}{#1}}
  },
  delay={for descendants={
    decision/.expanded/.wrap pgfmath arg={\getsecond#1\endget}{content},
    content/.expanded/.wrap pgfmath arg={\getfirst#1\endget}{content},
  }},
  [N,nomath
    [I;{p_1=0.5},nomath,elo={yshift=4pt}
      [{5,1};a]
      [II;b,nomath
        [{1,2};m]
        [{2,3};n]
      ]
    ]
    [II;{p_2=0.5},nomath,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 Known bugs

If you find a bug (there are bound to be some ...), please contact me at [saso.zivanovic@guest.arnes.si](mailto:saso.zivanovic@guest.arnes.si).

**System requirements** This package requires L<sup>A</sup>T<sub>E</sub>X and e<sub>T</sub><sub>E</sub>X. If you use something else: sorry.

The requirement for L<sup>A</sup>T<sub>E</sub>X 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 e<sub>T</sub><sub>E</sub>X will probably stay. If nothing else, FOREST is heavy on boxes: every node requires its own ... and consequently, I have freely used e<sub>T</sub><sub>E</sub>X constructs in the code ...

**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`.

```

\begin{forest}
  calign=first
  [R[AAAAA\AAAAA\AAAAA,align=center,base=bottom] [B]]
\end{forest}

```

(83)

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.

```

\begin{forest}
  calign=last
  [R[AAAAA\AAAAA\AAAAA,align=center,base=bottom] [B]]
\end{forest}

```

(84)

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 to say about the desired distance between them: in this sense, node C below is an “orphan.”

```

\begin{forest}
  for tree={s sep=0,draw},
  [R[A] [B] [C,l*=2] [D] [E]]
\end{forest}

```

(85)

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}

```

(86)

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.

## Part II

# Implementation

A disclaimer: the code could've been much cleaner and better-documented ...  
Identification.

```

1 \ProvidesPackage{forest}[2012/10/31 v1.0 Drawing (linguistic) trees]
2
3 \RequirePackage{tikz}
4 \usetikzlibrary{shapes}
5 \usetikzlibrary{fit}
6 \usetikzlibrary{calc}
7 \usepgflibrary{intersections}
8
9 \RequirePackage{pgfplots}
10 \RequirePackage{etoolbox}
11 \RequirePackage{environ}
12
13 %\usepackage[trace]{trace-pgfkeys}
    /forest is the root of the key hierarchy.
14 \pgfkeys{/forest/.is family}
15 \def\forestset#1{\pgfqkeys{/forest}{#1}}
```

## 6 Patches

These patches apply to pgf/tikz 2.10.

Serious: forest cannot load if this is not patched; disable /handlers/.wrap n pgfmath for n=6,7,8 if you cannot patch.

```

16 \long\def\forest@original@pgfkeysdefnargs@#1#2#3#4{%
17   \ifcase#2\relax
18   \pgfkeyssetvalue{#1/.@args}{}%
19   \or
20   \pgfkeyssetvalue{#1/.@args}{##1}%
21   \or
22   \pgfkeyssetvalue{#1/.@args}{##1##2}%
23   \or
24   \pgfkeyssetvalue{#1/.@args}{##1##2##3}%
25   \or
26   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4}%
27   \or
28   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5}%
29   \or
30   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
31   \or
32   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
33   \or
34   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7}%
35   \or
36   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8}%
37   \or
38   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8##9}%
39   \else
40   \pgfkeys@error{\string\pgfkeysdefnargs: expected <= 9 arguments, got #2}%
41   \fi
42   \pgfkeysgetvalue{#1/.@args}\pgfkeys@tempargs
43   \def\pgfkeys@temp{\expandafter#4\cname pgfk@#1/.@body\endcsname}%
44   \expandafter\pgfkeys@temp\pgfkeys@tempargs{#3}%
45   % eliminate the \pgfeov at the end such that TeX gobbles spaces
```

```

46 % by using
47 % \pgfkeysdef{#1}{\pgfkeysvalueof{#1/.@@body}##1}
48 % (with expansion of '#1'):
49 \edef\pgfkeys@tempargs{\noexpand\pgfkeysvalueof{#1/.@@body}}%
50 \def\pgfkeys@temp{\pgfkeysdef{#1}}%
51 \expandafter\pgfkeys@temp\expandafter{\pgfkeys@tempargs##1}%
52 \pgfkeyssetvalue{#1/.@body}{#3}%
53 }
54
55 \long\def\forest@patched@pgfkeysdefnargs@#1#2#3#4{%
56   \ifcase#2\relax
57   \pgfkeyssetvalue{#1/.@args}{}%
58   \or
59   \pgfkeyssetvalue{#1/.@args}{##1}%
60   \or
61   \pgfkeyssetvalue{#1/.@args}{##1##2}%
62   \or
63   \pgfkeyssetvalue{#1/.@args}{##1##2##3}%
64   \or
65   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4}%
66   \or
67   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5}%
68   \or
69   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
70   %%%% removed:
71   %%%% \or
72   %%%% \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
73   \or
74   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7}%
75   \or
76   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8}%
77   \or
78   \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8##9}%
79   \else
80   \pgfkeys@error{\string\pgfkeysdefnargs: expected <= 9 arguments, got #2}%
81   \fi
82   \pgfkeysgetvalue{#1/.@args}\pgfkeys@tempargs
83   \def\pgfkeys@temp{\expandafter#4\csname pgfk@#1/.@@body\endcsname}%
84   \expandafter\pgfkeys@temp\pgfkeys@tempargs{#3}%
85   % eliminate the \pgfeov at the end such that TeX gobbles spaces
86   % by using
87   % \pgfkeysdef{#1}{\pgfkeysvalueof{#1/.@@body}##1}
88   % (with expansion of '#1'):
89   \edef\pgfkeys@tempargs{\noexpand\pgfkeysvalueof{#1/.@@body}}%
90   \def\pgfkeys@temp{\pgfkeysdef{#1}}%
91   \expandafter\pgfkeys@temp\expandafter{\pgfkeys@tempargs##1}%
92   \pgfkeyssetvalue{#1/.@body}{#3}%
93 }
94 \ifx\pgfkeysdefnargs@\forest@original@pgfkeysdefnargs@
95   \let\pgfkeysdefnargs@\forest@patched@pgfkeysdefnargs@
96 \fi

```

Minor: a leaking space in the very first line.

```

97 \def\forest@original@pgfpointintersectionoflines#1#2#3#4{%
98   {
99     %
100    % Compute orthogonal vector to #1--#2
101    %
102    \pgf@process{#2}%
103    \pgf@xa=\pgf@x%
104    \pgf@ya=\pgf@y%

```

```

105 \pgf@process{#1}%
106 \advance\pgf@xa by-\pgf@x%
107 \advance\pgf@ya by-\pgf@y%
108 \pgf@ya=-\pgf@ya%
109 % Normalise a bit
110 \c@pgf@counta=\pgf@xa%
111 \ifnum\c@pgf@counta<0\relax%
112 \c@pgf@counta=-\c@pgf@counta\relax%
113 \fi%
114 \c@pgf@countb=\pgf@ya%
115 \ifnum\c@pgf@countb<0\relax%
116 \c@pgf@countb=-\c@pgf@countb\relax%
117 \fi%
118 \advance\c@pgf@counta by\c@pgf@countb\relax%
119 \divide\c@pgf@counta by 65536\relax%
120 \ifnum\c@pgf@counta>0\relax%
121 \divide\pgf@xa by\c@pgf@counta\relax%
122 \divide\pgf@ya by\c@pgf@counta\relax%
123 \fi%
124 %
125 % Compute projection
126 %
127 \pgf@xc=\pgf@sys@tonumber{\pgf@ya}\pgf@x%
128 \advance\pgf@xc by\pgf@sys@tonumber{\pgf@xa}\pgf@y%
129 %
130 % The orthogonal vector is (\pgf@ya,\pgf@xa)
131 %
132 %
133 % Compute orthogonal vector to #3--#4
134 %
135 \pgf@process{#4}%
136 \pgf@xb=\pgf@x%
137 \pgf@yb=\pgf@y%
138 \pgf@process{#3}%
139 \advance\pgf@xb by-\pgf@x%
140 \advance\pgf@yb by-\pgf@y%
141 \pgf@yb=-\pgf@yb%
142 % Normalise a bit
143 \c@pgf@counta=\pgf@xb%
144 \ifnum\c@pgf@counta<0\relax%
145 \c@pgf@counta=-\c@pgf@counta\relax%
146 \fi%
147 \c@pgf@countb=\pgf@yb%
148 \ifnum\c@pgf@countb<0\relax%
149 \c@pgf@countb=-\c@pgf@countb\relax%
150 \fi%
151 \advance\c@pgf@counta by\c@pgf@countb\relax%
152 \divide\c@pgf@counta by 65536\relax%
153 \ifnum\c@pgf@counta>0\relax%
154 \divide\pgf@xb by\c@pgf@counta\relax%
155 \divide\pgf@yb by\c@pgf@counta\relax%
156 \fi%
157 %
158 % Compute projection
159 %
160 \pgf@yc=\pgf@sys@tonumber{\pgf@yb}\pgf@x%
161 \advance\pgf@yc by\pgf@sys@tonumber{\pgf@xb}\pgf@y%
162 %
163 % The orthogonal vector is (\pgf@yb,\pgf@xb)
164 %
165 % Setup transformation matrix (this is just to use the matrix

```



```

166 % inversion)
167 %
168 \pgfsettransform{\pgf@sys@tonumber\pgf@ya}{\pgf@sys@tonumber\pgf@yb}{\pgf@sys@tonumber\pgf@xa}{\pgf@sys@
169 \pgftransforminvert%
170 \pgf@process{\pgfpointtransformed{\pgfpoint{\pgf@xc}{\pgf@yc}}}%
171 }%
172 }
173 \def\forest@patched@pgfpointintersectionoflines#1#2#3#4{%
174 {% added the percent sign in this line
175 %
176 % Compute orthogonal vector to #1--#2
177 %
178 \pgf@process{#2}%
179 \pgf@xa=\pgf@x%
180 \pgf@ya=\pgf@y%
181 \pgf@process{#1}%
182 \advance\pgf@xa by-\pgf@x%
183 \advance\pgf@ya by-\pgf@y%
184 \pgf@ya=-\pgf@ya%
185 % Normalise a bit
186 \c@pgf@counta=\pgf@xa%
187 \ifnum\c@pgf@counta<0\relax%
188 \c@pgf@counta=-\c@pgf@counta\relax%
189 \fi%
190 \c@pgf@countb=\pgf@ya%
191 \ifnum\c@pgf@countb<0\relax%
192 \c@pgf@countb=-\c@pgf@countb\relax%
193 \fi%
194 \advance\c@pgf@counta by\c@pgf@countb\relax%
195 \divide\c@pgf@counta by 65536\relax%
196 \ifnum\c@pgf@counta>0\relax%
197 \divide\pgf@xa by\c@pgf@counta\relax%
198 \divide\pgf@ya by\c@pgf@counta\relax%
199 \fi%
200 %
201 % Compute projection
202 %
203 \pgf@xc=\pgf@sys@tonumber{\pgf@ya}\pgf@x%
204 \advance\pgf@xc by\pgf@sys@tonumber{\pgf@xa}\pgf@y%
205 %
206 % The orthogonal vector is (\pgf@ya,\pgf@xa)
207 %
208 %
209 % Compute orthogonal vector to #3--#4
210 %
211 \pgf@process{#4}%
212 \pgf@xb=\pgf@x%
213 \pgf@yb=\pgf@y%
214 \pgf@process{#3}%
215 \advance\pgf@xb by-\pgf@x%
216 \advance\pgf@yb by-\pgf@y%
217 \pgf@yb=-\pgf@yb%
218 % Normalise a bit
219 \c@pgf@counta=\pgf@xb%
220 \ifnum\c@pgf@counta<0\relax%
221 \c@pgf@counta=-\c@pgf@counta\relax%
222 \fi%
223 \c@pgf@countb=\pgf@yb%
224 \ifnum\c@pgf@countb<0\relax%
225 \c@pgf@countb=-\c@pgf@countb\relax%
226 \fi%

```

```

227 \advance\c@pgf@counta by\c@pgf@countb\relax%
228 \divide\c@pgf@counta by 65536\relax%
229 \ifnum\c@pgf@counta>0\relax%
230 \divide\pgf@xb by\c@pgf@counta\relax%
231 \divide\pgf@yb by\c@pgf@counta\relax%
232 \fi%
233 %
234 % Compute projection
235 %
236 \pgf@yc=\pgf@sys@tonumber{\pgf@yb}\pgf@x%
237 \advance\pgf@yc by\pgf@sys@tonumber{\pgf@xb}\pgf@y%
238 %
239 % The orthogonal vector is (\pgf@yb,\pgf@xb)
240 %
241 % Setup transformation matrix (this is just to use the matrix
242 % inversion)
243 %
244 \pgfsettransform{\pgf@sys@tonumber\pgf@ya}{\pgf@sys@tonumber\pgf@yb}{\pgf@sys@tonumber\pgf@xa}{\pgf@sys@tonumber\pgf@yb}{\pgf@sys@tonumber\pgf@xc}{\pgf@sys@tonumber\pgf@yc}%
245 \pgftransforminvert%
246 \pgf@process{\pgfpointtransformed{\pgfpoint{\pgf@xc}{\pgf@yc}}}%
247 }%
248 }
249
250 \ifx\pgfpointintersectionoflines\forest@original\pgfpointintersectionoflines
251 \let\pgfpointintersectionoflines\forest@patched\pgfpointintersectionoflines
252 \fi

```

## 7 Utilities

Escaping \ifs.

```

253 \long\def\@escapeif#1#2\fi{\fi#1}
254 \long\def\@escapeifif#1#2\fi#3\fi{\fi{\fi#1}}

```

A factory for creating \...loop... macros.

```

255 \def\newloop#1{%
256 \count@=\escapechar
257 \escapechar=-1
258 \expandafter\newloop@parse@loopname\string#1\newloop@end
259 \escapechar=\count@
260 }%
261 {\lccode'7='1 \lccode'8='o \lccode'9='p
262 \lowercase{\gdef\newloop@parse@loopname#17889#2\newloop@end{%
263 \edef\newloop@marshal{%
264 \noexpand\csdef{#1loop#2}####1\expandafter\noexpand\csname #1repeat#2\endcsname{%
265 \noexpand\csdef{#1iterate#2}{####1\relax\noexpand\expandafter\expandafter\noexpand\csname#1iterate#2\endcsname
266 \expandafter\noexpand\csname#1iterate#2\endcsname
267 \let\expandafter\noexpand\csname#1iterate#2\endcsname\relax
268 }%
269 }%
270 \newloop@marshal
271 }%
272 }%
273 }%

```

Additional loops (for embedding).

```

274 \newloop\forest@loop
275 \newloop\forest@loopa
276 \newloop\forest@loopb
277 \newloop\forest@loopc
278 \newloop\forest@sort@loop

```



```

332 \def\forest@cotu@nospaceB{%
333   \ifcat\forest@cotu@nextchar a%
334     \let\forest@cotu@next\forest@cotu@have@alphanum
335   \else
336     \ifcat\forest@cotu@nextchar 0%
337       \let\forest@cotu@next\forest@cotu@have@alphanum
338     \else
339       \let\forest@cotu@next\forest@cotu@have@other
340     \fi
341   \fi
342   \forest@cotu@next
343 }
344 \def\forest@cotu@have@alphanum#1{%
345   \appto\forest@cotu@result{#1}%
346   \forest@cotu
347 }
348 \def\forest@cotu@have@other#1{%
349   \appto\forest@cotu@result{_}%
350   \forest@cotu
351 }

```

Additional list macros.

```

352 \def\forest@listedel#1#2{% #1 = list, #2 = item
353   \edef\forest@marshal{\noexpand\forest@listdel\noexpand#1{#2}}%
354   \forest@marshal
355 }
356 \def\forest@listcsdel#1#2{%
357   \expandafter\forest@listdel\csname #1\endcsname{#2}%
358 }
359 \def\forest@listcsedel#1#2{%
360   \expandafter\forest@listedel\csname #1\endcsname{#2}%
361 }
362 \edef\forest@restorelistsepcatcode{\noexpand\catcode'\the\catcode'\relax}%
363 \catcode'\|=3
364 \gdef\forest@listdel#1#2{%
365   \def\forest@listedel@A##1|#2|##2\forest@END{%
366     \forest@listedel@B##1|##2\forest@END%|
367   }%
368   \def\forest@listedel@B|#1|\forest@END{%|
369     \def#1{##1}%
370   }%
371   \expandafter\forest@listedel@A\expandafter|#1|\forest@END%|
372 }
373 \forest@restorelistsepcatcode

```

Strip (the first level of) braces from all the tokens in the argument.

```

374 \def\forest@strip@braces#1{%
375   \forest@strip@braces@A#1\forest@strip@braces@preend\forest@strip@braces@end
376 }
377 \def\forest@strip@braces@A#1#2\forest@strip@braces@end{%
378   #1\ifx\forest@strip@braces@preend#2\else\@escapeif{\forest@strip@braces@A#2\forest@strip@braces@end}\fi
379 }

```

## 7.1 Sorting

Macro `\forest@sort` is the user interface to sorting.

The user should prepare the data in an arbitrarily encoded array,<sup>18</sup> and provide the sorting macro (given in #1) and the array let macro (given in #2): these are the only ways in which sorting algo-

<sup>18</sup>In forest, arrays are encoded as families of macros. An array-macro name consists of the (optional, but recommended) prefix, the index, and the (optional) suffix (e.g. `\forest@42x`). Prefix establishes the “namespace”, while using more than one suffix simulates an array of named tuples. The length of the array is stored in macro `\<prefix>n`.

gorithms access the data. Both user-given macros should take two parameters, which expand to array indices. The comparison macro should compare the given array items and call `\forest@sort@cmp@gt`, `\forest@sort@cmp@lt` or `\forest@sort@cmp@eq` to signal that the first item is greater than, less than, or equal to the second item. The let macro should “copy” the contents of the second item onto the first item.

The sorting direction is be given in #3: it can one of `\forest@sort@ascending` and `\forest@sort@descending`. #4 and #5 must expand to the lower and upper (both inclusive) indices of the array to be sorted.

`\forest@sort` is just a wrapper for the central sorting macro `\forest@@sort`, storing the comparison macro, the array let macro and the direction. The central sorting macro and the algorithm-specific macros take only two arguments: the array bounds.

```
380 \def\forest@sort#1#2#3#4#5{%
381   \let\forest@sort@cmp#1\relax
382   \let\forest@sort@let#2\relax
383   \let\forest@sort@direction#3\relax
384   \forest@@sort{#4}{#5}%
385 }
```

The central sorting macro. Here it is decided which sorting algorithm will be used: for arrays at least `\forest@quicksort@minarraylength` long, quicksort is used; otherwise, insertion sort.

```
386 \def\forest@quicksort@minarraylength{10000}
387 \def\forest@@sort#1#2{%
388   \ifnum#1<#2\relax\@escapeif{%
389     \forest@sort@m=#2
390     \advance\forest@sort@m -#1
391     \ifnum\forest@sort@m>\forest@quicksort@minarraylength\relax\@escapeif{%
392       \forest@quicksort{#1}{#2}%
393     }\else\@escapeif{%
394       \forest@insertionsort{#1}{#2}%
395     }\fi
396   }\fi
397 }
```

Various counters and macros needed by the sorting algorithms.

```
398 \newcount\forest@sort@m\newcount\forest@sort@k\newcount\forest@sort@p
399 \def\forest@sort@ascending{>}
400 \def\forest@sort@descending{<}
401 \def\forest@sort@cmp{%
402   \PackageError{sort}{You must define forest@sort@cmp function before calling
403     sort}{The macro must take two arguments, indices of the array
404     elements to be compared, and return '=' if the elements are equal
405     and '>'/<' if the first is greater /less than the secong element.}%
406 }
407 \def\forest@sort@cmp@gt{\def\forest@sort@cmp@result{>}}
408 \def\forest@sort@cmp@lt{\def\forest@sort@cmp@result{<}}
409 \def\forest@sort@cmp@eq{\def\forest@sort@cmp@result{=}}
410 \def\forest@sort@let{%
411   \PackageError{sort}{You must define forest@sort@let function before calling
412     sort}{The macro must take two arguments, indices of the array:
413     element 2 must be copied onto element 1.}%
414 }
```

Quick sort macro (adapted from [laansort](#)).

```
415 \def\forest@quicksort#1#2{%
  Compute the index of the middle element (\forest@sort@m).
416   \forest@sort@m=#2
417   \advance\forest@sort@m -#1
418   \ifodd\forest@sort@m\relax\advance\forest@sort@m1 \fi
419   \divide\forest@sort@m 2
420   \advance\forest@sort@m #1
```

The pivot element is the median of the first, the middle and the last element.

```

421 \forest@sort@cmp{#1}{#2}%
422 \if\forest@sort@cmp@result=%
423   \forest@sort@p=#1
424 \else
425   \if\forest@sort@cmp@result>%
426     \forest@sort@p=#1\relax
427   \else
428     \forest@sort@p=#2\relax
429   \fi
430   \forest@sort@cmp{\the\forest@sort@p}{\the\forest@sort@m}%
431   \if\forest@sort@cmp@result<%
432     \else
433       \forest@sort@p=\the\forest@sort@m
434   \fi
435 \fi

```

Exchange the pivot and the first element.

```

436 \forest@sort@xch{#1}{\the\forest@sort@p}%

```

Counter \forest@sort@m will hold the final location of the pivot element.

```

437 \forest@sort@m=#1\relax

```

Loop through the list.

```

438 \forest@sort@k=#1\relax
439 \forest@sort@loop
440 \ifnum\forest@sort@k<#2\relax
441   \advance\forest@sort@k 1

```

Compare the pivot and the current element.

```

442 \forest@sort@cmp{#1}{\the\forest@sort@k}%

```

If the current element is smaller (ascending) or greater (descending) than the pivot element, move it into the first part of the list, and adjust the final location of the pivot.

```

443 \ifx\forest@sort@direction\forest@sort@cmp@result
444   \advance\forest@sort@m 1
445   \forest@sort@xch{\the\forest@sort@m}{\the\forest@sort@k}
446 \fi
447 \forest@sort@repeat

```

Move the pivot element into its final position.

```

448 \forest@sort@xch{#1}{\the\forest@sort@m}%

```

Recursively call sort on the two parts of the list: elements before the pivot are smaller (ascending order) / greater (descending order) than the pivot; elements after the pivot are greater (ascending order) / smaller (descending order) than the pivot.

```

449 \forest@sort@k=\forest@sort@m
450 \advance\forest@sort@k -1
451 \advance\forest@sort@m 1
452 \edef\forest@sort@marshal{%
453   \noexpand\forest@@sort{#1}{\the\forest@sort@k}%
454   \noexpand\forest@@sort{\the\forest@sort@m}{#2}%
455 }%
456 \forest@sort@marshal
457 }
458 % We defines the item-exchange macro in terms of the (user-provided)
459 % array let macro.
460 %   \begin{macrocode}
461 \def\forest@sort@xch#1#2{%
462   \forest@sort@let{aux}{#1}%
463   \forest@sort@let{#1}{#2}%
464   \forest@sort@let{#2}{aux}%
465 }

```

Insertion sort.

```

466 \def\forest@insertionsort#1#2{%
467   \forest@sort@m=#1
468   \edef\forest@insertionsort@low{#1}%
469   \forest@sort@loopA
470   \ifnum\forest@sort@m<#2
471     \advance\forest@sort@m 1
472     \forest@insertionsort@Qbody
473   \forest@sort@repeatA
474 }
475 \newif\ifforest@insertionsort@loop
476 \def\forest@insertionsort@Qbody{%
477   \forest@sort@let{aux}{\the\forest@sort@m}%
478   \forest@sort@k\forest@sort@m
479   \advance\forest@sort@k -1
480   \forest@insertionsort@looptrue
481   \forest@sort@loop
482   \ifforest@insertionsort@loop
483     \forest@insertionsort@Qbody
484   \forest@sort@repeat
485   \advance\forest@sort@k 1
486   \forest@sort@let{\the\forest@sort@k}{aux}%
487 }
488 \def\forest@insertionsort@Qbody{%
489   \forest@sort@cmp{\the\forest@sort@k}{aux}%
490   \ifx\forest@sort@direction\forest@sort@cmp@result\relax
491     \forest@sort@p=\forest@sort@k
492     \advance\forest@sort@p 1
493     \forest@sort@let{\the\forest@sort@p}{\the\forest@sort@k}%
494     \advance\forest@sort@k -1
495     \ifnum\forest@sort@k<\forest@insertionsort@low\relax
496       \forest@insertionsort@loopfalse
497     \fi
498   \else
499     \forest@insertionsort@loopfalse
500   \fi
501 }

```

Below, several helpers for writing comparison macros are provided. They take two (pairs of) control sequence names and compare their contents.

Compare numbers.

```

502 \def\forest@sort@cmpnumcs#1#2{%
503   \ifnum\csname#1\endcsname>\csname#2\endcsname\relax
504     \forest@sort@cmp@gt
505   \else
506     \ifnum\csname#1\endcsname<\csname#2\endcsname\relax
507       \forest@sort@cmp@lt
508     \else
509       \forest@sort@cmp@eq
510     \fi
511   \fi
512 }

```

Compare dimensions.

```

513 \def\forest@sort@cmpdimcs#1#2{%
514   \ifdim\csname#1\endcsname>\csname#2\endcsname\relax
515     \forest@sort@cmp@gt
516   \else
517     \ifdim\csname#1\endcsname<\csname#2\endcsname\relax
518       \forest@sort@cmp@lt
519     \else

```

```

520     \forest@sort@cmp@eq
521   \fi
522 \fi
523 }

```

Compare points (pairs of dimension) (#1,#2) and (#3,#4).

```

524 \def\forest@sort@cmptwodimcs#1#2#3#4{%
525   \ifdim\csname#1\endcsname>\csname#3\endcsname\relax
526     \forest@sort@cmp@gt
527   \else
528     \ifdim\csname#1\endcsname<\csname#3\endcsname\relax
529       \forest@sort@cmp@lt
530     \else
531       \ifdim\csname#2\endcsname>\csname#4\endcsname\relax
532         \forest@sort@cmp@gt
533       \else
534         \ifdim\csname#2\endcsname<\csname#4\endcsname\relax
535           \forest@sort@cmp@lt
536         \else
537           \forest@sort@cmp@eq
538         \fi
539       \fi
540     \fi
541   \fi
542 }

```

The following macro reverses an array. The arguments: #1 is the array let macro; #2 is the start index (inclusive), and #3 is the end index (exclusive).

```

543 \def\forest@reversearray#1#2#3{%
544   \let\forest@sort@let#1%
545   \c@pgf@countc=#2
546   \c@pgf@countd=#3
547   \advance\c@pgf@countd -1
548   \forest@loopa
549   \ifnum\c@pgf@countc<\c@pgf@countd\relax
550     \forest@sort@xch{\the\c@pgf@countc}{\the\c@pgf@countd}%
551     \advance\c@pgf@countc 1
552     \advance\c@pgf@countd -1
553   \forest@repeata
554 }

```

## 8 The bracket representation parser

### 8.1 The user interface macros

Settings.

```

555 \def\bracketset#1{\pgfqkeys{/bracket}{#1}}%
556 \bracketset{%
557   /bracket/.is family,
558   /handlers/.let/.style={\pgfkeyscurrentpath/.code={\let#1##1}},
559   opening bracket/.let=\bracket@openingBracket,
560   closing bracket/.let=\bracket@closingBracket,
561   action character/.let=\bracket@actionCharacter,
562   opening bracket=[,
563   closing bracket=],
564   action character,
565   new node/.code n args={3}{% #1=preamble, #2=node spec, #3=cs receiving the id
566     \forest@node@new#3%
567     \forest@set{#3}{given options}{content'=#2}%
568     \ifblank{#1}{}{%

```



```

569     \forestOpretto{#3}{given options}{#1,}%
570   }%
571 },
572 set afterthought/.code 2 args={% #1=node id, #2=afterthought
573   \ifblank{#2}{\forestOappto{#1}{given options}{,afterthought={#2}}}%
574 }
575 }

```

`\bracketParse` is the macro that should be called to parse a balanced bracket representation. It takes five parameters: `#1` is the code that will be run after parsing the bracket; `#2` is a control sequence that will receive the id of the root of the created tree structure. (The bracket representation should follow (after optional spaces), but is is not a formal parameter of the macro.)

```

576 \newtoks\bracket@content
577 \newtoks\bracket@afterthought
578 \def\bracketParse#1#2={%
579   \def\bracketEndParsingHook{#1}%
580   \def\bracket@saveRootNodeTo{#2}%

```

Content and afterthought will be appended to these macros. (The `\bracket@afterthought` toks register is abused for storing the preamble as well — that’s ok, the preamble comes before any afterthoughts.)

```

581   \bracket@content={}%
582   \bracket@afterthought={}%

```

The parser can be in three states: in content (0), in afterthought (1), or starting (2). While in the content/afterthought state, the parser appends all non-control tokens to the content/afterthought macro.

```

583   \let\bracket@state\bracket@state@starting
584   \bracket@ignoreSPACEtrue

```

By default, don’t expand anything.

```

585   \bracket@expandtokensfalse

```

We initialize several control sequences that are used to store some nodes while parsing.

```

586   \def\bracket@parentNode{0}%
587   \def\bracket@rootNode{0}%
588   \def\bracket@newNode{0}%
589   \def\bracket@afterthoughtNode{0}%

```

Finally, we start the parser.

```

590   \bracket@Parse
591 }

```

The other macro that an end user (actually a power user) can use, is actually just a synonym for `\bracket@Parse`. It should be used to resume parsing when the action code has finished its work.

```

592 \def\bracketResume{\bracket@Parse}%

```

## 8.2 Parsing

We first check if the next token is a space. Spaces need special treatment because they are eaten by both the `\romannumeral` trick and T<sub>E</sub>Xs (undelimited) argument parsing algorithm. If a space is found, remember that, eat it up, and restart the parsing.

```

593 \def\bracket@Parse{%
594   \futurelet\bracket@next@token\bracket@Parse@checkForSpace
595 }
596 \def\bracket@Parse@checkForSpace{%
597   \expandafter\ifx\space\bracket@next@token\@escapeif{%
598     \ifbracket@ignoreSPACES\else
599       \bracket@haveSPACEtrue
600     \fi
601     \expandafter\bracket@Parse\romannumeral-‘0%
602   }\else\@escapeif{%
603     \bracket@Parse@maybeexpand
604   }\fi
605 }

```

We either fully expand the next token (using a popular T<sub>E</sub>Xnical trick ...) or don't expand it at all, depending on the state of `\ifbracket@expandtokens`.

```

606 \newif\ifbracket@expandtokens
607 \def\bracket@Parse@maybeexpand{%
608   \ifbracket@expandtokens\@escapeif{%
609     \expandafter\bracket@Parse@peekAhead\romannumeral-‘0%
610   }\else\@escapeif{%
611     \bracket@Parse@peekAhead
612   }\fi
613 }

```

We then look ahead to see what's coming.

```

614 \def\bracket@Parse@peekAhead{%
615   \futurelet\bracket@next@token\bracket@Parse@checkForTeXGroup
616 }

```

If the next token is a begin-group token, we append the whole group to the content or afterthought macro, depending on the state.

```

617 \def\bracket@Parse@checkForTeXGroup{%
618   \ifx\bracket@next@token\bgroup%
619     \@escapeif{\bracket@Parse@appendGroup}%
620   \else
621     \@escapeif{\bracket@Parse@token}%
622   \fi
623 }

```

This is easy: if a control token is found, run the appropriate macro; otherwise, append the token to the content or afterthought macro, depending on the state.

```

624 \long\def\bracket@Parse@token#1{%
625   \ifx#1\bracket@openingBracket
626     \@escapeif{\bracket@Parse@openingBracketFound}%
627   \else
628     \@escapeif{%
629       \ifx#1\bracket@closingBracket
630         \@escapeif{\bracket@Parse@closingBracketFound}%
631       \else
632         \@escapeif{%
633           \ifx#1\bracket@actionCharacter
634             \@escapeif{\futurelet\bracket@next@token\bracket@Parse@actionCharacterFound}%
635           \else
636             \@escapeif{\bracket@Parse@appendToken#1}%
637           \fi
638         }%
639       \fi
640     }%
641   \fi
642 }

```

Append the token or group to the content or afterthought macro. If a space was found previously, append it as well.

```

643 \newif\ifbracket@haveSpace
644 \newif\ifbracket@ignorespaces
645 \def\bracket@Parse@appendSpace{%
646   \ifbracket@haveSpace
647     \ifcase\bracket@state\relax
648       \eapptotoks\bracket@content\space
649     \or
650       \eapptotoks\bracket@afterthought\space
651     \or
652       \eapptotoks\bracket@afterthought\space
653     \fi
654   \bracket@haveSpacefalse

```

```

655 \fi
656 }
657 \long\def\bracket@Parse@appendToken#1{%
658 \bracket@Parse@appendSpace
659 \ifcase\bracket@state\relax
660 \lapptotoks\bracket@content{#1}%
661 \or
662 \lapptotoks\bracket@afterthought{#1}%
663 \or
664 \lapptotoks\bracket@afterthought{#1}%
665 \fi
666 \bracket@ignorespacesfalse
667 \bracket@Parse
668 }
669 \def\bracket@Parse@appendGroup#1{%
670 \ifcase\bracket@state\relax
671 \apptotoks\bracket@content{{#1}}%
672 \or
673 \apptotoks\bracket@afterthought{{#1}}%
674 \or
675 \apptotoks\bracket@afterthought{{#1}}%
676 \fi
677 \bracket@ignorespacesfalse
678 \bracket@Parse
679 }

```

Declare states.

```

680 \def\bracket@state@inContent{0}
681 \def\bracket@state@inAfterthought{1}
682 \def\bracket@state@starting{2}

```

Welcome to the jungle. In the following two macros, new nodes are created, content and afterthought are sent to them, parents and states are changed... Altogether, we distinguish six cases, as shown below: in the schemas, we have just crossed the symbol after the dots. (In all cases, we reset the `\if` for spaces.)

```

683 \def\bracket@Parse@openingBracketFound{%
684 \bracket@haveSpacefalse
685 \ifcase\bracket@state\relax% in content [ ... [

```

[...[: we have just finished gathering the content and are about to begin gathering the content of another node. We create a new node (and put the content (...) into it). Then, if there is a parent node, we append the new node to the list of its children. Next, since we have just crossed an opening bracket, we declare the newly created node to be the parent of the coming node. The state does not change. Finally, we continue parsing.

```

686 \escapeif{%
687 \bracket@createNode
688 \ifnum\bracket@parentNode=0 \else
689 \forest@node@Append{\bracket@parentNode}{\bracket@newNode}%
690 \fi
691 \let\bracket@parentNode\bracket@newNode
692 \bracket@Parse
693 }%
694 \or % in afterthought ] ... [

```

]...[: we have just finished gathering the afterthought and are about to begin gathering the content of another node. We add the afterthought (...) to the “afterthought node” and change into the content state. The parent does not change. Finally, we continue parsing.

```

695 \escapeif{%
696 \bracket@addAfterthought
697 \let\bracket@state\bracket@state@inContent
698 \bracket@Parse
699 }%
700 \else % starting

```

{start}...[: we have just started. Nothing to do yet (we couldn't have collected any content yet), just get into the content state and continue parsing.

```

701 \escapeif{%
702   \let\bracket@state\bracket@state@inContent
703   \bracket@Parse
704 }%
705 \fi
706 }
707 \def\bracket@Parse@closingBracketFound{%
708   \bracket@haveSpacefalse
709   \ifcase\bracket@state\relax % in content [ ... ]

```

[...]: we have just finished gathering the content of a node and are about to begin gathering its afterthought. We create a new node (and put the content (...) into it). If there is no parent node, we're done with parsing. Otherwise, we set the newly created node to be the "afterthought node", i.e. the node that will receive the next afterthought, change into the afterthought mode, and continue parsing.

```

710 \escapeif{%
711   \bracket@createNode
712   \ifnum\bracket@parentNode=0
713     \escapeif\bracket@endParsingHook
714   \else
715     \escapeif{%
716       \let\bracket@afterthoughtNode\bracket@newNode
717       \let\bracket@state\bracket@state@inAfterthought
718       \forest@node@Append{\bracket@parentNode}{\bracket@newNode}%
719       \bracket@Parse
720     }%
721   \fi
722 }%
723 \or % in afterthought ] ... ]

```

]...]: we have finished gathering an afterthought of some node and will begin gathering the afterthought of its parent. We first add the afterthought to the afterthought node and set the current parent to be the next afterthought node. We change the parent to the current parent's parent and check if that node is null. If it is, we're done with parsing (ignore the trailing spaces), otherwise we continue.

```

724 \escapeif{%
725   \bracket@addAfterthought
726   \let\bracket@afterthoughtNode\bracket@parentNode
727   \edef\bracket@parentNode{\forestOve{\bracket@parentNode}{@parent}}%
728   \ifnum\bracket@parentNode=0
729     \expandafter\bracket@endParsingHook
730   \else
731     \expandafter\bracket@Parse
732   \fi
733 }%
734 \else % starting

```

{start}...[: something's obviously wrong with the input here...

```

735 \PackageError{forest}{You're attempting to start a bracket representation
736   with a closing bracket}{}%
737 \fi
738 }

```

The action character code. What happens is determined by the next token.

```

739 \def\bracket@Parse@actionCharacterFound{%

```

If a braced expression follows, its contents will be fully expanded.

```

740 \ifx\bracket@next@token\bgroup\escapeif{%
741   \bracket@Parse@action@expandgroup
742 } \else \escapeif{%
743   \bracket@Parse@action@notagroup
744 } \fi

```

```

745 }
746 \def\bracket@Parse@action@expandgroup#1{%
747   \edef\bracket@Parse@action@expandgroup@macro{#1}%
748   \expandafter\bracket@Parse\bracket@Parse@action@expandgroup@macro
749 }
750 \let\bracket@action@fullyexpandCharacter+
751 \let\bracket@action@dontexpandCharacter-
752 \let\bracket@action@executeCharacter!
753 \def\bracket@Parse@action@notagroup#1{%
  If + follows, tokens will be fully expanded from this point on.
754   \ifx#1\bracket@action@fullyexpandCharacter\@escapeif{%
755     \bracket@expandtokenstrue\bracket@Parse
756   }\else\@escapeif{%
  If - follows, tokens will not be expanded from this point on. (This is the default behaviour.)
757     \ifx#1\bracket@action@dontexpandCharacter\@escapeif{%
758       \bracket@expandtokensfalse\bracket@Parse
759     }\else\@escapeif{%
  Inhibit expansion of the next token.
760       \ifx#10\@escapeif{%
761         \bracket@Parse@appendToken
762       }\else\@escapeif{%
  If another action characted follows, we yield the control. The user is expected to resume the parser
  manually, using \bracketResume.
763       \ifx#1\bracket@action@Character
764       \else\@escapeif{%
  Anything else will be expanded once.
765         \expandafter\bracket@Parse#1%
766       }\fi
767     }\fi
768   }\fi
769 }\fi
770 }

```

### 8.3 The tree-structure interface

This macro creates a new node and sets its content (and preamble, if it's a root node). Bracket user must define a 3-arg key `/bracket/new node=<preamble><node specification><node cs>`. User's key must define `<node cs>` to be a macro holding the node's id.

```

771 \def\bracket@createNode{%
772   \ifnum\bracket@rootNode=0
773     % root node
774     \bracketset{new node/.expanded=%
775       {\the\bracket@afterthought}%
776       {\the\bracket@content}%
777       \noexpand\bracket@newNode
778     }%
779     \bracket@afterthought={}%
780     \let\bracket@rootNode\bracket@newNode
781     \expandafter\let\bracket@saveRootNodeTo\bracket@newNode
782   \else
783     % other nodes
784     \bracketset{new node/.expanded=%
785       {}%
786       {\the\bracket@content}%
787       \noexpand\bracket@newNode
788     }%
789   \fi

```

```

790 \bracket@content={}%
791 }

```

This macro sets the afterthought. Bracket user must define a 2-arg key `/bracket/set afterthought=⟨node id⟩⟨afterthought⟩`.

```

792 \def\bracket@addAfterthought{%
793 \bracketset{%
794   set afterthought/.expanded={\bracket@afterthoughtNode}{\the\bracket@afterthought}%
795 }%
796 \bracket@afterthought={}%
797 }

```

## 9 Nodes

Nodes have numeric ids. The node option values of node  $n$  are saved in the `\pgfkeys` tree in path `/forest/@node/n`.

### 9.1 Option setting and retrieval

Macros for retrieving/setting node options of the current node.

```

798 % full expansion expands precisely to the value
799 \def\forestov#1{\expandafter\expandafter\expandafter\expandonce
800 \pgfkeysvalueof{/forest/@node/\forest@cn/#1}}
801 % full expansion expands all the way
802 \def\forestove#1{\pgfkeysvalueof{/forest/@node/\forest@cn/#1}}
803 % full expansion expands to the cs holding the value
804 \def\forestom#1{\expandafter\expandonce\expandafter{\pgfkeysvalueof{/forest/@node/\forest@cn/#1}}\def\forest
805 \def\forestogget#1#2{\pgfkeysgetvalue{/forest/@node/\forest@cn/#1}{#2}}
806 \def\forestolet#1#2{\pgfkeyslet{/forest/@node/\forest@cn/#1}{#2}}
807 \def\forestoset#1#2{\pgfkeyssetvalue{/forest/@node/\forest@cn/#1}{#2}}
808 \def\forestoeset#1#2{%
809 \edef\forest@option@temp{%
810 \noexpand\pgfkeyssetvalue{/forest/@node/\forest@cn/#1}{#2}%
811 }\forest@option@temp
812 }
813 \def\forestoaappto#1#2{%
814 \forestoeset{#1}{\forestov{#1}\unexpanded{#2}}%
815 }
816 \def\forestoiifdefined#1#2#3{%
817 \pgfkeysifdefined{/forest/@node/\forest@cn/#1}{#2}{#3}%
818 }

```

User macros for retrieving node options of the current node.

```

819 \let\forestoption\forestov
820 \let\foresteooption\forestove

```

Macros for retrieving node options of a node given by its id.

```

821 \def\forestOv#1#2{\expandafter\expandafter\expandafter\expandonce
822 \pgfkeysvalueof{/forest/@node/#1/#2}}
823 \def\forestOve#1#2{\pgfkeysvalueof{/forest/@node/#1/#2}}
824 % full expansion expands to the cs holding the value
825 \def\forestOm#1#2{\expandafter\expandonce\expandafter{\pgfkeysvalueof{/forest/@node/#1/#2}}}
826 \def\forestOget#1#2#3{\pgfkeysgetvalue{/forest/@node/#1/#2}{#3}}
827 \def\forestOget#1#2#3{\pgfkeysgetvalue{/forest/@node/#1/#2}{#3}}
828 \def\forestOlet#1#2#3{\pgfkeyslet{/forest/@node/#1/#2}{#3}}
829 \def\forestOset#1#2#3{\pgfkeyssetvalue{/forest/@node/#1/#2}{#3}}
830 \def\forestOset#1#2#3{%
831 \edef\forestoption@temp{%
832 \noexpand\pgfkeyssetvalue{/forest/@node/#1/#2}{#3}%
833 }\forestoption@temp

```

```

834 }
835 \def\forest0appto#1#2#3{%
836   \forest0eset{#1}{#2}{\forest0v{#1}{#2}\unexpanded{#3}}%
837 }
838 \def\forest0eappto#1#2#3{%
839   \forest0eset{#1}{#2}{\forest0v{#1}{#2}#3}%
840 }
841 \def\forest0preto#1#2#3{%
842   \forest0eset{#1}{#2}{\unexpanded{#3}\forest0v{#1}{#2}}%
843 }
844 \def\forest0epreto#1#2#3{%
845   \forest0eset{#1}{#2}{#3\forest0v{#1}{#2}}%
846 }
847 \def\forest0ifdefined#1#2#3#4{%
848   \pgfkeysifdefined{/forest/@node/#1/#2}{#3}{#4}%
849 }
850 \def\forest0let0#1#2#3#4{% option #2 of node #1 <-- option #4 of node #3
851   \forest0get{#3}{#4}\forestoption@temp
852   \forest0let{#1}{#2}\forestoption@temp}
853 \def\forest0leto#1#2#3{%
854   \forest0get{#3}\forestoption@temp
855   \forest0let{#1}{#2}\forestoption@temp}
856 \def\forest0tolet0#1#2#3{%
857   \forest0get{#2}{#3}\forestoption@temp
858   \forest0let{#1}\forestoption@temp}
859 \def\forest0toleto#1#2{%
860   \forest0get{#2}\forestoption@temp
861   \forest0let{#1}\forestoption@temp}

Node initialization. Node option declarations append to \forest@node@init.

862 \def\forest@node@init{%
863   \forestoset{@parent}{0}%
864   \forestoset{@previous}{0}% previous sibling
865   \forestoset{@next}{0}% next sibling
866   \forestoset{@first}{0}% primary child
867   \forestoset{@last}{0}% last child
868 }
869 \def\forest0init#1{%
870   \pgfkeysgetvalue{/forest/#1}\forest0init@temp
871   \forest0let{#1}\forest0init@temp
872 }
873 \newcount\forest@node@maxid
874 \def\forest@node@new#1{% #1 = cs receiving the new node id
875   \advance\forest@node@maxid1
876   \forest@fornode{\the\forest@node@maxid}{%
877     \forest@node@init
878     \forest@node@setname{node@\forest@cn}%
879     \forest@initializefromstandardnode
880     \edef#1{\forest@cn}%
881   }%
882 }
883 \let\forest0init@orig\forest0init
884 \def\forest@node@copy#1#2{% #1=from node id, cs receiving the new node id
885   \advance\forest@node@maxid1
886   \def\forest0init##1{\forest0let0{##1}{#1}{##1}}%
887   \forest@fornode{\the\forest@node@maxid}{%
888     \forest@node@init
889     \forest@node@setname{\forest@copy@name@template{\forest0ve{#1}{name}}}%
890     \edef#2{\forest@cn}%
891   }%
892   \let\forest0init\forest0init@orig

```

```

893 }
894 \forestset{
895   copy name template/.code={\def\forest@copy@name@template##1{#1}},
896   copy name template/.default={node@the\forest@node@maxid},
897   copy name template
898 }
899 \def\forest@tree@copy#1#2{% #1=from node id, #2=cs receiving the new node id
900   \forest@node@copy{#1}\forest@node@copy@temp@id
901   \forest@for@node{\forest@node@copy@temp@id}{%
902     \expandafter\forest@tree@copy\expandafter{\forest@node@copy@temp@id}{#1}%
903     \edef#2{\forest@cn}%
904   }%
905 }
906 \def\forest@tree@copy@#1#2{%
907   \forest@node@Foreachchild{#2}{%
908     \expandafter\forest@tree@copy\expandafter{\forest@cn}\forest@node@copy@temp@childid
909     \forest@node@Append{#1}{\forest@node@copy@temp@childid}%
910   }%
911 }

```

Macro `\forest@cn` holds the current node id (a number). Node 0 is a special “null” node which is used to signal the absence of a node.

```

912 \def\forest@cn{0}
913 \forest@node@init

```

## 9.2 Tree structure

Node insertion/removal.

For the lowercase variants, `\forest@cn` is the parent/removed node. For the uppercase variants, `#1` is the parent/removed node. For efficiency, the public macros all expand the arguments before calling the internal macros.

```

914 \def\forest@node@append#1{\expandtwonumberargs\forest@node@Append{\forest@cn}{#1}}
915 \def\forest@node@prepend#1{\expandtwonumberargs\forest@node@Insertafter{\forest@cn}{#1}{0}}
916 \def\forest@node@insertafter#1#2{%
917   \expandthreenumberargs\forest@node@Insertafter{\forest@cn}{#1}{#2}}
918 \def\forest@node@insertbefore#1#2{%
919   \expandthreenumberargs\forest@node@Insertafter{\forest@cn}{#1}{\forestOve{#2}{@previous}}}%
920 }
921 \def\forest@node@remove{\expandnumberarg\forest@node@Remove{\forest@cn}}
922 \def\forest@node@Append#1#2{\expandtwonumberargs\forest@node@Append@{#1}{#2}}
923 \def\forest@node@Prepend#1#2{\expandtwonumberargs\forest@node@Insertafter{#1}{#2}{0}}
924 \def\forest@node@Insertafter#1#2#3{% #2 is inserted after #3
925   \expandthreenumberargs\forest@node@Insertafter@{#1}{#2}{#3}%
926 }
927 \def\forest@node@Insertbefore#1#2#3{% #2 is inserted before #3
928   \expandthreenumberargs\forest@node@Insertafter{#1}{#2}{\forestOve{#3}{@previous}}}%
929 }
930 \def\forest@node@Remove#1{\expandnumberarg\forest@node@Remove@{#1}}
931 \def\forest@node@Insertafter@#1#2#3{%
932   \ifnum\forestOve{#2}{@parent}=0
933   \else
934     \PackageError{forest}{Insertafter(#1,#2,#3):
935       node #2 already has a parent (\forestOve{#2}{@parent})}{}%
936   \fi
937   \ifnum#3=0
938   \else
939     \ifnum#1=\forestOve{#3}{@parent}
940     \else
941       \PackageError{forest}{Insertafter(#1,#2,#3): node #1 is not the parent of the
942         intended sibling #3 (with parent \forestOve{#3}{@parent})}{}%

```



```

943 \fi
944 \fi
945 \forestOset{#2}{@parent}{#1}%
946 \forestOset{#2}{@previous}{#3}%
947 \ifnum#3=0
948 \forestOget{#1}{@first}\forest@node@temp
949 \forestOset{#1}{@first}{#2}%
950 \else
951 \forestOget{#3}{@next}\forest@node@temp
952 \forestOset{#3}{@next}{#2}%
953 \fi
954 \forestOset{#2}{@next}{\forest@node@temp}%
955 \ifnum\forest@node@temp=0
956 \forestOset{#1}{@last}{#2}%
957 \else
958 \forestOset{\forest@node@temp}{@previous}{#2}%
959 \fi
960 }
961 \def\forest@node@Append@#1#2{%
962 \ifnum\forestOve{#2}{@parent}=0
963 \else
964 \PackageError{forest}{Append(#1,#2):
965 node #2 already has a parent (\forestOve{#2}{@parent})}{}%
966 \fi
967 \forestOset{#2}{@parent}{#1}%
968 \forestOget{#1}{@last}\forest@node@temp
969 \forestOset{#1}{@last}{#2}%
970 \forestOset{#2}{@previous}{\forest@node@temp}%
971 \ifnum\forest@node@temp=0
972 \forestOset{#1}{@first}{#2}%
973 \else
974 \forestOset{\forest@node@temp}{@next}{#2}%
975 \fi
976 }
977 \def\forest@node@Remove@#1{%
978 \forestOget{#1}{@parent}\forest@node@temp@parent
979 \ifnum\forest@node@temp@parent=0
980 \else
981 \forestOget{#1}{@previous}\forest@node@temp@previous
982 \forestOget{#1}{@next}\forest@node@temp@next
983 \ifnum\forest@node@temp@previous=0
984 \forestOset{\forest@node@temp@parent}{@first}{\forest@node@temp@next}%
985 \else
986 \forestOset{\forest@node@temp@previous}{@next}{\forest@node@temp@next}%
987 \fi
988 \ifnum\forest@node@temp@next=0
989 \forestOset{\forest@node@temp@parent}{@last}{\forest@node@temp@previous}%
990 \else
991 \forestOset{\forest@node@temp@next}{@previous}{\forest@node@temp@previous}%
992 \fi
993 \forestOset{#1}{@parent}{0}%
994 \forestOset{#1}{@previous}{0}%
995 \forestOset{#1}{@next}{0}%
996 \fi
997 }

```

Looping methods.

```

998 \def\forest@forthis#1{%
999 \edef\forest@node@marshal{\unexpanded{#1}\def\noexpand\forest@cn}%
1000 \expandafter\forest@node@marshal\expandafter{\forest@cn}%
1001 }

```

```

1002 \def\forest@fornode#1#2{%
1003   \edef\forest@node@marshal{\edef\noexpand\forest@cn{#1}\unexpanded{#2}\def\noexpand\forest@cn}%
1004   \expandafter\forest@node@marshal\expandafter{\forest@cn}%
1005 }
1006 \def\forest@fornode@ifexists#1#2{%
1007   \edef\forest@node@temp{#1}%
1008   \ifnum\forest@node@temp=0
1009   \else
1010     \@escapeif{\expandnumberarg\forest@fornode{\forest@node@temp}{#2}}%
1011   \fi
1012 }
1013 \def\forest@node@foreachchild#1{\forest@node@Foreachchild{\forest@cn}{#1}}
1014 \def\forest@node@Foreachchild#1#2{%
1015   \forest@fornode{\forest@ve{#1}{@first}}{\forest@node@@forselfandfollowingsiblings{#2}}%
1016 }
1017 \def\forest@node@@forselfandfollowingsiblings#1{%
1018   \ifnum\forest@cn=0
1019   \else
1020     \forest@forthis{#1}%
1021     \@escapeif{%
1022       \edef\forest@cn{\forest@ve{@next}}%
1023       \forest@node@@forselfandfollowingsiblings{#1}%
1024     }%
1025   \fi
1026 }
1027 \def\forest@node@foreach#1{\forest@node@Foreach{\forest@cn}{#1}}
1028 \def\forest@node@Foreach#1#2{%
1029   \forest@fornode{#1}{\forest@node@@foreach{#2}}%
1030 }
1031 \def\forest@node@@foreach#1{%
1032   \forest@forthis{#1}%
1033   \ifnum\forest@ve{@first}=0
1034   \else\@escapeif{%
1035     \edef\forest@cn{\forest@ve{@first}}%
1036     \forest@node@@forselfandfollowingsiblings{\forest@node@@foreach{#1}}%
1037   }%
1038   \fi
1039 }
1040 \def\forest@node@foreachdescendant#1{\forest@node@Foreachdescendant{\forest@cn}{#1}}
1041 \def\forest@node@Foreachdescendant#1#2{%
1042   \forest@node@Foreachchild{#1}{%
1043     \forest@node@foreach{#2}%
1044   }%
1045 }
1046 \def\forest@node@Compute@numeric@ts@info@#1{%
1047   \forest@node@Foreach{#1}{\forest@node@@compute@numeric@ts@info}%
1048   \ifnum\forest@ve{@parent}=0
1049   \else
1050     \fornode{#1}{\forest@node@@compute@numeric@ts@info@nbar}%
1051   \fi
1052   \forest@node@Foreachdescendant{#1}{\forest@node@@compute@numeric@ts@info@nbar}%
1053 }
1054 \def\forest@node@@compute@numeric@ts@info{%
1055   \forest@set{n children}{0}%
1056   %
1057   \edef\forest@node@temp{\forest@ve{@previous}}%
1058   \ifnum\forest@node@temp=0
1059     \forest@set{n}{1}%
1060   \else

```

Compute  $n$ ,  $n'$ ,  $n$  children and level.

```

1061 \forestoeset{n}{\number\numexpr\forestOve{\forest@node@temp}{n}+1}%
1062 \fi
1063 %
1064 \edef\forest@node@temp{\forestove{@parent}}%
1065 \ifnum\forest@node@temp=0
1066 \forestoset{n}{0}%
1067 \forestoset{n'}{0}%
1068 \forestoset{level}{0}%
1069 \else
1070 \forestOset{\forest@node@temp}{n children}{%
1071 \number\numexpr\forestOve{\forest@node@temp}{n children}+1%
1072 }%
1073 \forestoeset{level}{%
1074 \number\numexpr\forestOve{\forest@node@temp}{level}+1%
1075 }%
1076 \fi
1077 }
1078 \def\forest@node@@compute@numeric@ts@info@nbar{%
1079 \forestoeset{n'}{\number\numexpr\forestOve{\forestove{@parent}}{n children}-\forestove{n}+1}%
1080 }
1081 \def\forest@node@compute@numeric@ts@info#1{%
1082 \expandnumberarg\forest@node@Compute@numeric@ts@info@{\forest@cn}%
1083 }
1084 \def\forest@node@Compute@numeric@ts@info#1{%
1085 \expandnumberarg\forest@node@Compute@numeric@ts@info@{#1}%
1086 }

```

Tree structure queries.

```

1087 \def\forest@node@rootid{%
1088 \expandnumberarg\forest@node@Rootid{\forest@cn}%
1089 }
1090 \def\forest@node@Rootid#1{% #1=node
1091 \ifnum\forestOve{#1}{@parent}=0
1092 #1%
1093 \else
1094 \@escapeif{\expandnumberarg\forest@node@Rootid{\forestOve{#1}{@parent}}}%
1095 \fi
1096 }
1097 \def\forest@node@nthchildid#1{% #1=n
1098 \ifnum#1<1
1099 0%
1100 \else
1101 \expandnumberarg\forest@node@nthchildid@{\number\forestove{@first}}{#1}%
1102 \fi
1103 }
1104 \def\forest@node@nthchildid@#1#2{%
1105 \ifnum#1=0
1106 0%
1107 \else
1108 \ifnum#2>1
1109 \@escapeifif{\expandtwonumberargs
1110 \forest@node@nthchildid@{\forestOve{#1}{@next}}{\numexpr#2-1}}%
1111 \else
1112 #1%
1113 \fi
1114 \fi
1115 }
1116 \def\forest@node@nbarthchildid#1{% #1=n
1117 \expandnumberarg\forest@node@nbarthchildid@{\number\forestove{@last}}{#1}%
1118 }
1119 \def\forest@node@nbarthchildid@#1#2{%

```

```

1120 \ifnum#1=0
1121 0%
1122 \else
1123 \ifnum#2>1
1124 \escapeiff{\expandtwonumberargs
1125 \forest@node@nbarthchildid@{\forestOve{#1}{@previous}}{\numexpr#2-1}}%
1126 \else
1127 #1%
1128 \fi
1129 \fi
1130 }
1131 \def\forest@node@nornbarthchildid#1{%
1132 \ifnum#1>0
1133 \forest@node@nthchildid{#1}%
1134 \else
1135 \ifnum#1<0
1136 \forest@node@nbarthchildid{-#1}%
1137 \else
1138 \forest@node@nornbarthchildid@error
1139 \fi
1140 \fi
1141 }
1142 \def\forest@node@nornbarthchildid@error{%
1143 \PackageError{forest}{In \string\forest@node@nornbarthchildid, n should !=0}{}%
1144 }
1145 \def\forest@node@previousleafid{%
1146 \expandnumberarg\forest@node@Previousleafid{\forest@cn}%
1147 }
1148 \def\forest@node@Previousleafid#1{%
1149 \ifnum\forestOve{#1}{@previous}=0
1150 \escapeiff{\expandnumberarg\forest@node@previousleafid@Goup{#1}}%
1151 \else
1152 \expandnumberarg\forest@node@previousleafid@Gdown{\forestOve{#1}{@previous}}%
1153 \fi
1154 }
1155 \def\forest@node@previousleafid@Goup#1{%
1156 \ifnum\forestOve{#1}{@parent}=0
1157 \PackageError{forest}{get previous leaf: this is the first leaf}{}%
1158 \else
1159 \escapeiff{\expandnumberarg\forest@node@Previousleafid{\forestOve{#1}{@parent}}}%
1160 \fi
1161 }
1162 \def\forest@node@previousleafid@Gdown#1{%
1163 \ifnum\forestOve{#1}{@last}=0
1164 #1%
1165 \else
1166 \escapeiff{\expandnumberarg\forest@node@previousleafid@Gdown{\forestOve{#1}{@last}}}%
1167 \fi
1168 }
1169 \def\forest@node@nextleafid{%
1170 \expandnumberarg\forest@node@Nextleafid{\forest@cn}%
1171 }
1172 \def\forest@node@Nextleafid#1{%
1173 \ifnum\forestOve{#1}{@next}=0
1174 \escapeiff{\expandnumberarg\forest@node@nextleafid@Goup{#1}}%
1175 \else
1176 \expandnumberarg\forest@node@nextleafid@Gdown{\forestOve{#1}{@next}}%
1177 \fi
1178 }
1179 \def\forest@node@nextleafid@Goup#1{%
1180 \ifnum\forestOve{#1}{@parent}=0

```

```

1181 \PackageError{forest}{get next leaf: this is the last leaf}{}%
1182 \else
1183 \escapeif{\expandnumberarg\forest@node@nextleafid{\forestOve{#1}{@parent}}}%
1184 \fi
1185 }
1186 \def\forest@node@nextleafid@Gdown#1{%
1187 \ifnum\forestOve{#1}{@first}=0
1188 #1%
1189 \else
1190 \escapeif{\expandnumberarg\forest@node@nextleafid@Gdown{\forestOve{#1}{@first}}}%
1191 \fi
1192 }
1193 \def\forest@node@linearnextid{%
1194 \ifnum\forestove{@first}=0
1195 \expandafter\forest@node@linearnextnotdescendantid
1196 \else
1197 \forestove{@first}%
1198 \fi
1199 }
1200 \def\forest@node@linearnextnotdescendantid{%
1201 \expandnumberarg\forest@node@Linearnextnotdescendantid{\forest@cn}%
1202 }
1203 \def\forest@node@Linearnextnotdescendantid#1{%
1204 \ifnum\forestOve{#1}{@next}=0
1205 \escapeif{\expandnumberarg\forest@node@Linearnextnotdescendantid{\forestOve{#1}{@parent}}}%
1206 \else
1207 \forestOve{#1}{@next}%
1208 \fi
1209 }
1210 \def\forest@node@linearpreviousid{%
1211 \ifnum\forestove{@previous}=0
1212 \forestove{@parent}%
1213 \else
1214 \forest@node@previousleafid
1215 \fi
1216 }
1217 \def\forest@ifancestorof#1{% is the current node an ancestor of #1? Yes: #2, no: #3
1218 \expandnumberarg\forest@ifancestorof@{\forestOve{#1}{@parent}}%
1219 }
1220 \def\forest@ifancestorof@#1#2#3{%
1221 \ifnum#1=0
1222 \def\forest@ifancestorof@next{\@secondoftwo}%
1223 \else
1224 \ifnum\forest@cn=#1
1225 \def\forest@ifancestorof@next{\@firstoftwo}%
1226 \else
1227 \def\forest@ifancestorof@next{\expandnumberarg\forest@ifancestorof@{\forestOve{#1}{@parent}}}%
1228 \fi
1229 \fi
1230 \forest@ifancestorof@next{#2}{#3}%
1231 }

```

### 9.3 Node walk

```

1232 \newloop\forest@nodewalk@loop
1233 \forestset{
1234 @handlers@save@currentpath/.code={%
1235 \edef\pgfkeyscurrentkey{\pgfkeyscurrentpath}%
1236 \let\forest@currentkey\pgfkeyscurrentkey
1237 \pgfkeys@split@path

```

```

1238 \edef\forest@currentpath{\pgfkeyscurrentpath}%
1239 \let\forest@currentname\pgfkeyscurrentname
1240 },
1241 /handlers/.step 0 args/.style={
1242 /forest/@handlers@save@currentpath,
1243 \forest@currentkey/.code={#1\forestset{node walk/every step}},
1244 /forest/for \forest@currentname/.style/.expanded={%
1245 for={\forest@currentname}{####1}%
1246 }
1247 },
1248 /handlers/.step 1 arg/.style={%
1249 /forest/@handlers@save@currentpath,
1250 \forest@currentkey/.code={#1\forestset{node walk/every step}},
1251 /forest/for \forest@currentname/.style 2 args/.expanded={%
1252 for={\forest@currentname=####1}{####2}%
1253 }
1254 },
1255 node walk/.code={%
1256 \forestset{%
1257 node walk/before walk,%
1258 node walk/.cd,
1259 #1,%
1260 /forest/.cd,
1261 node walk/after walk
1262 }%
1263 },
1264 for/.code 2 args={%
1265 \forest@forthis{%
1266 \pgfkeysalso{%
1267 node walk/before walk/.style={},%
1268 node walk/every step/.style={},%
1269 node walk/after walk/.style={/forest,if id=0}{#2}},%
1270 %node walk/after walk/.style={#2},%
1271 node walk={#1}%
1272 }%
1273 }%
1274 },
1275 node walk/.cd,
1276 before walk/.code={},
1277 every step/.code={},
1278 after walk/.code={},
1279 current/.step 0 args={},
1280 current/.default=1,
1281 next/.step 0 args={\edef\forest@cn{\forestove{@next}}},
1282 next/.default=1,
1283 previous/.step 0 args={\edef\forest@cn{\forestove{@previous}}},
1284 previous/.default=1,
1285 parent/.step 0 args={\edef\forest@cn{\forestove{@parent}}},
1286 parent/.default=1,
1287 first/.step 0 args={\edef\forest@cn{\forestove{@first}}},
1288 first/.default=1,
1289 last/.step 0 args={\edef\forest@cn{\forestove{@last}}},
1290 last/.default=1,
1291 n/.step 1 arg={%
1292 \def\forest@nodewalk@temp{#1}%
1293 \ifx\forest@nodewalk@temp\pgfkeysnovalue@text
1294 \edef\forest@cn{\forestove{@next}}%
1295 \else
1296 \edef\forest@cn{\forest@node@nthchildid{#1}}%
1297 \fi
1298 },

```

```

1299 n'/.step 1 arg={\edef\forest@cn{\forest@node@nbarthchildid{#1}}},
1300 sibling/.step 0 args={%
1301   \edef\forest@cn{%
1302     \ifnum\forest@previous=0
1303       \forest@next}%
1304   \else
1305     \forest@previous}%
1306   \fi
1307 }%
1308 },
1309 previous leaf/.step 0 args={\edef\forest@cn{\forest@node@previousleafid}},
1310 previous leaf/.default=1,
1311 next leaf/.step 0 args={\edef\forest@cn{\forest@node@nextleafid}},
1312 next leaf/.default=1,
1313 linear next/.step 0 args={\edef\forest@cn{\forest@node@linearnextid}},
1314 linear previous/.step 0 args={\edef\forest@cn{\forest@node@linearpreviousid}},
1315 first leaf/.step 0 args={%
1316   \forest@nodewalk@loop
1317   \edef\forest@cn{\forest@previous}%
1318   \unless\ifnum\forest@previous=0
1319     \forest@nodewalk@repeat
1320 },
1321 last leaf/.step 0 args={%
1322   \forest@nodewalk@loop
1323   \edef\forest@cn{\forest@last}%
1324   \unless\ifnum\forest@last=0
1325     \forest@nodewalk@repeat
1326 },
1327 to tier/.step 1 arg={%
1328   \def\forest@nodewalk@giventier{#1}%
1329   \forest@nodewalk@loop
1330     \forest@toget{tier}\forest@nodewalk@tier
1331   \unless\ifx\forest@nodewalk@tier\forest@nodewalk@giventier
1332     \forest@toget{parent}\forest@cn
1333   \forest@nodewalk@repeat
1334 },
1335 next on tier/.step 0 args={\forest@nodewalk@nextontier},
1336 next on tier/.default=1,
1337 previous on tier/.step 0 args={\forest@nodewalk@previousontier},
1338 previous on tier/.default=1,
1339 name/.step 1 arg={\edef\forest@cn{\forest@node@Nametoid{#1}}},
1340 root/.step 0 args={\edef\forest@cn{\forest@node@rootid}},
1341 root'/.step 0 args={\edef\forest@cn{\forest@root}},
1342 id/.step 1 arg={\edef\forest@cn{#1}},
1343 % maybe it's not wise to have short-step sequences and names potentially clashing
1344 % .unknown/.code={%
1345 %   \forest@node@ifnamedefined{\pgfkeyscurrentname}%
1346 %     {\pgfkeysalso{name=\pgfkeyscurrentname}}%
1347 %     {\expandafter\forest@nodewalk@shortsteps\pgfkeyscurrentname\forest@nodewalk@endshortsteps}%
1348 % },
1349 .unknown/.code={%
1350   \expandafter\forest@nodewalk@shortsteps\pgfkeyscurrentname\forest@nodewalk@endshortsteps
1351 },
1352 node walk/.style={/forest/node walk={#1}},
1353 trip/.code={\forest@forthis{\pgfkeysalso{#1}}},
1354 group/.code={\forest@go{#1}\forestset{node walk/every step}},
1355 % repeat is taken later from /forest/repeat
1356 p/.style={previous=1},
1357 %n/.style={next=1}, % defined in "long" n
1358 u/.style={parent=1},
1359 s/.style={sibling},

```

```

1360 c/.style={current=1},
1361 r/.style={root},
1362 P/.style={previous leaf=1},
1363 N/.style={next leaf=1},
1364 F/.style={first leaf=1},
1365 L/.style={last leaf=1},
1366 >/.style={next on tier=1},
1367 </.style={previous on tier=1},
1368 1/.style={n=1},
1369 2/.style={n=2},
1370 3/.style={n=3},
1371 4/.style={n=4},
1372 5/.style={n=5},
1373 6/.style={n=6},
1374 7/.style={n=7},
1375 8/.style={n=8},
1376 9/.style={n=9},
1377 1/.style={last=1},
1378 %{\dots} is short for group={...}
1379 }
1380 \def\forest@nodewalk@nextontier{%
1381   \foresttoget{tier}\forest@nodewalk@giventier
1382   \edef\forest@cn{\forest@node@linearnextnotdescendantid}%
1383   \forest@nodewalk@loop
1384     \foresttoget{tier}\forest@nodewalk@tier
1385   \unless\ifx\forest@nodewalk@tier\forest@nodewalk@giventier
1386     \edef\forest@cn{\forest@node@linearnextid}%
1387   \forest@nodewalk@repeat
1388 }
1389 \def\forest@nodewalk@previousontier{%
1390   \foresttoget{tier}\forest@nodewalk@giventier
1391   \forest@nodewalk@loop
1392     \edef\forest@cn{\forest@node@linearpreviousid}%
1393   \foresttoget{tier}\forest@nodewalk@tier
1394   \unless\ifx\forest@nodewalk@tier\forest@nodewalk@giventier
1395   \forest@nodewalk@repeat
1396 }
1397 \def\forest@nodewalk@shortsteps{%
1398   \futurelet\forest@nodewalk@nexttoken\forest@nodewalk@shortsteps@
1399 }
1400 \def\forest@nodewalk@shortsteps@#1{%
1401   \ifx\forest@nodewalk@nexttoken\forest@nodewalk@endshortsteps
1402   \else
1403     \ifx\forest@nodewalk@nexttoken\bgroup
1404       \pgfkeysalso{group=#1}%
1405       \@escapeifif\forest@nodewalk@shortsteps
1406     \else
1407       \pgfkeysalso{#1}%
1408       \@escapeifif\forest@nodewalk@shortsteps
1409     \fi
1410   \fi
1411 }
1412 \def\forest@go#1{%
1413   {%
1414     \forestset{%
1415       node walk/before walk/.code={},%
1416       node walk/every step/.code={},%
1417       node walk/after walk/.code={},%
1418       node walk={#1}%
1419     }%
1420     \expandafter

```



```

1421 }%
1422 \expandafter\def\expandafter\forest@cn\expandafter{\forest@cn}%
1423 }

```

## 9.4 Node options

### 9.4.1 Option-declaration mechanism

Common code for declaring options.

```

1424 \def\forest@declarehandler#1#2#3{%#1=handler for specific type,#2=option name,#3=default value
1425   \pgfkeyssetvalue{/forest/#2}{#3}%
1426   \appto\forest@node@init{\forest@init{#2}}%
1427   \forest@convert@others@to@underscores{#2}\forest@pgfmathoptionname
1428   \edef\forest@marshal{%
1429     \noexpand#1{/forest/#2}{/forest}{#2}\forest@pgfmathoptionname}%
1430   }\forest@marshal
1431 }
1432 \def\forest@def@with@pgfeov#1#2{% \pgfeov mustn't occur in the arg of the .code handler!!!
1433   \long\def#1##1\pgfeov{#2}%
1434 }

```

Option-declaration handlers.

```

1435 \newtoks\forest@temp@toks
1436 \def\forest@declaretoks@handler#1#2#3#4{%
1437   \forest@declaretoks@handler@A{#1}{#2}{#3}{#4}{}%
1438 }
1439 \def\forest@declarekeylist@handler#1#2#3#4{%
1440   \forest@declaretoks@handler@A{#1}{#2}{#3}{#4}{,}%
1441   \pgfkeysgetvalue{#1/.@cmd}\forest@temp
1442   \pgfkeyslet{#1'/.@cmd}\forest@temp
1443   \pgfkeyssetvalue{#1'/option@name}{#3}%
1444   \pgfkeysgetvalue{#1+/.@cmd}\forest@temp
1445   \pgfkeyslet{#1/.@cmd}\forest@temp
1446 }
1447 \def\forest@declaretoks@handler@A#1#2#3#4#5{% #1=key,#2=path,#3=name,#4=pgfmathname,#5=infix
1448   \pgfkeysalso{%
1449     #1/.code={\forest@set{\forest@setter@node}{#3}{##1}},
1450     #1+/.code={\forest@appto{\forest@setter@node}{#3}{##1}},
1451     #1-/.code={\forest@preto{\forest@setter@node}{#3}{##1#5}},
1452     #2/if #3/.code n args={3}{%
1453       \forest@toget{#3}\forest@temp@option@value
1454       \edef\forest@temp@compared@value{\unexpanded{##1}}%
1455       \ifx\forest@temp@option@value\forest@temp@compared@value
1456         \pgfkeysalso{##2}%
1457       \else
1458         \pgfkeysalso{##3}%
1459       \fi
1460     },
1461     #2/if in #3/.code n args={3}{%
1462       \forest@toget{#3}\forest@temp@option@value
1463       \edef\forest@temp@compared@value{\unexpanded{##1}}%
1464       \expandafter\expandafter\expandafter\pgfutil@in@\expandafter\expandafter\expandafter{\expandafter\forest@temp@option@value}
1465       \ifpgfutil@in@
1466         \pgfkeysalso{##2}%
1467       \else
1468         \pgfkeysalso{##3}%
1469       \fi
1470     },
1471     #2/where #3/.style n args={3}{for tree={#2/if #3={##1}{##2}{##3}}},
1472     #2/where in #3/.style n args={3}{for tree={#2/if in #3={##1}{##2}{##3}}}
1473   }%
1474   \pgfkeyssetvalue{#1'/option@name}{#3}%

```

```

1475 \pgfkeyssetvalue{#1+/option@name}{#3}%
1476 \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@toks{##1}{#3}}%
1477 }
1478 \def\forest@declareautowrappedtoks@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname,#5=infix
1479 \forest@declaretoks@handler{#1}{#2}{#3}{#4}%
1480 \pgfkeysgetvalue{#1/.@cmd}\forest@temp
1481 \pgfkeyslet{#1'/.@cmd}\forest@temp
1482 \pgfkeysalso{#1/.style={#1'/.wrap value={##1}}}%
1483 \pgfkeyssetvalue{#1'/option@name}{#3}%
1484 \pgfkeysgetvalue{#1+/.@cmd}\forest@temp
1485 \pgfkeyslet{#1+/.@cmd}\forest@temp
1486 \pgfkeysalso{#1+/.style={#1+/.wrap value={##1}}}%
1487 \pgfkeyssetvalue{#1+'/option@name}{#3}%
1488 \pgfkeysgetvalue{#1-/.@cmd}\forest@temp
1489 \pgfkeyslet{#1-'/@cmd}\forest@temp
1490 \pgfkeysalso{#1-/.style={#1-'/wrap value={##1}}}%
1491 \pgfkeyssetvalue{#1-'/option@name}{#3}%
1492 }
1493 \def\forest@declarereadonlydimen@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
1494 \pgfkeysalso{%
1495 #2/if #3/.code n args={3}{%
1496 \forestoget{#3}\forest@temp@option@value
1497 \ifdim\forest@temp@option@value=##1\relax
1498 \pgfkeysalso{##2}%
1499 \else
1500 \pgfkeysalso{##3}%
1501 \fi
1502 },
1503 #2/where #3/.style n args={3}{for tree={#2/if #3={##1}{##2}{##3}}},
1504 }%
1505 \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@dimen{##1}{#3}}%
1506 }
1507 \def\forest@declaredimen@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
1508 \forest@declarereadonlydimen@handler{#1}{#2}{#3}{#4}%
1509 \pgfkeysalso{%
1510 #1/.code={%
1511 \pgfmathsetlengthmacro\forest@temp{##1}%
1512 \forestOlet{\forest@setter@node}{#3}\forest@temp
1513 },
1514 #1+/.code={%
1515 \pgfmathsetlengthmacro\forest@temp{##1}%
1516 \pgfutil@tempdima=\forestove{#3}
1517 \advance\pgfutil@tempdima\forest@temp\relax
1518 \forestOset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1519 },
1520 #1-/.code={%
1521 \pgfmathsetlengthmacro\forest@temp{##1}%
1522 \pgfutil@tempdima=\forestove{#3}
1523 \advance\pgfutil@tempdima-\forest@temp\relax
1524 \forestOset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1525 },
1526 #1*/.style={%
1527 #1={#4()}*(##1)}%
1528 },
1529 #1:/ .style={%
1530 #1={#4()/ (##1)}%
1531 },
1532 #1'/.code={%
1533 \pgfutil@tempdima=##1\relax
1534 \forestOset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1535 },

```

```

1536 #1'+/.code={%
1537   \pgfutil@tempdima=\forestove{#3}\relax
1538   \advance\pgfutil@tempdima##1\relax
1539   \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1540 },
1541 #1'-/.code={%
1542   \pgfutil@tempdima=\forestove{#3}\relax
1543   \advance\pgfutil@tempdima-##1\relax
1544   \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1545 },
1546 #1'*/.style={%
1547   \pgfutil@tempdima=\forestove{#3}\relax
1548   \multiply\pgfutil@tempdima##1\relax
1549   \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1550 },
1551 #1':/.style={%
1552   \pgfutil@tempdima=\forestove{#3}\relax
1553   \divide\pgfutil@tempdima##1\relax
1554   \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
1555 },
1556 }%
1557 \pgfkeyssetvalue{#1/option@name}{#3}%
1558 \pgfkeyssetvalue{#1+/option@name}{#3}%
1559 \pgfkeyssetvalue{#1-/option@name}{#3}%
1560 \pgfkeyssetvalue{#1*/option@name}{#3}%
1561 \pgfkeyssetvalue{#1:/option@name}{#3}%
1562 \pgfkeyssetvalue{#1'/option@name}{#3}%
1563 \pgfkeyssetvalue{#1'+/option@name}{#3}%
1564 \pgfkeyssetvalue{#1'-/option@name}{#3}%
1565 \pgfkeyssetvalue{#1'* /option@name}{#3}%
1566 \pgfkeyssetvalue{#1': /option@name}{#3}%
1567 }
1568 \def\forest@declarereadonlycount@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
1569   \pgfkeysalso{
1570     #2/if #3/.code n args={3}{%
1571       \foresttoget{#3}\forest@temp@option@value
1572       \ifnum\forest@temp@option@value=##1\relax
1573         \pgfkeysalso{##2}%
1574       \else
1575         \pgfkeysalso{##3}%
1576       \fi
1577     },
1578     #2/where #3/.style n args={3}{for tree={#2/if #3={##1}{##2}{##3}}},
1579   }%
1580   \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@count{##1}{#3}}%
1581 }
1582 \def\forest@declarecount@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
1583   \forest@declarereadonlycount@handler{#1}{#2}{#3}{#4}%
1584   \pgfkeysalso{
1585     #1/.code={%
1586       \pgfmathtruncatemacro\forest@temp{##1}%
1587       \forestOlet{\forest@setter@node}{#3}\forest@temp
1588     },
1589     #1+/.code={%
1590       \pgfmathsetlengthmacro\forest@temp{##1}%
1591       \c@pgf@counta=\forestove{#3}\relax
1592       \advance\c@pgf@counta\forest@temp\relax
1593       \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1594     },
1595     #1-/.code={%
1596       \pgfmathsetlengthmacro\forest@temp{##1}%

```

```

1597     \c@pgf@counta=\forestove{#3}\relax
1598     \advance\c@pgf@counta-\forest@temp\relax
1599     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1600 },
1601 #1*/.code={%
1602     \pgfmathsetlengthmacro\forest@temp{##1}%
1603     \c@pgf@counta=\forestove{#3}\relax
1604     \multiply\c@pgf@counta\forest@temp\relax
1605     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1606 },
1607 #1:/.code={%
1608     \pgfmathsetlengthmacro\forest@temp{##1}%
1609     \c@pgf@counta=\forestove{#3}\relax
1610     \divide\c@pgf@counta\forest@temp\relax
1611     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1612 },
1613 #1'/.code={%
1614     \c@pgf@counta=##1\relax
1615     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1616 },
1617 #1'+/.code={%
1618     \c@pgf@counta=\forestove{#3}\relax
1619     \advance\c@pgf@counta##1\relax
1620     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1621 },
1622 #1'-/.code={%
1623     \c@pgf@counta=\forestove{#3}\relax
1624     \advance\c@pgf@counta-##1\relax
1625     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1626 },
1627 #1'*/.style={%
1628     \c@pgf@counta=\forestove{#3}\relax
1629     \multiply\c@pgf@counta##1\relax
1630     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1631 },
1632 #1':/.style={%
1633     \c@pgf@counta=\forestove{#3}\relax
1634     \divide\c@pgf@counta##1\relax
1635     \forestOset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
1636 },
1637 }%
1638 \pgfkeyssetvalue{#1/option@name}{#3}%
1639 \pgfkeyssetvalue{#1+/option@name}{#3}%
1640 \pgfkeyssetvalue{#1-/option@name}{#3}%
1641 \pgfkeyssetvalue{#1*/option@name}{#3}%
1642 \pgfkeyssetvalue{#1:/option@name}{#3}%
1643 \pgfkeyssetvalue{#1'/option@name}{#3}%
1644 \pgfkeyssetvalue{#1'+/option@name}{#3}%
1645 \pgfkeyssetvalue{#1'-/option@name}{#3}%
1646 \pgfkeyssetvalue{#1'*/option@name}{#3}%
1647 \pgfkeyssetvalue{#1':/option@name}{#3}%
1648 }
1649 \def\forest@declareboolean@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
1650     \pgfkeysalso{%
1651         #1/.code={%
1652             \ifstrequal{##1}{1}{%
1653                 \forestOset{\forest@setter@node}{#3}{1}%
1654             }{%
1655                 \pgfmathifthenelse{##1}{1}{0}%
1656                 \forestOlet{\forest@setter@node}{#3}\pgfmathresult
1657             }%

```

```

1658 },
1659 #1/.default=1,
1660 #2/not #3/.code={\forest0set{\forest@setter@node}{#3}{0}},
1661 #2/if #3/.code 2 args={%
1662   \forestoget{#3}\forest@temp@option@value
1663   \ifnum\forest@temp@option@value=1
1664     \pgfkeysalso{##1}%
1665   \else
1666     \pgfkeysalso{##2}%
1667   \fi
1668 },
1669 #2/where #3/.style 2 args={for tree={#2/if #3={##1}{##2}}}
1670 }%
1671 \pgfkeyssetvalue{#1/option@name}{#3}%
1672 \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@count{##1}{#3}}%
1673 }
1674 \pgfkeys{/forest,
1675   declare toks/.code 2 args={%
1676     \forest@declarehandler\forest@declaretoks@handler{#1}{#2}%
1677   },
1678   declare autowrapped toks/.code 2 args={%
1679     \forest@declarehandler\forest@declareautowrappedtoks@handler{#1}{#2}%
1680   },
1681   declare keylist/.code 2 args={%
1682     \forest@declarehandler\forest@declarekeylist@handler{#1}{#2}%
1683   },
1684   declare readonly dimen/.code={%
1685     \forest@declarehandler\forest@declarereadonlydimen@handler{#1}{}%
1686   },
1687   declare dimen/.code 2 args={%
1688     \forest@declarehandler\forest@declaredimen@handler{#1}{#2}%
1689   },
1690   declare readonly count/.code={%
1691     \forest@declarehandler\forest@declarereadonlycount@handler{#1}{}%
1692   },
1693   declare count/.code 2 args={%
1694     \forest@declarehandler\forest@declarecount@handler{#1}{#2}%
1695   },
1696   declare boolean/.code 2 args={%
1697     \forest@declarehandler\forest@declareboolean@handler{#1}{#2}%
1698   },
1699   /handlers/.pgfmath/.code={%
1700     \pgfmathparse{#1}%
1701     \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\pgfmathresult}%
1702   },
1703   /handlers/.wrap value/.code={%
1704     \edef\forest@handlers@wrap@currentpath{\pgfkeyscurrentpath}%
1705     \pgfkeysgetvalue{\forest@handlers@wrap@currentpath/option@name}\forest@currentoptionname
1706     \expandafter\forestoget\expandafter{\forest@currentoptionname}\forest@option@value
1707     \forest@def@withpgfeov\forest@wrap@code{#1}%
1708     \expandafter\edef\expandafter\forest@wrapped@value\expandafter{\expandafter\expandonce\expandafter{\expand
1709     \pgfkeysalso{\forest@handlers@wrap@currentpath/.expand once=\forest@wrapped@value}%
1710   },
1711   /handlers/.wrap pgfmath arg/.code 2 args={%
1712     \pgfmathparse{#2}\let\forest@wrap@arg@i\pgfmathresult
1713     \edef\forest@wrap@args{{\expandonce\forest@wrap@arg@i}}%
1714     \def\forest@wrap@code##1{#1}%
1715     \expandafter\expandafter\expandafter\forest@temp@toks\expandafter\expandafter\expandafter{\expandafter\fo
1716     \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\the\forest@temp@toks}%
1717   },
1718   /handlers/.wrap 2 pgfmath args/.code n args={3}{%

```

```

1719 \pgfmthparse{#2}\let\forest@wrap@arg@i\pgfmthresult
1720 \pgfmthparse{#3}\let\forest@wrap@arg@ii\pgfmthresult
1721 \edef\forest@wrap@args{\expandonce\forest@wrap@arg@i}{\expandonce\forest@wrap@arg@ii}}%
1722 \def\forest@wrap@code##1##2{#1}%
1723 \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\forest@wrapped\expandafter\exp
1724 \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\forest@wrapped}%
1725 },
1726 /handlers/.wrap 3 pgfmth args/.code n args={4}{%
1727 \forest@wrap@n@pgfmth@args{#2}{#3}{#4}{-}{-}{-}{-}{3}%
1728 \forest@wrap@n@pgfmth@do{#1}{3}},
1729 /handlers/.wrap 4 pgfmth args/.code n args={5}{%
1730 \forest@wrap@n@pgfmth@args{#2}{#3}{#4}{#5}{-}{-}{-}{4}%
1731 \forest@wrap@n@pgfmth@do{#1}{4}},
1732 /handlers/.wrap 5 pgfmth args/.code n args={6}{%
1733 \forest@wrap@n@pgfmth@args{#2}{#3}{#4}{#5}{#6}{-}{-}{5}%
1734 \forest@wrap@n@pgfmth@do{#1}{5}},
1735 /handlers/.wrap 6 pgfmth args/.code n args={7}{%
1736 \forest@wrap@n@pgfmth@args{#2}{#3}{#4}{#5}{#6}{#7}{-}{6}%
1737 \forest@wrap@n@pgfmth@do{#1}{6}},
1738 /handlers/.wrap 7 pgfmth args/.code n args={8}{%
1739 \forest@wrap@n@pgfmth@args{#2}{#3}{#4}{#5}{#6}{#7}{#8}{-}{7}%
1740 \forest@wrap@n@pgfmth@do{#1}{7}},
1741 /handlers/.wrap 8 pgfmth args/.code n args={9}{%
1742 \forest@wrap@n@pgfmth@args{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}{8}%
1743 \forest@wrap@n@pgfmth@do{#1}{8}},
1744 }
1745 \def\forest@wrap@n@pgfmth@args#1#2#3#4#5#6#7#8#9{%
1746 \pgfmthparse{#1}\let\forest@wrap@arg@i\pgfmthresult
1747 \ifnum#9>1 \pgfmthparse{#2}\let\forest@wrap@arg@ii\pgfmthresult\fi
1748 \ifnum#9>2 \pgfmthparse{#3}\let\forest@wrap@arg@iii\pgfmthresult\fi
1749 \ifnum#9>3 \pgfmthparse{#4}\let\forest@wrap@arg@iv\pgfmthresult\fi
1750 \ifnum#9>4 \pgfmthparse{#5}\let\forest@wrap@arg@v\pgfmthresult\fi
1751 \ifnum#9>5 \pgfmthparse{#6}\let\forest@wrap@arg@vi\pgfmthresult\fi
1752 \ifnum#9>6 \pgfmthparse{#7}\let\forest@wrap@arg@vii\pgfmthresult\fi
1753 \ifnum#9>7 \pgfmthparse{#8}\let\forest@wrap@arg@viii\pgfmthresult\fi
1754 \edef\forest@wrap@args{%
1755 {\expandonce\forest@wrap@arg@i}
1756 \ifnum#9>1 {\expandonce\forest@wrap@arg@ii}\fi
1757 \ifnum#9>2 {\expandonce\forest@wrap@arg@iii}\fi
1758 \ifnum#9>3 {\expandonce\forest@wrap@arg@iv}\fi
1759 \ifnum#9>4 {\expandonce\forest@wrap@arg@v}\fi
1760 \ifnum#9>5 {\expandonce\forest@wrap@arg@vi}\fi
1761 \ifnum#9>6 {\expandonce\forest@wrap@arg@vii}\fi
1762 \ifnum#9>7 {\expandonce\forest@wrap@arg@viii}\fi
1763 }%
1764 }
1765 \def\forest@wrap@n@pgfmth@do#1#2{%
1766 \ifcase#2\relax
1767 \or\def\forest@wrap@code##1{#1}%
1768 \or\def\forest@wrap@code##1##2{#1}%
1769 \or\def\forest@wrap@code##1##2##3{#1}%
1770 \or\def\forest@wrap@code##1##2##3##4{#1}%
1771 \or\def\forest@wrap@code##1##2##3##4##5{#1}%
1772 \or\def\forest@wrap@code##1##2##3##4##5##6{#1}%
1773 \or\def\forest@wrap@code##1##2##3##4##5##6##7{#1}%
1774 \or\def\forest@wrap@code##1##2##3##4##5##6##7##8{#1}%
1775 \fi
1776 \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\forest@wrapped\expandafter\exp
1777 \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\forest@wrapped}%
1778 }

```

## 9.4.2 Declaring options

```

1779 \def\forest@node@setname#1{%
1780   \forestoeset{name}{#1}%
1781   \csedef{forest@id@of@#1}{\forest@cn}%
1782 }
1783 \def\forest@node@Nametoid#1{% #1 = name
1784   \csname forest@id@of@#1\endcsname
1785 }
1786 \def\forest@node@ifnamedefined#1{% #1 = name, #2=true,#3=false
1787   \ifcsname forest@id@of@#1\endcsname
1788     \expandafter\@firstoftwo
1789   \else
1790     \expandafter\@secondoftwo
1791   \fi
1792 }
1793 \def\forest@node@setalias#1{%
1794   \csedef{forest@id@of@#1}{\forest@cn}%
1795 }
1796 \def\forest@node@Setalias#1#2{%
1797   \csedef{forest@id@of@#2}{#1}%
1798 }
1799 \forestset{
1800   TeX/.code={#1},
1801   TeX'/.code={\appto\forest@externalize@loadimages{#1}#1},
1802   TeX''/.code={\appto\forest@externalize@loadimages{#1}},
1803   declare toks={name}{},
1804   name/.code={% override the default setter
1805     \forest@node@setname{#1}%
1806   },
1807   alias/.code={\forest@node@setalias{#1}},
1808   declare autowrapped toks={content}{},
1809   declare count={grow}{270},
1810   TeX={% a hack for grow-reversed connection, and compass-based grow specification
1811     \pgfkeysgetvalue{/forest/grow/.@cmd}\forest@temp
1812     \pgfkeyslet{/forest/grow@/.@cmd}\forest@temp
1813   },
1814   grow/.style={grow@={#1},reversed=0},
1815   grow'/.style={grow@={#1},reversed=1},
1816   grow''/.style={grow@={#1}},
1817   grow@/.is choice,
1818   grow@/east/.style={/forest/grow@=0},
1819   grow@/north east/.style={/forest/grow@=45},
1820   grow@/north/.style={/forest/grow@=90},
1821   grow@/north west/.style={/forest/grow@=135},
1822   grow@/west/.style={/forest/grow@=180},
1823   grow@/south west/.style={/forest/grow@=225},
1824   grow@/south/.style={/forest/grow@=270},
1825   grow@/south east/.style={/forest/grow@=315},
1826   grow@/.unknown/.code={\let\forest@temp@grow\pgfkeyscurrentname
1827     \pgfkeysalso{/forest/grow@/.expand once=\forest@temp@grow}},
1828   declare boolean={reversed}{0},
1829   declare toks={parent anchor}{},
1830   declare toks={child anchor}{},
1831   declare toks={anchor}{base},
1832   declare toks={calign}{midpoint},
1833   TeX={%
1834     \pgfkeysgetvalue{/forest/calign/.@cmd}\forest@temp
1835     \pgfkeyslet{/forest/calign'/.@cmd}\forest@temp
1836   },
1837   calign/.is choice,

```

```

1838 calign/child/.style={calign'=child},
1839 calign/first/.style={calign'=child,calign primary child=1},
1840 calign/last/.style={calign'=child,calign primary child=-1},
1841 calign with current/.style={for parent/.wrap pgfmath arg={calign=child,calign primary child=##1}{n}},
1842 calign with current edge/.style={for parent/.wrap pgfmath arg={calign=child edge,calign primary child=##1}{n}},
1843 calign/child edge/.style={calign'=child edge},
1844 calign/midpoint/.style={calign'=midpoint},
1845 calign/center/.style={calign'=midpoint,calign primary child=1,calign secondary child=-1},
1846 calign/edge midpoint/.style={calign'=edge midpoint},
1847 calign/fixed angles/.style={calign'=fixed angles},
1848 calign/fixed edge angles/.style={calign'=fixed edge angles},
1849 calign/.unknown/.code={\PackageError{forest}{unknown calign '\pgfkeyscurrentname'}{}}},
1850 declare count={calign primary child}{1},
1851 declare count={calign secondary child}{-1},
1852 declare count={calign primary angle}{-35},
1853 declare count={calign secondary angle}{35},
1854 calign child/.style={calign primary child=##1},
1855 calign angle/.style={calign primary angle=-##1,calign secondary angle=##1},
1856 declare toks={tier}{},
1857 declare toks={fit}{tight},
1858 declare boolean={ignore}{0},
1859 declare boolean={ignore edge}{0},
1860 no edge/.style={edge'={},ignore edge},
1861 declare keylist={edge}{draw},
1862 declare toks={edge path}{%
1863   \noexpand\path[\forestoption{edge}]%
1864   (\forestOve{\forestove{@parent}}{name}.parent anchor)--(\forestove{name}.child anchor)\forestoption{edge
1865 triangle/.style={edge path={%
1866   \noexpand\path[\forestoption{edge}]%
1867   (\forestove{name}.north east)--(\forestOve{\forestove{@parent}}{name}.south)--(\forestove{name}.north w
1868 declare toks={edge label}{},
1869 declare boolean={phantom}{0},
1870 baseline/.style={alias={forest@baseline@node}},
1871 declare readonly count={n},
1872 declare readonly count={n'},
1873 declare readonly count={n children},
1874 declare readonly count={level},
1875 declare dimen=x{},
1876 declare dimen=y{},
1877 declare dimen={s}{0pt},
1878 declare dimen={l}{6ex}, % just in case: should be set by the calibration
1879 declare dimen={s sep}{0.6666em},
1880 declare dimen={l sep}{1ex}, % just in case: calibration!
1881 declare keylist={node options}{},
1882 declare toks={tikz}{},
1883 afterthought/.style={tikz+=##1},
1884 declare toks={tikz preamble}{},
1885 label/.style={tikz={\path[late options={%
1886   name=\forestoption{name},label=##1}};}},
1887 pin/.style={tikz={\path[late options={%
1888   name=\forestoption{name},pin=##1}};}},
1889 declare toks={content format}{\forestoption{content}},
1890 declare toks={node format}{%
1891   \noexpand\node
1892   [\forestoption{node options},anchor=\forestoption{anchor}]%
1893   (\forestoption{name})%
1894   {\forestoption{content format}};%
1895 },
1896 tabular@environment/.style={content format={%
1897   \noexpand\begin{tabular}[\forestoption{base}]{\forestoption{align}}%
1898   \forestoption{content}%

```



```

1899     \noexpand\end{tabular}%
1900   }},
1901   declare toks={align}{},
1902   TeX={\pgfkeysgetvalue{/forest/align/.@cmd}\forest@temp
1903     \pgfkeyslet{/forest/align'/.@cmd}\forest@temp},
1904   align/.is choice,
1905   align/.unknown/.code={%
1906     \edef\forest@marshal{%
1907       \noexpand\pgfkeysalso{%
1908         align'={\pgfkeyscurrentname},%
1909         tabular@environment
1910       }%
1911     }\forest@marshal
1912   },
1913   align/center/.style={align'={@{}c@{}}},tabular@environment},
1914   align/left/.style={align'={@{}l@{}}},tabular@environment},
1915   align/right/.style={align'={@{}r@{}}},tabular@environment},
1916   declare toks={base}{t},
1917   TeX={\pgfkeysgetvalue{/forest/base/.@cmd}\forest@temp
1918     \pgfkeyslet{/forest/base'/.@cmd}\forest@temp},
1919   base/.is choice,
1920   base/top/.style={base'=t},
1921   base/bottom/.style={base'=b},
1922   base/.unknown/.style={base'/.expand once=\pgfkeyscurrentname},
1923   .unknown/.code={%
1924     \expandafter\pgfutil@in@\expandafter.\expandafter{\pgfkeyscurrentname}%
1925     \ifpgfutil@in@
1926       \expandafter\forest@relatednode@option@setter\pgfkeyscurrentname=#1\forest@END
1927     \else
1928       \edef\forest@marshal{%
1929         \noexpand\pgfkeysalso{node options={\pgfkeyscurrentname=\unexpanded{#1}}}%
1930       }\forest@marshal
1931     \fi
1932   },
1933   get node boundary/.code={%
1934     \forestoget{boundary}\forest@node@boundary
1935     \def#1{%
1936       \forest@extendpath#1\forest@node@boundary{\pgfpoint{\forestove{x}}{\forestove{y}}}%
1937     },
1938   % get min l tree boundary/.code={%
1939   %   \forest@get@tree@boundary{negative}{\the\numexpr\forestove{grow}-90\relax}#1},
1940   % get max l tree boundary/.code={%
1941   %   \forest@get@tree@boundary{positive}{\the\numexpr\forestove{grow}-90\relax}#1},
1942   get min s tree boundary/.code={%
1943     \forest@get@tree@boundary{negative}{\forestove{grow}}#1},
1944   get max s tree boundary/.code={%
1945     \forest@get@tree@boundary{positive}{\forestove{grow}}#1},
1946   fit to tree/.code={%
1947     \pgfkeysalso{%
1948       /forest/get min s tree boundary=\forest@temp@negative@boundary,
1949       /forest/get max s tree boundary=\forest@temp@positive@boundary
1950     }%
1951     \edef\forest@temp@boundary{\expandonce{\forest@temp@negative@boundary}\expandonce{\forest@temp@positive@b
1952   \forest@path@getboundingrectangle@xy\forest@temp@boundary
1953   \pgfkeysalso{inner sep=0,fit/.expanded={(\the\pgf@xa,\the\pgf@ya)(\the\pgf@xb,\the\pgf@yb)}}%
1954 },
1955 use as bounding box/.style={%
1956   before drawing tree={
1957     tikz+/.expanded={%
1958       \noexpand\pgfresetboundingbox
1959       \noexpand\useasboundingbox

```

```

1960      ($(.anchor)+(\forestoption{min x},\forestoption{min y}))$)
1961      rectangle
1962      ($(.anchor)+(\forestoption{max x},\forestoption{max y}))$)
1963      ;
1964  }
1965  }
1966  },
1967  use as bounding box'/.style={%
1968    before drawing tree={
1969      tikz+/.expanded={%
1970        \noexpand\pgfresetboundingbox
1971        \noexpand\useasboundingbox
1972        ($(.anchor)+(\forestoption{min x}+\pgfkeysvalueof{/pgf/outer xsep}/2+\pgfkeysvalueof{/pgf/inner xsep})
1973        rectangle
1974        ($(.anchor)+(\forestoption{max x}-\pgfkeysvalueof{/pgf/outer xsep}/2-\pgfkeysvalueof{/pgf/inner xsep})
1975        ;
1976      }
1977    }
1978  },
1979 }%
1980 \def\forest@get@tree@boundary#1#2#3{%#1=pos/neg,#2=grow,#3=receiving cs
1981   \def#3{}%
1982   \forest@node@getedge{#1}{#2}\forest@temp@boundary
1983   \forest@extendpath#3\forest@temp@boundary{\pgfpoint{\forestove{x}}{\forestove{y}}}%
1984 }
1985 \def\forest@setter@node{\forest@cn}%
1986 \def\forest@relatednode@option@setter#1.#2=#3\forest@END{%
1987   \forest@forthis{%
1988     \forest@nameandgo{#1}%
1989     \let\forest@setter@node\forest@cn
1990   }%
1991   \pgfkeysalso{#2={#3}}%
1992   \def\forest@setter@node{\forest@cn}%
1993 }%

```

### 9.4.3 Option propagation

The propagators targeting single nodes are automatically defined by node walk steps definitions.

```

1994 \forestset{
1995   for tree/.code={\forest@node@foreach{\pgfkeysalso{#1}}},
1996   if/.code n args={3}{%
1997     \pgfmathparse{#1}%
1998     \ifnum\pgfmathresult=0 \pgfkeysalso{#3}\else\pgfkeysalso{#2}\fi
1999   },
2000   where/.style n args={3}{for tree={if={#1}{#2}{#3}}},
2001   for descendants/.code={\forest@node@foreachdescendant{\pgfkeysalso{#1}}},
2002   for all next/.style={for next={#1,for all next={#1}}},
2003   for all previous/.style={for previous={#1,for all previous={#1}}},
2004   for siblings/.style={for all previous={#1,for all next={#1}}},
2005   for ancestors/.style={for parent={#1,for ancestors={#1}}},
2006   for ancestors'/.style={#1,for ancestors={#1}},
2007   for children/.code={\forest@node@foreachchild{\pgfkeysalso{#1}}},
2008   for c-commanded={for sibling={for tree={#1}}},
2009   for c-commanders={for sibling={#1},for parent={for c-commanders={#1}}}
2010 }

```

A bit of complication to allow for nested repeats without  $\text{\TeX}$  groups.

```

2011 \newcount\forest@repeat@key@depth
2012 \forestset{%
2013   repeat/.code 2 args={%
2014     \advance\forest@repeat@key@depth1

```

```

2015 \pgfmathparse{int(#1)}%
2016 \csedef{forest@repeat@key@the\forest@repeat@key@depth}{\pgfmathresult}%
2017 \expandafter\newloop\csname forest@repeat@key@loop@the\forest@repeat@key@depth\endcsname
2018 \def\forest@marshal{%
2019 \csname forest@repeat@key@loop@the\forest@repeat@key@depth\endcsname
2020 \forest@temp@count=\csname forest@repeat@key@the\forest@repeat@key@depth\endcsname\relax
2021 \ifnum\forest@temp@count>0
2022 \advance\forest@temp@count-1
2023 \csedef{forest@repeat@key@the\forest@repeat@key@depth}{\the\forest@temp@count}%
2024 \pgfkeysalso{#2}%
2025 }%
2026 \expandafter\forest@marshal\csname forest@repeat@key@repeat@the\forest@repeat@key@depth\endcsname
2027 \advance\forest@repeat@key@depth-1
2028 },
2029 }
2030 \pgfkeysgetvalue{/forest/repeat/.@cmd}\forest@temp
2031 \pgfkeyslet{/forest/node walk/repeat/.@cmd}\forest@temp
2032 %

```

#### 9.4.4 pgfmath extensions

```

2033 \pgfmathdeclarefunction{strequal}{2}{%
2034 \ifstrequal{#1}{#2}{\def\pgfmathresult{1}}{\def\pgfmathresult{0}}%
2035 }
2036 \pgfmathdeclarefunction{instr}{2}{%
2037 \pgfutil@in@{#1}{#2}%
2038 \ifpgfutil@in@\def\pgfmathresult{1}\else\def\pgfmathresult{0}\fi
2039 }
2040 \pgfmathdeclarefunction{strcat}{...}{%
2041 \edef\pgfmathresult{\forest@strip@braces{#1}}%
2042 }
2043 \def\forest@pgfmathhelper@attribute@toks#1#2{%
2044 \forest@forthis{%
2045 \forest@nameandgo{#1}%
2046 \forest@toget{#2}\pgfmathresult
2047 }%
2048 }
2049 \def\forest@pgfmathhelper@attribute@dimen#1#2{%
2050 \forest@forthis{%
2051 \forest@nameandgo{#1}%
2052 \forest@toget{#2}\forest@temp
2053 \pgfmathparse{+\forest@temp}%
2054 }%
2055 }
2056 \def\forest@pgfmathhelper@attribute@count#1#2{%
2057 \forest@forthis{%
2058 \forest@nameandgo{#1}%
2059 \forest@toget{#2}\forest@temp
2060 \pgfmathtruncatemacro\pgfmathresult{\forest@temp}%
2061 }%
2062 }
2063 \pgfmathdeclarefunction{id}{1}{%
2064 \forest@forthis{%
2065 \forest@nameandgo{#1}%
2066 \let\pgfmathresult\forest@cn
2067 }%
2068 }
2069 \forestset{%
2070 if id/.code n args={3}{%
2071 \ifnum#1=\forest@cn\relax
2072 \pgfkeysalso{#2}%

```

```

2073     \else
2074     \pgfkeysalso{#3}%
2075     \fi
2076 },
2077 where id/.style n args={3}{for tree={if id={#1}{#2}{#3}}}
2078 }

```

## 9.5 Dynamic tree

```

2079 \def\forest@last@node{0}
2080 \def\forest@nodehandleby@name@nodewalk@or@bracket#1{%
2081   \ifx\pgfkeysnovalue#1%
2082     \edef\forest@last@node{\forest@node@Nametoid{forest@last@node}}%
2083   \else
2084     \forest@nodehandleby@nnb@checkfirst#1\forest@END
2085   \fi
2086 }
2087 \def\forest@nodehandleby@nnb@checkfirst#1#2\forest@END{%
2088   \ifx[#1%
2089     \forest@create@node{#1#2}%
2090   \else
2091     \forest@forthis{%
2092       \forest@nameandgo{#1#2}%
2093       \let\forest@last@node\forest@cn
2094     }%
2095   \fi
2096 }
2097 \def\forest@create@node#1{% #1=bracket representation
2098   \bracketParse{\forest@create@collectafterthought}%
2099   \forest@last@node=#1\forest@end@create@node
2100 }
2101 \def\forest@create@collectafterthought#1\forest@end@create@node{%
2102   \forest@let0{\forest@last@node}{delay}{\forest@last@node}{given options}%
2103   \forest@set{\forest@last@node}{given options}{}%
2104   \forest@eappto{\forest@last@node}{delay}{,\unexpanded{#1}}%
2105 }
2106 \def\forest@remove@node#1{%
2107   \forest@node@Remove{#1}%
2108 }
2109 \def\forest@append@node#1#2{%
2110   \forest@node@Remove{#2}%
2111   \forest@node@Append{#1}{#2}%
2112 }
2113 \def\forest@prepend@node#1#2{%
2114   \forest@node@Remove{#2}%
2115   \forest@node@Prepend{#1}{#2}%
2116 }
2117 \def\forest@insertafter@node#1#2{%
2118   \forest@node@Remove{#2}%
2119   \forest@node@Insertafter{\forest@ve{#1}{@parent}}{#2}{#1}%
2120 }
2121 \def\forest@insertbefore@node#1#2{%
2122   \forest@node@Remove{#2}%
2123   \forest@node@Insertbefore{\forest@ve{#1}{@parent}}{#2}{#1}%
2124 }
2125 \def\forest@appto@do@ynamics#1#2{%
2126   \forest@nodehandleby@name@nodewalk@or@bracket{#2}%
2127   \ifcase\forest@dynamics@copyhow\relax\or
2128     \forest@tree@copy{\forest@last@node}\forest@last@node
2129   \or
2130     \forest@node@copy{\forest@last@node}\forest@last@node

```

```

2131 \fi
2132 \forest@node@ifnameddefined{forest@last@node}{%
2133 \forest@preto{forest@last@node}{delay}
2134 {for id={forest@node@Nametoid{forest@last@node}}{alias=forest@last@node},}%
2135 }{}%
2136 \forest@havedelayedoptionstrue
2137 \edef\forest@marshal{%
2138 \noexpand\apptotoks\noexpand\forest@do@ dynamics{%
2139 \noexpand#1{\forest@cn}{\forest@last@node}}%
2140 }\forest@marshal
2141 }
2142 \forestset{%
2143 create/.code={\forest@create@node{#1}},
2144 append/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ dynamics\forest@append@node{#1}},
2145 prepend/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ dynamics\forest@prepend@node{#1}},
2146 insert after/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ dynamics\forest@insertafter@node{#1}},
2147 insert before/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ dynamics\forest@insertbefore@node{#1}},
2148 append'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ dynamics\forest@append@node{#1}},
2149 prepend'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ dynamics\forest@prepend@node{#1}},
2150 insert after'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ dynamics\forest@insertafter@node{#1}},
2151 insert before'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ dynamics\forest@insertbefore@node{#1}},
2152 append''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ dynamics\forest@append@node{#1}},
2153 prepend''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ dynamics\forest@prepend@node{#1}},
2154 insert after''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ dynamics\forest@insertafter@node{#1}},
2155 insert before''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ dynamics\forest@insertbefore@node{#1}},
2156 remove/.code={%
2157 \pgfkeysalso{alias=forest@last@node}%
2158 \expandafter\apptotoks\expandafter\forest@do@ dynamics\expandafter{%
2159 \expandafter\forest@remove@node\expandafter{\forest@cn}}%
2160 },
2161 set root/.code={%
2162 \forest@nodehandleby@name@nodewalk@or@bracket{#1}%
2163 \edef\forest@marshal{%
2164 \noexpand\apptotoks\noexpand\forest@do@ dynamics{%
2165 \def\noexpand\forest@root{\forest@last@node}%
2166 }%
2167 }\forest@marshal
2168 },
2169 replace by/.code={\forest@replaceby@code{#1}{insert after}},
2170 replace by'/.code={\forest@replaceby@code{#1}{insert after'}},
2171 replace by''/.code={\forest@replaceby@code{#1}{insert after''}},
2172 }
2173 \def\forest@replaceby@code#1#2{%#1=node spec,#2=insert after['']['']}
2174 \ifnum\forestove{@parent}=0
2175 \pgfkeysalso{set root={#1}}%
2176 \else
2177 \pgfkeysalso{alias=forest@last@node,#2={#1}}%
2178 \eapptotoks\forest@do@ dynamics{%
2179 \noexpand\ifnum\noexpand\forest@ove{\forest@cn}{@parent}=\forestove{@parent}
2180 \noexpand\forest@remove@node{\forest@cn}%
2181 \noexpand\fi
2182 }%
2183 \fi
2184 }

```

## 10 Stages

```

2185 \forestset{
2186 stages/.style={
2187 before typesetting nodes hook,
2188 typeset nodes,

```

```

2189     after typesetting nodes hook,
2190     before packing hook,
2191     pack,
2192     after packing hook,
2193     before computing xy hook,
2194     compute xy,
2195     after computing xy hook,
2196     before drawing tree hook,
2197     draw tree,
2198     after drawing tree hook,
2199 },
2200 before typesetting nodes hook/.style={process keylist=before typesetting nodes},
2201 after typesetting nodes hook/.style={},
2202 before packing hook/.style={process keylist=before packing},
2203 after packing hook/.style={},
2204 before computing xy hook/.style={process keylist=before computing xy},
2205 after computing xy hook/.style={},
2206 before drawing tree hook/.style={process keylist=before drawing tree},
2207 after drawing tree hook/.style={},
2208 process keylist/.code={\forest@process@hook@keylist{#1}},
2209 declare keylist={given options}{},
2210 declare keylist={before typesetting nodes}{},
2211 declare keylist={before packing}{},
2212 declare keylist={before computing xy}{},
2213 declare keylist={before drawing tree}{},
2214 declare keylist={delay}{},
2215 delay/.append code={\forest@havedelayedoptionstrue},
2216 typeset nodes/.code={%
2217   \forest@drawtree@preservenodeboxes@false
2218   \forest@node@Foreach{\forest@root}{\forest@node@typeset}},
2219 typeset nodes'/.code={%
2220   \forest@drawtree@preservenodeboxes@true
2221   \forest@node@Foreach{\forest@root}{\forest@node@typeset}},
2222 typeset node/.code={%
2223   \forest@drawtree@preservenodeboxes@false
2224   \forest@node@typeset
2225 },
2226 pack/.code={%
2227   \forest@fornode{\forest@root}{\forest@pack}},
2228 compute xy/.code={%
2229   \forest@fornode{\forest@root}{\forest@node@computeabsolutepositions}},
2230 draw tree box/.store in=\forest@drawtreebox,
2231 draw tree box,
2232 draw tree/.code={%
2233   \forest@drawtree@preservenodeboxes@false
2234   \forest@fornode{\forest@root}{\forest@node@drawtree}%
2235 },
2236 draw tree'/.code={%
2237   \forest@drawtree@preservenodeboxes@true
2238   \forest@fornode{\forest@root}{\forest@node@drawtree}%
2239 },
2240 }
2241 \newtoks\forest@do@dynamics
2242 \newif\ifforest@havedelayedoptions
2243 \def\forest@process@hook@keylist#1{%
2244   \forest@loopa
2245   \forest@havedelayedoptionsfalse
2246   \forest@do@dynamics={}%
2247   \forest@fornode{\forest@root}{\forest@process@hook@keylist@{#1}}%
2248   \expandafter\ifstrempy\expandafter{\the\forest@do@dynamics}{}%
2249   \the\forest@do@dynamics

```

```

2250 \forest@node@Compute@numeric@ts@info{\forest@root}%
2251 \forest@havedelayedoptionstrue
2252 }%
2253 \ifforest@havedelayedoptions
2254 \forest@node@Foreach{\forest@root}{%
2255 \forest@toget{delay}\forest@temp@delayed
2256 \forest@tolet{#1}\forest@temp@delayed
2257 \forest@toset{delay}{}%
2258 }%
2259 \forest@repeata
2260 }
2261 \def\forest@process@hook@keylist@#1{%
2262 \forest@node@foreach{%
2263 \forest@toget{#1}\forest@temp@keys
2264 \ifdefvoid\forest@temp@keys{}{%
2265 \forest@toset{#1}{}%
2266 \expandafter\forest@set\expandafter{\forest@temp@keys}%
2267 }%
2268 }%
2269 }

```

## 10.1 Typesetting nodes

```

2270 \def\forest@node@typeset{%
2271 \let\forest@next\forest@node@typeset@
2272 \forest@ifdefined{box}{%
2273 \ifforest@drawtree@preservenodeboxes@
2274 \let\forest@next\relax
2275 \fi
2276 }{%
2277 \locbox\forest@temp@box
2278 \forest@tolet{box}\forest@temp@box
2279 }%
2280 \forest@next
2281 }
2282 \newbox\forest@box
2283 \def\forest@node@typeset@{%
2284 \forest@toget{name}\forest@nodename
2285 \edef\forest@temp@nodeformat{\forest@tove{node format}}%
2286 \gdef\forest@smuggle{}%
2287 \setbox0=\hbox{%
2288 \begin{tikzpicture}%
2289 \pgfpositionnodelater{\forest@positionnodelater@save}%
2290 \forest@temp@nodeformat
2291 \pgfinterruptpath
2292 \pgfpointanchor{not yet positioned@\forest@nodename}{forest@computenodeboundary}%
2293 \endpgfinterruptpath
2294 %\forest@compute@node@boundary\forest@temp
2295 %\xappto\forest@smuggle{\noexpand\forest@toset{boundary}{\expandonce\forest@temp}}%
2296 \if\relax\forest@tove{parent anchor}\relax
2297 \pgfpointanchor{not yet positioned@\forest@nodename}{center}%
2298 \else
2299 \pgfpointanchor{not yet positioned@\forest@nodename}{\forest@tove{parent anchor}}%
2300 \fi
2301 \xappto\forest@smuggle{%
2302 \noexpand\forest@toset{parent@anchor}{%
2303 \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
2304 \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
2305 \if\relax\forest@tove{child anchor}\relax
2306 \pgfpointanchor{not yet positioned@\forest@nodename}{center}%
2307 \else

```

```

2308     \pgfpointanchor{not yet positioned@\forest@nodename}{\forestve{child anchor}}}%
2309 \fi
2310 \xappto\forest@smuggle{%
2311     \noexpand\forestoeset{child@anchor}{%
2312         \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
2313         \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
2314 \if\relax\forestve{anchor}\relax
2315     \pgfpointanchor{not yet positioned@\forest@nodename}{center}%
2316 \else
2317     \pgfpointanchor{not yet positioned@\forest@nodename}{\forestve{anchor}}}%
2318 \fi
2319 \xappto\forest@smuggle{%
2320     \noexpand\forestoeset{@anchor}{%
2321         \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
2322         \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
2323 \end{tikzpicture}%
2324 }%
2325 \setbox\forestve{box}=\box\forest@box % smuggle the box
2326 \forestolet{boundary}\forest@global@boundary
2327 \forest@smuggle % ... and the rest
2328 }
2329 \forestset{
2330     declare readonly dimen={min x},
2331     declare readonly dimen={min y},
2332     declare readonly dimen={max x},
2333     declare readonly dimen={max y},
2334 }
2335 \def\forest@patch@enormouscoordinateboxbounds@plus#1{%
2336     \expandafter\ifstrequal\expandafter{#1}{16000.0pt}{\def#1{0.0pt}}}%
2337 }
2338 \def\forest@patch@enormouscoordinateboxbounds@minus#1{%
2339     \expandafter\ifstrequal\expandafter{#1}{-16000.0pt}{\def#1{0.0pt}}}%
2340 }
2341 \def\forest@positionnodelater@save{%
2342     \global\setbox\forest@box=\box\pgfpositionnodelaterbox
2343     \xappto\forest@smuggle{\noexpand\forestoeset{later@name}{\pgfpositionnodelatername}}%
2344     % a bug in pgf? ---well, here's a patch
2345     \forest@patch@enormouscoordinateboxbounds@plus\pgfpositionnodelaterminx
2346     \forest@patch@enormouscoordinateboxbounds@plus\pgfpositionnodelaterminy
2347     \forest@patch@enormouscoordinateboxbounds@minus\pgfpositionnodelatermaxx
2348     \forest@patch@enormouscoordinateboxbounds@minus\pgfpositionnodelatermaxy
2349     % end of patch
2350     \xappto\forest@smuggle{\noexpand\forestoeset{min x}{\pgfpositionnodelaterminx}}%
2351     \xappto\forest@smuggle{\noexpand\forestoeset{min y}{\pgfpositionnodelaterminy}}%
2352     \xappto\forest@smuggle{\noexpand\forestoeset{max x}{\pgfpositionnodelatermaxx}}%
2353     \xappto\forest@smuggle{\noexpand\forestoeset{max y}{\pgfpositionnodelatermaxy}}%
2354 }
2355 \def\forest@node@forest@positionnodelater@restore{%
2356     \ifforest@drawtree@preservenodeboxes@
2357         \let\forest@boxorcopy\copy
2358     \else
2359         \let\forest@boxorcopy\box
2360     \fi
2361     \forestoget{box}\forest@temp
2362     \setbox\pgfpositionnodelaterbox=\forest@boxorcopy\forest@temp
2363     \edef\pgfpositionnodelatername{\forestve{later@name}}%
2364     \edef\pgfpositionnodelaterminx{\forestve{min x}}%
2365     \edef\pgfpositionnodelaterminy{\forestve{min y}}%
2366     \edef\pgfpositionnodelatermaxx{\forestve{max x}}%
2367     \edef\pgfpositionnodelatermaxy{\forestve{max y}}%
2368 }

```



## 10.2 Packing

Method `pack` should be called to calculate the positions of descendant nodes; the positions are stored in attributes `l` and `s` of these nodes, in a level/sibling coordinate system with origin at the parent's anchor.

```

2369 \def\forest@pack{%
2370   \forest@pack@computetiers
2371   \forest@pack@computegrowthuniformity
2372   \forest@@pack
2373 }
2374 \def\forest@@pack{%
2375   \ifnum\forestove{n children}>0
2376     \ifnum\forestove{uniform growth}>0
2377       \forest@pack@level@uniform
2378       \forest@pack@aligntiers@ofsubtree
2379       \forest@pack@sibling@uniform@recursive
2380     \else
2381       \forest@node@foreachchild{\forest@@pack}%
2382       \forest@pack@level@nonuniform
2383       \forest@pack@aligntiers
2384       \forest@pack@sibling@uniform@applyreversed
2385     \fi
2386   \fi
2387 }

```

Compute growth uniformity for the subtree. A tree grows uniformly if all its branching nodes have the same `grow`.

```

2388 \def\forest@pack@computegrowthuniformity{%
2389   \forest@node@foreachchild{\forest@pack@computegrowthuniformity}%
2390   \edef\forest@pack@cgu@uniformity{%
2391     \ifnum\forestove{n children}=0
2392     2\else 1\fi
2393   }%
2394   \forestoget{grow}\forest@pack@cgu@parentgrow
2395   \forest@node@foreachchild{%
2396     \ifnum\forestove{uniform growth}=0
2397       \def\forest@pack@cgu@uniformity{0}%
2398     \else
2399       \ifnum\forestove{uniform growth}=1
2400         \ifnum\forestove{grow}=\forest@pack@cgu@parentgrow\relax\else
2401           \def\forest@pack@cgu@uniformity{0}%
2402         \fi
2403       \fi
2404     \fi
2405   }%
2406   \forestolet{uniform growth}\forest@pack@cgu@uniformity
2407 }

```

Pack children in the level dimension in a uniform tree.

```

2408 \def\forest@pack@level@uniform{%
2409   \let\forest@plu@minchildl\relax
2410   \forestoget{grow}\forest@plu@grow
2411   \forest@node@foreachchild{%
2412     \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
2413     \advance\pgf@xa\forestove{l}\relax
2414     \ifx\forest@plu@minchildl\relax
2415       \edef\forest@plu@minchildl{\the\pgf@xa}%
2416     \else
2417       \ifdim\pgf@xa<\forest@plu@minchildl\relax
2418         \edef\forest@plu@minchildl{\the\pgf@xa}%
2419       \fi
2420     \fi

```

```

2421 }%
2422 \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
2423 \pgfutil@tempdima=\pgf@xb\relax
2424 \advance\pgfutil@tempdima -\forest@plu@minchildl\relax
2425 \advance\pgfutil@tempdima \forestove{1 sep}\relax
2426 \ifdim\pgfutil@tempdima>0pt
2427   \forest@node@foreachchild{%
2428     \forestoeset{1}{\the\dimexpr\forestove{1}+\the\pgfutil@tempdima}%
2429   }%
2430 \fi
2431 \forest@node@foreachchild{%
2432   \ifnum\forestove{n children}>0
2433     \forest@pack@level@uniform
2434   \fi
2435 }%
2436 }

```

Pack children in the level dimension in a non-uniform tree. (Expects the children to be fully packed.)

```

2437 \def\forest@pack@level@nonuniform{%
2438   \let\forest@plu@minchildl\relax
2439   \forestoget{grow}\forest@plu@grow
2440   \forest@node@foreachchild{%
2441     \forest@node@getedge{negative}{\forest@plu@grow}{\forest@plnu@negativechilddedge}%
2442     \forest@node@getedge{positive}{\forest@plu@grow}{\forest@plnu@positivechilddedge}%
2443     \def\forest@plnu@childdedge{\forest@plnu@negativechilddedge\forest@plnu@positivechilddedge}%
2444     \forest@path@getboundingrectangle@ls\forest@plnu@childdedge{\forest@plu@grow}%
2445     \advance\pgf@xa\forestove{1}\relax
2446     \ifx\forest@plu@minchildl\relax
2447       \edef\forest@plu@minchildl{\the\pgf@xa}%
2448     \else
2449       \ifdim\pgf@xa<\forest@plu@minchildl\relax
2450         \edef\forest@plu@minchildl{\the\pgf@xa}%
2451       \fi
2452     \fi
2453   }%
2454   \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
2455   \pgfutil@tempdima=\pgf@xb\relax
2456   \advance\pgfutil@tempdima -\forest@plu@minchildl\relax
2457   \advance\pgfutil@tempdima \forestove{1 sep}\relax
2458   \ifdim\pgfutil@tempdima>0pt
2459     \forest@node@foreachchild{%
2460       \forestoeset{1}{\the\dimexpr\the\pgfutil@tempdima+\forestove{1}}%
2461     }%
2462   \fi
2463 }

```

Align tiers.

```

2464 \def\forest@pack@aligntiers{%
2465   \forestoget{grow}\forest@temp@parentgrow
2466   \forestoget{@tiers}\forest@temp@tiers
2467   \forlistloop\forest@pack@aligntier@\forest@temp@tiers
2468 }
2469 \def\forest@pack@aligntiers@ofsubtree{%
2470   \forest@node@foreach{\forest@pack@aligntiers}%
2471 }
2472 \def\forest@pack@aligntiers@computeabs1{%
2473   \forestoleto{abs@1}{1}%
2474   \forest@node@foreachdescendant{\forest@pack@aligntiers@computeabs1}%
2475 }
2476 \def\forest@pack@aligntiers@computeabs1@{%
2477   \forestoeset{abs@1}{\the\dimexpr\forestove{1}+\forestove{\forestove{@parent}}{abs@1}}%
2478 }

```

```

2479 \def\forest@pack@alignntier@#1{%
2480   \forest@pack@alignntiers@computeabs1
2481   \pgfutil@tempdima=-\maxdimen\relax
2482   \def\forest@temp@currenttier{#1}%
2483   \forest@node@foreach{%
2484     \forestoget{tier}\forest@temp@tier
2485     \ifx\forest@temp@currenttier\forest@temp@tier
2486       \ifdim\pgfutil@tempdima<\forestove{abs@1}\relax
2487         \pgfutil@tempdima=\forestove{abs@1}\relax
2488       \fi
2489     \fi
2490   }%
2491   \ifdim\pgfutil@tempdima=-\maxdimen\relax\else
2492     \forest@node@foreach{%
2493       \forestoget{tier}\forest@temp@tier
2494       \ifx\forest@temp@currenttier\forest@temp@tier
2495         \forestoeset{1}{\the\dimexpr\pgfutil@tempdima-\forestove{abs@1}+\forestove{1}}%
2496       \fi
2497     }%
2498   \fi
2499 }

```

Pack children in the sibling dimension in a uniform tree: recursion.

```

2500 \def\forest@pack@sibling@uniform@recursive{%
2501   \forest@node@foreachchild{\forest@pack@sibling@uniform@recursive}%
2502   \forest@pack@sibling@uniform@applyreversed
2503 }

```

Pack children in the sibling dimension in a uniform tree: applyreversed.

```

2504 \def\forest@pack@sibling@uniform@applyreversed{%
2505   \ifnum\forestove{n children}>1
2506     \ifnum\forestove{reversed}=0
2507       \pack@sibling@uniform@main{first}{last}{next}{previous}%
2508     \else
2509       \pack@sibling@uniform@main{last}{first}{previous}{next}%
2510     \fi
2511   \fi
2512 }

```

Pack children in the sibling dimension in a uniform tree: the main routine.

```

2513 \def\pack@sibling@uniform@main#1#2#3#4{%

```

Loop through the children. At each iteration, we compute the distance between the negative edge of the current child and the positive edge of the block of the previous children, and then set the `s` attribute of the current child accordingly.

We start the loop with the second (to last) child, having initialized the positive edge of the previous children to the positive edge of the first child.

```

2514   \forestoget{@#1}\forest@child
2515   \edef\forest@temp{%
2516     \noexpand\forest@fornode{\forestove{@#1}}{%
2517       \noexpand\forest@node@getedge
2518         {positive}
2519         {\forestove{grow}}
2520       \noexpand\forest@temp@edge
2521     }%
2522   }\forest@temp
2523   \forest@pack@pgfpoint@childsp@position\forest@child
2524   \let\forest@previous@positive@edge\pgfutil@empty
2525   \forest@extendpath\forest@previous@positive@edge\forest@temp@edge{%
2526     \forest0get{\forest@child}{@#3}\forest@child

```

Loop until the current child is the null node.

```

2527 \edef\forest@previous@child@s{0pt}%
2528 \forest@loopb
2529 \unless\ifnum\forest@child=0
    Get the negative edge of the child.
2530 \edef\forest@temp{%
2531 \noexpand\forest@for@node{\forest@child}{%
2532 \noexpand\forest@node@getedge
2533 {negative}
2534 {\forest@ve{grow}}
2535 \noexpand\forest@temp@edge
2536 }%
2537 }\forest@temp
    Set \pgf@x and \pgf@y to the position of the child (in the coordinate system of this node).
2538 \forest@pack@pgfpoint@child@position\forest@child
    Translate the edge of the child by the child's position.
2539 \let\forest@child@negative@edge\pgfutil@empty
2540 \forest@extendpath\forest@child@negative@edge\forest@temp@edge{}%
    Setup the grow line: the angle is given by this node's grow attribute.
2541 \forest@setupgrowline{\forest@ve{grow}}%
    Get the distance (wrt the grow line) between the positive edge of the previous children and the negative
    edge of the current child. (The distance can be negative!)
2542 \forest@distance@between@edge@paths\forest@previous@positive@edge\forest@child@negative@edge\forest@csdis
    If the distance is \relax, the projections of the edges onto the grow line don't overlap: do nothing.
    Otherwise, shift the current child so that its distance to the block of previous children is s sep.
2543 \ifx\forest@csdistance\relax
2544 %\forest@set{\forest@child}{s}{\forest@previous@child@s}%
2545 \else
2546 \advance\pgfutil@tempdimb-\forest@csdistance\relax
2547 \advance\pgfutil@tempdimb\forest@ve{s sep}\relax
2548 \forest@set{\forest@child}{s}{\the\dimexpr\forest@ve{s}-\forest@csdistance+\forest@ve{s sep}}%
2549 \fi
    Retain monotonicity (is this ok?). (This problem arises when the adjacent children's l are too far apart.)
2550 \ifdim\forest@ve{\forest@child}{s}<\forest@previous@child@s\relax
2551 \forest@set{\forest@child}{s}{\forest@previous@child@s}%
2552 \fi
    Prepare for the next iteration: add the current child's positive edge to the positive edge of the previous
    children, and set up the next current child.
2553 \forest@get{\forest@child}{s}\forest@child@s
2554 \edef\forest@previous@child@s{\forest@child@s}%
2555 \edef\forest@temp{%
2556 \noexpand\forest@for@node{\forest@child}{%
2557 \noexpand\forest@node@getedge
2558 {positive}
2559 {\forest@ve{grow}}
2560 \noexpand\forest@temp@edge
2561 }%
2562 }\forest@temp
2563 \forest@pack@pgfpoint@child@position\forest@child
2564 \forest@extendpath\forest@previous@positive@edge\forest@temp@edge{}%
2565 \forest@getpositivetightedgeofpath\forest@previous@positive@edge\forest@previous@positive@edge
2566 \forest@get{\forest@child}{@#3}\forest@child
2567 \forest@repeatb
    Shift the position of all children to achieve the desired alignment of the parent and its children.
2568 \csname forest@calign\forest@ve{calign}\endcsname
2569 }

```

Get the position of child #1 in the current node, in node's l-s coordinate system.

```

2570 \def\forest@pack@pgfpoint@childposition#1{%
2571   {%
2572     \pgftransformreset
2573     \pgftransformrotate{\forestove{grow}}%
2574     \forest@for node{#1}{%
2575       \pgfpointtransformed{\pgfqpoint{\forestove{1}}{\forestove{s}}}%
2576     }%
2577   }%
2578 }

```

Get the position of the node in the grow (#1)-rotated coordinate system.

```

2579 \def\forest@pack@pgfpoint@positioningrow#1{%
2580   {%
2581     \pgftransformreset
2582     \pgftransformrotate{#1}%
2583     \pgfpointtransformed{\pgfqpoint{\forestove{1}}{\forestove{s}}}%
2584   }%
2585 }

```

Child alignment.

```

2586 \def\forest@calign@s@shift#1{%
2587   \pgfutil@tempdima=#1\relax
2588   \forest@node@foreach child{%
2589     \forestoeset{s}{\the\dimexpr\forestove{s}+\pgfutil@tempdima}%
2590   }%
2591 }
2592 \def\forest@calign@child{%
2593   \forest@calign@s@shift{-\forestOve{\forest@node@nornbarthchildid{\forestove{calign primary child}}}{s}}%
2594 }
2595 \csdef{forest@calign@child edge}{%
2596   {%
2597     \edef\forest@temp@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2598     \pgftransformreset
2599     \pgftransformrotate{\forestove{grow}}%
2600     \pgfpointtransformed{\pgfqpoint{\forestOve{\forest@temp@child}{1}}{\forestOve{\forest@temp@child}{s}}}%
2601     \pgf@xa=\pgf@x\relax\pgf@ya=\pgf@y\relax
2602     \forestOve{\forest@temp@child}{child@anchor}%
2603     \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2604     \forestove{parent@anchor}%
2605     \advance\pgf@xa-\pgf@x\relax\advance\pgf@ya-\pgf@y\relax
2606     \edef\forest@marshal{%
2607       \noexpand\pgftransformreset
2608       \noexpand\pgftransformrotate{-\forestove{grow}}%
2609       \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
2610     }\forest@marshal
2611   }%
2612   \forest@calign@s@shift{\the\dimexpr-\the\pgf@y}%
2613 }
2614 \csdef{forest@calign@midpoint}{%
2615   \forest@calign@s@shift{\the\dimexpr Opt -%
2616     (\forestOve{\forest@node@nornbarthchildid{\forestove{calign primary child}}}{s}%
2617     +\forestOve{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}{s}%
2618     )/2\relax
2619   }%
2620 }
2621 \csdef{forest@calign@edge midpoint}{%
2622   {%
2623     \edef\forest@temp@firstchild{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2624     \edef\forest@temp@secondchild{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
2625     \pgftransformreset

```

```

2626 \pgftransformrotate{\forestove{grow}}%
2627 \pgfpointtransformed{\pgfqpoint{\forestOve{\forest@temp@firstchild}{1}}{\forestOve{\forest@temp@firstchild}}%
2628 \pgf@xa=\pgf@x\relax\pgf@ya=\pgf@y\relax
2629 \forestOve{\forest@temp@firstchild}{child@anchor}%
2630 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2631 \edef\forest@marshal{%
2632 \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\forestOve{\forest@temp@secondchild}{1}}{\forestOve{\forest@temp@secondchild}}%
2633 }\forest@marshal
2634 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2635 \forestOve{\forest@temp@secondchild}{child@anchor}%
2636 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2637 \divide\pgf@xa2 \divide\pgf@ya2
2638 \edef\forest@marshal{%
2639 \noexpand\pgftransformreset
2640 \noexpand\pgftransformrotate{-\forestove{grow}}%
2641 \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
2642 }\forest@marshal
2643 }%
2644 \forest@calign@s@shift{\the\dimexpr-\the\pgf@y}%
2645 }

```

Aligns the children to the center of the angles given by the options `calign first angle` and `calign second angle` and spreads them additionally if needed to fill the whole space determined by the option. The version `fixed angles` calculates the angles between node anchors; the version `fixes edge angles` calculates the angles between the node edges.

```

2646 \csdef{forest@calign@fixed angles}{%
2647 \edef\forest@ca@first@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2648 \edef\forest@ca@second@child{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
2649 \ifnum\forestove{reversed}=1
2650 \let\forest@temp\forest@ca@first@child
2651 \let\forest@ca@first@child\forest@ca@second@child
2652 \let\forest@ca@second@child\forest@temp
2653 \fi
2654 \forestOget{\forest@ca@first@child}{1}\forest@ca@first@l
2655 \forestOget{\forest@ca@second@child}{1}\forest@ca@second@l
2656 \pgfmathsetlengthmacro\forest@ca@desired@s@distance{%
2657 tan(\forestove{calign secondary angle})*\forest@ca@second@l
2658 -tan(\forestove{calign primary angle})*\forest@ca@first@l
2659 }%
2660 \forestOget{\forest@ca@first@child}{s}\forest@ca@first@s
2661 \forestOget{\forest@ca@second@child}{s}\forest@ca@second@s
2662 \pgfmathsetlengthmacro\forest@ca@actual@s@distance{%
2663 \forest@ca@second@s-\forest@ca@first@s}%
2664 \ifdim\forest@ca@desired@s@distance>\forest@ca@actual@s@distance\relax
2665 \ifdim\forest@ca@actual@s@distance=0pt
2666 \pgfmathsetlength\pgfutil@tempdima{tan(\forestove{calign primary angle})*\forest@ca@second@l}%
2667 \pgfmathsetlength\pgfutil@tempdimb{\forest@ca@desired@s@distance/(\forestove{n children}-1)}%
2668 \forest@node@foreachchild{%
2669 \forestoeset{s}{\the\pgfutil@tempdima}%
2670 \advance\pgfutil@tempdima\pgfutil@tempdimb
2671 }%
2672 \def\forest@calign@anchor{0pt}%
2673 \else
2674 \pgfmathsetmacro\forest@ca@ratio{%
2675 \forest@ca@desired@s@distance/\forest@ca@actual@s@distance}%
2676 \forest@node@foreachchild{%
2677 \pgfmathsetlengthmacro\forest@temp{\forest@ca@ratio*\forestove{s}}%
2678 \forestolet{s}\forest@temp
2679 }%
2680 \pgfmathsetlengthmacro\forest@calign@anchor{%
2681 -tan(\forestove{calign primary angle})*\forest@ca@first@l}%

```

```

2682 \fi
2683 \else
2684 \ifdim\forest@ca@desired@s@distance<\forest@ca@actual@s@distance\relax
2685 \pgfmathsetlengthmacro\forest@ca@ratio{%
2686 \forest@ca@actual@s@distance/\forest@ca@desired@s@distance}%
2687 \forest@node@foreachchild{%
2688 \pgfmathsetlengthmacro\forest@temp{\forest@ca@ratio*\forestove{1}}%
2689 \forestolet{1}\forest@temp
2690 }%
2691 \forestOget{\forest@ca@first@child}{1}\forest@ca@first@l
2692 \pgfmathsetlengthmacro\forest@calign@anchor{%
2693 -tan(\forestove{calign primary angle})*\forest@ca@first@l}%
2694 \fi
2695 \fi
2696 \forest@calign@s@shift{-\forest@calign@anchor}%
2697 }
2698 \csdef{forest@calign@fixed edge angles}{%
2699 \edef\forest@ca@first@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2700 \edef\forest@ca@second@child{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
2701 \ifnum\forestove{reversed}=1
2702 \let\forest@temp\forest@ca@first@child
2703 \let\forest@ca@first@child\forest@ca@second@child
2704 \let\forest@ca@second@child\forest@temp
2705 \fi
2706 \forestOget{\forest@ca@first@child}{1}\forest@ca@first@l
2707 \forestOget{\forest@ca@second@child}{1}\forest@ca@second@l
2708 \forestOget{parent@anchor}\forest@ca@parent@anchor
2709 \forest@ca@parent@anchor
2710 \edef\forest@ca@parent@anchor@s{\the\pgf@x}%
2711 \edef\forest@ca@parent@anchor@l{\the\pgf@y}%
2712 \forestOget{\forest@ca@first@child}{child@anchor}\forest@ca@first@child@anchor
2713 \forest@ca@first@child@anchor
2714 \edef\forest@ca@first@child@anchor@s{\the\pgf@x}%
2715 \edef\forest@ca@first@child@anchor@l{\the\pgf@y}%
2716 \forestOget{\forest@ca@second@child}{child@anchor}\forest@ca@second@child@anchor
2717 \forest@ca@second@child@anchor
2718 \edef\forest@ca@second@child@anchor@s{\the\pgf@x}%
2719 \edef\forest@ca@second@child@anchor@l{\the\pgf@y}%
2720 \pgfmathsetlengthmacro\forest@ca@desired@second@edge@s{tan(\forestove{calign secondary angle})*%
2721 (\forest@ca@second@l-\forest@ca@second@child@anchor@l+\forest@ca@parent@anchor@l)}%
2722 \pgfmathsetlengthmacro\forest@ca@desired@first@edge@s{tan(\forestove{calign primary angle})*%
2723 (\forest@ca@first@l-\forest@ca@first@child@anchor@l+\forest@ca@parent@anchor@l)}%
2724 \pgfmathsetlengthmacro\forest@ca@desired@s@distance{\forest@ca@desired@second@edge@s-\forest@ca@desired@fir
2725 \forestOget{\forest@ca@first@child}{s}\forest@ca@first@s
2726 \forestOget{\forest@ca@second@child}{s}\forest@ca@second@s
2727 \pgfmathsetlengthmacro\forest@ca@actual@s@distance{%
2728 \forest@ca@second@s+\forest@ca@second@child@anchor@s
2729 -\forest@ca@first@s-\forest@ca@first@child@anchor@s}%
2730 \ifdim\forest@ca@desired@s@distance>\forest@ca@actual@s@distance\relax
2731 \ifdim\forest@ca@actual@s@distance=0pt
2732 \forestOget{n children}\forest@temp@n@children
2733 \forest@node@foreachchild{%
2734 \forestOget{child@anchor}\forest@temp@child@anchor
2735 \forest@temp@child@anchor
2736 \edef\forest@temp@child@anchor@s{\the\pgf@x}%
2737 \pgfmathsetlengthmacro\forest@temp{%
2738 \forest@ca@desired@first@edge@s+(\forestove{n}-1)*\forest@ca@desired@s@distance/(\forest@temp@n@chi
2739 \forestolet{s}\forest@temp
2740 }%
2741 \def\forest@calign@anchor{0pt}%
2742 \else

```

```

2743 \pgfmathsetmacro\forest@ca@ratio{%
2744 \forest@ca@desired@s@distance/\forest@ca@actual@s@distance}%
2745 \forest@node@foreachchild{%
2746 \forest@toget{child@anchor}\forest@temp@child@anchor
2747 \forest@temp@child@anchor
2748 \edef\forest@temp@child@anchor@s{\the\pgf@x}%
2749 \pgfmathsetlengthmacro\forest@temp{%
2750 \forest@ca@ratio*(%
2751 \forest@ve{s}-\forest@ca@first@s
2752 +\forest@temp@child@anchor@s-\forest@ca@first@child@anchor@s)%
2753 +\forest@ca@first@s
2754 +\forest@ca@first@child@anchor@s-\forest@temp@child@anchor@s}%
2755 \forest@tolet{s}\forest@temp
2756 }%
2757 \pgfmathsetlengthmacro\forest@calign@anchor{%
2758 -tan(\forest@ve{calign primary angle})*(\forest@ca@first@l-\forest@ca@first@child@anchor@l+\forest@ca
2759 +\forest@ca@first@child@anchor@s-\forest@ca@parent@anchor@s
2760 }%
2761 \fi
2762 \else
2763 \ifdim\forest@ca@desired@s@distance<\forest@ca@actual@s@distance\relax
2764 \pgfmathsetlengthmacro\forest@ca@ratio{%
2765 \forest@ca@actual@s@distance/\forest@ca@desired@s@distance}%
2766 \forest@node@foreachchild{%
2767 \forest@toget{child@anchor}\forest@temp@child@anchor
2768 \forest@temp@child@anchor
2769 \edef\forest@temp@child@anchor@l{\the\pgf@y}%
2770 \pgfmathsetlengthmacro\forest@temp{%
2771 \forest@ca@ratio*(%
2772 \forest@ve{l}+\forest@ca@parent@anchor@l-\forest@temp@child@anchor@l)
2773 -\forest@ca@parent@anchor@l+\forest@temp@child@anchor@l}%
2774 \forest@tolet{l}\forest@temp
2775 }%
2776 \forest@toget{\forest@ca@first@child}{l}\forest@ca@first@l
2777 \pgfmathsetlengthmacro\forest@calign@anchor{%
2778 -tan(\forest@ve{calign primary angle})*(\forest@ca@first@l+\forest@ca@parent@anchor@l-\forest@temp@ch
2779 +\forest@ca@first@child@anchor@s-\forest@ca@parent@anchor@s
2780 }%
2781 \fi
2782 \fi
2783 \forest@calign@s@shift{-\forest@calign@anchor}%
2784 }

```

Get edge: #1 = positive/negative, #2 = grow (in degrees), #3 = the control sequence receiving the resulting path. The edge is taken from the cache (attribute #1@edge@#2) if possible; otherwise, both positive and negative edge are computed and stored in the cache.

```

2785 \def\forest@node@getedge#1#2#3{%
2786 \forest@toget{#1@edge@#2}#3%
2787 \ifx#3\relax
2788 \forest@node@foreachchild{%
2789 \forest@node@getedge{#1}{#2}{\forest@temp@edge}%
2790 }%
2791 \forest@forthis{\forest@node@getedges{#2}}%
2792 \forest@toget{#1@edge@#2}#3%
2793 \fi
2794 }

```

Get edges. #1 = grow (in degrees). The result is stored in attributes negative@edge@#1 and positive@edge@#1. This method expects that the children's edges are already cached.

```

2795 \def\forest@node@getedges#1{%

```

Run the computation in a  $\text{\TeX}$  group.



```

2796  %%
      Setup the grow line.
2797  \forest@setupgrowline{#1}%
      Get the edge of the node itself.
2798  \ifnum\forest@ignore=0
2799  \forest@get{boundary}\forest@node@boundary
2800  \else
2801  \def\forest@node@boundary{%
2802  \fi
2803  \csname forest@getboth\forest@fit\edgesofpath\endcsname
2804  \forest@node@boundary\forest@negative@node@edge\forest@positive@node@edge
2805  \forest@let{negative@edge@#1}\forest@negative@node@edge
2806  \forest@let{positive@edge@#1}\forest@positive@node@edge
      Add the edges of the children.
2807  \get@edges@merge{negative}{#1}%
2808  \get@edges@merge{positive}{#1}%
2809  }%%
2810 }

      Merge the #1 (=negative or positive) edge of the node with #1 edges of the children. #2 = grow
      angle.
2811 \def\get@edges@merge#1#2{%
2812 \ifnum\forest@n children>0
2813 \forest@get{#1@edge@#2}\forest@node@edge

      Remember the node's parent anchor and add it to the path (for breaking).
2814 \forest@parent@anchor}%
2815 \edef\forest@getedge@pa@l{\the\pgf@x}%
2816 \edef\forest@getedge@pa@s{\the\pgf@y}%
2817 \eappto\forest@node@edge{\noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@pa@l}{\forest@getedge@pa@s}}

      Switch to this node's (l,s) coordinate system (origin at the node's anchor).
2818 \pgftransformreset
2819 \pgftransformrotate{\forest@grow}%

      Get the child's (cached) edge, translate it by the child's position, and add it to the path holding all
      edges. Also add the edge from parent to the child to the path. This gets complicated when the child
      and/or parent anchor is empty, i.e. automatic border: we can get self-intersecting paths. So we store all
      the parent-child edges to a safe place first, compute all the possible breaking points (i.e. all the points
      in node@edge path), and break the parent-child edges on these points.
2820 \def\forest@all@edges{%
2821 \forest@node@foreachchild{%
2822 \forest@get{#1@edge@#2}\forest@temp@edge
2823 \pgfpointtransformed{\pgfpoint{\forest@l}{\forest@s}}%
2824 \forest@extendpath\forest@node@edge\forest@temp@edge}%
2825 \ifnum\forest@ignore edge=0
2826 \pgfpointadd
2827 {\pgfpointtransformed{\pgfpoint{\forest@l}{\forest@s}}}%
2828 {\forest@child@anchor}}%
2829 \pgfgetlastxy{\forest@getedge@ca@l}{\forest@getedge@ca@s}%
2830 \eappto\forest@all@edges{%
2831 \noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@pa@l}{\forest@getedge@pa@s}%
2832 \noexpand\pgfsyssoftpath@linetotoken{\forest@getedge@ca@l}{\forest@getedge@ca@s}}%
2833 }%
2834 % this deals with potential overlap of the edges:
2835 \eappto\forest@node@edge{\noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@ca@l}{\forest@getedge@ca@s}}
2836 \fi
2837 }%
2838 \ifdefempty{\forest@all@edges}{%
2839 \pgfintersectionofpaths{\pgfsetpath\forest@all@edges}{\pgfsetpath\forest@node@edge}%

```

```

2840 \def\forest@edgenode@intersections{%
2841 \forest@merge@intersectionloop
2842 \eappto\forest@node@edge{\expandonce{\forest@all@edges}\expandonce{\forest@edgenode@intersections}}%
2843 }%

```

Process the path into an edge and store the edge.

```

2844 \csname forest@get#1\forestove{fit}edgeofpath\endcsname\forest@node@edge\forest@node@edge
2845 \forestolet{#1@edge@#2}\forest@node@edge
2846 \fi
2847 }
2848 \newloop\forest@merge@loop
2849 \def\forest@merge@intersectionloop{%
2850 \c@pgf@counta=0
2851 \forest@merge@loop
2852 \ifnum\c@pgf@counta<\pgfintersectionsolutions\relax
2853 \advance\c@pgf@counta1
2854 \pgfpointintersectionsolution{\the\c@pgf@counta}%
2855 \eappto\forest@edgenode@intersections{\noexpand\pgfsyssoftpath@movetotoken
2856 {\the\pgf@x}{\the\pgf@y}}%
2857 \forest@merge@repeat
2858 }

```

Get the bounding rectangle of the node (without descendants). #1 = grow.

```

2859 \def\forest@node@getboundingrectangle@ls#1{%
2860 \forestoget{boundary}\forest@node@boundary
2861 \forest@path@getboundingrectangle@ls\forest@node@boundary{#1}%
2862 }

```

Applies the current coordinate transformation to the points in the path #1. Returns via the current path (so that the coordinate transformation can be set up as local).

```

2863 \def\forest@pgfpathtransformed#1{%
2864 \forest@save@pgfsyssoftpath@tokendef
2865 \let\pgfsyssoftpath@movetotoken\forest@pgfpathtransformed@moveto
2866 \let\pgfsyssoftpath@linetotoken\forest@pgfpathtransformed@lineto
2867 \pgfsyssoftpath@setcurrentpath\pgfutil@empty
2868 #1%
2869 \forest@restore@pgfsyssoftpath@tokendef
2870 }
2871 \def\forest@pgfpathtransformed@moveto#1#2{%
2872 \forest@pgfpathtransformed@op\pgfsyssoftpath@moveto{#1}{#2}%
2873 }
2874 \def\forest@pgfpathtransformed@lineto#1#2{%
2875 \forest@pgfpathtransformed@op\pgfsyssoftpath@lineto{#1}{#2}%
2876 }
2877 \def\forest@pgfpathtransformed@op#1#2#3{%
2878 \pgfpointtransformed{\pgfpoint{#2}{#3}}%
2879 \edef\forest@temp{%
2880 \noexpand#1{\the\pgf@x}{\the\pgf@y}%
2881 }%
2882 \forest@temp
2883 }

```

### 10.2.1 Tiers

Compute tiers to be aligned at a node. The result is saved in attribute @tiers.

```

2884 \def\forest@pack@computetiers{%
2885 {%
2886 \forest@pack@tiers@getalltiersinsubtree
2887 \forest@pack@tiers@computetierhierarchy
2888 \forest@pack@tiers@findcontainers
2889 \forest@pack@tiers@raisecontainers
2890 \forest@pack@tiers@computeprocessingorder

```

```

2891 \gdef\forest@smuggle{%
2892 \forest@pack@tiers@write
2893 }%
2894 \forest@node@foreach{\forest@set{@tiers}}{}%
2895 \forest@smuggle
2896 }

```

Puts all tiers contained in the subtree into attribute `tiers`.

```

2897 \def\forest@pack@tiers@getalltiersinsubtree{%
2898 \ifnum\forest@ve{n children}>0
2899 \forest@node@foreachchild{\forest@pack@tiers@getalltiersinsubtree}%
2900 \fi
2901 \forest@get{tier}\forest@temp@mytier
2902 \def\forest@temp@mytiers{}%
2903 \ifdefempty\forest@temp@mytier{}{%
2904 \listadd\forest@temp@mytiers\forest@temp@mytier
2905 }%
2906 \ifnum\forest@ve{n children}>0
2907 \forest@node@foreachchild{%
2908 \forest@get{tiers}\forest@temp@tiers
2909 \forlistloop\forest@pack@tiers@forhandlerA\forest@temp@tiers
2910 }%
2911 \fi
2912 \forest@let{tiers}\forest@temp@mytiers
2913 }
2914 \def\forest@pack@tiers@forhandlerA#1{%
2915 \ifinlist{#1}\forest@temp@mytiers{}{%
2916 \listadd\forest@temp@mytiers{#1}%
2917 }%
2918 }

```

Compute a set of higher and lower tiers for each tier. Tier A is higher than tier B iff a node on tier A is an ancestor of a node on tier B.

```

2919 \def\forest@pack@tiers@computetierhierarchy{%
2920 \def\forest@tiers@ancestors{}%
2921 \forest@get{tiers}\forest@temp@mytiers
2922 \forlistloop\forest@pack@tiers@cth@init\forest@temp@mytiers
2923 \forest@pack@tiers@computetierhierarchy@
2924 }
2925 \def\forest@pack@tiers@cth@init#1{%
2926 \csdef\forest@tiers@higher@#1{}%
2927 \csdef\forest@tiers@lower@#1{}%
2928 }
2929 \def\forest@pack@tiers@computetierhierarchy@{%
2930 \forest@get{tier}\forest@temp@mytier
2931 \ifdefempty\forest@temp@mytier{}{%
2932 \forlistloop\forest@pack@tiers@forhandlerB\forest@tiers@ancestors
2933 \listadd\forest@tiers@ancestors\forest@temp@mytier
2934 }%
2935 \forest@node@foreachchild{%
2936 \forest@pack@tiers@computetierhierarchy@
2937 }%
2938 \forest@get{tier}\forest@temp@mytier
2939 \ifdefempty\forest@temp@mytier{}{%
2940 \forest@listedel\forest@tiers@ancestors\forest@temp@mytier
2941 }%
2942 }
2943 \def\forest@pack@tiers@forhandlerB#1{%
2944 \def\forest@temp@tier{#1}%
2945 \ifx\forest@temp@tier\forest@temp@mytier
2946 \PackageError{forest}{Circular tier hierarchy (tier \forest@temp@mytier)}{}%

```

```

2947 \fi
2948 \ifinlistcs{#1}{forest@tiers@higher@\forest@temp@mytier}{}%
2949 \listcsadd{forest@tiers@higher@\forest@temp@mytier}{#1}}%
2950 \xifinlistcs\forest@temp@mytier{forest@tiers@lower@#1}{}%
2951 \listcseadd{forest@tiers@lower@#1}{\forest@temp@mytier}}%
2952 }
2953 \def\forest@pack@tiers@findcontainers{%
2954 \foresttoget{tiers}\forest@temp@tiers
2955 \forlistloop\forest@pack@tiers@findcontainer\forest@temp@tiers
2956 }
2957 \def\forest@pack@tiers@findcontainer#1{%
2958 \def\forest@temp@tier{#1}%
2959 \foresttoget{tier}\forest@temp@mytier
2960 \ifx\forest@temp@tier\forest@temp@mytier
2961 \csedef{forest@tiers@container@#1}{\forest@cn}%
2962 \else\@escapeif{%
2963 \forest@pack@tiers@findcontainerA{#1}%
2964 }\fi%
2965 }
2966 \def\forest@pack@tiers@findcontainerA#1{%
2967 \c@pgf@counta=0
2968 \forest@node@foreachchild{%
2969 \foresttoget{tiers}\forest@temp@tiers
2970 \ifinlist{#1}\forest@temp@tiers{%
2971 \advance\c@pgf@counta 1
2972 \let\forest@temp@child\forest@cn
2973 }}%
2974 }%
2975 \ifnum\c@pgf@counta>1
2976 \csedef{forest@tiers@container@#1}{\forest@cn}%
2977 \else\@escapeif{% surely =1
2978 \forest@fornode{\forest@temp@child}{%
2979 \forest@pack@tiers@findcontainer{#1}%
2980 }%
2981 }\fi
2982 }
2983 \def\forest@pack@tiers@raisecontainers{%
2984 \foresttoget{tiers}\forest@temp@mytiers
2985 \forlistloop\forest@pack@tiers@rc@forhandlerA\forest@temp@mytiers
2986 }
2987 \def\forest@pack@tiers@rc@forhandlerA#1{%
2988 \edef\forest@tiers@temptier{#1}%
2989 \letcs\forest@tiers@containernodeoftier{forest@tiers@container@#1}%
2990 \letcs\forest@temp@lowertiers{forest@tiers@lower@#1}%
2991 \forlistloop\forest@pack@tiers@rc@forhandlerB\forest@temp@lowertiers
2992 }
2993 \def\forest@pack@tiers@rc@forhandlerB#1{%
2994 \letcs\forest@tiers@containernodeoflowertier{forest@tiers@container@#1}%
2995 \forest@get{\forest@tiers@containernodeoflowertier}{content}\lowercontent
2996 \forest@get{\forest@tiers@containernodeoftier}{content}\uppercontent
2997 \forest@fornode{\forest@tiers@containernodeoflowertier}{%
2998 \forest@ifancestorof
2999 {\forest@tiers@containernodeoftier}
3000 {\csletcs{forest@tiers@container@\forest@tiers@temptier}{forest@tiers@container@#1}}%
3001 }%
3002 }%
3003 }
3004 \def\forest@pack@tiers@computeprocessingorder{%
3005 \def\forest@tiers@processingorder{%
3006 \foresttoget{tiers}\forest@tiers@cpo@tierstodo
3007 \forest@loopa

```

```

3008 \ifdefempty\forest@tiers@cpo@tierstodo{\forest@tempfalse}{\forest@temptrue}%
3009 \ifforest@temp
3010 \def\forest@tiers@cpo@tiersremaining{%
3011 \def\forest@tiers@cpo@tiersindependent{%
3012 \forlistloop\forest@pack@tiers@cpo@forhandlerA\forest@tiers@cpo@tierstodo
3013 \ifdefempty\forest@tiers@cpo@tiersindependent{%
3014 \PackageError{forest}{Circular tiers!}{}}}%
3015 \forlistloop\forest@pack@tiers@cpo@forhandlerB\forest@tiers@cpo@tiersremaining
3016 \let\forest@tiers@cpo@tierstodo\forest@tiers@cpo@tiersremaining
3017 \forest@repeata
3018 }
3019 \def\forest@pack@tiers@cpo@forhandlerA#1{%
3020 \ifcsempy{forest@tiers@higher@#1}{%
3021 \listadd\forest@tiers@cpo@tiersindependent{#1}%
3022 \listadd\forest@tiers@processingorder{#1}%
3023 }{%
3024 \listadd\forest@tiers@cpo@tiersremaining{#1}%
3025 }%
3026 }
3027 \def\forest@pack@tiers@cpo@forhandlerB#1{%
3028 \def\forest@pack@tiers@cpo@aremainingtier{#1}%
3029 \forlistloop\forest@pack@tiers@cpo@forhandlerC\forest@tiers@cpo@tiersindependent
3030 }
3031 \def\forest@pack@tiers@cpo@forhandlerC#1{%
3032 \ifinlistcs{#1}{forest@tiers@higher@\forest@pack@tiers@cpo@aremainingtier}{%
3033 \forest@listcsdel{forest@tiers@higher@\forest@pack@tiers@cpo@aremainingtier}{#1}%
3034 }{}%
3035 }
3036 \def\forest@pack@tiers@write{%
3037 \forlistloop\forest@pack@tiers@write@forhandler\forest@tiers@processingorder
3038 }
3039 \def\forest@pack@tiers@write@forhandler#1{%
3040 \forest@fornode{\csname forest@tiers@container@#1\endcsname}{%
3041 \forest@pack@tiers@check{#1}%
3042 }%
3043 \xappto\forest@smuggle{%
3044 \noexpand\listadd
3045 \forest@om{\csname forest@tiers@container@#1\endcsname}{@tiers}%
3046 {#1}%
3047 }%
3048 }
3049 % checks if the tier is compatible with growth changes and calign=node/edge angle
3050 \def\forest@pack@tiers@check#1{%
3051 \def\forest@temp@currenttier{#1}%
3052 \forest@node@foreachdescendant{%
3053 \ifnum\forestove{grow}=\forestove{\forestove{@parent}}{grow}
3054 \else
3055 \forest@pack@tiers@check@grow
3056 \fi
3057 \ifnum\forestove{n children}>1
3058 \forest@toget{calign}\forest@temp
3059 \ifx\forest@temp\forest@pack@tiers@check@nodeangle
3060 \forest@pack@tiers@check@calign
3061 \fi
3062 \ifx\forest@temp\forest@pack@tiers@check@edgeangle
3063 \forest@pack@tiers@check@calign
3064 \fi
3065 \fi
3066 }%
3067 }
3068 \def\forest@pack@tiers@check@nodeangle{node angle}%

```

```

3069 \def\forest@pack@tiers@check@edgeangle{edge angle}%
3070 \def\forest@pack@tiers@check@grow{%
3071   \forestoget{content}\forest@temp@content
3072   \let\forest@temp@currentnode\forest@cn
3073   \forest@node@foreachdescendant{%
3074     \forestoget{tier}\forest@temp
3075     \ifx\forest@temp@currentttier\forest@temp
3076       \forest@pack@tiers@check@grow@error
3077     \fi
3078   }%
3079 }
3080 \def\forest@pack@tiers@check@grow@error{%
3081   \PackageError{forest}{Tree growth direction changes in node \forest@temp@currentnode\space
3082     (content: \forest@temp@content), while tier '\forest@temp' is specified for nodes both
3083     out- and inside the subtree rooted in node \forest@temp@currentnode. This will not work.}{}%
3084 }
3085 \def\forest@pack@tiers@check@calign{%
3086   \forest@node@foreachchild{%
3087     \forestoget{tier}\forest@temp
3088     \ifx\forest@temp@currentttier\forest@temp
3089       \forest@pack@tiers@check@calign@warning
3090     \fi
3091   }%
3092 }
3093 \def\forest@pack@tiers@check@calign@warning{%
3094   \PackageWarning{forest}{Potential option conflict: node \forest@temp@parent (content:
3095     '\forest@temp@parent\forest@temp@parent') was given 'calign=\forest@temp@calign', while its
3096     child \forest@temp@child (content: '\forest@temp@child') was given 'tier=\forest@temp@tier'.
3097     The parent's 'calign' will only work if the child was the lowest node on its tier before the
3098     alignment.}{}
3099 }

```

### 10.2.2 Node boundary

Compute the node boundary: it will be put in the pgf's current path. The computation is done within a generic anchor so that the shape's saved anchors and macros are available.

```

3100 \pgfdeclaregenericanchor{forest@compute@node@boundary}{%
3101   \letcs\forest@temp@boundary@macro\forest@compute@node@boundary@#1}%
3102   \ifcsname forest@compute@node@boundary@#1\endcsname
3103     \csname forest@compute@node@boundary@#1\endcsname
3104   \else
3105     \forest@compute@node@boundary@rectangle
3106   \fi
3107   \pgfsyssoftpath@getcurrentpath\forest@temp
3108   \global\let\forest@global@boundary\forest@temp
3109 }
3110 \def\forest@mt#1{%
3111   \expandafter\pgfpointanchor\expandafter{\pgfreferencednodename}{#1}%
3112   \pgfsyssoftpath@moveto{\the\pgf@x}{\the\pgf@y}%
3113 }%
3114 \def\forest@lt#1{%
3115   \expandafter\pgfpointanchor\expandafter{\pgfreferencednodename}{#1}%
3116   \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
3117 }%
3118 \def\forest@compute@node@boundary@coordinate{%
3119   \forest@mt{center}%
3120 }
3121 \def\forest@compute@node@boundary@circle{%
3122   \forest@mt{east}%
3123   \forest@lt{north east}%

```

```

3124 \forest@lt{north}%
3125 \forest@lt{north west}%
3126 \forest@lt{west}%
3127 \forest@lt{south west}%
3128 \forest@lt{south}%
3129 \forest@lt{south east}%
3130 \forest@lt{east}%
3131 }
3132 \def\forest@compute@node@boundary@rectangle{%
3133 \forest@mt{south west}%
3134 \forest@lt{south east}%
3135 \forest@lt{north east}%
3136 \forest@lt{north west}%
3137 \forest@lt{south west}%
3138 }
3139 \def\forest@compute@node@boundary@diamond{%
3140 \forest@mt{east}%
3141 \forest@lt{north}%
3142 \forest@lt{west}%
3143 \forest@lt{south}%
3144 \forest@lt{east}%
3145 }
3146 \let\forest@compute@node@boundary@ellipse\forest@compute@node@boundary@circle
3147 \def\forest@compute@node@boundary@trapezium{%
3148 \forest@mt{top right corner}%
3149 \forest@lt{top left corner}%
3150 \forest@lt{bottom left corner}%
3151 \forest@lt{bottom right corner}%
3152 \forest@lt{top right corner}%
3153 }
3154 \def\forest@compute@node@boundary@semicircle{%
3155 \forest@mt{arc start}%
3156 \forest@lt{north}%
3157 \forest@lt{east}%
3158 \forest@lt{north east}%
3159 \forest@lt{apex}%
3160 \forest@lt{north west}%
3161 \forest@lt{west}%
3162 \forest@lt{arc end}%
3163 \forest@lt{arc start}%
3164 }
3165 \newloop\forest@computenodeboundary@loop
3166 \csdef{forest@compute@node@boundary@regular polygon}{%
3167 \forest@mt{corner 1}%
3168 \c@pgf@counta=\sides\relax
3169 \forest@computenodeboundary@loop
3170 \ifnum\c@pgf@counta>0
3171 \forest@lt{corner \the\c@pgf@counta}%
3172 \advance\c@pgf@counta-1
3173 \forest@computenodeboundary@repeat
3174 }%
3175 \def\forest@compute@node@boundary@star{%
3176 \forest@mt{outer point 1}%
3177 \c@pgf@counta=\totalstarpoints\relax
3178 \divide\c@pgf@counta2
3179 \forest@computenodeboundary@loop
3180 \ifnum\c@pgf@counta>0
3181 \forest@lt{inner point \the\c@pgf@counta}%
3182 \forest@lt{outer point \the\c@pgf@counta}%
3183 \advance\c@pgf@counta-1
3184 \forest@computenodeboundary@repeat

```

```

3185 }%
3186 \csdef{forest@compute@node@boundary@isosceles triangle}{%
3187   \forest@mt{apex}%
3188   \forest@lt{left corner}%
3189   \forest@lt{right corner}%
3190   \forest@lt{apex}%
3191 }
3192 \def\forest@compute@node@boundary@kite{%
3193   \forest@mt{upper vertex}%
3194   \forest@lt{left vertex}%
3195   \forest@lt{lower vertex}%
3196   \forest@lt{right vertex}%
3197   \forest@lt{upper vertex}%
3198 }
3199 \def\forest@compute@node@boundary@dart{%
3200   \forest@mt{tip}%
3201   \forest@lt{left tail}%
3202   \forest@lt{tail center}%
3203   \forest@lt{right tail}%
3204   \forest@lt{tip}%
3205 }
3206 \csdef{forest@compute@node@boundary@circular sector}{%
3207   \forest@mt{sector center}%
3208   \forest@lt{arc start}%
3209   \forest@lt{arc center}%
3210   \forest@lt{arc end}%
3211   \forest@lt{sector center}%
3212 }
3213 \def\forest@compute@node@boundary@cylinder{%
3214   \forest@mt{top}%
3215   \forest@lt{after top}%
3216   \forest@lt{before bottom}%
3217   \forest@lt{bottom}%
3218   \forest@lt{after bottom}%
3219   \forest@lt{before top}%
3220   \forest@lt{top}%
3221 }
3222 \cslet{forest@compute@node@boundary@forbidden sign}\forest@compute@node@boundary@circle
3223 \cslet{forest@compute@node@boundary@magnifying glass}\forest@compute@node@boundary@circle
3224 \def\forest@compute@node@boundary@cloud{%
3225   \getradii
3226   \forest@mt{puff 1}%
3227   \c@pgf@counta=\puffs\relax
3228   \forest@computenodeboundary@loop
3229   \ifnum\c@pgf@counta>0
3230     \forest@lt{puff \the\c@pgf@counta}%
3231     \advance\c@pgf@counta-1
3232   \forest@computenodeboundary@repeat
3233 }
3234 \def\forest@compute@node@boundary@starburst{
3235   \calculatestarburstpoints
3236   \forest@mt{outer point 1}%
3237   \c@pgf@counta=\totalpoints\relax
3238   \divide\c@pgf@counta2
3239   \forest@computenodeboundary@loop
3240   \ifnum\c@pgf@counta>0
3241     \forest@lt{inner point \the\c@pgf@counta}%
3242     \forest@lt{outer point \the\c@pgf@counta}%
3243     \advance\c@pgf@counta-1
3244   \forest@computenodeboundary@repeat
3245 }%

```



```

3246 \def\forest@compute@node@boundary@signal{%
3247   \forest@mt{east}%
3248   \forest@lt{south east}%
3249   \forest@lt{south west}%
3250   \forest@lt{west}%
3251   \forest@lt{north west}%
3252   \forest@lt{north east}%
3253   \forest@lt{east}%
3254 }
3255 \def\forest@compute@node@boundary@tape{%
3256   \forest@mt{north east}%
3257   \forest@lt{60}%
3258   \forest@lt{north}%
3259   \forest@lt{120}%
3260   \forest@lt{north west}%
3261   \forest@lt{south west}%
3262   \forest@lt{240}%
3263   \forest@lt{south}%
3264   \forest@lt{310}%
3265   \forest@lt{south east}%
3266   \forest@lt{north east}%
3267 }
3268 \csdef{forest@compute@node@boundary@single arrow}{%
3269   \forest@mt{tip}%
3270   \forest@lt{after tip}%
3271   \forest@lt{after head}%
3272   \forest@lt{before tail}%
3273   \forest@lt{after tail}%
3274   \forest@lt{before head}%
3275   \forest@lt{before tip}%
3276   \forest@lt{tip}%
3277 }
3278 \csdef{forest@compute@node@boundary@double arrow}{%
3279   \forest@mt{tip 1}%
3280   \forest@lt{after tip 1}%
3281   \forest@lt{after head 1}%
3282   \forest@lt{before head 2}%
3283   \forest@lt{before tip 2}%
3284   \forest@mt{tip 2}%
3285   \forest@lt{after tip 2}%
3286   \forest@lt{after head 2}%
3287   \forest@lt{before head 1}%
3288   \forest@lt{before tip 1}%
3289   \forest@lt{tip 1}%
3290 }
3291 \csdef{forest@compute@node@boundary@arrow box}{%
3292   \forest@mt{before north arrow}%
3293   \forest@lt{before north arrow head}%
3294   \forest@lt{before north arrow tip}%
3295   \forest@lt{north arrow tip}%
3296   \forest@lt{after north arrow tip}%
3297   \forest@lt{after north arrow head}%
3298   \forest@lt{after north arrow}%
3299   \forest@lt{north east}%
3300   \forest@lt{before east arrow}%
3301   \forest@lt{before east arrow head}%
3302   \forest@lt{before east arrow tip}%
3303   \forest@lt{east arrow tip}%
3304   \forest@lt{after east arrow tip}%
3305   \forest@lt{after east arrow head}%
3306   \forest@lt{after east arrow}%

```

```

3307 \forest@lt{south east}%
3308 \forest@lt{before south arrow}%
3309 \forest@lt{before south arrow head}%
3310 \forest@lt{before south arrow tip}%
3311 \forest@lt{south arrow tip}%
3312 \forest@lt{after south arrow tip}%
3313 \forest@lt{after south arrow head}%
3314 \forest@lt{after south arrow}%
3315 \forest@lt{south west}%
3316 \forest@lt{before west arrow}%
3317 \forest@lt{before west arrow head}%
3318 \forest@lt{before west arrow tip}%
3319 \forest@lt{west arrow tip}%
3320 \forest@lt{after west arrow tip}%
3321 \forest@lt{after west arrow head}%
3322 \forest@lt{after west arrow}%
3323 \forest@lt{north west}%
3324 \forest@lt{before north arrow}%
3325 }
3326 \cslet{forest@compute@node@boundary@circle split}\forest@compute@node@boundary@circle
3327 \cslet{forest@compute@node@boundary@circle solidus}\forest@compute@node@boundary@circle
3328 \cslet{forest@compute@node@boundary@ellipse split}\forest@compute@node@boundary@ellipse
3329 \cslet{forest@compute@node@boundary@rectangle split}\forest@compute@node@boundary@rectangle
3330 \def\forest@compute@node@boundary@@callout{%
3331 \beforecalloutpointer
3332 \pgfsyssoftpath@moveto{\the\pgf@x}{\the\pgf@y}%
3333 \calloutpointeranchor
3334 \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
3335 \aftercalloutpointer
3336 \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
3337 }
3338 \csdef{forest@compute@node@boundary@rectangle callout}{%
3339 \forest@compute@node@boundary@rectangle
3340 \rectanglecalloutpoints
3341 \forest@compute@node@boundary@@callout
3342 }
3343 \csdef{forest@compute@node@boundary@ellipse callout}{%
3344 \forest@compute@node@boundary@ellipse
3345 \ellipsecalloutpoints
3346 \forest@compute@node@boundary@@callout
3347 }
3348 \csdef{forest@compute@node@boundary@cloud callout}{%
3349 \forest@compute@node@boundary@cloud
3350 % at least a first approx...
3351 \forest@mt{center}%
3352 \forest@lt{pointer}%
3353 }%
3354 \csdef{forest@compute@node@boundary@cross out}{%
3355 \forest@mt{south east}%
3356 \forest@lt{north west}%
3357 \forest@mt{south west}%
3358 \forest@lt{north east}%
3359 }%
3360 \csdef{forest@compute@node@boundary@strike out}{%
3361 \forest@mt{north east}%
3362 \forest@lt{south west}%
3363 }%
3364 \cslet{forest@compute@node@boundary@rounded rectangle}\forest@compute@node@boundary@rectangle
3365 \csdef{forest@compute@node@boundary@chamfered rectangle}{%
3366 \forest@mt{before south west}%
3367 \forest@mt{after south west}%

```

```

3368 \forest@lt{before south east}%
3369 \forest@lt{after south east}%
3370 \forest@lt{before north east}%
3371 \forest@lt{after north east}%
3372 \forest@lt{before north west}%
3373 \forest@lt{after north west}%
3374 \forest@lt{before south west}%
3375 }%

```

### 10.3 Compute absolute positions

Computes absolute positions of descendants relative to this node. Stores the results in attributes `x` and `y`.

```

3376 \def\forest@node@computeabsolutepositions{%
3377   \forestoset{x}{0pt}%
3378   \forestoset{y}{0pt}%
3379   \edef\forest@marshal{%
3380     \noexpand\forest@node@foreachchild{%
3381       \noexpand\forest@node@computeabsolutepositions@{0pt}{0pt}{\forestove{grow}}%
3382     }%
3383   }\forest@marshal
3384 }
3385 \def\forest@node@computeabsolutepositions@#1#2#3{%
3386   \pgfpointadd{\pgfpoint{#1}{#2}}{%
3387     \pgfpointadd{\pgfpolar{#3}{\forestove{1}}}{\pgfpolar{90 + #3}{\forestove{s}}}%
3388     \pgfgetlastxy\forest@temp@x\forest@temp@y
3389     \forestolet{x}\forest@temp@x
3390     \forestolet{y}\forest@temp@y
3391   }\edef\forest@marshal{%
3392     \noexpand\forest@node@foreachchild{%
3393       \noexpand\forest@node@computeabsolutepositions@{\forest@temp@x}{\forest@temp@y}{\forestove{grow}}%
3394     }%
3395   }\forest@marshal
3396 }

```

### 10.4 Drawing the tree

```

3397 \newif\ifforest@drawtree@preservenodeboxes@
3398 \def\forest@node@drawtree{%
3399   \expandafter\ifstrequal\expandafter{\forest@drawtreebox}{\pgfkeysnovalue}{%
3400     \let\forest@drawtree@beginbox\relax
3401     \let\forest@drawtree@endbox\relax
3402   }{%
3403     \edef\forest@drawtree@beginbox{\global\setbox\forest@drawtreebox=\hbox\bgroup}%
3404     \let\forest@drawtree@endbox\egroup
3405   }%
3406   \ifforest@external@
3407     \ifforest@externalize@tree@
3408       \forest@temptrue
3409     \else
3410       \tikzifexternalizing{%
3411         \ifforest@was@tikzexternalwasenable
3412           \forest@temptrue
3413           \pgfkeys{/tikz/external/optimize=false}%
3414           \let\forest@drawtree@beginbox\relax
3415           \let\forest@drawtree@endbox\relax
3416         \else
3417           \forest@tempfalse
3418         \fi
3419       }{%

```

```

3420     \forest@tempfalse
3421 }%
3422 \fi
3423 \ifforest@temp
3424   \advance\forest@externalize@inner@n 1
3425   \edef\forest@externalize@filename{%
3426     \tikzexternalrealjob-forest-\forest@externalize@outer@n
3427     \ifnum\forest@externalize@inner@n=0 \else.\the\forest@externalize@inner@n\fi}%
3428   \expandafter\tikzsetnextfilename\expandafter{\forest@externalize@filename}%
3429   \tikzexternalenable
3430   \pgfkeysalso{/tikz/external/remake next,/tikz/external/export next}%
3431 \fi
3432 \ifforest@externalize@tree@
3433   \typeout{forest: Invoking a recursive call to generate the external picture
3434     '\forest@externalize@filename' for the following context+code:
3435     '\expandafter\detokenize\expandafter{\forest@externalize@id}'}%
3436 \fi
3437 \fi
3438 %
3439 \ifforest\tikzcshack
3440   \let\forest@original@tikz@parse@node\tikz@parse@node
3441   \let\tikz@parse@node\forest@tikz@parse@node
3442 \fi
3443 \forest@drawtree@beginbox
3444 \tikz{%
3445   \forestove{tikz preamble}%
3446   \forest@node@drawtree@
3447 }%
3448 \forest@drawtree@endbox
3449 \ifforest\tikzcshack
3450   \let\tikz@parse@node\forest@original@tikz@parse@node
3451 \fi
3452 %
3453 \ifforest@external@
3454   \ifforest@externalize@tree@
3455     \tikzexternaldisable
3456     \eappto\forest@externalize@checkimages{%
3457       \noexpand\forest@includeexternal@check{\forest@externalize@filename}%
3458     }%
3459     \expandafter\ifstrequal\expandafter{\forest@drawtreebox}{\pgfkeysnovalue}{%
3460       \eappto\forest@externalize@loadimages{%
3461         \noexpand\forest@includeexternal{\forest@externalize@filename}%
3462       }%
3463     }{%
3464       \eappto\forest@externalize@loadimages{%
3465         \noexpand\forest@includeexternal@box\forest@drawtreebox{\forest@externalize@filename}%
3466       }%
3467     }%
3468   \fi
3469 \fi
3470 }
3471 \def\forest@node@drawtree@{%
3472   \forest@node@foreach{\forest@draw@node}%
3473   \forest@node@ifnamedefined{forest@baseline@node}{%
3474     \edef\forest@temp{%
3475       \noexpand\pgfsetbaselinepointlater{%
3476         \noexpand\pgfpointanchor
3477           {\forestOve{\forest@node@Nametoid{forest@baseline@node}}{name}}
3478           {\forestOve{\forest@node@Nametoid{forest@baseline@node}}{anchor}}
3479       }%
3480     }\forest@temp

```

```

3481 }{}%
3482 \forest@node@foreachdescendant{\forest@draw@edge}%
3483 \forest@node@foreach{\forest@draw@tikz}%
3484 }
3485 \def\forest@draw@node{%
3486   \ifnum\forestove{phantom}=0
3487     \forest@node@forest@positionnodelater@restore
3488     \ifforest@drawtree@preservenodeboxes@
3489       \pgfnodelias{\forest@temp}{\forestove{later@name}}%
3490     \fi
3491     \pgfpositionnodelater{\pgfqpoint{\forestove{x}}{\forestove{y}}}%
3492     \ifforest@drawtree@preservenodeboxes@
3493       \pgfnodelias{\forestove{later@name}}{\forest@temp}%
3494     \fi
3495   \fi
3496 }
3497 \def\forest@draw@edge{%
3498   \ifnum\forestove{phantom}=0
3499     \ifnum\forestove{\forestove{parent}}{phantom}=0
3500       \edef\forest@temp{\forestove{edge path}}%
3501       \forest@temp
3502     \fi
3503   \fi
3504 }
3505 \def\forest@draw@tikz{%
3506   \forestove{tikz}%
3507 }

```

A hack into TikZ's coordinate parser: implements relative node names!

```

3508 \def\forest@tikz@parse@node#1(#2){%
3509   \pgfutil@in@.#2}%
3510   \ifpgfutil@in@
3511     \expandafter\forest@tikz@parse@node@checkiftikzname@withdot
3512   \else%
3513     \expandafter\forest@tikz@parse@node@checkiftikzname@withoutdot
3514   \fi%
3515   #1(#2)\forest@end
3516 }
3517 \def\forest@tikz@parse@node@checkiftikzname@withdot#1(#2.#3)\forest@end{%
3518   \forest@tikz@parse@node@checkiftikzname#1{#2}{.#3}%
3519 }
3520 \def\forest@tikz@parse@node@checkiftikzname@withoutdot#1(#2)\forest@end{%
3521   \forest@tikz@parse@node@checkiftikzname#1{#2}{}%
3522 }
3523 \def\forest@tikz@parse@node@checkiftikzname#1#2#3{%
3524   \expandafter\ifx\csname pgf@sh@ns@#2\endcsname\relax
3525     \forest@forthis{%
3526       \forest@nameandgo{#2}%
3527       \edef\forest@temp@relativenodename{\forestove{name}}%
3528     }%
3529   \else
3530     \def\forest@temp@relativenodename{#2}%
3531   \fi
3532   \expandafter\forest@original@tikz@parse@node\expandafter#1\expandafter(\forest@temp@relativenodename#3)%
3533 }
3534 \def\forest@nameandgo#1{%
3535   \pgfutil@in@!{#1}%
3536   \ifpgfutil@in@
3537     \forest@nameandgo@(#1)%
3538   \else
3539     \ifstrempy{#1}{-}{\edef\forest@cn{\forest@node@Nametoid{#1}}}%
3540   \fi
3541 }

```

```

3540 \def\forest@nameandgo@(#1!#2){%
3541   \ifstrempy{#1}{\edef\forest@cn{\forest@node@Nametoid{#1}}}%
3542   \forest@go{#2}%
3543 }

parent/child anchor are generic anchors which forward to the real one. There's a hack in there to
deal with link pointing to the "border" anchor.
3544 \pgfdeclaregenericanchor{parent anchor}{%
3545   \forest@generic@parent@child@anchor{parent }{#1}}
3546 \pgfdeclaregenericanchor{child anchor}{%
3547   \forest@generic@parent@child@anchor{child }{#1}}
3548 \pgfdeclaregenericanchor{anchor}{%
3549   \forest@generic@parent@child@anchor{}{#1}}
3550 \def\forest@generic@parent@child@anchor#1#2{%
3551   \forest@get{\forest@node@Nametoid{\pgfreferencednodename}}{#1anchor}\forest@temp@parent@anchor
3552   \ifdefempty\forest@temp@parent@anchor{%
3553     \pgf@sh@reanchor{#2}{center}%
3554     \xdef\forest@hack@tikzshapeborder{%
3555       \noexpand\tikzshapebordertrue
3556       \def\noexpand\tikzshapeborder@name{\pgfreferencednodename}%
3557     }\aftergroup\forest@hack@tikzshapeborder
3558   }{%
3559     \pgf@sh@reanchor{#2}{\forest@temp@parent@anchor}%
3560   }%
3561 }

```

## 11 Geometry

A  $\alpha$  *grow line* is a line through the origin at angle  $\alpha$ . The following macro sets up the grow line, which can then be used by other code (the change is local to the  $\text{\TeX}$  group). More precisely, two normalized vectors are set up: one  $(x_g, y_g)$  on the grow line, and one  $(x_s, y_s)$  orthogonal to it—to get  $(x_s, y_s)$ , rotate  $(x_g, y_g)$   $90^\circ$  counter-clockwise.

```

3562 \newdimen\forest@xg
3563 \newdimen\forest@yg
3564 \newdimen\forest@xs
3565 \newdimen\forest@ys
3566 \def\forest@setupgrowline#1{%
3567   \edef\forest@grow{#1}%
3568   \pgfpointpolar\forest@grow{1pt}%
3569   \forest@xg=\pgf@x
3570   \forest@yg=\pgf@y
3571   \forest@xs=-\pgf@y
3572   \forest@ys=\pgf@x
3573 }

```

### 11.1 Projections

The following macro belongs to the `\pgfpoint...` family: it projects point `#1` on the grow line. (The result is returned via `\pgf@x` and `\pgf@y`.) The implementation is based on code from `tikzlibrarycalc`, but optimized for projecting on grow lines, and split to optimize serial usage in `\forest@projectpath`.

```

3574 \def\forest@pgfpointprojectiontogrowline#1{%
3575   \pgf@process{#1}%

```

Calculate the scalar product of  $(x, y)$  and  $(x_g, y_g)$ : that's the distance of  $(x, y)$  to the grow line.

```

3576   \pgfutil@tempdima=\pgf@sys@tonumber{\pgf@x}\forest@xg%
3577   \advance\pgfutil@tempdima by\pgf@sys@tonumber{\pgf@y}\forest@yg%

```

The projection is  $(x_g, y_g)$  scaled by the distance.

```

3578   \global\pgf@x=\pgf@sys@tonumber{\pgfutil@tempdima}\forest@xg%
3579   \global\pgf@y=\pgf@sys@tonumber{\pgfutil@tempdima}\forest@yg%
3580 }

```

The following macro calculates the distance of point #2 to the grow line and stores the result in TeX-dimension #1. The distance is the scalar product of the point vector and the normalized vector orthogonal to the grow line.

```

3581 \def\forest@distancetogrowline#1#2{%
3582   \pgf@process{#2}%
3583   #1=\pgf@sys@tonumber{\pgf@x}\forest@xs\relax
3584   \advance#1 by\pgf@sys@tonumber{\pgf@y}\forest@ys\relax
3585 }

```

Note that the distance to the grow line is positive for points on one of its sides and negative for points on the other side. (It is positive on the side which  $(x_s, y_s)$  points to.) We thus say that the grow line partitions the plane into a *positive* and a *negative* side.

The following macro projects all segment edges (“points”) of a simple<sup>19</sup> path #1 onto the grow line. The result is an array of tuples (xo, yo, xp, yp), where xo and yo stand for the *original* point, and xp and yp stand for its *projection*. The prefix of the array is given by #2. If the array already exists, the new items are appended to it. The array is not sorted: the order of original points in the array is their order in the path. The computation does not destroy the current path. All result-macros have local scope.

The macro is just a wrapper for \forest@projectpath@process.

```

3586 \let\forest@pp@n\relax
3587 \def\forest@projectpath@trowline#1#2{%
3588   \edef\forest@pp@prefix{#2}%
3589   \forest@save@pgfsyssoftpath@tokendef
3590   \let\pgfsyssoftpath@movetotoken\forest@projectpath@processpoint
3591   \let\pgfsyssoftpath@linetotoken\forest@projectpath@processpoint
3592   \c@pgf@counta=0
3593   #1%
3594   \csedef{#2n}{\the\c@pgf@counta}%
3595   \forest@restore@pgfsyssoftpath@tokendef
3596 }

```

For each point, remember the point and its projection to grow line.

```

3597 \def\forest@projectpath@processpoint#1#2{%
3598   \pgf@point{#1}{#2}%
3599   \expandafter\edef\c@pgf@counta\forest@pp@prefix\the\c@pgf@counta xo\endcsname{\the\pgf@x}%
3600   \expandafter\edef\c@pgf@counta\forest@pp@prefix\the\c@pgf@counta yo\endcsname{\the\pgf@y}%
3601   \forest@pgfpointprojectiontrowline{}%
3602   \expandafter\edef\c@pgf@counta\forest@pp@prefix\the\c@pgf@counta xp\endcsname{\the\pgf@x}%
3603   \expandafter\edef\c@pgf@counta\forest@pp@prefix\the\c@pgf@counta yp\endcsname{\the\pgf@y}%
3604   \advance\c@pgf@counta 1\relax
3605 }

```

Sort the array (prefix #1) produced by \forest@projectpath@trowline by (xp,yp), in the ascending order.

```

3606 \def\forest@sortprojections#1{%
3607   % todo: optimize in cases when we know that the array is actually a
3608   % merger of sorted arrays; when does this happen? in
3609   % distance_between_paths, and when merging the edges of the parent
3610   % and its children in a uniform growth tree
3611   \edef\forest@ppi@inputprefix{#1}%
3612   \c@pgf@counta=\c@pgf@counta\forest@ppi@inputprefix\relax
3613   \advance\c@pgf@counta -1
3614   \forest@sort\forest@ppi@inputprefix\forest@ppi@inputprefix\let\forest@sort@ascending{0}{\the\c@pgf@counta}%
3615 }

```

The following macro processes the data gathered by (possibly more than one invocation of) \forest@projectpath@trowline into array with prefix #1. The resulting data is the following.

- Array of projections (prefix #2)

---

<sup>19</sup>A path is *simple* if it consists of only move-to and line-to operations.

- its items are tuples (x,y) (the array is sorted by x and y), and
- an inner array of original points (prefix #2N@, where N is the index of the item in array #2. The items of #2N@ are x, y and d: x and y are the coordinates of the original point; d is its distance to the grow line. The inner array is not sorted.

- A dictionary #2: keys are the coordinates (x,y) of the original points; a value is the index of the original point's projection in array #2.<sup>20</sup>

```
3616 \def\forest@processprojectioninfo#1#2{%
3617   \edef\forest@ppi@inputprefix{#1}%
```

Loop (counter \c@pgf@counta) through the sorted array of raw data.

```
3618   \c@pgf@counta=0
3619   \c@pgf@countb=-1
3620   \loop
3621   \ifnum\c@pgf@counta<\csname#1n\endcsname\relax
```

Check if the projection tuple in the current raw item equals the current projection.

```
3622     \letcs\forest@xo{#1\the\c@pgf@counta xo}%
3623     \letcs\forest@yo{#1\the\c@pgf@counta yo}%
3624     \letcs\forest@xp{#1\the\c@pgf@counta xp}%
3625     \letcs\forest@yp{#1\the\c@pgf@counta yp}%
3626     \ifnum\c@pgf@countb<0
3627       \forest@equaltotolerancefalse
3628     \else
3629       \forest@equaltotolerance
3630       {\pgfqpoint\forest@xp\forest@yp}%
3631       {\pgfqpoint
3632        {\csname#2\the\c@pgf@countb x\endcsname}%
3633        {\csname#2\the\c@pgf@countb y\endcsname}%
3634       }%
3635     \fi
3636     \ifforest@equaltotolerance\else
```

It not, we will append a new item to the outer result array.

```
3637     \advance\c@pgf@countb 1
3638     \cslet{#2\the\c@pgf@countb x}\forest@xp
3639     \cslet{#2\the\c@pgf@countb y}\forest@yp
3640     \csdef{#2\the\c@pgf@countb @n}{0}%
3641     \fi
```

If the projection is actually a projection of one a point in our path:

```
3642     % todo: this is ugly!
3643     \ifdefined\forest@xo\ifx\forest@xo\relax\else
3644     \ifdefined\forest@yo\ifx\forest@yo\relax\else
```

Append the point of the current raw item to the inner array of points projecting to the current projection.

```
3645     \forest@append@point@to@inner@array
3646     \forest@xo\forest@yo
3647     {#2\the\c@pgf@countb @}%
```

Put a new item in the dictionary: key = the original point, value = the projection index.

```
3648     \csedef{#2(\forest@xo,\forest@yo)}{\the\c@pgf@countb}%
3649     \fi\fi
3650     \fi\fi
```

Clean-up the raw array item.

```
3651     \cslet{#1\the\c@pgf@counta xo}\relax
3652     \cslet{#1\the\c@pgf@counta yo}\relax
3653     \cslet{#1\the\c@pgf@counta xp}\relax
```

<sup>20</sup>At first sight, this information could be cached “at the source”: by forest@pgfpntprojectiontogrowline. However, due to imprecise intersecting (in breakpath), we cheat and merge very adjacent projection points, expecting that the points to project to the merged projection point. All this depends on the given path, so a generic cache is not feasible.



```

3654 \cslet{#1\the\c@pgf@counta yp}\relax
3655 \advance\c@pgf@counta 1
3656 \repeat

Clean up the raw array length.
3657 \cslet{#1n}\relax

Store the length of the outer result array.
3658 \advance\c@pgf@countb 1
3659 \csedef{#2n}{\the\c@pgf@countb}%
3660 }

Item-exchange macro for quicksorting the raw projection data. (#1 is copied into #2.)
3661 \def\forest@ppiraw@let#1#2{%
3662 \csletcs{\forest@ppi@inputprefix#1xo}{\forest@ppi@inputprefix#2xo}%
3663 \csletcs{\forest@ppi@inputprefix#1yo}{\forest@ppi@inputprefix#2yo}%
3664 \csletcs{\forest@ppi@inputprefix#1xp}{\forest@ppi@inputprefix#2xp}%
3665 \csletcs{\forest@ppi@inputprefix#1yp}{\forest@ppi@inputprefix#2yp}%
3666 }

Item comparison macro for quicksorting the raw projection data.
3667 \def\forest@ppiraw@cmp#1#2{%
3668 \forest@sort@cmptwodimcs
3669 {\forest@ppi@inputprefix#1xp}{\forest@ppi@inputprefix#1yp}%
3670 {\forest@ppi@inputprefix#2xp}{\forest@ppi@inputprefix#2yp}%
3671 }

Append the point (#1,#2) to the (inner) array of points (prefix #3).
3672 \def\forest@append@point@to@inner@array#1#2#3{%
3673 \c@pgf@countc=\csname#3n\endcsname\relax
3674 \csedef{#3\the\c@pgf@countc x}{#1}%
3675 \csedef{#3\the\c@pgf@countc y}{#2}%
3676 \forest@distancetogrowline\pgfutil@tempdima{\pgfqpoint#1#2}%
3677 \csedef{#3\the\c@pgf@countc d}{\the\pgfutil@tempdima}%
3678 \advance\c@pgf@countc 1
3679 \csedef{#3n}{\the\c@pgf@countc}%
3680 }

```

## 11.2 Break path

The following macro computes from the given path (#1) a “broken” path (#3) that contains the same points of the plane, but has potentially more segments, so that, for every point from a given set of points on the grow line, a line through this point perpendicular to the grow line intersects the broken path only at its edge segments (i.e. not between them).

The macro works only for *simple* paths, i.e. paths built using only move-to and line-to operations. Furthermore, `\forest@processprojectioninfo` must be called before calling `\forest@breakpath`: we expect information with prefix #2. The macro updates the information compiled by `\forest@processprojectioninfo` with information about points added by path-breaking.

```

3681 \def\forest@breakpath#1#2#3{%

Store the current path in a macro and empty it, then process the stored path. The processing creates a new current path.
3682 \edef\forest@bp@prefix{#2}%
3683 \forest@save@pgfsyssoftpath@tokendefs
3684 \let\pgfsyssoftpath@movetotoken\forest@breakpath@processfirstpoint
3685 \let\pgfsyssoftpath@linetotoken\forest@breakpath@processfirstpoint
3686 %\pgfusepath{}% empty the current path. ok?
3687 #1%
3688 \forest@restore@pgfsyssoftpath@tokendefs
3689 \pgfsyssoftpath@getcurrentpath#3%
3690 }

```

The original and the broken path start in the same way. (This code implicitly “repairs” a path that starts illegally, with a line-to operation.)

```

3691 \def\forest@breakpath@processfirstpoint#1#2{%
3692   \forest@breakpath@processmoveto{#1}-{#2}%
3693   \let\pgfsyssoftpath@movetotoken\forest@breakpath@processmoveto
3694   \let\pgfsyssoftpath@linetotoken\forest@breakpath@processlineto
3695 }

```

When a move-to operation is encountered, it is simply copied to the broken path, starting a new subpath. Then we remember the last point, its projection’s index (the point dictionary is used here) and the actual projection point.

```

3696 \def\forest@breakpath@processmoveto#1#2{%
3697   \pgfsyssoftpath@moveto{#1}-{#2}%
3698   \def\forest@previous@x{#1}%
3699   \def\forest@previous@y{#2}%
3700   \expandafter\let\expandafter\forest@previous@i
3701   \csname\forest@bp@prefix(#1,#2)\endcsname
3702   \expandafter\let\expandafter\forest@previous@px
3703   \csname\forest@bp@prefix\forest@previous@i x\endcsname
3704   \expandafter\let\expandafter\forest@previous@py
3705   \csname\forest@bp@prefix\forest@previous@i y\endcsname
3706 }

```

This is the heart of the path-breaking procedure.

```

3707 \def\forest@breakpath@processlineto#1#2{%

```

Usually, the broken path will continue with a line-to operation (to the current point (#1,#2)).

```

3708   \let\forest@breakpath@op\pgfsyssoftpath@lineto

```

Get the index of the current point’s projection and the projection itself. (The point dictionary is used here.)

```

3709   \expandafter\let\expandafter\forest@i
3710   \csname\forest@bp@prefix(#1,#2)\endcsname
3711   \expandafter\let\expandafter\forest@px
3712   \csname\forest@bp@prefix\forest@i x\endcsname
3713   \expandafter\let\expandafter\forest@py
3714   \csname\forest@bp@prefix\forest@i y\endcsname

```

Test whether the projections of the previous and the current point are the same.

```

3715   \forest@equaltotolerance
3716   {\pgfqpoint{\forest@previous@px}{\forest@previous@py}}%
3717   {\pgfqpoint{\forest@px}{\forest@py}}%
3718   \ifforest@equaltotolerance

```

If so, we are dealing with a segment, perpendicular to the grow line. This segment must be removed, so we change the operation to move-to.

```

3719   \let\forest@breakpath@op\pgfsyssoftpath@moveto
3720   \else

```

Figure out the “direction” of the segment: in the order of the array of projections, or in the reversed order? Setup the loop step and the test condition.

```

3721   \forest@temp@count=\forest@previous@i\relax
3722   \ifnum\forest@previous@i<\forest@i\relax
3723     \def\forest@breakpath@step{1}%
3724     \def\forest@breakpath@test{\forest@temp@count<\forest@i\relax}%
3725   \else
3726     \def\forest@breakpath@step{-1}%
3727     \def\forest@breakpath@test{\forest@temp@count>\forest@i\relax}%
3728   \fi

```

Loop through all the projections between (in the (possibly reversed) array order) the projections of the previous and the current point (both exclusive).

```

3729   \loop

```

```

3730      \advance\forest@temp@count\forest@breakpath@step\relax
3731      \expandafter\ifnum\forest@breakpath@test

Intersect the current segment with the line through the current (in the loop!) projection perpendicular
to the grow line. (There will be an intersection.)
3732      \pgfpointintersectionoflines
3733      {\pgfqpoint
3734       {\csname\forest@bp@prefix\the\forest@temp@count x\endcsname}%
3735       {\csname\forest@bp@prefix\the\forest@temp@count y\endcsname}%
3736      }%
3737      {\pgfpointadd
3738       {\pgfqpoint
3739        {\csname\forest@bp@prefix\the\forest@temp@count x\endcsname}%
3740        {\csname\forest@bp@prefix\the\forest@temp@count y\endcsname}%
3741       }%
3742       {\pgfqpoint{\forest@xs}{\forest@ys}}%
3743      }%
3744      {\pgfqpoint{\forest@previous@x}{\forest@previous@y}}%
3745      {\pgfqpoint{#1}{#2}}%

Break the segment at the intersection.
3746      \pgfgetlastxy\forest@last@x\forest@last@y
3747      \pgfsyssoftpath@lineto\forest@last@x\forest@last@y

Append the breaking point to the inner array for the projection.
3748      \forest@append@point@to@inner@array
3749      \forest@last@x\forest@last@y
3750      {\forest@bp@prefix\the\forest@temp@count @}%

Cache the projection of the new segment edge.
3751      \csedef{\forest@bp@prefix(\the\pgf@x,\the\pgf@y)}{\the\forest@temp@count}%
3752      \repeat
3753      \fi

Add the current point.
3754      \forest@breakpath@op{#1}{#2}%

Setup new “previous” info: the segment edge, its projection’s index, and the projection.
3755      \def\forest@previous@x{#1}%
3756      \def\forest@previous@y{#2}%
3757      \let\forest@previous@i\forest@i
3758      \let\forest@previous@px\forest@px
3759      \let\forest@previous@py\forest@py
3760 }

```

### 11.3 Get tight edge of path

This is one of the central algorithms of the package. Given a simple path and a grow line, this method computes its (negative and positive) “tight edge”, which we (informally) define as follows.

Imagine an infinitely long light source parallel to the grow line, on the grow line’s negative/positive side.<sup>21</sup> Furthermore imagine that the path is opaque. Then the negative/positive tight edge of the path is the part of the path that is illuminated.

This macro takes three arguments: **#1** is the path; **#2** and **#3** are macros which will receive the negative and the positive edge, respectively. The edges are returned in the softpath format. Grow line should be set before calling this macro.

Enclose the computation in a  $\TeX$  group. This is actually quite crucial: if there was no enclosure, the temporary data (the segment dictionary, to be precise) computed by the prior invocations of the macro could corrupt the computation in the current invocation.

```

3761 \def\forest@getnegativetightedgeofpath#1#2{%
3762   \forest@get@onetightedgeofpath#1\forest@sort@ascending#2}

```

<sup>21</sup>For the definition of negative/positive side, see `forest@distancetogrowline` in §11.1

```

3763 \def\forest@getpositivetightedgeofpath#1#2{%
3764   \forest@get@onetightedgeofpath#1\forest@sort@descending#2}
3765 \def\forest@get@onetightedgeofpath#1#2#3{%
3766   {%
3767     \forest@get@one@tightedgeofpath#1#2\forest@gep@edge
3768     \global\let\forest@gep@global@edge\forest@gep@edge
3769   }%
3770   \let#3\forest@gep@global@edge
3771 }
3772 \def\forest@get@one@tightedgeofpath#1#2#3{%

```

Project the path to the grow line and compile some useful information.

```

3773   \forest@projectpathtogrowline#1{forest@pp@}%
3774   \forest@sortprojections{forest@pp@}%
3775   \forest@processprojectioninfo{forest@pp@}{forest@pi@}%

```

Break the path.

```

3776   \forest@breakpath#1{forest@pi@}\forest@brokenpath

```

Compile some more useful information.

```

3777   \forest@sort@inner@arrays{forest@pi@}#2%
3778   \forest@pathtodict\forest@brokenpath{forest@pi@}%

```

The auxiliary data is set up: do the work!

```

3779   \forest@gettightedgeofpath@getedge
3780   \pgfsyssoftpath@getcurrentpath\forest@edge

```

Where possible, merge line segments of the path into a single line segment. This is an important optimization, since the edges of the subtrees are computed recursively. Not simplifying the edge could result in a wild growth of the length of the edge (in the sense of the number of segments).

```

3781   \forest@simplifypath\forest@edge#3%
3782 }

```

Get both negative (stored in #2) and positive (stored in #3) edge of the path #1.

```

3783 \def\forest@getbothtightedgesofpath#1#2#3{%
3784   {%
3785     \forest@get@one@tightedgeofpath#1\forest@sort@ascending\forest@gep@firstedge

```

Reverse the order of items in the inner arrays.

```

3786     \c@pgf@counta=0
3787     \loop
3788     \ifnum\c@pgf@counta<\forest@pi@n\relax
3789       \forest@ppi@deflet{forest@pi@\the\c@pgf@counta @}%
3790       \forest@reversearray\forest@ppi@let
3791       {0}%
3792       {\csname forest@pi@\the\c@pgf@counta @n\endcsname}%
3793       \advance\c@pgf@counta 1
3794     \repeat

```

Calling `\forest@gettightedgeofpath@getedge` now will result in the positive edge.

```

3795   \forest@gettightedgeofpath@getedge
3796   \pgfsyssoftpath@getcurrentpath\forest@edge
3797   \forest@simplifypath\forest@edge\forest@gep@secondedge

```

Smuggle the results out of the enclosing  $\text{\TeX}$  group.

```

3798   \global\let\forest@gep@global@firstedge\forest@gep@firstedge
3799   \global\let\forest@gep@global@secondedge\forest@gep@secondedge
3800   }%
3801   \let#2\forest@gep@global@firstedge
3802   \let#3\forest@gep@global@secondedge
3803 }

```

Sort the inner arrays of original points wrt the distance to the grow line. `#2 = \forest@sort@ascending/\forest@sort@loopa` is used here because quicksort uses `\loop`.)

```

3804 \def\forest@sort@inner@arrays#1#2{%
3805   \c@pgf@counta=0
3806   \forest@loopa
3807   \ifnum\c@pgf@counta<\csname#1n\endcsname
3808     \c@pgf@countb=\csname#1\the\c@pgf@counta @n\endcsname\relax
3809     \ifnum\c@pgf@countb>1
3810       \advance\c@pgf@countb -1
3811       \forest@ppi@deflet{#1\the\c@pgf@counta @}%
3812       \forest@ppi@defcmp{#1\the\c@pgf@counta @}%
3813       \forest@sort\forest@ppi@cmp\forest@ppi@let#2{0}{\the\c@pgf@countb}%
3814     \fi
3815     \advance\c@pgf@counta 1
3816   \forest@repeata
3817 }

```

A macro that will define the item exchange macro for quicksorting the inner arrays of original points.  
It takes one argument: the prefix of the inner array.

```

3818 \def\forest@ppi@deflet#1{%
3819   \edef\forest@ppi@let##1##2{%
3820     \noexpand\csletcs{#1##1x}{#1##2x}%
3821     \noexpand\csletcs{#1##1y}{#1##2y}%
3822     \noexpand\csletcs{#1##1d}{#1##2d}%
3823   }%
3824 }

```

A macro that will define the item-compare macro for quicksorting the embedded arrays of original points.  
It takes one argument: the prefix of the inner array.

```

3825 \def\forest@ppi@defcmp#1{%
3826   \edef\forest@ppi@cmp##1##2{%
3827     \noexpand\forest@sort@cmpdimcs{#1##1d}{#1##2d}%
3828   }%
3829 }

```

Put path segments into a “segment dictionary”: for each segment of the path from  $(x_1, y_1)$  to  $(x_2, y_2)$   
let  $\text{\forest@}(x_1, y_1) \text{--}(x_2, y_2)$  be  $\text{\forest@inpath}$  (which can be anything but  $\text{\relax}$ ).

```

3830 \let\forest@inpath\advance

```

This macro is just a wrapper to process the path.

```

3831 \def\forest@pathtodict#1#2{%
3832   \edef\forest@pathtodict@prefix{#2}%
3833   \forest@save@pgfsyssoftpath@tokendefs
3834   \let\pgfsyssoftpath@movetotoken\forest@pathtodict@movetotop
3835   \let\pgfsyssoftpath@linetotoken\forest@pathtodict@linetotop
3836   \def\forest@pathtodict@subpathstart{}%
3837   #1%
3838   \forest@restore@pgfsyssoftpath@tokendefs
3839 }

```

When a move-to operation is encountered:

```

3840 \def\forest@pathtodict@movetotop#1#2{%

```

If a subpath had just started, it was a degenerate one (a point). No need to store that (i.e. no code would use this information). So, just remember that a new subpath has started.

```

3841   \def\forest@pathtodict@subpathstart{(#1,#2)-}%
3842 }

```

When a line-to operation is encountered:

```

3843 \def\forest@pathtodict@linetotop#1#2{%

```

If the subpath has just started, its start is also the start of the current segment.

```

3844 \if\relax\forest@pathtodict@subpathstart\relax\else
3845   \let\forest@pathtodict@from\forest@pathtodict@subpathstart
3846 \fi

```

Mark the segment as existing.

```
3847 \expandafter\let\csname\forest@pathtodict@prefix\forest@pathtodict@from-({#1,#2})\endcsname\forest@inpath
```

Set the start of the next segment to the current point, and mark that we are in the middle of a subpath.

```
3848 \def\forest@pathtodict@from({#1,#2})-%
3849 \def\forest@pathtodict@subpathstart{}%
3850 }
```

In this macro, the edge is actually computed.

```
3851 \def\forest@gettightedgeofpath@getedge{%
```

Clear the path and the last projection.

```
3852 \pgfsyssoftpath@setcurrentpath\pgfutil@empty
3853 \let\forest@lastx\relax
3854 \let\forest@lasty\relax
```

Loop through the (ordered) array of projections. (Since we will be dealing with the current and the next projection in each iteration of the loop, we loop the counter from the first to the second-to-last projection.)

```
3855 \c@pgf@counta=0
3856 \forest@temp@count=\forest@pi@n\relax
3857 \advance\forest@temp@count -1
3858 \edef\forest@nminusone{\the\forest@temp@count}%
3859 \forest@loopa
3860 \ifnum\c@pgf@counta<\forest@nminusone\relax
3861 \forest@gettightedgeofpath@getedge@loopa
3862 \forest@repeata
```

A special case: the edge ends with a degenerate subpath (a point).

```
3863 \ifnum\forest@nminusone<\forest@n\relax\else
3864 \ifnum\csname forest@pi@\forest@nminusone @n\endcsname>0
3865 \forest@gettightedgeofpath@maybemoveto{\forest@nminusone}{0}%
3866 \fi
3867 \fi
3868 }
```

The body of a loop containing an embedded loop must be put in a separate macro because it contains the `\if...` of the embedded `\loop...` without the matching `\fi`: `\fi` is “hiding” in the embedded `\loop`, which has not been expanded yet.

```
3869 \def\forest@gettightedgeofpath@getedge@loopa{%
3870 \ifnum\csname forest@pi@\the\c@pgf@counta @n\endcsname>0
```

Degenerate case: a subpath of the edge is a point.

```
3871 \forest@gettightedgeofpath@maybemoveto{\the\c@pgf@counta}{0}%
```

Loop through points projecting to the current projection. The preparations above guarantee that the points are ordered (either in the ascending or the descending order) with respect to their distance to the grow line.

```
3872 \c@pgf@countb=0
3873 \forest@loopb
3874 \ifnum\c@pgf@countb<\csname forest@pi@\the\c@pgf@counta @n\endcsname\relax
3875 \forest@gettightedgeofpath@getedge@loopb
3876 \forest@repeatb
3877 \fi
3878 \advance\c@pgf@counta 1
3879 }
```

Loop through points projecting to the next projection. Again, the points are ordered.

```
3880 \def\forest@gettightedgeofpath@getedge@loopb{%
3881 \c@pgf@countc=0
3882 \advance\c@pgf@counta 1
3883 \edef\forest@aplusone{\the\c@pgf@counta}%
3884 \advance\c@pgf@counta -1
```

```

3885 \forest@loopc
3886 \ifnum\c@pgf@countc<\csname forest@pi@\forest@aplusone @n\endcsname\relax

```

Test whether [the current point]–[the next point] or [the next point]–[the current point] is a segment in the (broken) path. The first segment found is the one with the minimal/maximal distance (depending on the sort order of arrays of points projecting to the same projection) to the grow line.

Note that for this to work in all cases, the original path should have been broken on its self-intersections. However, a careful reader will probably remember that `\forest@breakpath` does *not* break the path at its self-intersections. This is omitted for performance reasons. Given the intended use of the algorithm (calculating edges of subtrees), self-intersecting paths cannot arise anyway, if only the node boundaries are non-self-intersecting. So, a warning: if you develop a new shape and write a macro computing its boundary, make sure that the computed boundary path is non-self-intersecting!

```

3887 \forest@tempfalse
3888 \expandafter\ifx\csname forest@pi@(%
3889 \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb x\endcsname,%
3890 \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb y\endcsname)--(%
3891 \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname,%
3892 \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname)%
3893 \endcsname\forest@inpath
3894 \forest@temptrue
3895 \else
3896 \expandafter\ifx\csname forest@pi@(%
3897 \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname,%
3898 \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname)--(%
3899 \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb x\endcsname,%
3900 \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb y\endcsname)%
3901 \endcsname\forest@inpath
3902 \forest@temptrue
3903 \fi
3904 \fi
3905 \ifforest@temp

```

We have found the segment with the minimal/maximal distance to the grow line. So let's add it to the edge path.

First, deal with the start point of the edge: check if the current point is the last point. If that is the case (this happens if the current point was the end point of the last segment added to the edge), nothing needs to be done; otherwise (this happens if the current point will start a new subpath of the edge), move to the current point, and update the last-point macros.

```

3906 \forest@gettightedgeofpath@maybemoveto{\the\c@pgf@counta}{\the\c@pgf@countb}%

```

Second, create a line to the end point.

```

3907 \edef\forest@last@x{%
3908 \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname}%
3909 \edef\forest@last@y{%
3910 \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname}%
3911 \pgfsyssoftpath@lineto\forest@last@x\forest@last@y

```

Finally, “break” out of the `\forest@loopc` and `\forest@loopb`.

```

3912 \c@pgf@countc=\csname forest@pi@\forest@aplusone @n\endcsname
3913 \c@pgf@countb=\csname forest@pi@\the\c@pgf@counta @n\endcsname
3914 \fi
3915 \advance\c@pgf@countc 1
3916 \forest@repeatc
3917 \advance\c@pgf@countb 1
3918 }

```

`\forest@#1@` is an (ordered) array of points projecting to projection with index #1. Check if #2th point of that array equals the last point added to the edge: if not, add it.

```

3919 \def\forest@gettightedgeofpath@maybemoveto#1#2{%
3920 \forest@temptrue
3921 \ifx\forest@last@x\relax\else

```

```

3922 \ifdim\forest@last@x=\csname forest@pi@#1@#2x\endcsname\relax
3923 \ifdim\forest@last@y=\csname forest@pi@#1@#2y\endcsname\relax
3924 \forest@tempfalse
3925 \fi
3926 \fi
3927 \fi
3928 \ifforest@temp
3929 \edef\forest@last@x{\csname forest@pi@#1@#2x\endcsname}%
3930 \edef\forest@last@y{\csname forest@pi@#1@#2y\endcsname}%
3931 \pgfsyssoftpath@moveto\forest@last@x\forest@last@y
3932 \fi
3933 }

```

Simplify the resulting path by “unbreaking” segments where possible. (The macro itself is just a wrapper for path processing macros below.)

```

3934 \def\forest@simplifypath#1#2{%
3935 \pgfsyssoftpath@setcurrentpath\pgfutil@empty
3936 \forest@save\pgfsyssoftpath@tokendefs
3937 \let\pgfsyssoftpath@movetotoken\forest@simplifypath@moveto
3938 \let\pgfsyssoftpath@linetotoken\forest@simplifypath@lineto
3939 \let\forest@last@x\relax
3940 \let\forest@last@y\relax
3941 \let\forest@last@atan\relax
3942 #1%
3943 \ifx\forest@last@x\relax\else
3944 \ifx\forest@last@atan\relax\else
3945 \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
3946 \fi
3947 \fi
3948 \forest@restore\pgfsyssoftpath@tokendefs
3949 \pgfsyssoftpath@getcurrentpath#2%
3950 }

```

When a move-to is encountered, we flush whatever segment we were building, make the move, remember the last position, and set the slope to unknown.

```

3951 \def\forest@simplifypath@moveto#1#2{%
3952 \ifx\forest@last@x\relax\else
3953 \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
3954 \fi
3955 \pgfsyssoftpath@moveto{#1}{#2}%
3956 \def\forest@last@x{#1}%
3957 \def\forest@last@y{#2}%
3958 \let\forest@last@atan\relax
3959 }

```

How much may the segment slopes differ that we can still merge them? (Ignore pt, these are degrees.) Also, how good is this number?

```

3960 \def\forest@getedgeofpath@precision{1pt}

```

When a line-to is encountered...

```

3961 \def\forest@simplifypath@lineto#1#2{%
3962 \ifx\forest@last@x\relax

```

If we’re not in the middle of a merger, we need to nothing but start it.

```

3963 \def\forest@last@x{#1}%
3964 \def\forest@last@y{#2}%
3965 \let\forest@last@atan\relax
3966 \else

```

Otherwise, we calculate the slope of the current segment (i.e. the segment between the last and the current point), ...

```

3967 \pgfpointdiff{\pgfpoint{#1}{#2}}{\pgfpoint{\forest@last@x}{\forest@last@y}}%
3968 \ifdim\pgf@x<\pgfintersectiontolerance

```



```

3969     \ifdim-\pgf@x<\pgfintersectiontolerance
3970     \pgf@x=0pt
3971     \fi
3972 \fi
3973 \csname pgfmatahan2\endcsname{\pgf@x}{\pgf@y}%
3974 \let\forest@current@atan\pgfmathresult
3975 \ifx\forest@last@atan\relax

```

If this is the first segment in the current merger, simply remember the slope and the last point.

```

3976     \def\forest@last@x{#1}%
3977     \def\forest@last@y{#2}%
3978     \let\forest@last@atan\forest@current@atan
3979 \else

```

Otherwise, compare the first and the current slope.

```

3980     \pgfutil@tempdima=\forest@current@atan pt
3981     \advance\pgfutil@tempdima -\forest@last@atan pt
3982     \ifdim\pgfutil@tempdima<0pt\relax
3983     \multiply\pgfutil@tempdima -1
3984     \fi
3985     \ifdim\pgfutil@tempdima<\forest@getedgeofpath@precision\relax
3986     \else

```

If the slopes differ too much, flush the path up to the previous segment, and set up a new first slope.

```

3987     \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
3988     \let\forest@last@atan\forest@current@atan
3989     \fi

```

In any event, update the last point.

```

3990     \def\forest@last@x{#1}%
3991     \def\forest@last@y{#2}%
3992 \fi
3993 \fi
3994 }

```

## 11.4 Get rectangle/band edge

```

3995 \def\forest@getnegativerectangleedgeofpath#1#2{%
3996   \forest@getnegativerectangleorbandedgeofpath{#1}{#2}{\the\pgf@xb}}
3997 \def\forest@getpositiverectangleedgeofpath#1#2{%
3998   \forest@getpositiverectangleorbandedgeofpath{#1}{#2}{\the\pgf@xb}}
3999 \def\forest@getbothrectangleedgesofpath#1#2#3{%
4000   \forest@getbothrectangleorbandedgesofpath{#1}{#2}{#3}{\the\pgf@xb}}
4001 \def\forest@bandlength{5000pt} % something large (ca. 180cm), but still manageable for TeX without producing
4002 \def\forest@getnegativebandedgeofpath#1#2{%
4003   \forest@getnegativerectangleorbandedgeofpath{#1}{#2}{\forest@bandlength}}
4004 \def\forest@getpositivebandedgeofpath#1#2{%
4005   \forest@getpositiverectangleorbandedgeofpath{#1}{#2}{\forest@bandlength}}
4006 \def\forest@getbothbandedgesofpath#1#2#3{%
4007   \forest@getbothrectangleorbandedgesofpath{#1}{#2}{#3}{\forest@bandlength}}
4008 \def\forest@getnegativerectangleorbandedgeofpath#1#2#3{%
4009   \forest@path@getboundingrectangle@ls#1{\forest@grow}%
4010   \edef\forest@gre@path{%
4011     \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@ya}%
4012     \noexpand\pgfsyssoftpath@linetotoken{#3}{\the\pgf@ya}%
4013   }%
4014   {%
4015     \pgftransformreset
4016     \pgftransformrotate{\forest@grow}%
4017     \forest@pgfpathtransformed\forest@gre@path
4018   }%
4019   \pgfsyssoftpath@getcurrentpath#2%

```

```

4020 }
4021 \def\forest@getpositiverectangleorbandedgeofpath#1#2#3{%
4022   \forest@path@getboundingrectangle@ls#1{\forest@grow}%
4023   \edef\forest@gre@path{%
4024     \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@yb}%
4025     \noexpand\pgfsyssoftpath@linetotoken{#3}{\the\pgf@yb}%
4026   }%
4027   {%
4028     \pgftransformreset
4029     \pgftransformrotate{\forest@grow}%
4030     \forest@pgfpathtransformed\forest@gre@path
4031   }%
4032   \pgfsyssoftpath@getcurrentpath#2%
4033 }
4034 \def\forest@getbothrectangleorbandedgesofpath#1#2#3#4{%
4035   \forest@path@getboundingrectangle@ls#1{\forest@grow}%
4036   \edef\forest@gre@negpath{%
4037     \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@ya}%
4038     \noexpand\pgfsyssoftpath@linetotoken{#4}{\the\pgf@ya}%
4039   }%
4040   \edef\forest@gre@pospath{%
4041     \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@yb}%
4042     \noexpand\pgfsyssoftpath@linetotoken{#4}{\the\pgf@yb}%
4043   }%
4044   {%
4045     \pgftransformreset
4046     \pgftransformrotate{\forest@grow}%
4047     \forest@pgfpathtransformed\forest@gre@negpath
4048   }%
4049   \pgfsyssoftpath@getcurrentpath#2%
4050   {%
4051     \pgftransformreset
4052     \pgftransformrotate{\forest@grow}%
4053     \forest@pgfpathtransformed\forest@gre@pospath
4054   }%
4055   \pgfsyssoftpath@getcurrentpath#3%
4056 }

```

## 11.5 Distance between paths

Another crucial part of the package.

```

4057 \def\forest@distance@between@edge@paths#1#2#3{%
4058   % #1, #2 = (edge) paths
4059   %
4060   % project paths
4061   \forest@projectpathtogrowline#1{\forest@p1@}%
4062   \forest@projectpathtogrowline#2{\forest@p2@}%
4063   % merge projections (the lists are sorted already, because edge
4064   % paths are |sorted|)
4065   \forest@dbep@mergeprojections
4066     {forest@p1@}{forest@p2@}%
4067     {forest@P1@}{forest@P2@}%
4068   % process projections
4069   \forest@processprojectioninfo{forest@P1@}{forest@PI1@}%
4070   \forest@processprojectioninfo{forest@P2@}{forest@PI2@}%
4071   % break paths
4072   \forest@breakpath#1{\forest@PI1@}\forest@broken@one
4073   \forest@breakpath#2{\forest@PI2@}\forest@broken@two
4074   % sort inner arrays ---optimize: it's enough to find max and min
4075   \forest@sort@inner@arrays{forest@PI1@}\forest@sort@descending
4076   \forest@sort@inner@arrays{forest@PI2@}\forest@sort@ascending

```

```

4077 % compute the distance
4078 \let\forest@distance\relax
4079 \c@pgf@countc=0
4080 \loop
4081 \ifnum\c@pgf@countc<\csname forest@PI1@n\endcsname\relax
4082 \ifnum\csname forest@PI1@the\c@pgf@countc @n\endcsname=0 \else
4083 \ifnum\csname forest@PI2@the\c@pgf@countc @n\endcsname=0 \else
4084 \pgfutil@tempdima=\csname forest@PI2@the\c@pgf@countc @0d\endcsname\relax
4085 \advance\pgfutil@tempdima -\csname forest@PI1@the\c@pgf@countc @0d\endcsname\relax
4086 \ifx\forest@distance\relax
4087 \edef\forest@distance{\the\pgfutil@tempdima}%
4088 \else
4089 \ifdim\pgfutil@tempdima<\forest@distance\relax
4090 \edef\forest@distance{\the\pgfutil@tempdima}%
4091 \fi
4092 \fi
4093 \fi
4094 \fi
4095 \advance\c@pgf@countc 1
4096 \repeat
4097 \let#3\forest@distance
4098 }
4099 % merge projections: we need two projection arrays, both containing
4100 % projection points from both paths, but each with the original
4101 % points from only one path
4102 \def\forest@dbep@mergeprojections#1#2#3#4{%
4103 % TODO: optimize: v bistvu ni treba sortirat, ker je edge path e sortiran
4104 \forest@sortprojections{#1}%
4105 \forest@sortprojections{#2}%
4106 \c@pgf@counta=0
4107 \c@pgf@countb=0
4108 \c@pgf@countc=0
4109 \edef\forest@input@prefix@one{#1}%
4110 \edef\forest@input@prefix@two{#2}%
4111 \edef\forest@output@prefix@one{#3}%
4112 \edef\forest@output@prefix@two{#4}%
4113 \forest@dbep@mp@iterate
4114 \csedef{#3n}{\the\c@pgf@countc}%
4115 \csedef{#4n}{\the\c@pgf@countc}%
4116 }
4117 \def\forest@dbep@mp@iterate{%
4118 \let\forest@dbep@mp@next\forest@dbep@mp@iterate
4119 \ifnum\c@pgf@counta<\csname\forest@input@prefix@one n\endcsname\relax
4120 \ifnum\c@pgf@countb<\csname\forest@input@prefix@two n\endcsname\relax
4121 \let\forest@dbep@mp@next\forest@dbep@mp@do
4122 \else
4123 \let\forest@dbep@mp@next\forest@dbep@mp@iteratefirst
4124 \fi
4125 \else
4126 \ifnum\c@pgf@countb<\csname\forest@input@prefix@two n\endcsname\relax
4127 \let\forest@dbep@mp@next\forest@dbep@mp@iteratesecond
4128 \else
4129 \let\forest@dbep@mp@next\relax
4130 \fi
4131 \fi
4132 \forest@dbep@mp@next
4133 }
4134 \def\forest@dbep@mp@do{%
4135 \forest@sort@cmptwodimcs%
4136 {\forest@input@prefix@one\the\c@pgf@counta xp}%
4137 {\forest@input@prefix@one\the\c@pgf@counta yp}%

```

```

4138     {\forest@input@prefix@two\the\c@pgf@countb xp}%
4139     {\forest@input@prefix@two\the\c@pgf@countb yp}%
4140 \if\forest@sort@cmp@result=%
4141     \forest@dbep@mp@@store@p\forest@input@prefix@one\c@pgf@counta
4142     \forest@dbep@mp@@store@o\forest@input@prefix@one
4143     \c@pgf@counta\forest@output@prefix@one
4144     \forest@dbep@mp@@store@o\forest@input@prefix@two
4145     \c@pgf@countb\forest@output@prefix@two
4146     \advance\c@pgf@counta 1
4147     \advance\c@pgf@countb 1
4148 \else
4149     \if\forest@sort@cmp@result>%
4150         \forest@dbep@mp@@store@p\forest@input@prefix@two\c@pgf@countb
4151         \forest@dbep@mp@@store@o\forest@input@prefix@two
4152         \c@pgf@countb\forest@output@prefix@two
4153         \advance\c@pgf@countb 1
4154     \else%<
4155         \forest@dbep@mp@@store@p\forest@input@prefix@one\c@pgf@counta
4156         \forest@dbep@mp@@store@o\forest@input@prefix@one
4157         \c@pgf@counta\forest@output@prefix@one
4158         \advance\c@pgf@counta 1
4159     \fi
4160 \fi
4161 \advance\c@pgf@countc 1
4162 \forest@dbep@mp@iterate
4163 }
4164 \def\forest@dbep@mp@@store@p#1#2{%
4165     \csletcs
4166     {\forest@output@prefix@one\the\c@pgf@countc xp}%
4167     {#1\the#2xp}%
4168     \csletcs
4169     {\forest@output@prefix@one\the\c@pgf@countc yp}%
4170     {#1\the#2yp}%
4171     \csletcs
4172     {\forest@output@prefix@two\the\c@pgf@countc xp}%
4173     {#1\the#2xp}%
4174     \csletcs
4175     {\forest@output@prefix@two\the\c@pgf@countc yp}%
4176     {#1\the#2yp}%
4177 }
4178 \def\forest@dbep@mp@@store@o#1#2#3{%
4179     \csletcs{#3\the\c@pgf@countc xo}{#1\the#2xo}%
4180     \csletcs{#3\the\c@pgf@countc yo}{#1\the#2yo}%
4181 }
4182 \def\forest@dbep@mp@iteratefirst{%
4183     \forest@dbep@mp@iterateone\forest@input@prefix@one\c@pgf@counta\forest@output@prefix@one
4184 }
4185 \def\forest@dbep@mp@iteratesecond{%
4186     \forest@dbep@mp@iterateone\forest@input@prefix@two\c@pgf@countb\forest@output@prefix@two
4187 }
4188 \def\forest@dbep@mp@iterateone#1#2#3{%
4189     \loop
4190     \ifnum#2<\csname#1n\endcsname\relax
4191         \forest@dbep@mp@@store@p#1#2%
4192         \forest@dbep@mp@@store@o#1#2#3%
4193         \advance\c@pgf@countc 1
4194         \advance#21
4195     \repeat
4196 }

```

## 11.6 Utilities

Equality test: points are considered equal if they differ less than `\pgfintersectiontolerance` in each coordinate.

```

4197 \newif\ifforest@equaltotolerance
4198 \def\forest@equaltotolerance#1#2{%
4199   \pgfpointdiff{#1}{#2}%
4200   \ifdim\pgf@x<0pt \multiply\pgf@x -1 \fi
4201   \ifdim\pgf@y<0pt \multiply\pgf@y -1 \fi
4202   \global\forest@equaltolerancefalse
4203   \ifdim\pgf@x<\pgfintersectiontolerance\relax
4204     \ifdim\pgf@y<\pgfintersectiontolerance\relax
4205       \global\forest@equaltolerancetrue
4206     \fi
4207   \fi
4208 }}
```

Save/restore pgfs `\pgfsyssoftpath@...` token definitions.

```

4209 \def\forest@save@pgfsyssoftpath@tokendefs{%
4210   \let\forest@origmovetotoken\pgfsyssoftpath@movetotoken
4211   \let\forest@origlinetotoken\pgfsyssoftpath@linetotoken
4212   \let\forest@origcurvetosupportatoken\pgfsyssoftpath@curvetosupportatoken
4213   \let\forest@origcurvetosupportbtoken\pgfsyssoftpath@curvetosupportbtoken
4214   \let\forest@origcurvetotoken\pgfsyssoftpath@curvetototoken
4215   \let\forest@origrectcornertoken\pgfsyssoftpath@rectcornertoken
4216   \let\forest@origrectsizetoken\pgfsyssoftpath@rectsizetoken
4217   \let\forest@origclosepathtoken\pgfsyssoftpath@closepathtoken
4218   \let\pgfsyssoftpath@movetotoken\forest@badtoken
4219   \let\pgfsyssoftpath@linetotoken\forest@badtoken
4220   \let\pgfsyssoftpath@curvetosupportatoken\forest@badtoken
4221   \let\pgfsyssoftpath@curvetosupportbtoken\forest@badtoken
4222   \let\pgfsyssoftpath@curvetototoken\forest@badtoken
4223   \let\pgfsyssoftpath@rectcornertoken\forest@badtoken
4224   \let\pgfsyssoftpath@rectsizetoken\forest@badtoken
4225   \let\pgfsyssoftpath@closepathtoken\forest@badtoken
4226 }
4227 \def\forest@badtoken{%
4228   \PackageError{forest}{This token should not be in this path}{}%
4229 }
4230 \def\forest@restore@pgfsyssoftpath@tokendefs{%
4231   \let\pgfsyssoftpath@movetotoken\forest@origmovetotoken
4232   \let\pgfsyssoftpath@linetotoken\forest@origlinetotoken
4233   \let\pgfsyssoftpath@curvetosupportatoken\forest@origcurvetosupportatoken
4234   \let\pgfsyssoftpath@curvetosupportbtoken\forest@origcurvetosupportbtoken
4235   \let\pgfsyssoftpath@curvetototoken\forest@origcurvetotoken
4236   \let\pgfsyssoftpath@rectcornertoken\forest@origrectcornertoken
4237   \let\pgfsyssoftpath@rectsizetoken\forest@origrectsizetoken
4238   \let\pgfsyssoftpath@closepathtoken\forest@origclosepathtoken
4239 }
```

Extend path #1 with path #2 translated by point #3.

```

4240 \def\forest@extendpath#1#2#3{%
4241   \pgf@process{#3}%
4242   \pgfsyssoftpath@setcurrentpath#1%
4243   \forest@save@pgfsyssoftpath@tokendefs
4244   \let\pgfsyssoftpath@movetotoken\forest@extendpath@moveto
4245   \let\pgfsyssoftpath@linetotoken\forest@extendpath@lineto
4246   #2%
4247   \forest@restore@pgfsyssoftpath@tokendefs
4248   \pgfsyssoftpath@getcurrentpath#1%
4249 }
4250 \def\forest@extendpath@moveto#1#2{%
```

```

4251 \forest@extendpath@do{#1}{#2}\pgfsyssoftpath@moveto
4252 }
4253 \def\forest@extendpath@lineto#1#2{%
4254 \forest@extendpath@do{#1}{#2}\pgfsyssoftpath@lineto
4255 }
4256 \def\forest@extendpath@do#1#2#3{%
4257 {%
4258 \advance\pgf@x #1
4259 \advance\pgf@y #2
4260 #3{\the\pgf@x}{\the\pgf@y}%
4261 }%
4262 }

Get bounding rectangle of the path. #1 = the path, #2 = grow. Returns (\pgf@xa=min x/l,
\pgf@ya=max y/s, \pgf@xb=min x/l, \pgf@yb=max y/s). (If path #1 is empty, the result is undefined.)
4263 \def\forest@path@getboundingrectangle@ls#1#2{%
4264 {%
4265 \pgftransformreset
4266 \pgftransformrotate{-(#2)}%
4267 \forest@pgfpathtransformed#1%
4268 }%
4269 \pgfsyssoftpath@getcurrentpath\forest@gbr@rotatedpath
4270 \forest@path@getboundingrectangle@xy\forest@gbr@rotatedpath
4271 }
4272 \def\forest@path@getboundingrectangle@xy#1{%
4273 \forest@save\pgfsyssoftpath@tokendefs
4274 \let\pgfsyssoftpath@movetotoken\forest@gbr@firstpoint
4275 \let\pgfsyssoftpath@linetotoken\forest@gbr@firstpoint
4276 #1%
4277 \forest@restore\pgfsyssoftpath@tokendefs
4278 }
4279 \def\forest@gbr@firstpoint#1#2{%
4280 \pgf@xa=#1 \pgf@xb=#1 \pgf@ya=#2 \pgf@yb=#2
4281 \let\pgfsyssoftpath@movetotoken\forest@gbr@point
4282 \let\pgfsyssoftpath@linetotoken\forest@gbr@point
4283 }
4284 \def\forest@gbr@point#1#2{%
4285 \ifdim#1<\pgf@xa\relax\pgf@xa=#1 \fi
4286 \ifdim#1>\pgf@xb\relax\pgf@xb=#1 \fi
4287 \ifdim#2<\pgf@ya\relax\pgf@ya=#2 \fi
4288 \ifdim#2>\pgf@yb\relax\pgf@yb=#2 \fi
4289 }

```

## 12 The outer UI

### 12.1 Package options

```

4290 \newif\ifforesttikzcshack
4291 \foresttikzcshacktrue
4292 \newif\ifforest@install@keys@to@tikz@path@
4293 \forest@install@keys@to@tikz@path@true
4294 \forestset{package@options/.cd,
4295 external/.is if=forest@external@,
4296 tikzcshack/.is if=foresttikzcshack,
4297 tikzinstallkeys/.is if=forest@install@keys@to@tikz@path@,
4298 }

```

### 12.2 Externalization

```

4299 \pgfkeys{/forest/external/.cd,
4300 copy command/.initial={cp "\source" "\target"},

```

```

4301 optimize/.is if=forest@external@optimize@,
4302 context/.initial={%
4303   \forestOve{\csname forest@id@of@standard node\endcsname}{environment@formula}},
4304 depends on macro/.style={context/.append/.expanded={%
4305   \expandafter\detokenize\expandafter{#1}}},
4306 }
4307 \def\forest@external@copy#1#2{%
4308   \pgfkeysgetvalue{/forest/external/copy command}\forest@copy@command
4309   \ifx\forest@copy@command\pgfkeysnovalue\else
4310     \IfFileExists{#1}{%
4311       {%
4312         \def\source{#1}%
4313         \def\target{#2}%
4314         \immediate\write18{\forest@copy@command}%
4315       }%
4316     }{%
4317     \fi
4318 }
4319 \newif\ifforest@external@
4320 \newif\ifforest@external@optimize@
4321 \forest@external@optimize@true
4322 \ProcessPgfPackageOptions{/forest/package@options}
4323 \ifforest@install@keys@to@tikz@path@
4324   \tikzset{fit to tree/.style={/forest/fit to tree}}
4325 \fi
4326 \ifforest@external@
4327   \ifdefined\tikzexternal@tikz@replacement\else
4328     \usetikzlibrary{external}%
4329   \fi
4330   \pgfkeys{%
4331     /tikz/external/failed ref warnings for={},
4332     /pgf/images/aux in dpth=false,
4333   }%
4334   \tikzifexternalizing{}{%
4335     \forest@external@copy{\jobname.aux}{\jobname.aux.copy}%
4336   }%
4337   \AtBeginDocument{%
4338     \tikzifexternalizing{%
4339       \IfFileExists{\tikzexternalrealjob.aux.copy}{%
4340         \makeatletter
4341         \input \tikzexternalrealjob.aux.copy
4342         \makeatother
4343       }{%
4344       }{%
4345         \newwrite\forest@auxout
4346         \immediate\openout\forest@auxout=\tikzexternalrealjob.for.tmp
4347       }%
4348       \IfFileExists{\tikzexternalrealjob.for}{%
4349         {%
4350           \makehashother\makeatletter
4351           \input \tikzexternalrealjob.for
4352         }%
4353       }{%
4354       }%
4355     }
4356     \AtEndDocument{%
4357       \tikzifexternalizing{}{%
4358         \immediate\closeout\forest@auxout
4359         \forest@external@copy{\jobname.for.tmp}{\jobname.for}%
4360       }%
4361     }
4362   \fi

```

## 12.3 The forest environment

There are three ways to invoke FOREST: the environment and the starless and the starred version of the macro. The latter creates no group.

Most of the code in this section deals with externalization.

```

4362 \newenvironment{forest}{\Collect@Body\forest@env}{\fi}
4363 \long\def\Forest{\@ifnextchar*\forest@nogroup{\forest@group}}
4364 \def\forest@group#1{\forest@env{#1}}
4365 \def\forest@nogroup*#1{\forest@env{#1}}
4366 \newif\ifforest@externalize@tree@
4367 \newif\ifforest@was@tikzexternalwasenable
4368 \long\def\forest@env#1{%
4369   \let\forest@external@next\forest@begin
4370   \forest@was@tikzexternalwasenablefalse
4371   \ifdefined\tikzexternal@tikz@replacement
4372     \ifx\tikz\tikzexternal@tikz@replacement
4373       \forest@was@tikzexternalwasenabletrue
4374       \tikzexternaldisable
4375     \fi
4376   \fi
4377   \forest@externalize@tree@false
4378   \ifforest@external@
4379     \ifforest@was@tikzexternalwasenable
4380     \tikzifexternalizing{%
4381       \let\forest@external@next\forest@begin@externalizing
4382     }{%
4383       \let\forest@external@next\forest@begin@externalize
4384     }%
4385   \fi
4386   \fi
4387   \forest@standardnode@calibrate
4388   \forest@external@next{#1}%
4389 }

```

We're externalizing, i.e. this code gets executed in the embedded call.

```

4390 \long\def\forest@begin@externalizing#1{%
4391   \forest@external@setup{#1}%
4392   \let\forest@external@next\forest@begin
4393   \forest@externalize@inner@n=-1
4394   \ifforest@external@optimize@\forest@externalizing@maybeoptimize\fi
4395   \forest@external@next{#1}%
4396   \tikzexternalenable
4397 }
4398 \def\forest@externalizing@maybeoptimize{%
4399   \edef\forest@temp{\tikzexternalrealjob-forest-\forest@externalize@outer@n}%
4400   \edef\forest@marshal{%
4401     \noexpand\pgfutil@in@
4402     {\expandafter\detokenize\expandafter{\forest@temp}.}
4403     {\expandafter\detokenize\expandafter{\jobname}.}%
4404   }\forest@marshal
4405   \ifpgfutil@in@
4406   \else
4407     \let\forest@external@next@gobble
4408   \fi
4409 }

```

Externalization is enabled, we're in the outer process, deciding if the picture is up-to-date.

```

4410 \long\def\forest@begin@externalize#1{%
4411   \forest@external@setup{#1}%
4412   \iftikzexternal@file@isuptodate
4413     \setbox0=\hbox{%

```



```

4414 \csname forest@externalcheck@\forest@externalize@outer@n\endcsname
4415 }%
4416 \fi
4417 \iftikzexternal@file@isuptodate
4418 \csname forest@externalload@\forest@externalize@outer@n\endcsname
4419 \else
4420 \forest@externalize@tree@true
4421 \forest@externalize@inner@n=-1
4422 \forest@begin{#1}%
4423 \ifcsdef{forest@externalize@@\forest@externalize@id}{}%
4424 \immediate\write\forest@auxout{%
4425 \noexpand\forest@external
4426 {\forest@externalize@outer@n}%
4427 {\expandafter\detokenize\expandafter{\forest@externalize@id}}%
4428 {\expandonce\forest@externalize@checkimages}%
4429 {\expandonce\forest@externalize@loadimages}%
4430 }%
4431 }%
4432 \fi
4433 \tikzexternalenable
4434 }
4435 \def\forest@includeexternal@check#1{%
4436 \tikzsetnextfilename{#1}%
4437 \tikzexternal@externalizefig@systemcall@uptodatecheck
4438 }
4439 \def\makehashother{\catcode'\#=12}%
4440 \long\def\forest@external@setup#1{%
4441 % set up \forest@externalize@id and \forest@externalize@outer@n
4442 % we need to deal with #s correctly (\write doubles them)
4443 \setbox0=\hbox{\makehashother\makeatletter
4444 \scantokens{\forest@temp@toks{#1}}\expandafter
4445 }%
4446 \expandafter\forest@temp@toks\expandafter{\the\forest@temp@toks}%
4447 \edef\forest@temp{\pgfkeysvalueof{/forest/external/context}}%
4448 \edef\forest@externalize@id{%
4449 \expandafter\detokenize\expandafter{\forest@temp}%
4450 @@%
4451 \expandafter\detokenize\expandafter{\the\forest@temp@toks}%
4452 }%
4453 \letcs\forest@externalize@outer@n\forest@externalize@@\forest@externalize@id}%
4454 \ifdefined\forest@externalize@outer@n
4455 \global\tikzexternal@file@isuptodate true
4456 \else
4457 \global\advance\forest@externalize@max@outer@n 1
4458 \edef\forest@externalize@outer@n{\the\forest@externalize@max@outer@n}%
4459 \global\tikzexternal@file@isuptodate false
4460 \fi
4461 \def\forest@externalize@loadimages{}%
4462 \def\forest@externalize@checkimages{}%
4463 }
4464 \newcount\forest@externalize@max@outer@n
4465 \global\forest@externalize@max@outer@n=0
4466 \newcount\forest@externalize@inner@n

```

The .for file is a string of calls of this macro.

```

4467 \long\def\forest@external#1#2#3#4{% #1=n,#2=context+source code,#3=update check code, #4=load code
4468 \ifnum\forest@externalize@max@outer@n<#1
4469 \global\forest@externalize@max@outer@n=#1
4470 \fi
4471 \global\csdef{forest@externalize@@\detokenize{#2}}{#1}%
4472 \global\csdef{forest@externalcheck@#1}{#3}%

```

```

4473 \global\csdef{forest@externalload@#1}{#4}%
4474 \tikzifexternalizing{}{%
4475   \immediate\write\forest@auxout{%
4476     \noexpand\forest@external{#1}%
4477     {\expandafter\detokenize\expandafter{#2}}}%
4478     {\unexpanded{#3}}}%
4479     {\unexpanded{#4}}}%
4480   }%
4481 }%
4482 }

```

These two macros include the external picture.

```

4483 \def\forest@includeexternal#1{%
4484   \edef\forest@temp{\pgfkeysvalueof{/forest/external/context}}%
4485   \typeout{forest: Including external picture '#1' for forest context+code:
4486     '\expandafter\detokenize\expandafter{\forest@externalize@id}'}%
4487   {%
4488     %\def\pgf@declaredraftimage##1##2{\def\pgf@image{\hbox{}}}%
4489     \tikzsetnextfilename{#1}%
4490     \tikzexternalenable
4491     \tikz{}%
4492   }%
4493 }
4494 \def\forest@includeexternal@box#1#2{%
4495   \global\setbox#1=\hbox{\forest@includeexternal{#2}}%
4496 }

```

This code runs the bracket parser and stage processing.

```

4497 \long\def\forest@begin#1{%
4498   \iffalse{\fi\forest@parsebracket#1}%
4499 }
4500 \def\forest@parsebracket{%
4501   \bracketParse{\forest@get@root@afterthought}\forest@root=%
4502 }
4503 \def\forest@get@root@afterthought{%
4504   \expandafter\forest@get@root@afterthought@expandafter{\iffalse}\fi
4505 }
4506 \long\def\forest@get@root@afterthought@#1{%
4507   \ifblank{#1}{\}{%
4508     \forest@oappto{\forest@root}{given options}{,afterthought={\unexpanded{#1}}}%
4509   }%
4510   \forest@do
4511 }
4512 \def\forest@do{%
4513   \forest@node@Compute@numeric@ts@info{\forest@root}%
4514   \forestset{process keylist=given options}%
4515   \forestset{stages}%
4516   \ifforest@was@tikzexternalwasenable
4517     \tikzexternalenable
4518   \fi
4519 }

```

## 12.4 Standard node

The standard node should be calibrated when entering the forest env: The standard node init does *not* initialize options from a(nother) standard node!

```

4520 \def\forest@standardnode@new{%
4521   \advance\forest@node@maxid1
4522   \forest@fornode{\the\forest@node@maxid}{%
4523     \forest@node@init
4524     \forest@node@setname{standard node}%

```

```

4525 }%
4526 }
4527 \def\forest@standardnode@calibrate{%
4528   \forest@fornode{\forest@node@Nametoid{standard node}}{%
4529     \edef\forest@environment{\forest@ve{environment@formula}}%
4530     \forest@toget{previous@environment}\forest@previous@environment
4531     \ifx\forest@environment\forest@previous@environment\else
4532       \forest@tolet{previous@environment}\forest@environment
4533       \forest@node@typeset
4534       \forest@toget{calibration@procedure}\forest@temp
4535       \expandafter\forestset\expandafter{\forest@temp}%
4536     \fi
4537   }%
4538 }

```

Usage: `\forestStandardNode [#1]{#2}{#3}{#4}`. #1 = standard node specification — specify it as any other node content (but without children, of course). #2 = the environment fingerprint: list the values of parameters that influence the standard node's height and depth; the standard will be adjusted whenever any of these parameters changes. #3 = the calibration procedure: a list of usual forest options which should calculating the values of exported options. #4 = a comma-separated list of exported options: every newly created node receives the initial values of exported options from the standard node. (The standard node definition is local to the  $\text{T}_\text{E}\text{X}$  group.)

```

4539 \def\forestStandardNode[#1]#2#3#4{%
4540   \let\forest@standardnode@restoretikzexternal\relax
4541   \ifdefined\tikzexternaldisable
4542     \ifx\tikz\tikzexternal\tikz@replacement
4543       \tikzexternaldisable
4544       \let\forest@standardnode@restoretikzexternal\tikzexternalenable
4545     \fi
4546   \fi
4547   \forest@standardnode@new
4548   \forest@fornode{\forest@node@Nametoid{standard node}}{%
4549     \forestset{content=#1}%
4550     \forest@set{environment@formula}{#2}%
4551     \edef\forest@temp{\unexpanded{#3}}%
4552     \forest@tolet{calibration@procedure}\forest@temp
4553     \def\forest@calibration@initializing@code{%
4554       \pgfkeys{/forest/initializing@code}{#4}%
4555       \forest@tolet{initializing@code}\forest@calibration@initializing@code
4556     \forest@standardnode@restoretikzexternal
4557   }
4558 }
4559 \forestset{initializing@code/.unknown/.code={%
4560   \eappto\forest@calibration@initializing@code{%
4561     \noexpand\forest@toget{\forest@node@Nametoid{standard node}}{\pgfkeyscurrentname}\noexpand\forest@temp
4562     \noexpand\forest@tolet{\pgfkeyscurrentname}\noexpand\forest@temp
4563   }%
4564 }
4565 }

```

This macro is called from a new (non-standard) node's init.

```

4566 \def\forest@initializefromstandardnode{%
4567   \forest@ve{\forest@node@Nametoid{standard node}}{initializing@code}%
4568 }

```

Define the default standard node. Standard content: dj — in Computer Modern font, d is the highest and j the deepest letter (not character!). Environment fingerprint: the height of the strut and the values of inner and outer seps. Calibration procedure: (i) `1 sep` equals the height of the strut plus the value of `inner ysep`, implementing both font-size and inner sep dependency; (ii) The effect of `l` on the standard node should be the same as the effect of `1 sep`, thus, we derive `l` from `1 sep` by adding to the latter the total height of the standard node (plus the double outer sep, one for the parent and one for the

child). (iii) `s sep` is straightforward: a double inner `xsep`. Exported options: options, calculated in the calibration. (Tricks: to change the default anchor, set it in #1 and export it; to set a non-forest node option (such as `draw` or `blue`) as default, set it in #1 and export the (internal) option `node options`.)

```

4569 \forestStandardNode[dj]
4570 {%
4571   \forestOve{\forest@node@Nametoid{standard node}}{content},%
4572   \the\ht\strutbox,\the\pgflinewidth,%
4573   \pgfkeysvalueof{/pgf/inner ysep},\pgfkeysvalueof{/pgf/outer ysep},%
4574   \pgfkeysvalueof{/pgf/inner xsep},\pgfkeysvalueof{/pgf/outer xsep}%
4575 }
4576 {
4577   l sep={\the\ht\strutbox+\pgfkeysvalueof{/pgf/inner ysep}},
4578   l={l sep()+abs(max_y()-min_y())+2*\pgfkeysvalueof{/pgf/outer ysep}},
4579   s sep={2*\pgfkeysvalueof{/pgf/inner xsep}}
4580 }
4581 {l sep,l,s sep}

```

## 12.5 ls coordinate system

```

4582 \pgfkeys{/forest/@cs}{%
4583   name/.code={%
4584     \edef\forest@cn{\forest@node@Nametoid{#1}}%
4585     \forest@forestcs@resetxy},
4586   id/.code={%
4587     \edef\forest@cn{#1}%
4588     \forest@forestcs@resetxy},
4589   go/.code={%
4590     \forest@go{#1}%
4591     \forest@forestcs@resetxy},
4592   anchor/.code={\forest@forestcs@anchor{#1}},
4593   l/.code={%
4594     \pgfmathsetlengthmacro\forest@forestcs@l{#1}%
4595     \forest@forestcs@ls
4596   },
4597   s/.code={%
4598     \pgfmathsetlengthmacro\forest@forestcs@s{#1}%
4599     \forest@forestcs@ls
4600   },
4601   .unknown/.code={%
4602     \expandafter\pgfutil@in@\expandafter.\expandafter{\pgfkeyscurrentname}%
4603     \ifpgfutil@in@
4604       \expandafter\forest@forestcs@namegoanchor\pgfkeyscurrentname\forest@end
4605     \else
4606       \expandafter\forest@nameandgo\expandafter{\pgfkeyscurrentname}%
4607       \forest@forestcs@resetxy
4608     \fi
4609   }
4610 }
4611 \def\forest@forestcs@resetxy{%
4612   \ifnum\forest@cn=0
4613   \else
4614     \global\pgf@x\forestove{x}%
4615     \global\pgf@y\forestove{y}%
4616   \fi
4617 }
4618 \def\forest@forestcs@ls{%
4619   \ifdefined\forest@forestcs@l
4620   \ifdefined\forest@forestcs@s
4621     {%
4622       \pgftransformreset

```

```

4623     \pgftransformrotate{\forestove{grow}}}%
4624     \pgfpointransformed{\pgfpoint{\forest@forestcs@l}{\forest@forestcs@s}}}%
4625   }%
4626   \global\advance\pgf@x\forestove{x}%
4627   \global\advance\pgf@y\forestove{y}%
4628   \fi
4629 \fi
4630 }
4631 \def\forest@forestcs@anchor#1{%
4632   \edef\forest@marshal{%
4633     \noexpand\forest@original@tikz@parse@node\relax
4634     (\forestove{name}\ifx\relax#1\relax\else.\fi#1)%
4635   }\forest@marshal
4636 }
4637 \def\forest@forestcs@namegoanchor#1.#2\forest@end{%
4638   \forest@nameandgo{#1}%
4639   \forest@forestcs@anchor{#2}%
4640 }
4641 \tikzdeclarecoordinatesystem{forest}{%
4642   \forest@forthis{%
4643     \forest@forestcs@resetxy
4644     \ifdefined\forest@forestcs@l\undef\forest@forestcs@l\fi
4645     \ifdefined\forest@forestcs@s\undef\forest@forestcs@s\fi
4646     \pgfqkeys{/forest/@cs}{#1}%
4647   }%
4648 }

```