

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

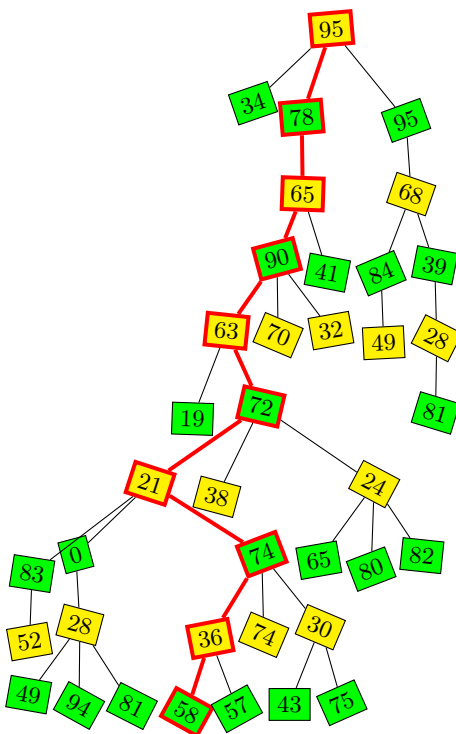
v1.0.10

Sašo Živanović\*

July 15, 2015

## 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>3</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Tutorial</b>	<b>3</b>
2.1	Basic usage . . . . .	3
2.2	Options . . . . .	5
2.3	Decorating the tree . . . . .	7
2.4	Node positioning . . . . .	9
2.4.1	The defaults, or the hairy details of vertical alignment . . . . .	15
2.5	Advanced option setting . . . . .	17
2.6	Externalization . . . . .	19
2.7	Expansion control in the bracket parser . . . . .	20
<b>3</b>	<b>Reference</b>	<b>21</b>
3.1	Environments . . . . .	21
3.2	The bracket representation . . . . .	21
3.3	Options and keys . . . . .	22
3.3.1	Node appearance . . . . .	24
3.3.2	Node position . . . . .	26
3.3.3	Edges . . . . .	31
3.3.4	Readonly . . . . .	33
3.3.5	Miscellaneous . . . . .	33
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 . . . . .	41
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>6</b>	<b>Changelog</b>	<b>54</b>
<b>II</b>	<b>Implementation</b>	<b>56</b>
<b>7</b>	<b>Patches</b>	<b>56</b>
<b>8</b>	<b>Utilities</b>	<b>60</b>
8.1	Sorting . . . . .	63
<b>9</b>	<b>The bracket representation parser</b>	<b>66</b>
9.1	The user interface macros . . . . .	66
9.2	Parsing . . . . .	67

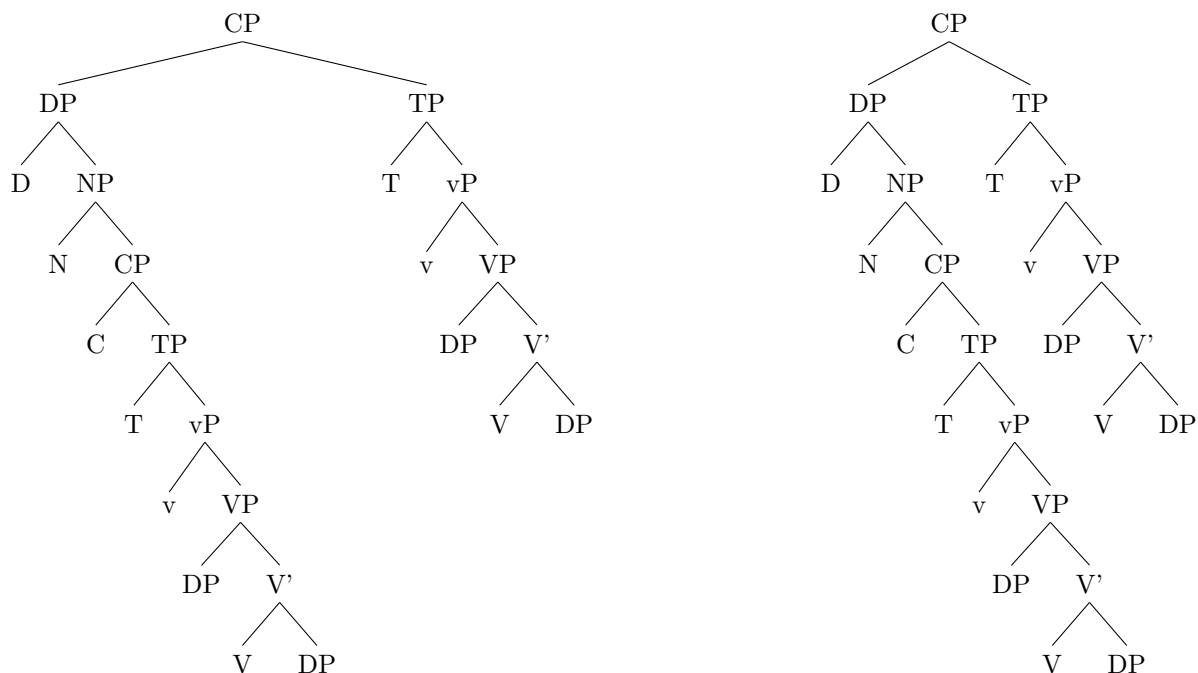
9.3	The tree-structure interface	71
<b>10</b>	<b>Nodes</b>	<b>72</b>
10.1	Option setting and retrieval	72
10.2	Tree structure	74
10.3	Node walk	80
10.4	Node options	83
10.4.1	Option-declaration mechanism	83
10.4.2	Declaring options	89
10.4.3	Option propagation	92
10.4.4	pgfmath extensions	93
10.5	Dynamic tree	94
<b>11</b>	<b>Stages</b>	<b>96</b>
11.1	Typesetting nodes	97
11.2	Packing	99
11.2.1	Tiers	109
11.2.2	Node boundary	113
11.3	Compute absolute positions	118
11.4	Drawing the tree	118
<b>12</b>	<b>Geometry</b>	<b>121</b>
12.1	Projections	121
12.2	Break path	124
12.3	Get tight edge of path	126
12.4	Get rectangle/band edge	132
12.5	Distance between paths	133
12.6	Utilities	136
<b>13</b>	<b>The outer UI</b>	<b>137</b>
13.1	Package options	137
13.2	Externalization	138
13.3	The forest environment	139
13.4	Standard node	142
13.5	ls coordinate system	143
	<b>References</b>	<b>144</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 `syntree`, 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 `LATEX` 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!

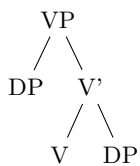
FOREST is [available](#) at [CTAN](#), and I have also started a [style repository](#) at [GitHub](#).

### 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.

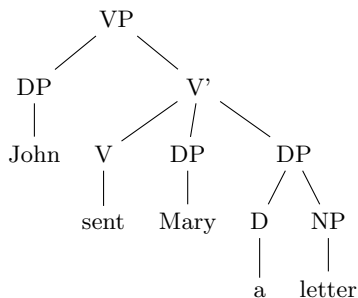


```

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

```

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.

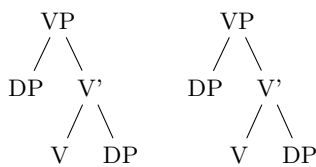


```

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

```

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

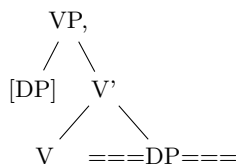


```

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

```

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 =.

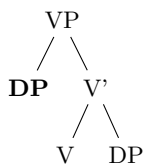


```

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

```

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

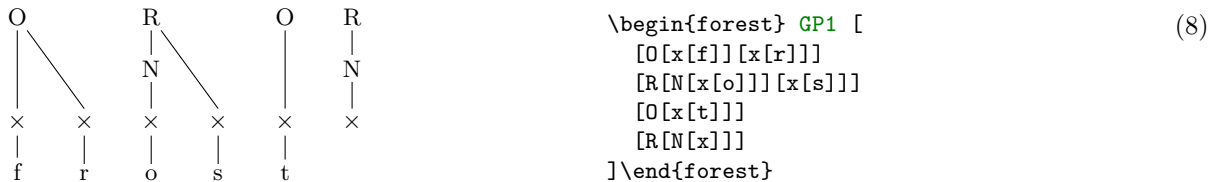


```

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

```

All the examples given above produced top-down trees with centered children. The other sections of this manual explain how various properties of a tree can be changed, making it possible to typeset radically different-looking trees. However, you don't have to learn everything about this package to profit from its power. Using styles, you can draw predefined types of trees with ease. For example, a phonologist can use the **GP1** style from §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.



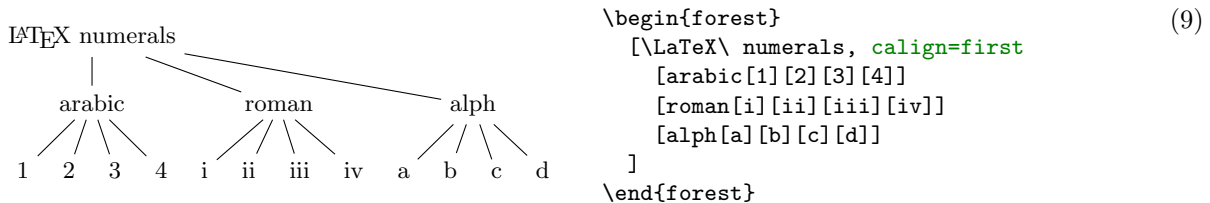
Of course, someone needs to develop the style — you, me, your local TeXnician . . . Fortunately, designing styles is not very difficult once you know your FOREST options. If you write one, please contribute!

I have started a [style repository](#) at GitHub. Hopefully, it will grow . . . Check it out, download the styles . . . and contribute them!

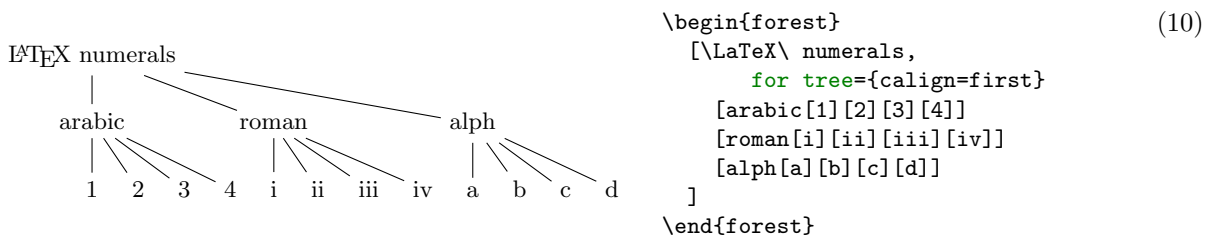
## 2.2 Options

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  $\langle option\ name \rangle = \langle option\ value \rangle$ . (There are also options which do not require a value or have a default value: these are given simply as  $\langle option\ name \rangle$ .)



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:

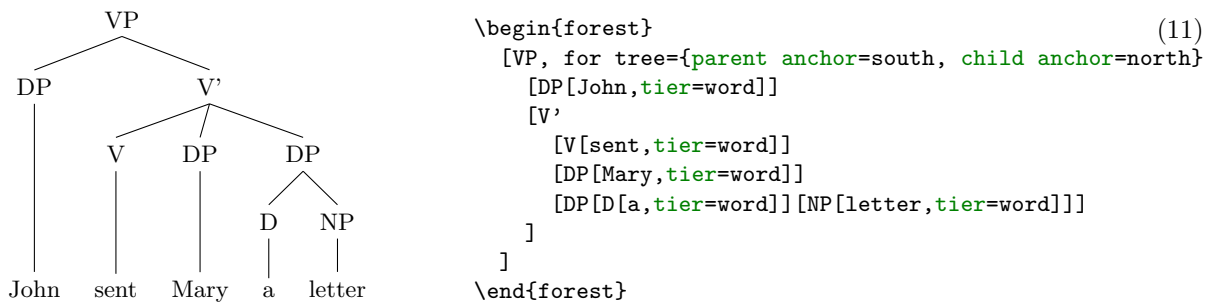


The value of propagator `for tree` is the option string that we want to process. This option string is propagated to all the nodes in the subtree<sup>1</sup> rooted in the current node (i.e. the node where `for tree` was given), including the node itself. (Propagator `for descendants` is just like `for 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.)

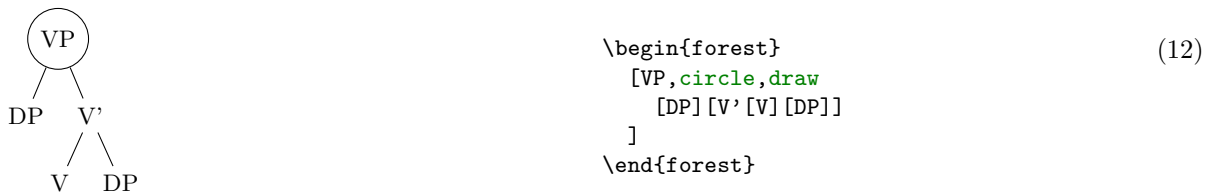
Some other useful options are `parent anchor`, `child anchor` and `tier`. The `parent anchor` and `child anchor` options tell where the parent's and child's endpoint of the edge between them should be, respectively: usually, the value is either empty (meaning a smartly determined border point [see ? , §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!)

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

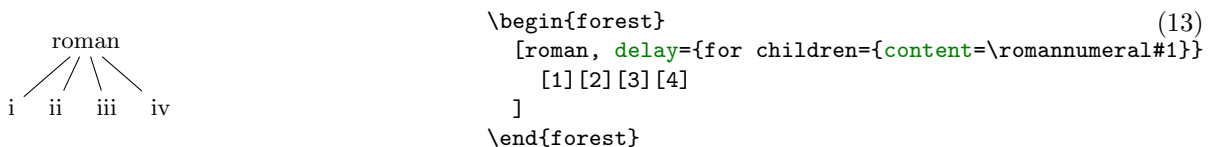
Option `tier` is what makes the skeletal points  $\times$  in example (8) align horizontally although they occur at different levels in the logical structure of the tree. Using option `tier` is very simple: just set `tier=tier name` at all the nodes that you want to align horizontally. Any tier name will do, as long as the tier names of different tiers are different ... (Yes, you can have multiple tiers!)



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>



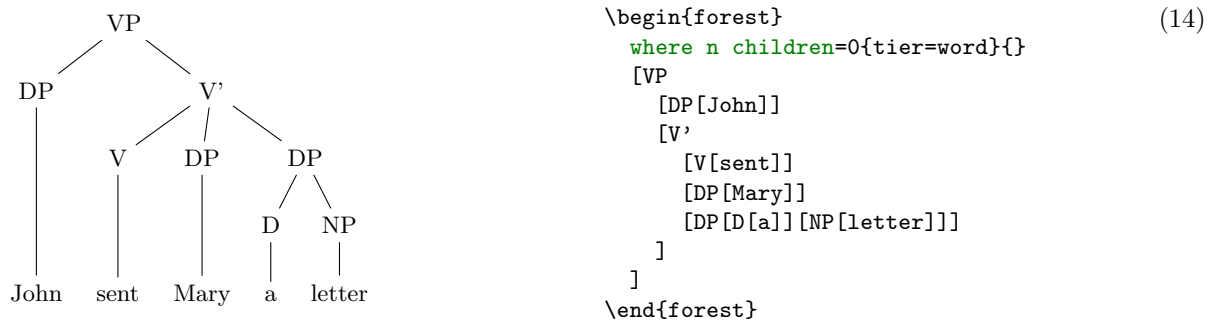
This example introduces another option: `delay`. Without it, the example wouldn't work: we would get arabic numerals. This is so because of the order in which the options are processed. The processing proceeds through the tree in a depth-first, parent-first fashion (first the parent is processed, and 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.

<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].

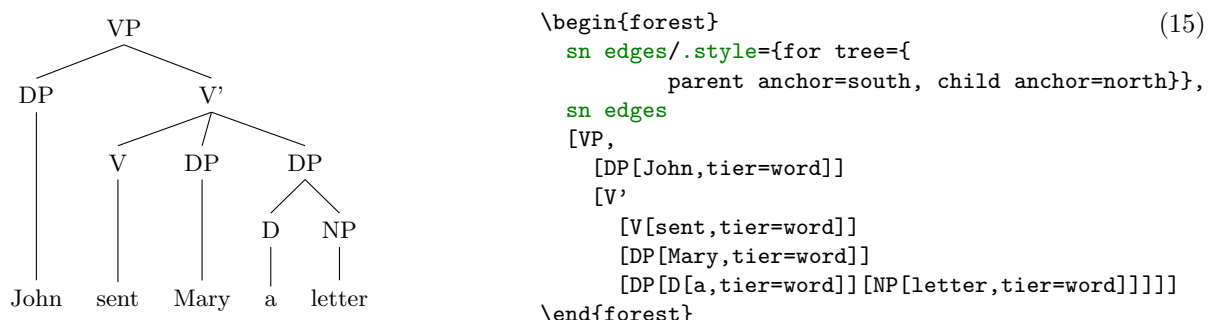
<sup>3</sup>This mechanism is called *wrapping*. `content` is the only option where wrapping works implicitly (simply because I assume that wrapping will be almost exclusively used with this option). To wrap values of other options, use handler `.wrap value`; see §3.4.

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 `<option>` equals `<value>`. If so, `<true options>` are processed, otherwise `<>false options>`. 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>



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.)



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}...`



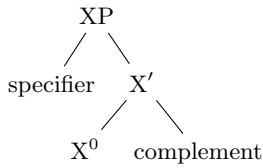


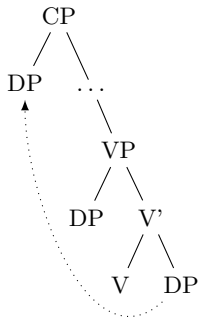
Figure 1: The X' template

```

\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.

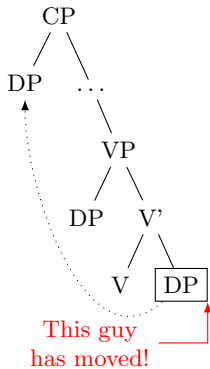


```

\begin{forest}
  [CP
    [DP, name=spec CP]
    [\dots
      [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
      [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}

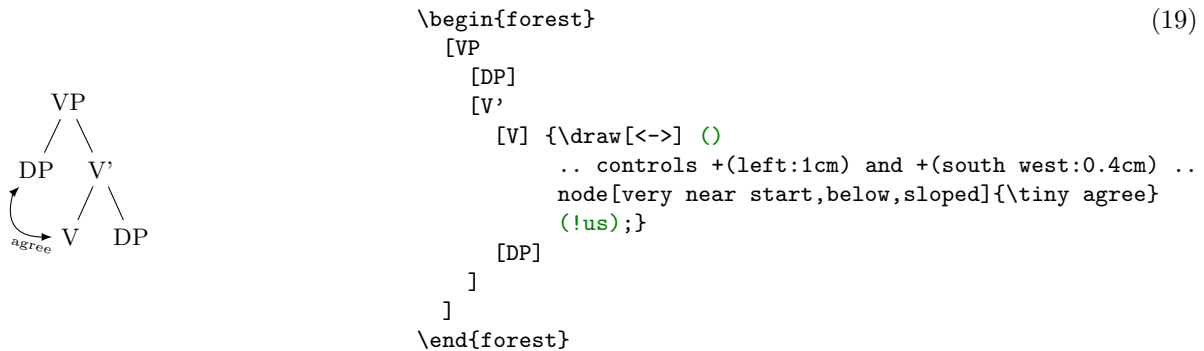
```

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.)

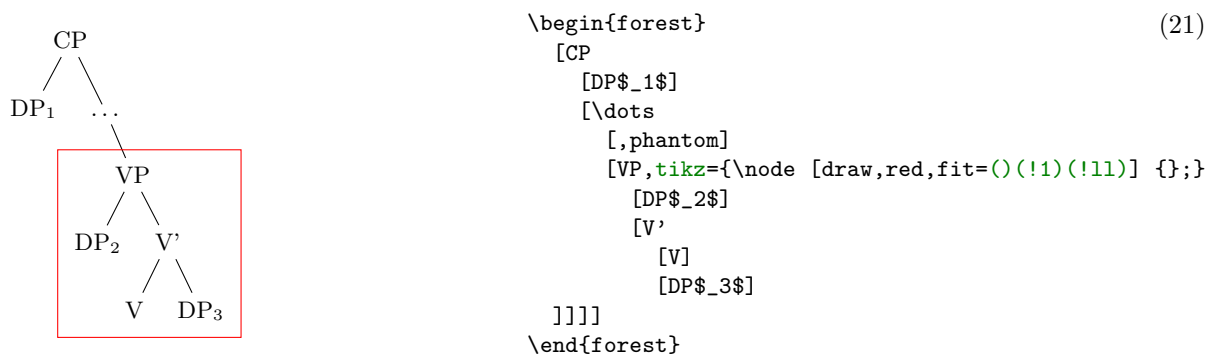
Finally, the most powerful tool in the node reference toolbox: *relative nodes*. It is possible to refer to other nodes which stand in some (most often geometrical) relation to the current node. To do this, follow the node's name with a `!` and a *node walk* specification.

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>

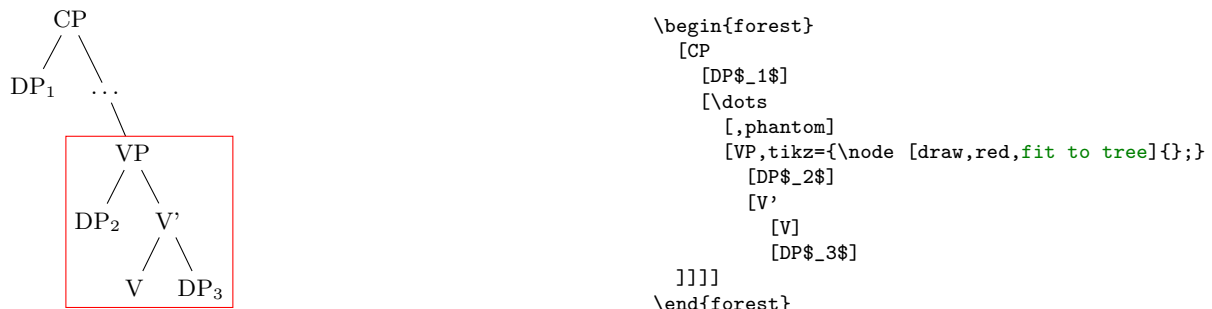


## 2.4 Node positioning

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

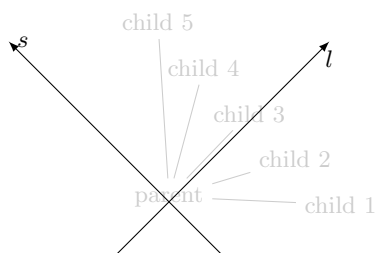
<sup>5</sup>Actually, FOREST distinguishes two kinds of steps in node walks: long and short steps. This section introduces only short steps. See §3.5.1.

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



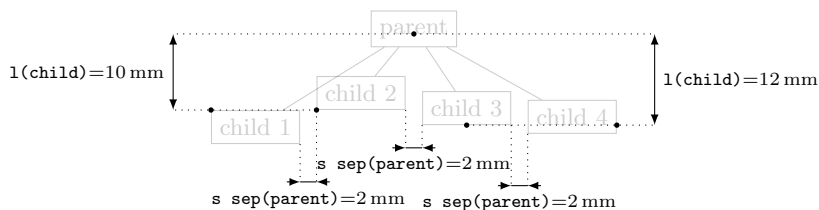
<sup>7</sup>The direction can be specified either in degrees (following the standard mathematical convention that 0 degrees is to the right, and that degrees increase counter-clockwise) or by the compass directions: `east`, `north east`, `north`, etc.

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>Opt
      \draw[dimension,<-] (md1)--+#7; \draw[dimension,<-] let \p1=(md1)-(md2) in (md2)--+(\p1);
    \else
      \draw[dimension,<-] let \p1=(md1)-(md2) in (md1)--+(\p1); \draw[dimension,<-] (md2)--+#7;
    \fi
  \draw[dimension,-] (md1)--(md2) node[#4]{\distancelabelwrapper{\uselengthunit{mm}\rndprintlength\absmd}};
\fi}
```

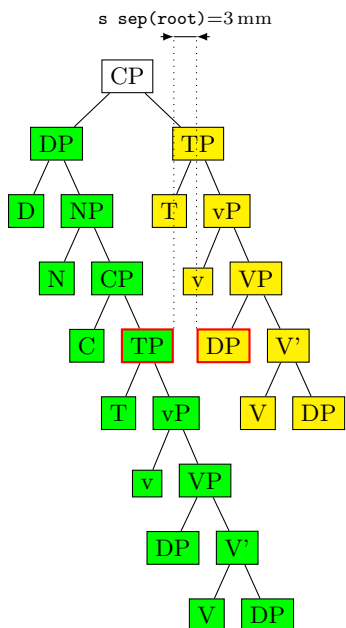
```

\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)}{(!1.anchor)+(5mm,0)}{right}
\measurexdistance[\texttt{s sep(parent)}=#1]{(!1.south east)}{(!2.south west)}{+(0,-5mm)}{below}
\measurexdistance[\texttt{s sep(parent)}=#1]{(!2.south east)}{(!3.south west)}{+(0,-5mm)}{below}
\measurexdistance[\texttt{s sep(parent)}=#1]{(!3.south east)}{(!4.south west)}{+(0,-8mm)}{below}
\end{forest}

```

Positioning the 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`.



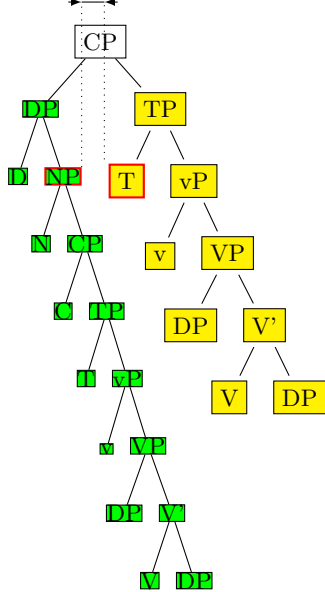
```

\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}

```

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



```

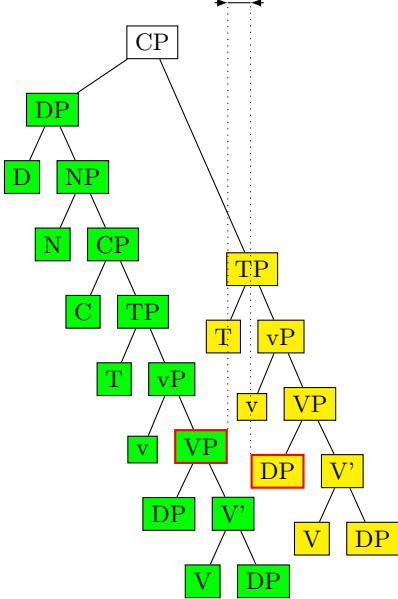
\begin{forest}
important/.style={name=#1,draw={red,thick}}
[CP, s sep=3mm, for tree=draw
  [DP, for tree={fill=green,inner sep=0}
    [D] [NP,important=left[N] [CP[C] [TP[T] [vP[v]
      [VP[DP] [V' [V] [DP]]]]]]]]]
  [TP,for tree={fill=yellow,outer sep=2pt}
    [T,important=right] [vP[v] [VP[DP] [V'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

```

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

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

s sep(root)=3 mm

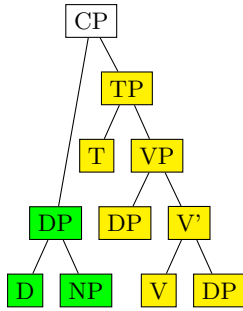


```

\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}

```

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.)



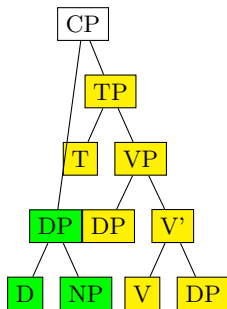
```

\begin{forest}
  (28)
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D] [NP]]
    [TP,for tree={fill=yellow}
      [T] [VP [DP] [V' [V] [DP]]]]]
  ]
\end{forest}

```

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

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

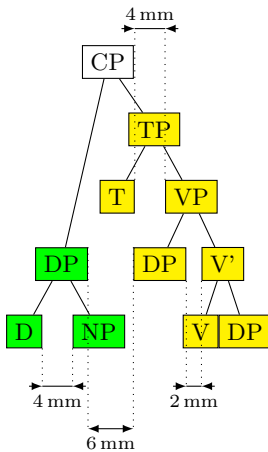


```

\begin{forest}
  (29)
  [CP, for tree=draw, s sep=0
    [DP, for tree={fill=green},l*=3
      [D] [NP]]
    [TP,for tree={fill=yellow}
      [T] [VP [DP] [V' [V] [DP]]]]]
  ]
\end{forest}

```

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



```

\begin{forest}
  (30)
  for tree={s sep=(3-level)*2mm}
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D] [NP]]
    [TP,for tree={fill=yellow}
      [T] [VP [DP] [V' [V] [DP]]]]]
  ]
  \measurexdistance{(!11.south east)}{(!12.south west)}{+(0,-5mm)}{below}
  \path(md2)-|coordinate(md){(!221.south east)};
  \measurexdistance{(!221.south east)}{(!222.south west)}{(md)}{below}
  \measurexdistance{(!21.north east)}{(!22.north west)}{+(0,2cm)}{above}
  \measurexdistance{(!1.north east)}{(!221.north west)}{+(0,-2.4cm)}{below}
\end{forest}

```

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

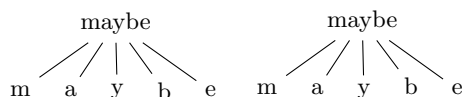
As you have probably noticed, this example also demonstrated that we can compute the value of an option using an (arbitrarily complex) formula. This is thanks to PGF's module `pgfmath`. FOREST provides an interface to `pgfmath` by defining `pgfmath` functions for every node option, and some other



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

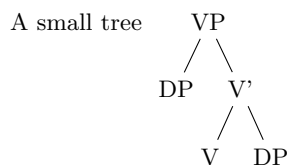
In this section we discuss the default values of options controlling the l-alignment of the nodes. The defaults are set with top-down trees in mind, so l-alignment is actually vertical alignment. There are two desired effects of the defaults. First, the spacing between the nodes of a tree should adjust 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:

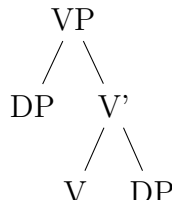


```
\begin{forest} for tree={anchor=center} (33)
  [maybe[m] [a] [y] [b] [e]]
\end{forest}\quad
\begin{forest}
  [maybe[m] [a] [y] [b] [e]]
\end{forest}
```

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



and a large tree

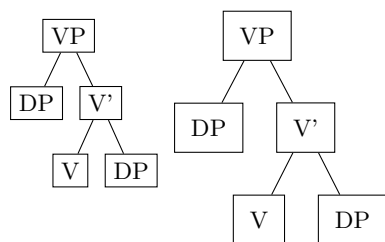


```
\hbox{\small A small tree (34)
\begin{forest} baseline
  [VP [DP] [V' [V] [DP]]]
\end{forest}
\normalsize and
\large
a large tree
\begin{forest} baseline
  [VP [DP] [V' [V] [DP]]]
\end{forest}}
```

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

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

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



```
\begin{forest} baseline,for tree=draw (35)
  [VP [DP] [V' [V] [DP]]]
\end{forest}
\pgfkeys{/pgf/inner sep=0.6666em}
\begin{forest} baseline,for tree=draw
  [VP [DP] [V' [V] [DP]]]
\end{forest}
```

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).



```

\begin{forest}
[default,baseline,for children={no edge}
[DP
[AdjP[Adj]
[D' [D] [NP,name=np]]]]]
\path (current bounding box.west)|-coordinate(1)(np.base);
\path (current bounding box.east)|-coordinate(r)(np.base);
\draw[dotted] (1)--(r);
\end{forest}
\begin{forest}
[{\l=0},baseline,for children={no edge}
[DP,for descendants={\l=0}
[AdjP[Adj]
[D' [D] [NP,name=np]]]]]
\path (current bounding box.west)|-coordinate(1)(np.base);
\path (current bounding box.east)|-coordinate(r)(np.base);
\draw[dotted] (1)--(r);
\end{forest}

```

default

l=0

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 ...

```

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

```

default

l sep=0

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.)

```

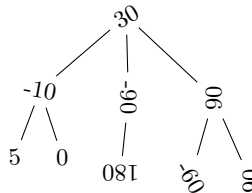
c1   o2   u3   n4   t5
\begin{forest}
  [,phantom,delay={for descendants={
    content=#1$_{\forestoption{n}}$}]
  [c] [o] [u] [n] [t]]
\end{forest}

```

(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:



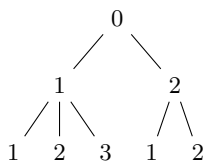
```

\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).



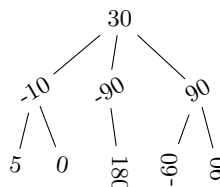
```

\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")}}]
  [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={1*=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.

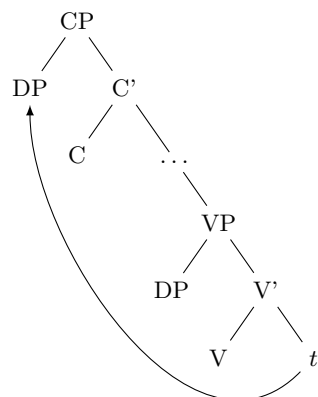
```

\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

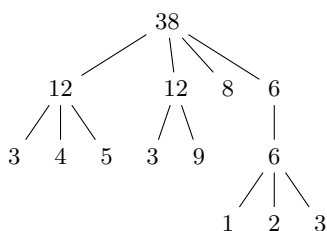
`##1`: `#1` is the argument of the style move, i.e. the given node walk, while `##1` is the original option value (in this case, `content`).



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

(45)

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.



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

(46)

## 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  $\TeX$  group.

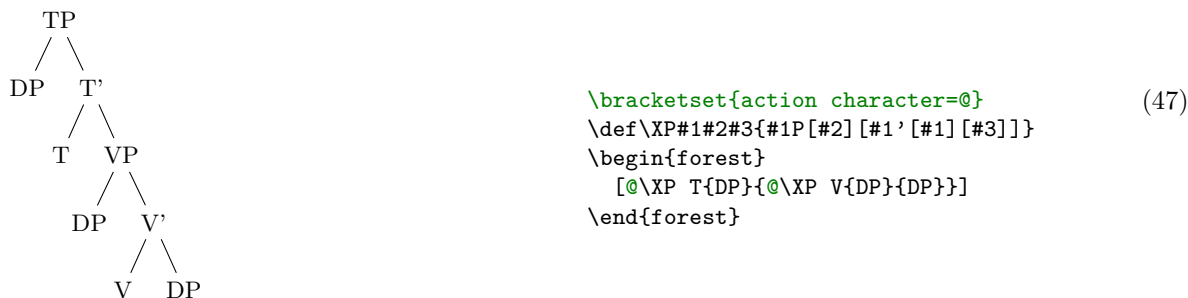
TikZ's externalization library promises a `\label` inside the externalized graphics to work out-of-box, while `\ref` inside the externalized graphics should work only if the externalization is run manually or by `make` [?, §32.4.1]. A bit surprisingly perhaps, the situation is roughly reversed in FOREST. `\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

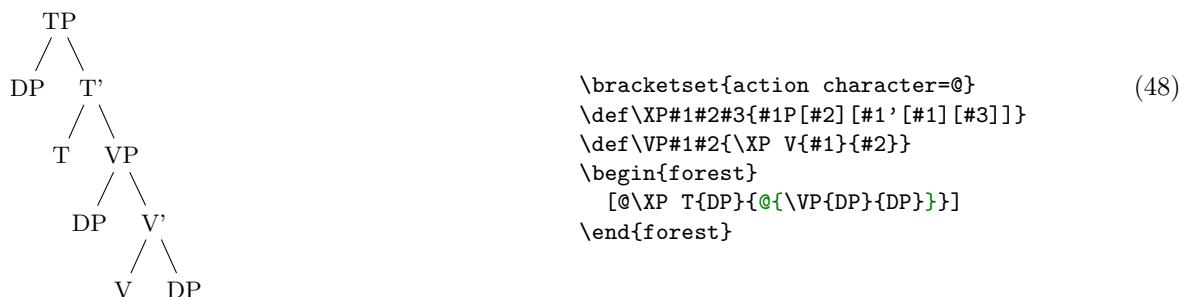
By default, macros in the bracket encoding of a tree are not expanded until nodes are being drawn — this way, node specification can contain formatting instructions, as illustrated in section 2.1. However, sometimes it is useful to expand macros while parsing the bracket representation, for example to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



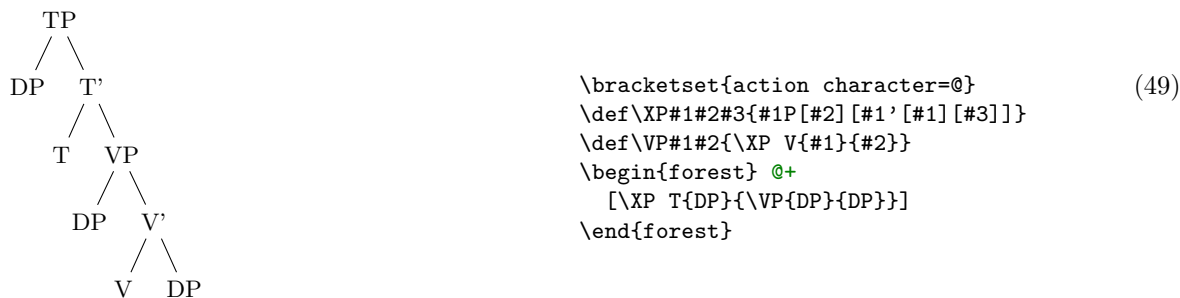
In the above example, the `\XP` macro is preceded by the *action character* `@`: as the result, the token following the action character was expanded before the parsing proceeded.

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  $\text{T}_{\text{E}}\text{X}$ 's macro expansion. There is one final action command,  $\text{@@}$ , which yields control to the user code and expects it to call  $\backslash\text{bracketResume}$  to resume parsing. This is useful to e.g. implement automatic node enumeration:

$\times_1$	$\times_2$	$\times_3$	$\times_4$	$\times_5$	$\times_6$	<pre style="margin: 0;"> \bracketset{action character=@} \newcount\xcount \def\x#1{@@\advance\xcount1   \edef\xtmp{[\$\noexpand\times_{\the\xcount}\\$#1]}%   \expandafter\bracketResume\xtmp } \begin{forest}   phantom,   delay={where level=1{content={\strut #1}}{}} @+   [\x{f}\x{o}\x{r}\x{e}\x{s}\x{t}] \end{forest} </pre>
f	o	r	e	s	t	

(50)

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

## 3 Reference

### 3.1 Environments

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

$\backslash\text{Forest}[*]\{\langle\text{tree}\rangle\}$

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  $\backslash\text{forest}$  namespace.)

### 3.2 The bracket representation

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

```

⟨tree⟩ = [⟨preamble⟩]⟨node⟩
⟨node⟩ = [ [⟨content⟩] [ ,⟨keylist⟩ ] [⟨children⟩] ]⟨afterthought⟩
⟨preamble⟩ = ⟨keylist⟩
⟨keylist⟩ = ⟨key-value⟩ [ ,⟨keylist⟩ ]
⟨key-value⟩ = ⟨key⟩ | ⟨key⟩=⟨value⟩
⟨children⟩ = ⟨node⟩ [⟨children⟩ ]

```

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.

$\langle\text{action character}\rangle-$	no-expansion mode (default): nothing is expanded
$\langle\text{action character}\rangle+$	expansion mode: everything is fully expanded
$\langle\text{action character}\rangle\langle\text{token}\rangle$	expand $\langle\text{token}\rangle$
$\langle\text{action character}\rangle\langle\text{T}_{\text{E}}\text{X-group}\rangle$	fully expand $\langle\text{T}_{\text{E}}\text{X-group}\rangle$
$\langle\text{action character}\rangle\langle\text{action character}\rangle$	yield control; upon finishing its job, user's code should call $\backslash\text{bracketResume}$

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

```
opening bracket=⟨character⟩ [
closing bracket=⟨character⟩ ]
action character=⟨character⟩ none
```

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

**new node**=⟨preamble⟩⟨node specification⟩⟨csize⟩. 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 ⟨csize⟩.

**set afterthought**=⟨afterthought⟩⟨node id⟩. 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 *keylist*. 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:

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

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:

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

Notes: (i) `⟨value⟩` is evaluated in the context of the current node. (ii) In general, the resolution of `⟨relative node name⟩` depends on the current node; see §3.5. (iii) `⟨option⟩` 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 `⟨value⟩` 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 TeX'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 TeX'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 TeX `<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 TeX assignment.

The two above-mentioned speed-up tricks work for the augmented assignments as well. The keys for the second, TeX-only trick are: `<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  $\TeX$  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
<code>left</code>	<code>@{}l@{}</code>
<code>center</code>	<code>@{}c@{}</code>
<code>right</code>	<code>@{}r@{}</code>

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}

```

(51)

<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.

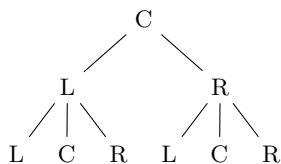
*option* `base`= $\langle$ *toks: vertical alignment* $\rangle$  t

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. {}

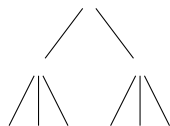
Normally, the value of option `content` is given implicitly by virtue of the special (initial) position of content in the bracket representation (see §3.2). However, the option also be set explicitly, as any other option.



```
\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 *text* 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`.

*style* **math content** The content of the node will be typeset in a math environment.

This style is just an abbreviation for `content format={\ensuremath{\forestoption{content}}}`.

*option* **node format**=(*toks*) \noexpand\node  

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

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

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

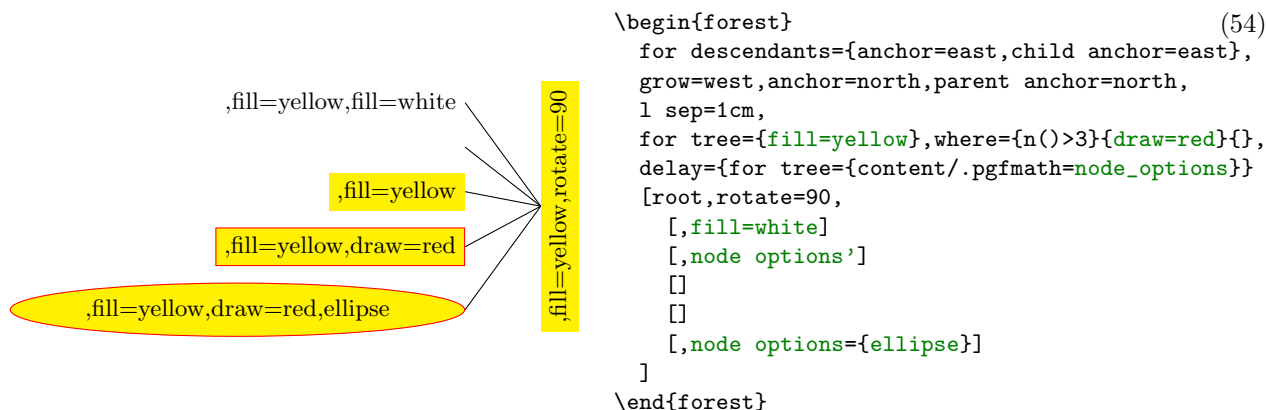
Important: the value of this option is first expanded using `\edef` and only then executed. Note that in its default value, `content format` is fully expanded using `\forestoption`: this is necessary for complex content formats, such as `tabular` environments.

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**=(*keylist*) {}

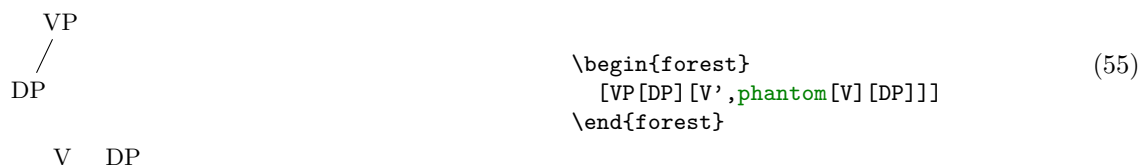
When the node is being typeset under the default conditions (see option `node format`), the content of this option is passed to TikZ as options to the TikZ node operation [?, §16].

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**=(*boolean*) false

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



### 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**=*<toks: TikZ anchor name>*

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.

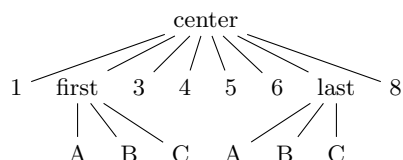
**calign=first** is an abbreviation for **calign=child,calign child=1**.

**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**.



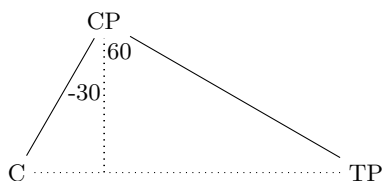
```
\begin{forest}
[center,calign=center[1]
[first,calign=first[A][B][C]][3][4][5][6]
[last,calign=last[A][B][C]][8]]
\end{forest}
```

**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=<count>` Sets the secondary child. (See `calign`.) -1  
 <count> is the child's sequence number. Negative numbers start counting at the last child.

`calign angle=<count>` is an abbreviation for `calign primary angle=-<count>`, `calign secondary angle=<count>`.

*option* `calign primary angle=<count>` Sets the primary angle. (See `calign`.) -35

*option* `calign secondary angle=<count>` Sets the secondary angle. (See `calign`.) 35

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

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

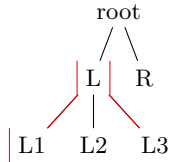
`calign with current edge` s-aligns the child anchor of the current node and the parent anchor of its parent. This key is an abbreviation for:

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

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

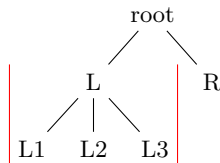
This option sets the type of the (s-)boundary that will be computed for the subtree rooted in the node, thereby determining how it will be packed into the subtree rooted in the node's parent. There are three choices:<sup>14</sup>

- `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.

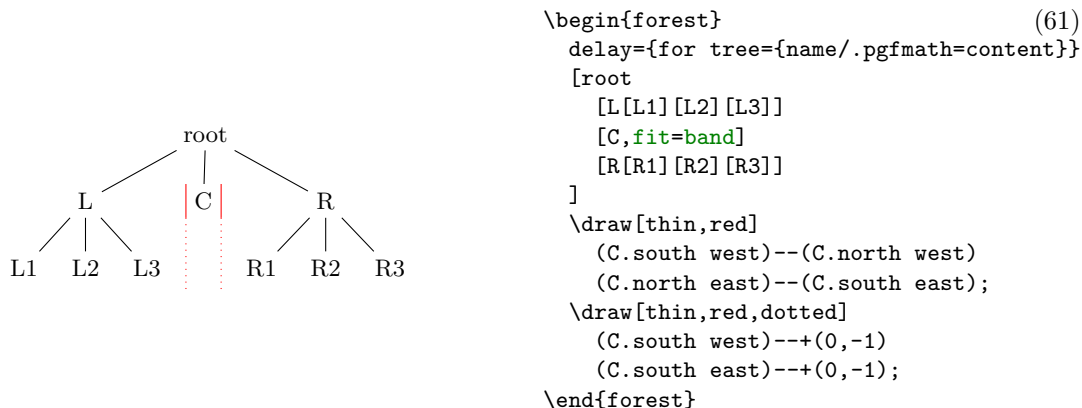


```
\begin{forest} (60)
  delay={for tree={name/.pgfmath=content}}
  [root
    [L,fit=rectangle,
      show boundary
      [L1] [L2] [L3]]
    [R]]
\end{forest}
```

<sup>14</sup>Below is the definition of style `show boundary`. The use path trick is adjusted from [TeX Stackexchange question Calling a previously named path in tikz](#).

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

- `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.

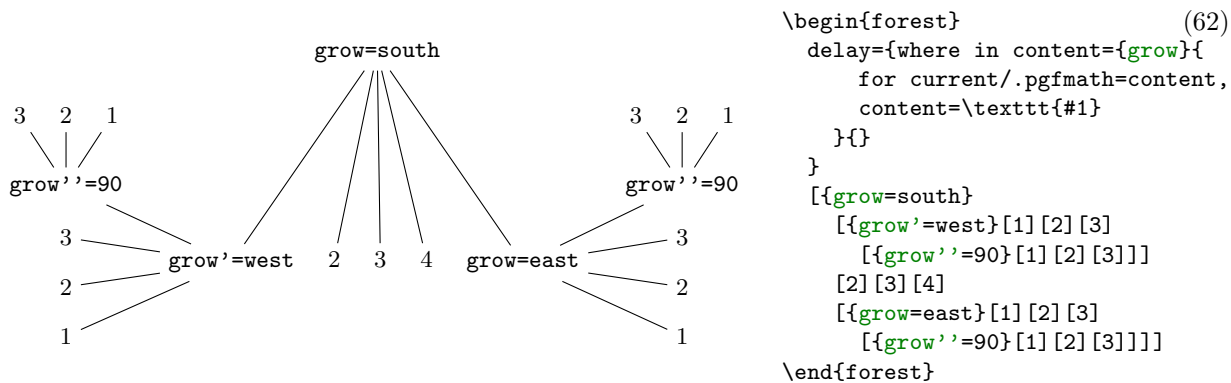


*option* `grow=(count)` The direction of the tree's growth at the node. 270

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.



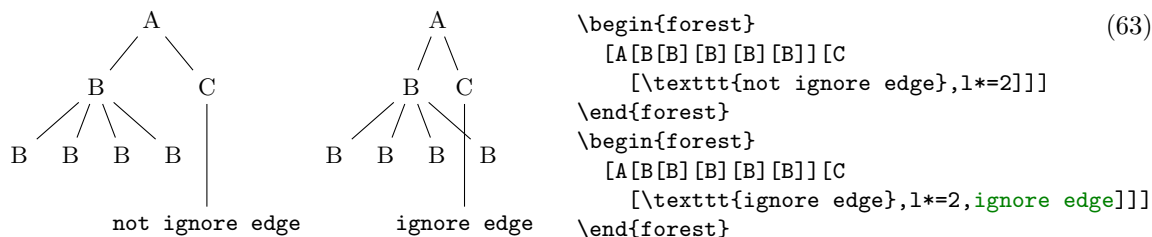
*option* `ignore=(boolean)` `false`

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

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

*option* `ignore edge=(boolean)` `false`

If this option is set, the packing mechanism ignores the edge from the node to the parent, i.e. nodes and other edges can overlap it. (See §5 for some problematic situations.)



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

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

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

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**=*(dimen)* The minimal l-distance between the node and its descendants.

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

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

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

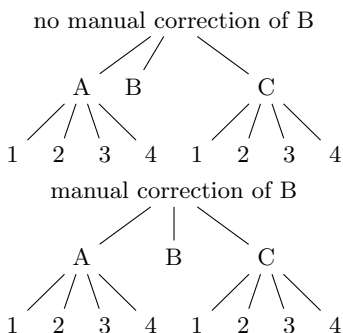
Note that despite the similar name, the semantics of l sep and s sep are quite different.

*option* **reversed**=*(boolean)* **false**

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

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

The value of s is computed 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**= $\langle dimen \rangle$

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**= $\langle toks \rangle$

{}

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.

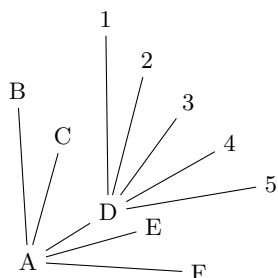
Tier alignment also does not work well with **calign=fixed (edge) angles**, because these child alignment methods may change the l-position of the children. When this might happen, FOREST will yield a warning.

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

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

**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**.

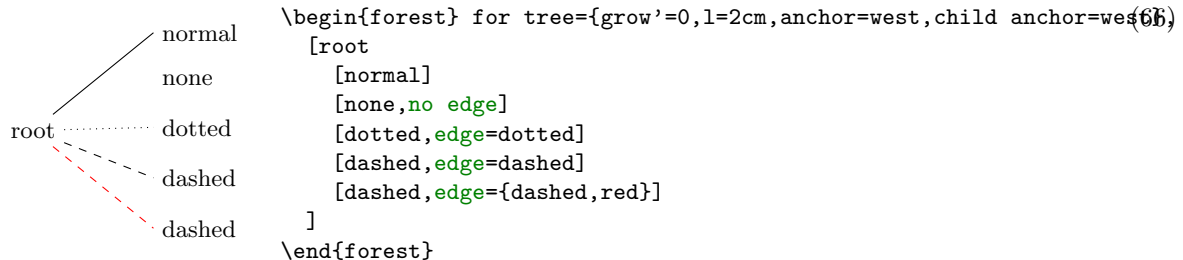


option `child anchor`= $\langle toks \rangle$  See `parent anchor`. {}

option `edge`= $\langle keylist \rangle$  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.

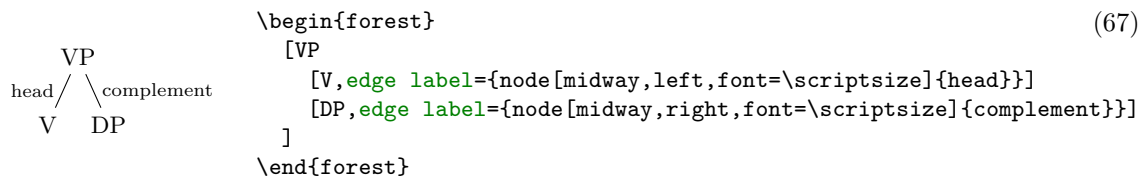
Also see key `no edge`.



option `edge label`= $\langle toks: TikZ code \rangle$  {}

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`= $\langle toks: TikZ code \rangle$  \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`= $\langle toks: TikZ anchor \rangle$  (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 T<sub>E</sub>X) 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   Baseline at the parent and baseline at the child.   child</p>	<pre style="font-family: monospace;">{\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>
---	--

```
begin draw/.code=<toks: TEX code> \begin{tikzpicture}
end draw/.code=<toks: TEX code> \end{tikzpicture}
```

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

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

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

```
begin forest/.code=<toks: TEX code> {}
end forest/.code=<toks: TEX code> {}
```

The code stored in these (`\pgfkeys`) keys is executed at the beginning and end of the `forest` environment / `\Forest` macro.

Using these keys is only effective *outside* the `forest` environment, and the effect lasts until the end of the current T<sub>E</sub>X group.

For example, executing `\forestset{begin forest/.code=\small}` will typeset all trees (and only trees) in the small font size.

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

This key should be used like `/tikz/fit` of the TikZ’s fitting library [see ? , §34]: as an option to TikZ’s `node` operation, the obvious restriction being that `fit to tree` must be used in the context of some FOREST node. For an example, see footnote 6.

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

```
get min s tree boundary=<cs>
get max s tree boundary=<cs>
```

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

**label**=`<toks: TikZ node>` 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**=*(toks)* Sets the name of the node. node@*(id)*

The expansion of *(toks)* becomes the *(forest node name)* 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**=*(node walk)* This key is the most general way to use a *(node walk)*.

Before starting the *(node walk)*, key **node walk/before walk** is processed. Then, the *(step)*s composing the *(node walk)* are processed: making a step (normally) changes the current node. After every step, key **node walk/every step** is processed. After the walk, key **node walk/after walk** is processed.

**node walk/before walk**, **node walk/every step** and **node walk/after walk** are processed with **/forest** as the default path: thus, FOREST's 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 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**=*(toks: TikZ node)* The current node gets a pin, see [? , §16.10].

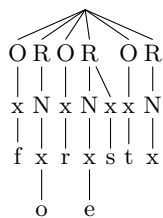
The technical details are the same as for **label**.

**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**=*(toks: TEX code)* The given code is executed immediately.

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



```

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

```

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

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

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

*option* **tikz**=*(toks: TikZ code)* “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**.

### 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.

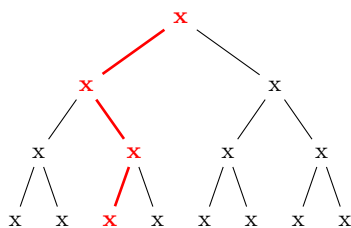
*key prefix* `for  $\langle step \rangle = \langle keylist \rangle$`  Walks a single-step node-walk  $\langle step \rangle$  from the current node and passes the given  $\langle keylist \rangle$  to the final (i.e. second) node.

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

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* **delay n**  $= \langle integer \rangle \langle keylist \rangle$  Defers the processing of the  $\langle keylist \rangle$  for  $n$  cycles.  $n$  may be 0, and it may be given as a `pgfmath` expression.

*propagator* **if have delayed**  $= \langle true\ keylist \rangle \langle false\ keylist \rangle$  If any options were delayed in the current cycle (more precisely, up to the point of the execution of this key), process  $\langle true\ keylist \rangle$ , otherwise process  $\langle false\ keylist \rangle$ . (**delay n** will trigger “true” for the intermediate cycles.)

*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. The bracket representation of the tree is parsed and stored in a data structure.
2. The given options are processed, including the options in the preamble, which are processed first (in the context of the root node).
3. Each node is typeset in its own `tikzpicture` environment, saved in a box and its measures are taken.
4. The nodes of the tree are *packed*, i.e. the relative positions of the nodes are computed so that the nodes don't overlap. That's difficult. The result: option `s` is set for all nodes. (Sometimes, the value of `l` is adjusted as well.)
5. Absolute positions, or rather, positions of the nodes relative to the root node are computed. That's easy. The result: options `x` and `y` are set.
6. The TikZ code that will draw the tree is produced. (The nodes are drawn by using the boxes typeset in step 3.)

Steps 1 and 2 collect user input and are thus “fixed”. However, the other steps, which do the actual work, are under user's control.

First, hooks exist which make it possible (and easy) to change node's properties between the processing stages. 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.

Second, the execution of the processing stages 3–6 is *completely* under user's control. To facilitate adjusting the processing flow, the approach is twofold. The outer level: FOREST initiates the processing by executing style `stages`, which by default executes the processing stages 3–6, preceding the execution of each stage by processing the options embedded in temporal propagators `before ...` (see §3.3.6). The inner level: each processing step is the sole resident of a stage-style, which makes it easy to adjust the workings of a single step. What follows is the default content of style `stages`, including the default content of the individual stage-styles.

```

style stages
    process keylist=before typesetting nodes
style typeset nodes stage                                {for root'=typeset nodes}
    process keylist=before packing
style pack stage                                         {for root'=pack}
    process keylist=before computing xy
style compute xy stage                                   {for root'=compute xy}
    process keylist=before drawing tree
style draw tree stage                                    {for root'=draw tree}

```

Both style `stages` and the individual stage-styles may be freely modified by the user. Obviously, a style must be redefined before it is processed, so it is safest to do so either outside the `forest` environment (using macro `\forestset`) or in the preamble (in a non-deferred fashion).

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

*stage* **typeset nodes** Typesets each node of the current node's subtree in its own `tikzpicture` environment. The result is saved in a box and its measures are taken.

*stage* **typeset nodes'** Like `typeset nodes`, but the node box's content is not overwritten if the box 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  $\times$ s is implemented in the **GP1** style.

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

**pack’** “Non-recursive” packing: packs the children of the current node only. (Experimental, use with care, especially when combining with tier alignment.)

*stage* **compute xy** Computes the positions of the nodes relative to the (formal) root node. The results are stored into options **x** and **y**.

*stage* **draw tree** Produces the TikZ code that will draw the tree. First, the nodes are drawn (using the boxes typeset in step 3), followed by edges and custom code (see option **tikz**).

*stage* **draw tree’** Like **draw tree**, but the node boxes are included in the picture using `\copy`, not `\box`, thereby preserving them.

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

**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:

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

$\langle \text{node} \rangle$ : 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.”

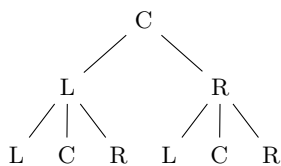
$\langle \text{relative node name} \rangle$ : the node  $\langle \text{relative node name} \rangle$  resolves to will be used.

Here is the list of dynamic tree keys:

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

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**=[*<node>*]

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

*dynamic tree* **insert after**=*<empty>* | [*<node>*] | *<relative node name>*

The specified node becomes the new following sibling of the current node. If the specified node had a parent, it is first removed from its old position.

*dynamic tree* **insert before**=*<empty>* | [*<node>*] | *<relative node name>*

The specified node becomes the new previous sibling of the current node. If the specified node had a parent, it is first removed from its old position.

*dynamic tree* **prepend**=*<empty>* | [*<node>*] | *<relative node name>*

The specified node becomes the new first child of the current node. If the specified node had a parent, it is first removed from its old position.

*dynamic tree* **remove**

The current node is removed from the tree and becomes the last node.

The node itself is not deleted: it is just not integrated in the tree anymore. Removing the root node has no effect.

*dynamic tree* **replace by**=*<empty>* | [*<node>*] | *<relative node name>*

The current node is replaced by the specified node. The current node becomes the last node.

If the specified node is a new node containing a dynamic tree key, it can refer to the replaced node by the *<empty>* specification. This works even if multiple replacements are made.

If **replace by** is used on the root node, the “replacement” becomes the root node (**set root** is used).

*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.

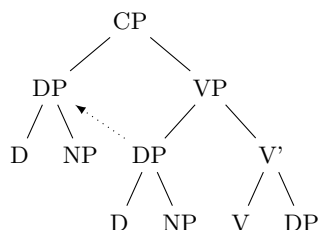
*dynamic tree* **append''**, **insert after''**, **insert before''**, **prepend''**, **replace by''**

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`= $\langle macro\ definition \rangle$  $\langle arg\ 1 \rangle$ ... $\langle arg\ n \rangle$

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 `<relative node name>.<option name>=<value>`, see §3.3.
- The `forest` coordinate system, both explicit and implicit; see §3.5.2.

### 3.5.1 Node walk

A *<node walk>* is a sequence of *<step>*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 *<node walk>* 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:

- <step>* `current` an “empty” step: the current node remains the same<sup>15</sup>
- <step>* `first` the primary child
- <step>* `first leaf` the first leaf (terminal node)
- <step>* `group=<node walk>` treat the given *<node walk>* as a single step
- <step>* `last` the last child
- <step>* `last leaf` the last leaf
- <step>* `id=<id>` the node with the given id
- <step>* `linear next` the next node, in the processing order
- <step>* `linear previous` the previous node, in the processing order
- <step>* `n=n` the *n*th child; counting starts at 1 (not 0)
- <step>* `n’=n` the *n*th child, starting the count from the last child
- <step>* `name` the node with the given name
- <step>* `next` the next sibling
- <step>* `next leaf` the next leaf  
(the current node need not be a leaf)
- <step>* `next on tier` the next node on the same tier as the current node
- <step>* `node walk=<node walk>` embed the given *<node walk>*  
(the `node walk/before walk` and `node walk/after walk` are processed)
- <step>* `parent` the parent
- <step>* `previous` the previous sibling

---

<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 *<keylist>* constructed using `.wrap (n) pgfmath arg(s)` or `.wrap value`.

- ⟨step⟩ **previous leaf** the previous leaf  
(the current node need not be a leaf)
- ⟨step⟩ **previous on tier** the next node on the same tier as the current node
- repeat**=*n*⟨node walk⟩ repeat the given ⟨node walk⟩ *n* times  
(each step in every repetition counts as a step)
- ⟨step⟩ **root** the root node
- ⟨step⟩ **root'** the formal root node (see **set root** in §3.3.8)
- ⟨step⟩ **sibling** the sibling  
(don't use if the parent doesn't have exactly two children ...)
- ⟨step⟩ **to tier**=⟨tier⟩ the first ancestor of the current node on the given ⟨tier⟩
- ⟨step⟩ **trip**=⟨node walk⟩ after walking the embedded ⟨node walk⟩, return to the current node; the return does not count as a step

For each long ⟨step⟩ except **node walk**, **group**, **trip** and **repeat**, propagator **for** ⟨step⟩ is also defined. Each such propagator takes a ⟨keylist⟩ argument. If the step takes an argument, then so does its propagator; this argument precedes the ⟨keylist⟩. 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.

- ⟨short step⟩ **1, 2, 3, 4, 5, 6, 7, 8, 9** the first, ..., ninth child
- ⟨short step⟩ **l** the last child
- ⟨short step⟩ **u** the parent (up)
- ⟨short step⟩ **p** the previous sibling
- ⟨short step⟩ **n** the next sibling
- ⟨short step⟩ **s** the sibling
- ⟨short step⟩ **P** the previous leaf
- ⟨short step⟩ **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⟩]`.

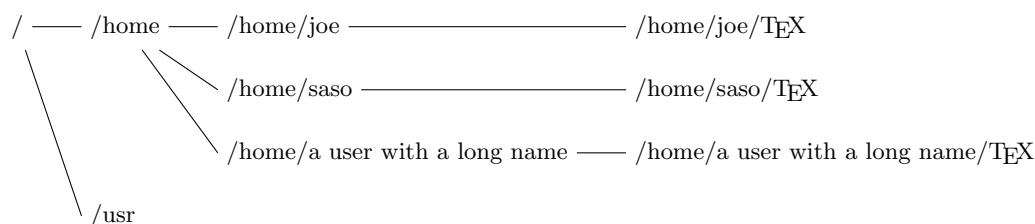
### 3.6 New pgfmath functions

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:



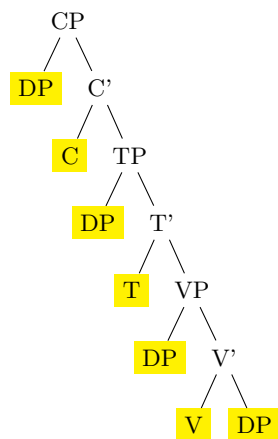
<sup>16</sup>Actually, the hack can be switched on and off on the fly, using `\ifforesttikzcshack`.

<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: `l+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}

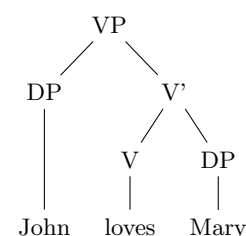
```



```

\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` $\langle node \rangle \langle environment fingerprint \rangle \langle calibration procedure \rangle \langle exported options \rangle$

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.

`<calibration procedure>` is a keylist (processed in the `/forest` path) which calculates the values of exported options.

`<exported options>` is a comma-separated list of exported options.

This is how the default standard node is created:

```
\forestStandardNode[dj]
  {%
    \forestOve{\csname forest@id@of@standard node@endcsname}{content},%
    \the\ht\strutbox,\the\pgflinewidth,%
    \pgfkeysvalueof{/pgf/inner ysep},\pgfkeysvalueof{/pgf/outer ysep},%
    \pgfkeysvalueof{/pgf/inner xsep},\pgfkeysvalueof{/pgf/outer xsep}%
  }
  {
    l sep={\the\ht\strutbox+\pgfkeysvalueof{/pgf/inner ysep}},
    l={l_sep()+abs(max_y()-min_y()+2*\pgfkeysvalueof{/pgf/outer ysep})},
    s sep={2*\pgfkeysvalueof{/pgf/inner xsep}}
  }
  {l sep,l,s sep}
```

### 3.8 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**=`<cs>` Adds the definition of macro `<cs>` to `external/context`. Thus, if the definition of `<cs>` is changed, the tree will be recompiled.

FOREST respects or is compatible with several (not all) keys and commands of 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.

Using `draw tree` and `draw tree'` multiple times *is* compatible with externalization, as is drawing the tree in the box (see `draw tree box`). If you are trying to externalize a `forest` environment which utilizes `TeX` to produce a visible effect, you will probably need to use `TeX'` and/or `TeX''`.

## 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>tikzcsback=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 `×`s by having a large enough `outer xsep` (adjustable), and then, before drawing (timing control option `before drawing tree`), setting `outer xsep` back to 0pt. The last step is important, otherwise the arrows between `×`s won't draw!



```

\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}}},
  e1/.style={content=\textsc{\textbf{##1}}},
  head/.style={content=\textsc{\textbf{\underline{##1}}}}
}

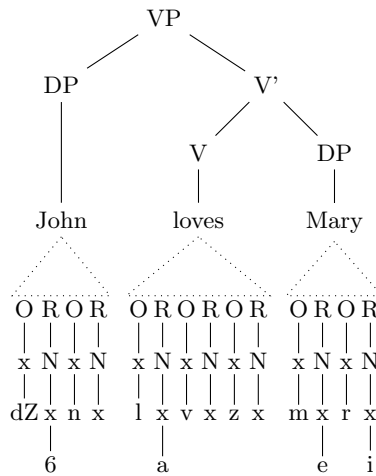
```

An example of an “embedded” GP1 style:

```

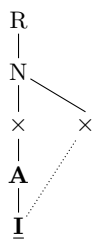
\begin{forest}
myGP1/.style={
  GP1,
  delay={where tier={x}{
    for children={content=\textipa{##1}}{}}},
  tikz={\draw[dotted](.south)--
    (!1.north west)--(!1.north east)--cycle;},
  for children={l+=5mm,no edge}
}
[VP[DP[John,tier=word,myGP1
  [O[x[dZ]]]
  [R[N[x[6]]]]
  [O[x[n]]]
  [R[N[x]]]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

```

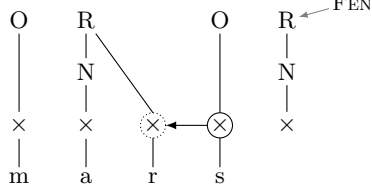


And an example of annotations.

[ei]



[mars]



\begin{forest}[,phantom,s sep=1cm (78)

```

[{{ei}}, GP1
  [R[N[x[A,eI[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}

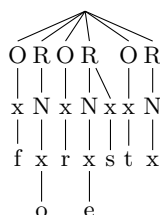
```

**rlap and llap** The FOREST versions of T<sub>E</sub>X’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

```

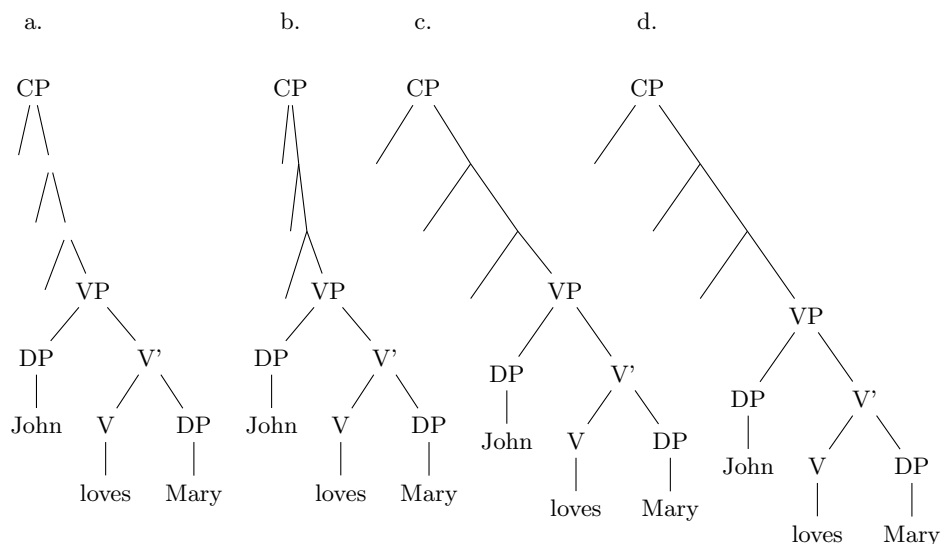
**nice empty nodes** We often need empty nodes: tree (a) shows how they look like by default: ugly. First, we don’t want the gaps: we change the shape of empty nodes to coordinate. We get tree (b). Second, the empty nodes seem too close to the other (especially empty) nodes (this is a result of a small 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.

```

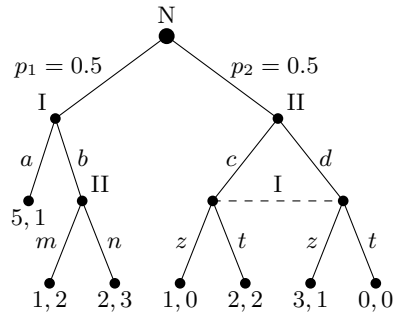
\forestset{
  nice empty nodes/.style={
    for tree={calign=fixed edge angles},
    delay={where content={}{shape=coordinate,for parent={for children={anchor=north}}}{}}
}
\begin{forest}
  [,xlist
  [CP,
    [ [ [ [ [VP [DP [John]] [V' [V [loves]] [DP [Mary]]]]]]]]]
  [CP, delay={where content={}{shape=coordinate}{}}
    [ [ [ [ [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 `e1o` 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).

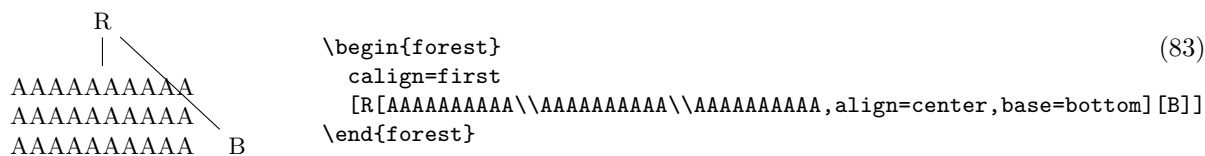
**System requirements** This package requires L<sup>A</sup>T<sub>E</sub>X and e<sub>T</sub><sub>E</sub>X. If you use something else: sorry.

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).

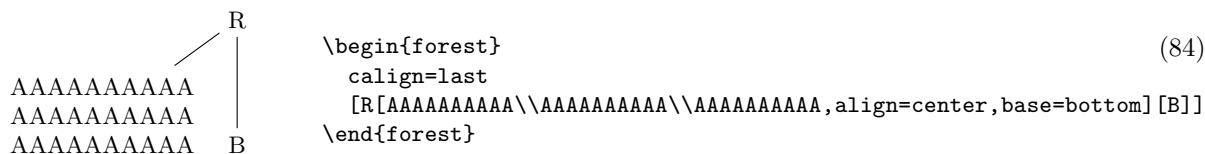
The requirement for e<sub>T</sub><sub>E</sub>X will probably stay. If nothing else, FOREST is heavy on boxes: every node requires its own ... and consequently, I have freely used e<sub>T</sub><sub>E</sub>X constructs in the code ...

**pgf internals** FOREST relies on some details of PGF implementation, like the name of the “not yet positioned” nodes. Thus, a new bug might appear with the development of PGF. If you notice one, please let me know.

**Edges cutting through sibling nodes** In the following example, the R–B edge crosses the AAA node, although `ignore edge` is set to the default `false`.

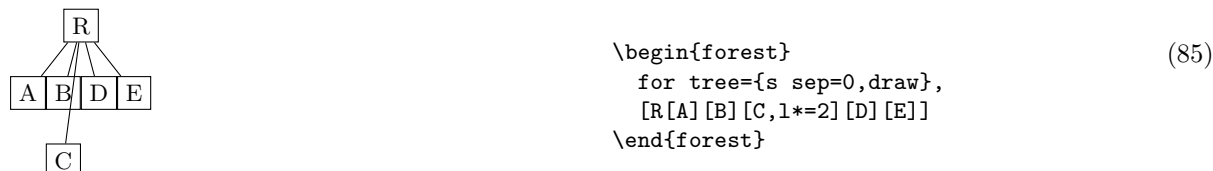


This happens because s-distances between the adjacent children are computed before child alignment (which is obviously the correct order in the general case), but child alignment non-linearly influences the edges. Observe that the with a different value of `calign`, the problem does not arise.



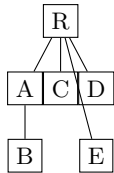
While it would be possible to fix the situation after child alignment (at least for some child alignment methods), I have decided against that, since the distances between siblings would soon become too large. If the AAA node in the example above was large enough, B could easily be pushed off the paper. 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.”



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.

## 6 Changelog

### v1.0.10 (2015/07/22)

- Bugfix: a left-over debugging `\typeout` command was interfering with a forest within `tabular`, see [this question on TeX.SE](#).
- A somewhat changed versioning scheme ...

### v1.09 (2015/07/15)

- Bugfix: child alignment was not done in nodes with a single child, see [this question on TeX.SE](#).

### v1.08 (2015/07/10)

- Fix externalization (compatibility with new `tikz` features).

### v1.07 (2015/05/29)

- Require package `elocalloc` for local boxes, which were previously defined by package `etex`.

### v1.06 (2015/05/04)

- Load `etex` package: since v2.1a, `etoolbox` doesn't do it anymore.

### v1.05 (2014/03/07)

- Fix the node boundary code for rounded rectangle. (Patch contributed by Paul Gaborit.)

### v1.04 (2013/10/17)

- Fixed an [externalization bug](#).

### v1.03 (2013/01/28)

- Bugfix: options of dynamically created nodes didn't get processed.
- Bugfix: the bracket parser was losing spaces before opening braces.
- Bugfix: a family of utility macros dealing with affixing token lists was not expanding content correctly.
- Added style [math content](#).
- Replace key `tikz preamble` with more general [begin draw](#) and [end draw](#).
- Add keys [begin forest](#) and [end forest](#).

### v1.02 (2013/01/20)

- Reworked style [stages](#): it's easier to modify the processing flow now.
- Individual stages must now be explicitly called in the context of some (usually root) node.
- Added [delay n](#) and [if have delayed](#).
- Added (experimental) [pack'](#).
- Added reference to the [style repository](#).

### v1.01 (2012/11/14)

- Compatibility with the `standalone` package: temporarily disable the effect of `standalone's` package option `tikz` while typesetting nodes.
- Require at least the [2010/08/21] (v2.0) release of package `etoolbox`.

- Require version [2010/10/13] (v2.10, rcs-revision 1.76) of PGF/TikZ. Future compatibility: adjust to the change of the “not yet positioned” node name (2.10 @ → 2.10-csv PGFINTERNAL).
- Add this changelog.

**v1.0 (2012/10/31)** First public version

**Acknowledgements** Many thanks to the T<sub>E</sub>X SE community and the people who have reported bugs! In the chronological order: Markus Pöchtrager, Timothy Dozat, Ignasi Furio.<sup>18</sup>

---

<sup>18</sup>If you're in the list but don't want to be, my apologies and please let me know about it!



## Part II

# Implementation

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

Identification.

```
1 \ProvidesPackage{forest}[2015/07/15 v1.0.10 Drawing (linguistic) trees]
2
3 \RequirePackage{tikz}[2010/10/13]
4 \usetikzlibrary{shapes}
5 \usetikzlibrary{fit}
6 \usetikzlibrary{calc}
7 \usepgflibrary{intersections}
8
9 \RequirePackage{pgfopts}
10 \RequirePackage{etoolbox}[2010/08/21]
11 \RequirePackage{elocalloc}% for \locbox
12 \RequirePackage{environ}
13
14 %\usepackage[trace]{trace-pgfkeys}
    /forest is the root of the key hierarchy.
15 \pgfkeys{/forest/.is family}
16 \def\forestset#1{\pgfkeys{/forest}{#1}}
```

## 7 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.

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

```

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

```

Minor: a leaking space in the very first line.

```

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

```

```

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

```

```

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

```

```

227 \fi%
228 \advance\c@pgf@counta by\c@pgf@countb\relax%
229 \divide\c@pgf@counta by 65536\relax%
230 \ifnum\c@pgf@counta>0\relax%
231 \divide\pgf@xb by\c@pgf@counta\relax%
232 \divide\pgf@yb by\c@pgf@counta\relax%
233 \fi%
234 %
235 % Compute projection
236 %
237 \pgf@yc=\pgf@sys@tonumber{\pgf@yb}\pgf@x%
238 \advance\pgf@yc by\pgf@sys@tonumber{\pgf@xb}\pgf@y%
239 %
240 % The orthogonal vector is (\pgf@yb,\pgf@xb)
241 %
242 % Setup transformation matrix (this is just to use the matrix
243 % inversion)
244 %
245 \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}}%
246 \pgftransforminvert%
247 \pgf@process{\pgfpointtransformed{\pgfpoint{\pgf@xc}{\pgf@yc}}}%
248 }%
249 }
250
251 \ifx\pgfpointintersectionoflines\forest@original@pgfpointintersectionoflines
252 \let\pgfpointintersectionoflines\forest@patched@pgfpointintersectionoflines
253 \fi
254
255 % hah: hacking forest --- it depends on some details of PGF implementation
256 \def\forest@pgf@notyetpositioned{not yet positionedPGFINTERNAL}%
257 \expandafter\ifstrequal\expandafter{\pgfversion}{2.10}{-%
258 \def\forest@pgf@notyetpositioned{not yet positioned@}%
259 }-}

```

## 8 Utilities

Escaping \ifs.

```

260 \long\def\@escapeif#1#2\fi{\fi#1}
261 \long\def\@escapeifif#1#2\fi#3\fi{\fi{\fi#1}
    A factory for creating \...loop... macros.
262 \def\newloop#1{%
263 \count@=\escapechar
264 \escapechar=-1
265 \expandafter\newloop@parse@loopname\string#1\newloop@end
266 \escapechar=\count@
267 }%
268 {\lccode'7='1 \lccode'8='o \lccode'9='p
269 \lowercase{\gdef\newloop@parse@loopname#17889#2\newloop@end{%
270 \edef\newloop@marshal{%
271 \noexpand\csdef{#1loop#2}####1\expandafter\noexpand\csname #1repeat#2\endcsname{%
272 \noexpand\csdef{#1iterate#2}{####1\relax\noexpand\expandafter\expandafter\noexpand\csname#1iterate#2\endcsname}
273 \expandafter\noexpand\csname#1iterate#2\endcsname
274 \let\expandafter\noexpand\csname#1iterate#2\endcsname\relax
275 }%
276 }%
277 \newloop@marshal
278 }%
279 }%
280 }%

```

Additional loops (for embedding).

```
281 \newloop\forest@loop
282 \newloop\forest@loopa
283 \newloop\forest@loopb
284 \newloop\forest@loopc
285 \newloop\forest@sort@loop
286 \newloop\forest@sort@loopA
```

New counters, dimens, ifs.

```
287 \newdimen\forest@temp@dimen
288 \newcount\forest@temp@count
289 \newcount\forest@n
290 \newif\ifforest@temp
291 \newcount\forest@temp@global@count
```

Appending and prepending to token lists.

```
292 \def\apptotoks#1#2{\expandafter#1\expandafter{\the#1#2}}
293 \long\def\lapptotoks#1#2{\expandafter#1\expandafter{\the#1#2}}
294 \def\eaaptotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter#1\expandafter\expandafter\expandafter
295 \def\pretotoks#1#2{\toks@=#2}\expandafter\expandafter\expandafter#1\expandafter\expandafter\expandafter{\the#1#2}}
296 \def\epretotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter#1\expandafter\expandafter\expandafter
297 \def\gapptotoks#1#2{\expandafter\global\expandafter#1\expandafter{\the#1#2}}
298 \def\xapptotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter\global\expandafter\expandafter\expandafter
299 \def\gpretotoks#1#2{\toks@=#2}\expandafter\expandafter\expandafter\global\expandafter\expandafter\expandafter
300 \def\xpretotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter\global\expandafter\expandafter\expandafter\exp
```

Expanding number arguments.

```
301 \def\expandnumberarg#1#2{\expandafter#1\expandafter{\number#2}}
302 \def\expandtwonumberargs#1#2#3{%
303   \expandafter\expandtwonumberargs@\expandafter#1\expandafter{\number#3}{#2}}
304 \def\expandtwonumberargs@#1#2#3{%
305   \expandafter#1\expandafter{\number#3}{#2}}
306 \def\expandthreenumberargs#1#2#3#4{%
307   \expandafter\expandthreenumberargs@\expandafter#1\expandafter{\number#4}{#2}{#3}}
308 \def\expandthreenumberargs@#1#2#3#4{%
309   \expandafter\expandthreenumberargs@@\expandafter#1\expandafter{\number#4}{#2}{#3}}
310 \def\expandthreenumberargs@@#1#2#3#4{%
311   \expandafter#1\expandafter{\number#4}{#2}{#3}}
```

A macro converting all non-letters in a string to `_`. #1 = string, #2 = receiving macro. Used for declaring pgfmath functions.

```
312 \def\forest@convert@others@to@underscores#1#2{%
313   \def\forest@cotu@result{}}
314 \forest@cotu#1\forest@end
315 \let#2\forest@cotu@result
316 }
317 \def\forest@cotu{%
318   \futurelet\forest@cotu@nextchar\forest@cotu@checkforspace
319 }
320 \def\forest@cotu@checkforspace{%
321   \expandafter\ifx\space\forest@cotu@nextchar
322   \let\forest@cotu@next\forest@cotu@havespace
323   \else
324     \let\forest@cotu@next\forest@cotu@nospace
325   \fi
326   \forest@cotu@next
327 }
328 \def\forest@cotu@havespace#1{%
329   \appto\forest@cotu@result{_}}
330 \forest@cotu#1%
331 }
332 \def\forest@cotu@nospace{%
```

```

333 \ifx\forest@cotu@nextchar\forest@end
334 \escapeif\@gobble
335 \else
336 \escapeif\forest@cotu@nospaceB
337 \fi
338 }
339 \def\forest@cotu@nospaceB{%
340 \ifcat\forest@cotu@nextchar a%
341 \let\forest@cotu@next\forest@cotu@have@alphanum
342 \else
343 \ifcat\forest@cotu@nextchar 0%
344 \let\forest@cotu@next\forest@cotu@have@alphanum
345 \else
346 \let\forest@cotu@next\forest@cotu@have@other
347 \fi
348 \fi
349 \forest@cotu@next
350 }
351 \def\forest@cotu@have@alphanum#1{%
352 \appto\forest@cotu@result{#1}%
353 \forest@cotu
354 }
355 \def\forest@cotu@have@other#1{%
356 \appto\forest@cotu@result{#1}%
357 \forest@cotu
358 }

Additional list macros.

359 \def\forest@listedel#1#2{% #1 = list, #2 = item
360 \edef\forest@marshal{\noexpand\forest@listedel\noexpand#1{#2}}%
361 \forest@marshal
362 }
363 \def\forest@listcsdel#1#2{%
364 \expandafter\forest@listdel\csname #1\endcsname{#2}%
365 }
366 \def\forest@listcsedel#1#2{%
367 \expandafter\forest@listedel\csname #1\endcsname{#2}%
368 }
369 \edef\forest@restorelistsepcatcode{\noexpand\catcode'\the\catcode'\relax}%
370 \catcode'\|=3
371 \gdef\forest@listdel#1#2{%
372 \def\forest@listedel@A##1|#2|##2\forest@END{%
373 \forest@listedel@B##1|##2\forest@END%|
374 }%
375 \def\forest@listedel@B|##1\forest@END{%|
376 \def#1{##1}%
377 }%
378 \expandafter\forest@listedel@A\expandafter|#1\forest@END%|
379 }
380 \forest@restorelistsepcatcode

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

381 \def\forest@strip@braces#1{%
382 \forest@strip@braces@A#1\forest@strip@braces@preend\forest@strip@braces@end
383 }
384 \def\forest@strip@braces@A#1#2\forest@strip@braces@end{%
385 #1\ifx\forest@strip@braces@preend#2\else\escapeif{\forest@strip@braces@A#2\forest@strip@braces@end}\fi
386 }

```

## 8.1 Sorting

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

The user should prepare the data in an arbitrarily encoded array,<sup>19</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 algorithms 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.

```
387 \def\forest@sort#1#2#3#4#5{%
388   \let\forest@sort@cmp#1\relax
389   \let\forest@sort@let#2\relax
390   \let\forest@sort@direction#3\relax
391   \forest@@sort{#4}{#5}%
392 }
```

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.

```
393 \def\forest@quicksort@minarraylength{10000}
394 \def\forest@@sort#1#2{%
395   \ifnum#1<#2\relax\@escapeif{%
396     \forest@sort@m=#2
397     \advance\forest@sort@m -#1
398     \ifnum\forest@sort@m>\forest@quicksort@minarraylength\relax\@escapeif{%
399       \forest@quicksort{#1}{#2}%
400     }\else\@escapeif{%
401       \forest@insertionsort{#1}{#2}%
402     }\fi
403   }\fi
404 }
```

Various counters and macros needed by the sorting algorithms.

```
405 \newcount\forest@sort@m\newcount\forest@sort@k\newcount\forest@sort@p
406 \def\forest@sort@ascending{>}
407 \def\forest@sort@descending{<}
408 \def\forest@sort@cmp{%
409   \PackageError{sort}{You must define forest@sort@cmp function before calling
410     sort}{The macro must take two arguments, indices of the array
411     elements to be compared, and return '=' if the elements are equal
412     and '>'/<' if the first is greater /less than the secong element.}%
413 }
414 \def\forest@sort@cmp@gt{\def\forest@sort@cmp@result{>}}
415 \def\forest@sort@cmp@lt{\def\forest@sort@cmp@result{<}}
416 \def\forest@sort@cmp@eq{\def\forest@sort@cmp@result{=}}
417 \def\forest@sort@let{%
418   \PackageError{sort}{You must define forest@sort@let function before calling
419     sort}{The macro must take two arguments, indices of the array:
420     element 2 must be copied onto element 1.}%
421 }
```

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

```
422 \def\forest@quicksort#1#2{%
```

---

<sup>19</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`.



Compute the index of the middle element (`\forest@sort@m`).

```
423 \forest@sort@m=#2
424 \advance\forest@sort@m -#1
425 \ifodd\forest@sort@m\relax\advance\forest@sort@m1 \fi
426 \divide\forest@sort@m 2
427 \advance\forest@sort@m #1
```

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

```
428 \forest@sort@cmp{#1}{#2}%
429 \if\forest@sort@cmp@result=%
430   \forest@sort@p=#1
431 \else
432   \if\forest@sort@cmp@result>%
433     \forest@sort@p=#1\relax
434   \else
435     \forest@sort@p=#2\relax
436   \fi
437 \forest@sort@cmp{\the\forest@sort@p}{\the\forest@sort@m}%
438 \if\forest@sort@cmp@result<%
439   \else
440     \forest@sort@p=\the\forest@sort@m
441   \fi
442 \fi
```

Exchange the pivot and the first element.

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

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

```
444 \forest@sort@m=#1\relax
```

Loop through the list.

```
445 \forest@sort@k=#1\relax
446 \forest@sort@loop
447 \ifnum\forest@sort@k<#2\relax
448   \advance\forest@sort@k 1
```

Compare the pivot and the current element.

```
449 \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.

```
450 \ifx\forest@sort@direction\forest@sort@cmp@result
451   \advance\forest@sort@m 1
452   \forest@sort@xch{\the\forest@sort@m}{\the\forest@sort@k}
453 \fi
454 \forest@sort@repeat
```

Move the pivot element into its final position.

```
455 \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.

```
456 \forest@sort@k=\forest@sort@m
457 \advance\forest@sort@k -1
458 \advance\forest@sort@m 1
459 \edef\forest@sort@marshal{%
460   \noexpand\forest@sort@{#1}{\the\forest@sort@k}%
461   \noexpand\forest@sort@{\the\forest@sort@m}{#2}%
462 }%
463 \forest@sort@marshal
464 }
```

```
465 % We defines the item-exchange macro in terms of the (user-provided)
```

```

466 % array let macro.
467 %   \begin{macrocode}
468 \def\forest@sort@xch#1#2{%
469   \forest@sort@let{aux}{#1}%
470   \forest@sort@let{#1}{#2}%
471   \forest@sort@let{#2}{aux}%
472 }

Insertion sort.
473 \def\forest@insertionsort#1#2{%
474   \forest@sort@m=#1
475   \edef\forest@insertionsort@low{#1}%
476   \forest@sort@loopA
477   \ifnum\forest@sort@m<#2
478     \advance\forest@sort@m 1
479     \forest@insertionsort@Qbody
480   \forest@sort@repeatA
481 }
482 \newif\ifforest@insertionsort@loop
483 \def\forest@insertionsort@Qbody{%
484   \forest@sort@let{aux}{\the\forest@sort@m}%
485   \forest@sort@k\forest@sort@m
486   \advance\forest@sort@k -1
487   \forest@insertionsort@looptrue
488   \forest@sort@loop
489   \ifforest@insertionsort@loop
490     \forest@insertionsort@qbody
491   \forest@sort@repeat
492   \advance\forest@sort@k 1
493   \forest@sort@let{\the\forest@sort@k}{aux}%
494 }
495 \def\forest@insertionsort@qbody{%
496   \forest@sort@cmp{\the\forest@sort@k}{aux}%
497   \ifx\forest@sort@direction\forest@sort@cmp@result\relax
498     \forest@sort@p=\forest@sort@k
499     \advance\forest@sort@p 1
500     \forest@sort@let{\the\forest@sort@p}{\the\forest@sort@k}%
501     \advance\forest@sort@k -1
502     \ifnum\forest@sort@k<\forest@insertionsort@low\relax
503       \forest@insertionsort@loopfalse
504     \fi
505   \else
506     \forest@insertionsort@loopfalse
507   \fi
508 }

```

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

Compare numbers.

```

509 \def\forest@sort@cmpnumcs#1#2{%
510   \ifnum\csname#1\endcsname>\csname#2\endcsname\relax
511     \forest@sort@cmp@gt
512   \else
513     \ifnum\csname#1\endcsname<\csname#2\endcsname\relax
514       \forest@sort@cmp@lt
515     \else
516       \forest@sort@cmp@eq
517     \fi
518   \fi
519 }

```

Compare dimensions.

```

520 \def\forest@sort@cmpdimcs#1#2{%
521   \ifdim\csname#1\endcsname>\csname#2\endcsname\relax
522     \forest@sort@cmp@gt
523   \else
524     \ifdim\csname#1\endcsname<\csname#2\endcsname\relax
525       \forest@sort@cmp@lt
526     \else
527       \forest@sort@cmp@eq
528     \fi
529 \fi
530 }

```

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

```

531 \def\forest@sort@cmptwodimcs#1#2#3#4{%
532   \ifdim\csname#1\endcsname>\csname#3\endcsname\relax
533     \forest@sort@cmp@gt
534   \else
535     \ifdim\csname#1\endcsname<\csname#3\endcsname\relax
536       \forest@sort@cmp@lt
537     \else
538       \ifdim\csname#2\endcsname>\csname#4\endcsname\relax
539         \forest@sort@cmp@gt
540       \else
541         \ifdim\csname#2\endcsname<\csname#4\endcsname\relax
542           \forest@sort@cmp@lt
543         \else
544           \forest@sort@cmp@eq
545         \fi
546       \fi
547     \fi
548 \fi
549 }

```

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).

```

550 \def\forest@reversearray#1#2#3{%
551   \let\forest@sort@let#1%
552   \c@pgf@countc=#2
553   \c@pgf@countd=#3
554   \advance\c@pgf@countd -1
555   \forest@loopa
556   \ifnum\c@pgf@countc<\c@pgf@countd\relax
557     \forest@sort@exch{\the\c@pgf@countc}{\the\c@pgf@countd}%
558     \advance\c@pgf@countc 1
559     \advance\c@pgf@countd -1
560   \forest@repeata
561 }

```

## 9 The bracket representation parser

### 9.1 The user interface macros

Settings.

```

562 \def\bracketset#1{\pgfkeys{/bracket}{#1}}%
563 \bracketset{%
564   /bracket/.is family,
565   /handlers/.let/.style={\pgfkeyscurrentpath/.code={\let#1##1}},
566   opening bracket/.let=\bracket@openingBracket,
567   closing bracket/.let=\bracket@closingBracket,
568   action character/.let=\bracket@actionCharacter,

```

```

569 opening bracket=[,
570 closing bracket=],
571 action character,
572 new node/.code n args={3}{% #1=preamble, #2=node spec, #3=cs receiving the id
573   \forest@node@new#3%
574   \forest@set{#3}{given options}{content'=#2}%
575   \ifblank{#1}{}{%
576     \forest@preto{#3}{given options}{#1,}%
577   }%
578 },
579 set afterthought/.code 2 args={% #1=node id, #2=afterthought
580   \ifblank{#2}{}{\forest@appto{#1}{given options}{,afterthought={#2}}}%
581 }
582 }

```

`\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.)

```

583 \newtoks\bracket@content
584 \newtoks\bracket@afterthought
585 \def\bracketParse#1#2={%
586   \def\bracketEndParsingHook{#1}%
587   \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.)

```

588 \bracket@content={}%
589 \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.

```

590 \let\bracket@state\bracket@state@starting
591 \bracket@ignorespacestrue

```

By default, don’t expand anything.

```

592 \bracket@expandtokensfalse

```

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

```

593 \def\bracket@parentNode{0}%
594 \def\bracket@rootNode{0}%
595 \def\bracket@newNode{0}%
596 \def\bracket@afterthoughtNode{0}%

```

Finally, we start the parser.

```

597 \bracket@Parse
598 }

```

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.

```

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

```

## 9.2 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 `TEXs` (undelimited) argument parsing algorithm. If a space is found, remember that, eat it up, and restart the parsing.

```

600 \def\bracket@Parse{%
601   \futurelet\bracket@next@token\bracket@Parse@checkForSpace
602 }
603 \def\bracket@Parse@checkForSpace{%
604   \expandafter\ifx\space\bracket@next@token\@escapeif{%

```

```

605 \ifbracket@ignorespaces\else
606 \bracket@haveSpace>true
607 \fi
608 \expandafter\bracket@Parse\romannumeral-‘0%
609 }\else\@escapeif{%
610 \bracket@Parse@maybeexpand
611 }\fi
612 }

```

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`.

```

613 \newif\ifbracket@expandtokens
614 \def\bracket@Parse@maybeexpand{%
615 \ifbracket@expandtokens\@escapeif{%
616 \expandafter\bracket@Parse@peekAhead\romannumeral-‘0%
617 }\else\@escapeif{%
618 \bracket@Parse@peekAhead
619 }\fi
620 }

```

We then look ahead to see what's coming.

```

621 \def\bracket@Parse@peekAhead{%
622 \futurelet\bracket@next@token\bracket@Parse@checkForTeXGroup
623 }

```

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

```

624 \def\bracket@Parse@checkForTeXGroup{%
625 \ifx\bracket@next@token\bgroup%
626 \@escapeif{\bracket@Parse@appendGroup}%
627 \else
628 \@escapeif{\bracket@Parse@token}%
629 \fi
630 }

```

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.

```

631 \long\def\bracket@Parse@token#1{%
632 \ifx#1\bracket@openingBracket
633 \@escapeif{\bracket@Parse@openingBracketFound}%
634 \else
635 \@escapeif{%
636 \ifx#1\bracket@closingBracket
637 \@escapeif{\bracket@Parse@closingBracketFound}%
638 \else
639 \@escapeif{%
640 \ifx#1\bracket@actionCharacter
641 \@escapeif{\futurelet\bracket@next@token\bracket@Parse@actionCharacterFound}%
642 \else
643 \@escapeif{\bracket@Parse@appendToken#1}%
644 \fi
645 }%
646 \fi
647 }%
648 \fi
649 }

```

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

```

650 \newif\ifbracket@haveSpace
651 \newif\ifbracket@ignorespaces
652 \def\bracket@Parse@appendSpace{%

```

```

653 \ifbracket@haveSpace
654   \ifcase\bracket@state\relax
655     \eapptotoks\bracket@content\space
656   \or
657     \eapptotoks\bracket@afterthought\space
658   \or
659     \eapptotoks\bracket@afterthought\space
660   \fi
661   \bracket@haveSpacefalse
662 \fi
663 }
664 \long\def\bracket@Parse@appendToken#1{%
665   \bracket@Parse@appendSpace
666   \ifcase\bracket@state\relax
667     \lapptotoks\bracket@content{#1}%
668   \or
669     \lapptotoks\bracket@afterthought{#1}%
670   \or
671     \lapptotoks\bracket@afterthought{#1}%
672   \fi
673   \bracket@ignorespacesfalse
674   \bracket@Parse
675 }
676 \def\bracket@Parse@appendGroup#1{%
677   \bracket@Parse@appendSpace
678   \ifcase\bracket@state\relax
679     \apptotoks\bracket@content{#{1}}%
680   \or
681     \apptotoks\bracket@afterthought{#{1}}%
682   \or
683     \apptotoks\bracket@afterthought{#{1}}%
684   \fi
685   \bracket@ignorespacesfalse
686   \bracket@Parse
687 }

```

Declare states.

```

688 \def\bracket@state@inContent{0}
689 \def\bracket@state@inAfterthought{1}
690 \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.)

```

691 \def\bracket@Parse@openingBracketFound{%
692   \bracket@haveSpacefalse
693   \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.

```

694     \@escapeif{%
695       \bracket@createNode
696       \ifnum\bracket@parentNode=0 \else
697         \forest@node@Append{\bracket@parentNode}{\bracket@newNode}%
698       \fi
699       \let\bracket@parentNode\bracket@newNode
700       \bracket@Parse
701     }%
702   \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.

```
703   \@escapeif{%
704     \bracket@addAfterthought
705     \let\bracket@state\bracket@state@inContent
706     \bracket@Parse
707   }%
708   \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.

```
709   \@escapeif{%
710     \let\bracket@state\bracket@state@inContent
711     \bracket@Parse
712   }%
713   \fi
714 }
```

```
715 \def\bracket@Parse@closingBracketFound{%
716   \bracket@haveSpacefalse
717   \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.

```
718   \@escapeif{%
719     \bracket@createNode
720     \ifnum\bracket@parentNode=0
721       \@escapeif\bracket@EndParsingHook
722     \else
723       \@escapeif{%
724         \let\bracket@afterthoughtNode\bracket@newNode
725         \let\bracket@state\bracket@state@inAfterthought
726         \forest@node@Append{\bracket@parentNode}{\bracket@newNode}%
727         \bracket@Parse
728       }%
729     \fi
730   }%
731   \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.

```
732   \@escapeif{%
733     \bracket@addAfterthought
734     \let\bracket@afterthoughtNode\bracket@parentNode
735     \edef\bracket@parentNode{\forest@ve{\bracket@parentNode}{@parent}}%
736     \ifnum\bracket@parentNode=0
737       \expandafter\bracket@EndParsingHook
738     \else
739       \expandafter\bracket@Parse
740     \fi
741   }%
742   \else % starting
```

{start}...]: something’s obviously wrong with the input here...

```
743   \PackageError{forest}{You’re attempting to start a bracket representation
744     with a closing bracket}{}%
745   \fi
746 }
```

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

```
747 \def\bracket@Parse@actionCharacterFound{%
```

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

```
748 \ifx\bracket@next@token\bgroup\@escapeif{%
749 \bracket@Parse@action@expandgroup
750 }\else\@escapeif{%
751 \bracket@Parse@action@notagroup
752 }\fi
753 }
754 \def\bracket@Parse@action@expandgroup#1{%
755 \edef\bracket@Parse@action@expandgroup@macro{#1}%
756 \expandafter\bracket@Parse\bracket@Parse@action@expandgroup@macro
757 }
```

```
758 \let\bracket@action@fullyexpandCharacter+
759 \let\bracket@action@dontexpandCharacter-
760 \let\bracket@action@executeCharacter!
761 \def\bracket@Parse@action@notagroup#1{%
```

If + follows, tokens will be fully expanded from this point on.

```
762 \ifx#1\bracket@action@fullyexpandCharacter\@escapeif{%
763 \bracket@expandtokenstrue\bracket@Parse
764 }\else\@escapeif{%
```

If - follows, tokens will not be expanded from this point on. (This is the default behaviour.)

```
765 \ifx#1\bracket@action@dontexpandCharacter\@escapeif{%
766 \bracket@expandtokensfalse\bracket@Parse
767 }\else\@escapeif{%
```

Inhibit expansion of the next token.

```
768 \ifx#10\@escapeif{%
769 \bracket@Parse@appendToken
770 }\else\@escapeif{%
```

If another action characted follows, we yield the control. The user is expected to resume the parser manually, using `\bracketResume`.

```
771 \ifx#1\bracket@actionCharacter
772 \else\@escapeif{%
```

Anything else will be expanded once.

```
773 \expandafter\bracket@Parse#1%
774 }\fi
775 }\fi
776 }\fi
777 }\fi
778 }
```

### 9.3 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.

```
779 \def\bracket@createNode{%
780 \ifnum\bracket@rootNode=0
781 % root node
782 \bracketset{new node/.expanded=%
783 {\the\bracket@afterthought}%
784 {\the\bracket@content}%
785 \noexpand\bracket@newNode
786 }%
787 \bracket@afterthought={}%
788 \let\bracket@rootNode\bracket@newNode
```



```

789 \expandafter\let\bracket@saveRootNodeTo\bracket@newNode
790 \else
791 % other nodes
792 \bracketset{new node/.expanded=%
793   {}%
794   {\the\bracket@content}}%
795 \noexpand\bracket@newNode
796   }%
797 \fi
798 \bracket@content={}%
799 }

```

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

```

800 \def\bracket@addAfterthought{%
801 \bracketset{%
802   set afterthought/.expanded={\bracket@afterthoughtNode}{\the\bracket@afterthought}}%
803 }%
804 \bracket@afterthought={}%
805 }

```

## 10 Nodes

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

### 10.1 Option setting and retrieval

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

```

806 % full expansion expands precisely to the value
807 \def\forestov#1{\expandafter\expandafter\expandafter\expandonce
808 \pgfkeysvalueof{/forest/@node/\forest@cn/#1}}
809 % full expansion expands all the way
810 \def\forestove#1{\pgfkeysvalueof{/forest/@node/\forest@cn/#1}}
811 % full expansion expands to the cs holding the value
812 \def\forestom#1{\expandafter\expandonce\expandafter{\pgfkeysvalueof{/forest/@node/\forest@cn/#1}}}\def\forest
813 \def\forestogget#1#2{\pgfkeysgetvalue{/forest/@node/\forest@cn/#1}{#2}}
814 \def\forestolet#1#2{\pgfkeyslet{/forest/@node/\forest@cn/#1}{#2}}
815 \def\forestoset#1#2{\pgfkeyssetvalue{/forest/@node/\forest@cn/#1}{#2}}
816 \def\forestoeset#1#2{%
817 \edef\forest@option@temp{%
818 \noexpand\pgfkeyssetvalue{/forest/@node/\forest@cn/#1}{#2}}%
819 }\forest@option@temp
820 }
821 \def\forestooppto#1#2{%
822 \forestoeset{#1}{\forestov{#1}\unexpanded{#2}}%
823 }
824 \def\forestoiifdefined#1#2#3{%
825 \pgfkeysifdefined{/forest/@node/\forest@cn/#1}{#2}{#3}%
826 }

```

User macros for retrieving node options of the current node.

```

827 \let\forestoption\forestov
828 \let\foresteooption\forestove

```

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

```

829 \def\forestov#1#2{\expandafter\expandafter\expandafter\expandonce
830 \pgfkeysvalueof{/forest/@node/#1/#2}}
831 \def\forestove#1#2{\pgfkeysvalueof{/forest/@node/#1/#2}}
832 % full expansion expands to the cs holding the value

```

```

833 \def\forest0m#1#2{\expandafter\expandonce\expandafter{\pgfkeysvalueof{/forest/@node/#1/#2}}}
834 \def\forest0get#1#2#3{\pgfkeysgetvalue{/forest/@node/#1/#2}{#3}}
835 \def\forest0let#1#2#3{\pgfkeysgetvalue{/forest/@node/#1/#2}{#3}}
836 \def\forest0let#1#2#3{\pgfkeyslet{/forest/@node/#1/#2}{#3}}
837 \def\forest0set#1#2#3{\pgfkeyssetvalue{/forest/@node/#1/#2}{#3}}
838 \def\forest0eset#1#2#3{%
839   \edef\forestoption@temp{%
840     \noexpand\pgfkeyssetvalue{/forest/@node/#1/#2}{#3}%
841   }\forestoption@temp
842 }
843 \def\forest0appto#1#2#3{%
844   \forest0eset{#1}{#2}{\forest0v{#1}{#2}\unexpanded{#3}}%
845 }
846 \def\forest0eappto#1#2#3{%
847   \forest0eset{#1}{#2}{\forest0v{#1}{#2}#3}%
848 }
849 \def\forest0preto#1#2#3{%
850   \forest0eset{#1}{#2}{\unexpanded{#3}\forest0v{#1}{#2}}%
851 }
852 \def\forest0epreto#1#2#3{%
853   \forest0eset{#1}{#2}{#3\forest0v{#1}{#2}}%
854 }
855 \def\forest0ifdefined#1#2#3#4{%
856   \pgfkeysifdefined{/forest/@node/#1/#2}{#3}{#4}%
857 }
858 \def\forest0let0#1#2#3#4{% option #2 of node #1 <-- option #4 of node #3
859   \forest0get{#3}{#4}\forestoption@temp
860   \forest0let{#1}{#2}\forestoption@temp}
861 \def\forest0leto#1#2#3{%
862   \forest0get{#3}\forestoption@temp
863   \forest0let{#1}{#2}\forestoption@temp}
864 \def\forest0leto0#1#2#3{%
865   \forest0get{#2}{#3}\forestoption@temp
866   \forest0let{#1}\forestoption@temp}
867 \def\forest0leto#1#2{%
868   \forest0get{#2}\forestoption@temp
869   \forest0let{#1}\forestoption@temp}

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

870 \def\forest@node@init{%
871   \forestoset{@parent}{0}%
872   \forestoset{@previous}{0}% previous sibling
873   \forestoset{@next}{0}%      next sibling
874   \forestoset{@first}{0}%    primary child
875   \forestoset{@last}{0}%     last child
876 }
877 \def\forest0toint#1{%
878   \pgfkeysgetvalue{/forest/#1}\forest0toint@temp
879   \forest0let{#1}\forest0toint@temp
880 }
881 \newcount\forest@node@maxid
882 \def\forest@node@new#1{% #1 = cs receiving the new node id
883   \advance\forest@node@maxid1
884   \forest@fornode{\the\forest@node@maxid}{%
885     \forest@node@init
886     \forest@node@setname{node@\forest@cn}%
887     \forest@initializefromstandardnode
888     \edef#1{\forest@cn}%
889   }%
890 }
891 \let\forest0toint@orig\forest0toint

```

```

892 \def\forest@node@copy#1#2{% #1=from node id, cs receiving the new node id
893   \advance\forest@node@maxid1
894   \def\forest@node@copy#1{\forest@node@copy{#1}{#1}}%
895   \forest@node@copy{\the\forest@node@maxid}{%
896     \forest@node@init
897     \forest@node@setname{\forest@copy@name@template{\forest@node@copy{#1}{name}}}%
898     \edef#2{\forest@cn}%
899   }%
900   \let\forest@node@copy\forest@node@copy@orig
901 }
902 \forestset{
903   copy name template/.code={\def\forest@copy@name@template##1{#1}},
904   copy name template/.default={node@\the\forest@node@maxid},
905   copy name template
906 }
907 \def\forest@tree@copy#1#2{% #1=from node id, #2=cs receiving the new node id
908   \forest@node@copy{#1}\forest@node@copy@temp@id
909   \forest@node@copy{\forest@node@copy@temp@id}{%
910     \expandafter\forest@tree@copy@expandafter{\forest@node@copy@temp@id}{#1}%
911     \edef#2{\forest@cn}%
912   }%
913 }
914 \def\forest@tree@copy@#1#2{%
915   \forest@node@foreachchild{#2}{%
916     \expandafter\forest@tree@copy@expandafter{\forest@cn}\forest@node@copy@temp@childid
917     \forest@node@append{#1}{\forest@node@copy@temp@childid}%
918   }%
919 }

```

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.

```

920 \def\forest@cn{0}
921 \forest@node@init

```

## 10.2 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.

```

922 \def\forest@node@append#1{\expandtwonumberargs\forest@node@append{\forest@cn}{#1}}
923 \def\forest@node@prepend#1{\expandtwonumberargs\forest@node@insertafter{\forest@cn}{#1}{0}}
924 \def\forest@node@insertafter#1#2{%
925   \expandthreenumberargs\forest@node@insertafter{\forest@cn}{#1}{#2}}
926 \def\forest@node@insertbefore#1#2{%
927   \expandthreenumberargs\forest@node@insertafter{\forest@cn}{#1}{\forest@node@copy@previous}}
928 }
929 \def\forest@node@remove{\expandnumberarg\forest@node@remove{\forest@cn}}
930 \def\forest@node@append#1#2{\expandtwonumberargs\forest@node@append@{#1}{#2}}
931 \def\forest@node@prepend#1#2{\expandtwonumberargs\forest@node@insertafter{#1}{#2}{0}}
932 \def\forest@node@insertafter#1#2#3{% #2 is inserted after #3
933   \expandthreenumberargs\forest@node@insertafter@{#1}{#2}{#3}}
934 }
935 \def\forest@node@insertbefore#1#2#3{% #2 is inserted before #3
936   \expandthreenumberargs\forest@node@insertafter{#1}{#2}{\forest@node@copy@previous}}
937 }
938 \def\forest@node@remove#1{\expandnumberarg\forest@node@remove@{#1}}
939 \def\forest@node@insertafter#1#2#3{%
940   \ifnum\forest@node@parent=0
941     \else

```

```

942 \PackageError{forest}{Insertafter(#1,#2,#3):
943   node #2 already has a parent (\forestOve{#2}{@parent})}{}%
944 \fi
945 \ifnum#3=0
946 \else
947   \ifnum#1=\forestOve{#3}{@parent}
948   \else
949     \PackageError{forest}{Insertafter(#1,#2,#3): node #1 is not the parent of the
950       intended sibling #3 (with parent \forestOve{#3}{@parent})}{}%
951   \fi
952 \fi
953 \forestOset{#2}{@parent}{#1}%
954 \forestOset{#2}{@previous}{#3}%
955 \ifnum#3=0
956   \forestOget{#1}{@first}\forest@node@temp
957   \forestOset{#1}{@first}{#2}%
958 \else
959   \forestOget{#3}{@next}\forest@node@temp
960   \forestOset{#3}{@next}{#2}%
961 \fi
962 \forestOset{#2}{@next}{\forest@node@temp}%
963 \ifnum\forest@node@temp=0
964   \forestOset{#1}{@last}{#2}%
965 \else
966   \forestOset{\forest@node@temp}{@previous}{#2}%
967 \fi
968 }
969 \def\forest@node@Append@#1#2{%
970   \ifnum\forestOve{#2}{@parent}=0
971   \else
972     \PackageError{forest}{Append(#1,#2):
973       node #2 already has a parent (\forestOve{#2}{@parent})}{}%
974   \fi
975   \forestOset{#2}{@parent}{#1}%
976   \forestOget{#1}{@last}\forest@node@temp
977   \forestOset{#1}{@last}{#2}%
978   \forestOset{#2}{@previous}{\forest@node@temp}%
979   \ifnum\forest@node@temp=0
980     \forestOset{#1}{@first}{#2}%
981   \else
982     \forestOset{\forest@node@temp}{@next}{#2}%
983   \fi
984 }
985 \def\forest@node@Remove@#1{%
986   \forestOget{#1}{@parent}\forest@node@temp@parent
987   \ifnum\forest@node@temp@parent=0
988   \else
989     \forestOget{#1}{@previous}\forest@node@temp@previous
990     \forestOget{#1}{@next}\forest@node@temp@next
991     \ifnum\forest@node@temp@previous=0
992       \forestOset{\forest@node@temp@parent}{@first}{\forest@node@temp@next}%
993     \else
994       \forestOset{\forest@node@temp@previous}{@next}{\forest@node@temp@next}%
995     \fi
996     \ifnum\forest@node@temp@next=0
997       \forestOset{\forest@node@temp@parent}{@last}{\forest@node@temp@previous}%
998     \else
999       \forestOset{\forest@node@temp@next}{@previous}{\forest@node@temp@previous}%
1000   \fi
1001   \forestOset{#1}{@parent}{0}%
1002   \forestOset{#1}{@previous}{0}%

```

```

1003   \forestOset{#1}{@next}{0}%
1004   \fi
1005 }

  Looping methods.
1006 \def\forest@forthis#1{%
1007   \edef\forest@node@marshal{\unexpanded{#1}\def\noexpand\forest@cn}%
1008   \expandafter\forest@node@marshal\expandafter{\forest@cn}%
1009 }
1010 \def\forest@fornode#1#2{%
1011   \edef\forest@node@marshal{\edef\noexpand\forest@cn{#1}\unexpanded{#2}\def\noexpand\forest@cn}%
1012   \expandafter\forest@node@marshal\expandafter{\forest@cn}%
1013 }
1014 \def\forest@fornode@ifexists#1#2{%
1015   \edef\forest@node@temp{#1}%
1016   \ifnum\forest@node@temp=0
1017   \else
1018     \@escapeif{\expandnumberarg\forest@fornode{\forest@node@temp}{#2}}%
1019   \fi
1020 }
1021 \def\forest@node@foreachchild#1{\forest@node@Foreachchild{\forest@cn}{#1}}
1022 \def\forest@node@Foreachchild#1#2{%
1023   \forest@fornode{\forestOve{#1}{@first}}{\forest@node@@forselfandfollowingsiblings{#2}}%
1024 }
1025 \def\forest@node@@forselfandfollowingsiblings#1{%
1026   \ifnum\forest@cn=0
1027   \else
1028     \forest@forthis{#1}%
1029     \@escapeif{%
1030       \edef\forest@cn{\forestove{@next}}%
1031       \forest@node@@forselfandfollowingsiblings{#1}%
1032     }%
1033   \fi
1034 }
1035 \def\forest@node@foreach#1{\forest@node@Foreach{\forest@cn}{#1}}
1036 \def\forest@node@Foreach#1#2{%
1037   \forest@fornode{#1}{\forest@node@@foreach{#2}}%
1038 }
1039 \def\forest@node@@foreach#1{%
1040   \forest@forthis{#1}%
1041   \ifnum\forestove{@first}=0
1042   \else\@escapeif{%
1043     \edef\forest@cn{\forestove{@first}}%
1044     \forest@node@@forselfandfollowingsiblings{\forest@node@@foreach{#1}}%
1045   }%
1046   \fi
1047 }
1048 \def\forest@node@foreachdescendant#1{\forest@node@Foreachdescendant{\forest@cn}{#1}}
1049 \def\forest@node@Foreachdescendant#1#2{%
1050   \forest@node@Foreachchild{#1}{%
1051     \forest@node@foreach{#2}%
1052   }%
1053 }

  Compute n, n', n children and level.
1054 \def\forest@node@Compute@numeric@ts@info@#1{%
1055   \forest@node@Foreach{#1}{\forest@node@@compute@numeric@ts@info}%
1056   \ifnum\forestOve{#1}{@parent}=0
1057   \else
1058     \fornode{#1}{\forest@node@@compute@numeric@ts@info@nbar}%
1059   \fi
1060   \forest@node@Foreachdescendant{#1}{\forest@node@@compute@numeric@ts@info@nbar}%

```

```

1061 }
1062 \def\forest@node@@compute@numeric@ts@info{%
1063   \forestset{n children}{0}%
1064   %
1065   \edef\forest@node@temp{\forestove{@previous}}%
1066   \ifnum\forest@node@temp=0
1067     \forestset{n}{1}%
1068   \else
1069     \forestoeset{n}{\number\numexpr\forestOve{\forest@node@temp}{n}+1}%
1070   \fi
1071   %
1072   \edef\forest@node@temp{\forestove{@parent}}%
1073   \ifnum\forest@node@temp=0
1074     \forestset{n}{0}%
1075     \forestset{n'}{0}%
1076     \forestset{level}{0}%
1077   \else
1078     \forestOset{\forest@node@temp}{n children}{%
1079       \number\numexpr\forestOve{\forest@node@temp}{n children}+1%
1080     }%
1081     \forestoeset{level}{%
1082       \number\numexpr\forestOve{\forest@node@temp}{level}+1%
1083     }%
1084   \fi
1085 }
1086 \def\forest@node@@compute@numeric@ts@info@nbar{%
1087   \forestoeset{n'}{\number\numexpr\forestOve{\forestove{@parent}}{n children}-\forestove{n}+1}%
1088 }
1089 \def\forest@node@compute@numeric@ts@info#1{%
1090   \expandnumberarg\forest@node@Compute@numeric@ts@info@{\forest@cn}%
1091 }
1092 \def\forest@node@Compute@numeric@ts@info#1{%
1093   \expandnumberarg\forest@node@Compute@numeric@ts@info@{#1}%
1094 }

```

Tree structure queries.

```

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

```

```

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

```

```

1181 \ifnum\forestOve{#1}{@next}=0
1182   \@escapeif{\expandnumberarg\forest@node@nextleafid@Goup{#1}}%
1183 \else
1184   \expandnumberarg\forest@node@nextleafid@Gdown{\forestOve{#1}{@next}}%
1185 \fi
1186 }
1187 \def\forest@node@nextleafid@Goup#1{%
1188   \ifnum\forestOve{#1}{@parent}=0
1189     \PackageError{forest}{get next leaf: this is the last leaf}{}%
1190   \else
1191     \@escapeif{\expandnumberarg\forest@node@Nextleafid{\forestOve{#1}{@parent}}}%
1192   \fi
1193 }
1194 \def\forest@node@nextleafid@Gdown#1{%
1195   \ifnum\forestOve{#1}{@first}=0
1196     #1%
1197   \else
1198     \@escapeif{\expandnumberarg\forest@node@nextleafid@Gdown{\forestOve{#1}{@first}}}%
1199   \fi
1200 }
1201 \def\forest@node@linearnextid{%
1202   \ifnum\forestove{@first}=0
1203     \expandafter\forest@node@linearnextnotdescendantid
1204   \else
1205     \forestove{@first}%
1206   \fi
1207 }
1208 \def\forest@node@linearnextnotdescendantid{%
1209   \expandnumberarg\forest@node@Linearnextnotdescendantid{\forest@cn}%
1210 }
1211 \def\forest@node@Linearnextnotdescendantid#1{%
1212   \ifnum\forestOve{#1}{@next}=0
1213     \@escapeif{\expandnumberarg\forest@node@Linearnextnotdescendantid{\forestOve{#1}{@parent}}}%
1214   \else
1215     \forestOve{#1}{@next}%
1216   \fi
1217 }
1218 \def\forest@node@linearpreviousid{%
1219   \ifnum\forestove{@previous}=0
1220     \forestove{@parent}%
1221   \else
1222     \forest@node@previousleafid
1223   \fi
1224 }
1225 \def\forest@ifancestorof#1{% is the current node an ancestor of #1? Yes: #2, no: #3
1226   \expandnumberarg\forest@ifancestorof{\forestOve{#1}{@parent}}%
1227 }
1228 \def\forest@ifancestorof@#1#2#3{%
1229   \ifnum#1=0
1230     \def\forest@ifancestorof@next{\@secondoftwo}%
1231   \else
1232     \ifnum\forest@cn=#1
1233       \def\forest@ifancestorof@next{\@firstoftwo}%
1234     \else
1235       \def\forest@ifancestorof@next{\expandnumberarg\forest@ifancestorof@\forestOve{#1}{@parent}}%
1236     \fi
1237   \fi
1238   \forest@ifancestorof@next{#2}{#3}%
1239 }

```



### 10.3 Node walk

```

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

```

```

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

```

```

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

```

```

1421 {%
1422   \forestset{%
1423     node walk/before walk/.code={},%
1424     node walk/every step/.code={},%
1425     node walk/after walk/.code={},%
1426     node walk={#1}%
1427   }%
1428   \expandafter
1429 }%
1430 \expandafter\def\expandafter\forest@cn\expandafter{\forest@cn}%
1431 }

```

## 10.4 Node options

### 10.4.1 Option-declaration mechanism

Common code for declaring options.

```

1432 \def\forest@declarehandler#1#2#3{%#1=handler for specific type,#2=option name,#3=default value
1433   \pgfkeyssetvalue{/forest/#2}{#3}%
1434   \appto\forest@node@init{\forest@init{#2}}%
1435   \forest@convert@others@to@underscores{#2}\forest@pgfmathoptionname
1436   \edef\forest@marshal{%
1437     \noexpand#1{/forest/#2}{/forest}{#2}{\forest@pgfmathoptionname}%
1438   }\forest@marshal
1439 }
1440 \def\forest@def@with@pgfeov#1#2{% \pgfeov mustn't occur in the arg of the .code handler!!!
1441   \long\def##1\pgfeov{#2}%
1442 }

```

Option-declaration handlers.

```

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

```

```

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

```

```

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

```

```

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

```

```

1658 \pgfkeysalso{%
1659   #1/.code={%
1660     \ifstrequal{##1}{1}{-%
1661       \forest0set{\forest@setter@node}{#3}{1}%
1662     }{%
1663       \pgfmathifthenelse{##1}{1}{0}%
1664       \forest0let{\forest@setter@node}{#3}\pgfmathresult
1665     }%
1666   },
1667   #1/.default=1,
1668   #2/not #3/.code={\forest0set{\forest@setter@node}{#3}{0}},
1669   #2/if #3/.code 2 args={%
1670     \forestoget{#3}\forest@temp@option@value
1671     \ifnum\forest@temp@option@value=1
1672       \pgfkeysalso{##1}%
1673     \else
1674       \pgfkeysalso{##2}%
1675     \fi
1676   },
1677   #2/where #3/.style 2 args={for tree={#2/if #3={##1}{##2}}}
1678 }%
1679 \pgfkeyssetvalue{#1/option@name}{#3}%
1680 \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@count{##1}{#3}}%
1681 }
1682 \pgfkeys{/forest,
1683   declare toks/.code 2 args={%
1684     \forest@declarehandler\forest@declaretoks@handler{#1}{#2}%
1685   },
1686   declare autowrapped toks/.code 2 args={%
1687     \forest@declarehandler\forest@declareautowrappedtoks@handler{#1}{#2}%
1688   },
1689   declare keylist/.code 2 args={%
1690     \forest@declarehandler\forest@declarekeylist@handler{#1}{#2}%
1691   },
1692   declare readonly dimen/.code={%
1693     \forest@declarehandler\forest@declarereadonlydimen@handler{#1}{}%
1694   },
1695   declare dimen/.code 2 args={%
1696     \forest@declarehandler\forest@declaredimen@handler{#1}{#2}%
1697   },
1698   declare readonly count/.code={%
1699     \forest@declarehandler\forest@declarereadonlycount@handler{#1}{}%
1700   },
1701   declare count/.code 2 args={%
1702     \forest@declarehandler\forest@declarecount@handler{#1}{#2}%
1703   },
1704   declare boolean/.code 2 args={%
1705     \forest@declarehandler\forest@declareboolean@handler{#1}{#2}%
1706   },
1707   /handlers/.pgfmath/.code={%
1708     \pgfmathparse{#1}%
1709     \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\pgfmathresult}%
1710   },
1711   /handlers/.wrap value/.code={%
1712     \edef\forest@handlers@wrap@currentpath{\pgfkeyscurrentpath}%
1713     \pgfkeysgetvalue{\forest@handlers@wrap@currentpath/option@name}\forest@currentoptionname
1714     \expandafter\forestoget\expandafter{\forest@currentoptionname}\forest@option@value
1715     \forest@def@with@pgfeov\forest@wrap@code{#1}%
1716     \expandafter\edef\expandafter\forest@wrapped@value\expandafter{\expandafter\expandonce\expandafter{\expa
1717     \pgfkeysalso{\forest@handlers@wrap@currentpath/.expand once=\forest@wrapped@value}%
1718   },

```



```

1719 /handlers/.wrap pgfmath arg/.code 2 args={%
1720   \pgfmathparse{#2}\let\forest@wrap@arg@i\pgfmathresult
1721   \edef\forest@wrap@args{\expandonce\forest@wrap@arg@i}%
1722   \def\forest@wrap@code##1{#1}%
1723   \expandafter\expandafter\expandafter\forest@temp@toks\expandafter\expandafter\expandafter{\expandafter\fo
1724   \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\the\forest@temp@toks}%
1725 },
1726 /handlers/.wrap 2 pgfmath args/.code n args={3}{-%
1727   \pgfmathparse{#2}\let\forest@wrap@arg@i\pgfmathresult
1728   \pgfmathparse{#3}\let\forest@wrap@arg@ii\pgfmathresult
1729   \edef\forest@wrap@args{\expandonce\forest@wrap@arg@i}{\expandonce\forest@wrap@arg@ii}%
1730   \def\forest@wrap@code##1##2{#1}%
1731   \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\forest@wrapped\expandafter\fo
1732   \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\forest@wrapped}%
1733 },
1734 /handlers/.wrap 3 pgfmath args/.code n args={4}{-%
1735   \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{-}{-}{-}{-}{3}%
1736   \forest@wrap@n@pgfmath@do{#1}{3}},
1737 /handlers/.wrap 4 pgfmath args/.code n args={5}{-%
1738   \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{-}{-}{-}{4}%
1739   \forest@wrap@n@pgfmath@do{#1}{4}},
1740 /handlers/.wrap 5 pgfmath args/.code n args={6}{-%
1741   \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{-}{-}{5}%
1742   \forest@wrap@n@pgfmath@do{#1}{5}},
1743 /handlers/.wrap 6 pgfmath args/.code n args={7}{-%
1744   \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{#7}{-}{6}%
1745   \forest@wrap@n@pgfmath@do{#1}{6}},
1746 /handlers/.wrap 7 pgfmath args/.code n args={8}{-%
1747   \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{#7}{#8}{-}{7}%
1748   \forest@wrap@n@pgfmath@do{#1}{7}},
1749 /handlers/.wrap 8 pgfmath args/.code n args={9}{-%
1750   \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}{8}%
1751   \forest@wrap@n@pgfmath@do{#1}{8}},
1752 }
1753 \def\forest@wrap@n@pgfmath@args#1#2#3#4#5#6#7#8#9{%
1754   \pgfmathparse{#1}\let\forest@wrap@arg@i\pgfmathresult
1755   \ifnum#9>1 \pgfmathparse{#2}\let\forest@wrap@arg@ii\pgfmathresult\fi
1756   \ifnum#9>2 \pgfmathparse{#3}\let\forest@wrap@arg@iii\pgfmathresult\fi
1757   \ifnum#9>3 \pgfmathparse{#4}\let\forest@wrap@arg@iv\pgfmathresult\fi
1758   \ifnum#9>4 \pgfmathparse{#5}\let\forest@wrap@arg@v\pgfmathresult\fi
1759   \ifnum#9>5 \pgfmathparse{#6}\let\forest@wrap@arg@vi\pgfmathresult\fi
1760   \ifnum#9>6 \pgfmathparse{#7}\let\forest@wrap@arg@vii\pgfmathresult\fi
1761   \ifnum#9>7 \pgfmathparse{#8}\let\forest@wrap@arg@viii\pgfmathresult\fi
1762   \edef\forest@wrap@args{%
1763     {\expandonce\forest@wrap@arg@i}
1764     \ifnum#9>1 {\expandonce\forest@wrap@arg@ii}\fi
1765     \ifnum#9>2 {\expandonce\forest@wrap@arg@iii}\fi
1766     \ifnum#9>3 {\expandonce\forest@wrap@arg@iv}\fi
1767     \ifnum#9>4 {\expandonce\forest@wrap@arg@v}\fi
1768     \ifnum#9>5 {\expandonce\forest@wrap@arg@vi}\fi
1769     \ifnum#9>6 {\expandonce\forest@wrap@arg@vii}\fi
1770     \ifnum#9>7 {\expandonce\forest@wrap@arg@viii}\fi
1771   }%
1772 }
1773 \def\forest@wrap@n@pgfmath@do#1#2{%
1774   \ifcase#2\relax
1775   \or\def\forest@wrap@code##1{#1}%
1776   \or\def\forest@wrap@code##1##2{#1}%
1777   \or\def\forest@wrap@code##1##2##3{#1}%
1778   \or\def\forest@wrap@code##1##2##3##4{#1}%
1779   \or\def\forest@wrap@code##1##2##3##4##5{#1}%

```

```

1780 \or\def\forest@wrap@code##1##2##3##4##5##6{#1}%
1781 \or\def\forest@wrap@code##1##2##3##4##5##6##7{#1}%
1782 \or\def\forest@wrap@code##1##2##3##4##5##6##7##8{#1}%
1783 \fi
1784 \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\forest@wrapped\expandafter\exp
1785 \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\forest@wrapped}%
1786 }

```

## 10.4.2 Declaring options

```

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

```

```

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

```

```

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

```

```

1960     /forest/get min s tree boundary=\forest@temp@negative@boundary,
1961     /forest/get max s tree boundary=\forest@temp@positive@boundary
1962 }%
1963 \edef\forest@temp@boundary{\expandonce{\forest@temp@negative@boundary}\expandonce{\forest@temp@positive@boundary}}
1964 \forest@path@getboundingrectangle@xy\forest@temp@boundary
1965 \pgfkeysalso{inner sep=0,fit/.expanded={(\the\pgf@xa,\the\pgf@ya)(\the\pgf@xb,\the\pgf@yb)}}%
1966 },
1967 use as bounding box/.style={%
1968   before drawing tree={
1969     tikz+/.expanded={%
1970       \noexpand\pgfresetboundingbox
1971       \noexpand\useasboundingbox
1972       ($(.anchor)+(\forestoption{min x},\forestoption{min y}))$)
1973       rectangle
1974       ($(.anchor)+(\forestoption{max x},\forestoption{max y}))$)
1975     };
1976   }
1977 },
1978 },
1979 use as bounding box'/.style={%
1980   before drawing tree={
1981     tikz+/.expanded={%
1982       \noexpand\pgfresetboundingbox
1983       \noexpand\useasboundingbox
1984       ($(.anchor)+(\forestoption{min x}+\pgfkeysvalueof{/pgf/outer xsep}/2+\pgfkeysvalueof{/pgf/inner xsep}))$)
1985       rectangle
1986       ($(.anchor)+(\forestoption{max x}-\pgfkeysvalueof{/pgf/outer xsep}/2-\pgfkeysvalueof{/pgf/inner xsep}))$)
1987     };
1988   }
1989 },
1990 },
1991 }%
1992 \def\forest@get@tree@boundary#1#2#3{#1=pos/neg,#2=grow,#3=receiving cs
1993   \def#3{}%
1994   \forest@node@getedge{#1}{#2}\forest@temp@boundary
1995   \forest@extendpath#3\forest@temp@boundary{\pgfpoint{\forestove{x}}{\forestove{y}}}%
1996 }
1997 \def\forest@setter@node{\forest@cn}%
1998 \def\forest@relatednode@option@setter#1.#2=#3\forest@END{%
1999   \forest@forthis{%
2000     \forest@nameandgo{#1}%
2001     \let\forest@setter@node\forest@cn
2002   }%
2003   \pgfkeysalso{#2={#3}}%
2004   \def\forest@setter@node{\forest@cn}%
2005 }%

```

### 10.4.3 Option propagation

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

```

2006 \forestset{
2007   for tree/.code={\forest@node@foreach{\pgfkeysalso{#1}}},
2008   if/.code n args={3}{%
2009     \pgfmathparse{#1}%
2010     \ifnum\pgfmathresult=0 \pgfkeysalso{#3}\else\pgfkeysalso{#2}\fi
2011   },
2012   where/.style n args={3}{for tree={if={#1}{#2}{#3}}},
2013   for descendants/.code={\forest@node@foreachdescendant{\pgfkeysalso{#1}}},
2014   for all next/.style={for next={#1,for all next={#1}}},
2015   for all previous/.style={for previous={#1,for all previous={#1}}},
2016   for siblings/.style={for all previous={#1,for all next={#1}},

```

```

2017 for ancestors/.style={for parent={#1,for ancestors={#1}},
2018 for ancestors'/.style={#1,for ancestors={#1}},
2019 for children/.code={\forest@node@foreachchild{\pgfkeysalso{#1}}},
2020 for c-commanded={for sibling={for tree={#1}}},
2021 for c-commanders={for sibling={#1},for parent={for c-commanders={#1}}}
2022 }

```

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

```

2023 \newcount\forest@repeat@key@depth
2024 \forestset{%
2025   repeat/.code 2 args={%
2026     \advance\forest@repeat@key@depth1
2027     \pgfmathparse{int(#1)}%
2028     \csedef{forest@repeat@key@the\forest@repeat@key@depth}{\pgfmathresult}%
2029     \expandafter\newloop\csname forest@repeat@key@loop@the\forest@repeat@key@depth\endcsname
2030     \def\forest@marshal{%
2031       \csname forest@repeat@key@loop@the\forest@repeat@key@depth\endcsname
2032       \forest@temp@count=\csname forest@repeat@key@the\forest@repeat@key@depth\endcsname\relax
2033       \ifnum\forest@temp@count>0
2034         \advance\forest@temp@count-1
2035         \csedef{forest@repeat@key@the\forest@repeat@key@depth}{\the\forest@temp@count}%
2036         \pgfkeysalso{#2}%
2037       }%
2038       \expandafter\forest@marshal\csname forest@repeat@key@repeat@the\forest@repeat@key@depth\endcsname
2039       \advance\forest@repeat@key@depth-1
2040     },
2041   }
2042   \pgfkeysgetvalue{/forest/repeat/.@cmd}\forest@temp
2043   \pgfkeyslet{/forest/node walk/repeat/.@cmd}\forest@temp
2044   %

```

#### 10.4.4 pgfmath extensions

```

2045 \pgfmathdeclarefunction{strequal}{2}{%
2046   \ifstrequal{#1}{#2}{\def\pgfmathresult{1}}{\def\pgfmathresult{0}}%
2047 }
2048 \pgfmathdeclarefunction{instr}{2}{%
2049   \pgfutil@in@{#1}{#2}%
2050   \ifpgfutil@in@\def\pgfmathresult{1}\else\def\pgfmathresult{0}\fi
2051 }
2052 \pgfmathdeclarefunction{strcat}{...}{%
2053   \edef\pgfmathresult{\forest@strip@braces{#1}}%
2054 }
2055 \def\forest@pgfmathhelper@attribute@toks#1#2{%
2056   \forest@forthis{%
2057     \forest@nameandgo{#1}%
2058     \forestoget{#2}\pgfmathresult
2059   }%
2060 }
2061 \def\forest@pgfmathhelper@attribute@dimen#1#2{%
2062   \forest@forthis{%
2063     \forest@nameandgo{#1}%
2064     \forestoget{#2}\forest@temp
2065     \pgfmathparse{+\forest@temp}%
2066   }%
2067 }
2068 \def\forest@pgfmathhelper@attribute@count#1#2{%
2069   \forest@forthis{%
2070     \forest@nameandgo{#1}%
2071     \forestoget{#2}\forest@temp
2072     \pgfmathtruncatemacro\pgfmathresult{\forest@temp}%

```

```

2073 }%
2074 }
2075 \pgfmathdeclarefunction{id}{1}{%
2076   \forest@forthis{%
2077     \forest@nameandgo{#1}%
2078     \let\pgfmathresult\forest@cn
2079   }%
2080 }
2081 \forestset{%
2082   if id/.code n args={3}{%
2083     \ifnum#1=\forest@cn\relax
2084       \pgfkeysalso{#2}%
2085     \else
2086       \pgfkeysalso{#3}%
2087     \fi
2088   },
2089   where id/.style n args={3}{for tree={if id={#1}{#2}{#3}}}
2090 }

```

## 10.5 Dynamic tree

```

2091 \def\forest@last@node{0}
2092 \def\forest@nodehandleby@name@nodewalk@or@bracket#1{%
2093   \ifx\pgfkeysnovalue#1%
2094     \edef\forest@last@node{\forest@node@Nametoid{forest@last@node}}%
2095   \else
2096     \forest@nodehandleby@nmb@checkfirst#1\forest@END
2097   \fi
2098 }
2099 \def\forest@nodehandleby@nmb@checkfirst#1#2\forest@END{%
2100   \ifx[#1%
2101     \forest@create@node{#1#2}%
2102   \else
2103     \forest@forthis{%
2104       \forest@nameandgo{#1#2}%
2105       \let\forest@last@node\forest@cn
2106     }%
2107   \fi
2108 }
2109 \def\forest@create@node#1{% #1=bracket representation
2110   \bracketParse{\forest@create@collectafterthought}%
2111   \forest@last@node=#1\forest@end@create@node
2112 }
2113 \def\forest@create@collectafterthought#1\forest@end@create@node{%
2114   \forest@let0{\forest@last@node}{delay}{\forest@last@node}{given options}%
2115   \forest@set{\forest@last@node}{given options}{}%
2116   \forest@eappto{\forest@last@node}{delay}{,\unexpanded{#1}}%
2117 }
2118 \def\forest@create@collectafterthought#1\forest@end@create@node{%
2119   \forest@node@Foreach{\forest@last@node}{%
2120     \forest@tolet0{delay}{given options}%
2121     \forest@set{given options}{}%
2122   }%
2123   \forest@eappto{\forest@last@node}{delay}{,\unexpanded{#1}}%
2124 }
2125 \def\forest@remove@node#1{%
2126   \forest@node@Remove{#1}%
2127 }
2128 \def\forest@append@node#1#2{%
2129   \forest@node@Remove{#2}%
2130   \forest@node@Append{#1}{#2}%

```

```

2131 }
2132 \def\forest@prepend@node#1#2{%
2133   \forest@node@Remove{#2}%
2134   \forest@node@Prepend{#1}{#2}%
2135 }
2136 \def\forest@insertafter@node#1#2{%
2137   \forest@node@Remove{#2}%
2138   \forest@node@Insertafter{\forestOve{#1}{@parent}}{#2}{#1}%
2139 }
2140 \def\forest@insertbefore@node#1#2{%
2141   \forest@node@Remove{#2}%
2142   \forest@node@Insertbefore{\forestOve{#1}{@parent}}{#2}{#1}%
2143 }
2144 \def\forest@appto@do@ynamics#1#2{%
2145   \forest@nodehandleby@name@nodewalk@or@bracket{#2}%
2146   \ifcase\forest@dynamics@copyhow\relax\or
2147     \forest@tree@copy{\forest@last@node}\forest@last@node
2148   \or
2149     \forest@node@copy{\forest@last@node}\forest@last@node
2150   \fi
2151   \forest@node@ifnamedefined{forest@last@node}{%
2152     \forestOpreto{\forest@last@node}{delay}
2153     {for id={\forest@node@Nametoid{forest@last@node}}{alias=forest@last@node},}%
2154   }{}%
2155   \forest@havedelayedoptionstrue
2156   \edef\forest@marshal{%
2157     \noexpand\apptotoks\noexpand\forest@do@ynamics{%
2158       \noexpand#1{\forest@cn}{\forest@last@node}}%
2159   }\forest@marshal
2160 }
2161 \forestset{%
2162   create/.code={\forest@create@node{#1}},
2163   append/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ynamics\forest@append@node{#1}},
2164   prepend/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ynamics\forest@prepend@node{#1}},
2165   insert after/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ynamics\forest@insertafter@node{#1}},
2166   insert before/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@ynamics\forest@insertbefore@node{#1}},
2167   append'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ynamics\forest@append@node{#1}},
2168   prepend'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ynamics\forest@prepend@node{#1}},
2169   insert after'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ynamics\forest@insertafter@node{#1}},
2170   insert before'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@ynamics\forest@insertbefore@node{#1}},
2171   append''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ynamics\forest@append@node{#1}},
2172   prepend''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ynamics\forest@prepend@node{#1}},
2173   insert after''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ynamics\forest@insertafter@node{#1}},
2174   insert before''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@ynamics\forest@insertbefore@node{#1}},
2175   remove/.code={%
2176     \pgfkeysalso{alias=forest@last@node}%
2177     \expandafter\apptotoks\expandafter\forest@do@ynamics\expandafter{%
2178       \expandafter\forest@remove@node\expandafter{\forest@cn}}%
2179   },
2180   set root/.code={%
2181     \forest@nodehandleby@name@nodewalk@or@bracket{#1}%
2182     \edef\forest@marshal{%
2183       \noexpand\apptotoks\noexpand\forest@do@ynamics{%
2184         \def\noexpand\forest@root{\forest@last@node}%
2185       }%
2186     }\forest@marshal
2187   },
2188   replace by/.code={\forest@replaceby@code{#1}{insert after}},
2189   replace by'/.code={\forest@replaceby@code{#1}{insert after'}},
2190   replace by''/.code={\forest@replaceby@code{#1}{insert after''}},
2191 }

```



```

2192 \def\forest@replaceby@code#1#2{#1=node spec,#2=insert after['][']}
2193 \ifnum\forest@parent=0
2194   \pgfkeysalso{set root={#1}}%
2195 \else
2196   \pgfkeysalso{alias=forest@last@node,#2={#1}}%
2197   \eapptotoks\forest@do@dynamics{%
2198     \noexpand\ifnum\noexpand\forest@ve{\forest@cn}{@parent}=\forest@parent}
2199     \noexpand\forest@remove@node{\forest@cn}%
2200   \noexpand\fi
2201   }%
2202 \fi
2203 }

```

## 11 Stages

```

2204 \forestset{
2205   stages/.style={
2206     process keylist=before typesetting nodes,
2207     typeset nodes stage,
2208     process keylist=before packing,
2209     pack stage,
2210     process keylist=before computing xy,
2211     compute xy stage,
2212     process keylist=before drawing tree,
2213     draw tree stage,
2214   },
2215   typeset nodes stage/.style={for root'=typeset nodes},
2216   pack stage/.style={for root'=pack},
2217   compute xy stage/.style={for root'=compute xy},
2218   draw tree stage/.style={for root'=draw tree},
2219   process keylist/.code={\forest@process@hook@keylist{#1}},
2220   declare keylist={given options}{},
2221   declare keylist={before typesetting nodes}{},
2222   declare keylist={before packing}{},
2223   declare keylist={before computing xy}{},
2224   declare keylist={before drawing tree}{},
2225   declare keylist={delay}{},
2226   delay/.append code={\forest@havedelayedoptionstrue},
2227   delay n/.style 2 args={if={#1==0}{#2}{delay@n={#1}{#2}}},
2228   delay@n/.style 2 args={
2229     if={#1==1}{delay={#2}}{delay={delay@n/.wrap pgfmath arg={{##1}{#2}}{#1-1}}}
2230   },
2231   if have delayed/.code 2 args={%
2232     \ifforest@havedelayedoptions\pgfkeysalso{#1}\else\pgfkeysalso{#2}\fi
2233   },
2234   typeset nodes/.code={%
2235     \forest@drawtree@preservenodeboxes@false
2236     \forest@node@foreach{\forest@node@typeset}},
2237   typeset nodes'/.code={%
2238     \forest@drawtree@preservenodeboxes@true
2239     \forest@node@foreach{\forest@node@typeset}},
2240   typeset node/.code={%
2241     \forest@drawtree@preservenodeboxes@false
2242     \forest@node@typeset
2243   },
2244   pack/.code={\forest@pack},
2245   pack'/.code={\forest@pack@onlythisnode},
2246   compute xy/.code={\forest@node@computeabsolutepositions},
2247   draw tree box/.store in=\forest@drawtreebox,
2248   draw tree box,
2249   draw tree/.code={%

```

```

2250 \forest@drawtree@preservenodeboxes@false
2251 \forest@node@drawtree
2252 },
2253 draw tree'/.code={%
2254 \forest@drawtree@preservenodeboxes@true
2255 \forest@node@drawtree
2256 },
2257 }
2258 \newtoks\forest@do@dynamics
2259 \newif\ifforest@havedelayedoptions
2260 \def\forest@process@hook@keylist#1{%
2261 \forest@loopa
2262 \forest@havedelayedoptionsfalse
2263 \forest@do@dynamics={}%
2264 \forest@fornode{\forest@root}{\forest@process@hook@keylist@{#1}}%
2265 \expandafter\ifstrempty\expandafter{\the\forest@do@dynamics}{}%
2266 \the\forest@do@dynamics
2267 \forest@node@Compute@numeric@ts@info{\forest@root}%
2268 \forest@havedelayedoptionstrue
2269 }%
2270 \ifforest@havedelayedoptions
2271 \forest@node@Foreach{\forest@root}{%
2272 \forest@toget{delay}\forest@temp@delayed
2273 \forest@tolet{#1}\forest@temp@delayed
2274 \forest@toset{delay}{}%
2275 }%
2276 \forest@repeata
2277 }
2278 \def\forest@process@hook@keylist@#1{%
2279 \forest@node@foreach{%
2280 \forest@toget{#1}\forest@temp@keys
2281 \ifdefvoid\forest@temp@keys}{}%
2282 \forest@toset{#1}{}%
2283 \expandafter\forest@set\expandafter{\forest@temp@keys}%
2284 }%
2285 }%
2286 }

```

## 11.1 Typesetting nodes

```

2287 \def\forest@node@typeset{%
2288 \let\forest@next\forest@node@typeset@
2289 \forest@toifdefined{box}{%
2290 \ifforest@drawtree@preservenodeboxes@
2291 \let\forest@next\relax
2292 \fi
2293 }{%
2294 \locbox\forest@temp@box
2295 \forest@tolet{box}\forest@temp@box
2296 }%
2297 \def\forest@node@typeset@restore{%
2298 \ifdefined\ifsa@tikz\forest@standalone@hack\fi
2299 \forest@next
2300 \forest@node@typeset@restore
2301 }
2302 \def\forest@standalone@hack{%
2303 \ifsa@tikz
2304 \let\forest@standalone@tikzpicture\tikzpicture
2305 \let\forest@standalone@endtikzpicture\endtikzpicture
2306 \let\tikzpicture\sa@orig@tikzpicture
2307 \let\endtikzpicture\sa@orig@endtikzpicture

```

```

2308 \def\forest@node@typeset@restore{%
2309 \let\tikzpicture\forest@standalone@tikzpicture
2310 \let\endtikzpicture\forest@standalone@endtikzpicture
2311 }%
2312 \fi
2313 }
2314 \newbox\forest@box
2315 \def\forest@node@typeset@{%
2316 \forestoget{name}\forest@nodename
2317 \edef\forest@temp@nodeformat{\forestove{node format}}%
2318 \gdef\forest@smuggle{}%
2319 \setbox0=\hbox{%
2320 \begin{tikzpicture}%
2321 \pgfpositionnodelater{\forest@positionnodelater@save}%
2322 \forest@temp@nodeformat
2323 \pgfinterruptpath
2324 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{forestcomputenodeboundary}%
2325 \endpgfinterruptpath
2326 %\forest@compute@node@boundary\forest@temp
2327 %\xappto\forest@smuggle{noexpand\forestoset{boundary}{\expandonce\forest@temp}}%
2328 \if\relax\forestove{parent anchor}\relax
2329 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{center}%
2330 \else
2331 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{\forestove{parent anchor}}%
2332 \fi
2333 \xappto\forest@smuggle{%
2334 \noexpand\forestoset{parent@anchor}{%
2335 \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
2336 \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
2337 \if\relax\forestove{child anchor}\relax
2338 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{center}%
2339 \else
2340 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{\forestove{child anchor}}%
2341 \fi
2342 \xappto\forest@smuggle{%
2343 \noexpand\forestoeset{child@anchor}{%
2344 \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
2345 \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
2346 \if\relax\forestove{anchor}\relax
2347 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{center}%
2348 \else
2349 \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{\forestove{anchor}}%
2350 \fi
2351 \xappto\forest@smuggle{%
2352 \noexpand\forestoeset{@anchor}{%
2353 \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
2354 \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
2355 \end{tikzpicture}%
2356 }%
2357 \setbox\forestove{box}=\box\forest@box % smuggle the box
2358 \forestolet{boundary}\forest@global@boundary
2359 \forest@smuggle % ... and the rest
2360 }
2361 \forestset{
2362 declare readonly dimen={min x},
2363 declare readonly dimen={min y},
2364 declare readonly dimen={max x},
2365 declare readonly dimen={max y},
2366 }
2367 \def\forest@patch@enormouscoordinateboxbounds@plus#1{%
2368 \expandafter\ifstrequal\expandafter{#1}{16000.0pt}{\def#1{0.0pt}}}%

```

```

2369 }
2370 \def\forest@patch@enormouscoordinateboxbounds@minus#1{%
2371   \expandafter\ifstrequal\expandafter{#1}{-16000.Opt}{\def#1{0.Opt}}{}}%
2372 }
2373 \def\forest@positionnodelater@save{%
2374   \global\setbox\forest@box=\box\pgfpositionnodelaterbox
2375   \xappto\forest@smuggle{\noexpand\forestoset{later@name}{\pgfpositionnodelatername}}%
2376   % a bug in pgf? ---well, here's a patch
2377   \forest@patch@enormouscoordinateboxbounds@plus\pgfpositionnodelaterminx
2378   \forest@patch@enormouscoordinateboxbounds@plus\pgfpositionnodelaterminy
2379   \forest@patch@enormouscoordinateboxbounds@minus\pgfpositionnodelatermaxx
2380   \forest@patch@enormouscoordinateboxbounds@minus\pgfpositionnodelatermaxy
2381   % end of patch
2382   \xappto\forest@smuggle{\noexpand\forestoset{min x}{\pgfpositionnodelaterminx}}%
2383   \xappto\forest@smuggle{\noexpand\forestoset{min y}{\pgfpositionnodelaterminy}}%
2384   \xappto\forest@smuggle{\noexpand\forestoset{max x}{\pgfpositionnodelatermaxx}}%
2385   \xappto\forest@smuggle{\noexpand\forestoset{max y}{\pgfpositionnodelatermaxy}}%
2386 }
2387 \def\forest@node@forest@positionnodelater@restore{%
2388   \ifforest@drawtree@preservenodeboxes@
2389     \let\forest@boxorcopy\copy
2390   \else
2391     \let\forest@boxorcopy\box
2392   \fi
2393   \forestoget{box}\forest@temp
2394   \setbox\pgfpositionnodelaterbox=\forest@boxorcopy\forest@temp
2395   \edef\pgfpositionnodelatername{\forestove{later@name}}%
2396   \edef\pgfpositionnodelaterminx{\forestove{min x}}%
2397   \edef\pgfpositionnodelaterminy{\forestove{min y}}%
2398   \edef\pgfpositionnodelatermaxx{\forestove{max x}}%
2399   \edef\pgfpositionnodelatermaxy{\forestove{max y}}%
2400 }

```

## 11.2 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.

```

2401 \def\forest@pack{%
2402   \forest@pack@computetiers
2403   \forest@pack@computegrowthuniformity
2404   \forest@@@pack
2405 }
2406 \def\forest@@@pack{%
2407   \ifnum\forestove{n children}>0
2408     \ifnum\forestove{uniform growth}>0
2409       \forest@pack@level@uniform
2410       \forest@pack@aligtiers@ofsubtree
2411       \forest@pack@sibling@uniform@recursive
2412     \else
2413       \forest@node@foreachchild{\forest@@@pack}%
2414       \forest@pack@level@nonuniform
2415       \forest@pack@aligtiers
2416       \forest@pack@sibling@uniform@applyreversed
2417     \fi
2418   \fi
2419 }
2420 \def\forest@pack@onlythisnode{%
2421   \ifnum\forestove{n children}>0
2422     \forest@pack@computetiers
2423     \forest@pack@level@nonuniform
2424     \forest@pack@aligtiers

```

```

2425     \forest@pack@sibling@uniform@applyreversed
2426 \fi
2427 }

```

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

```

2428 \def\forest@pack@computegrowthuniformity{%
2429 \forest@node@foreachchild{\forest@pack@computegrowthuniformity}%
2430 \edef\forest@pack@cgu@uniformity{%
2431 \ifnum\forestove{n children}=0
2432 2\else 1\fi
2433 }%
2434 \forestoget{grow}\forest@pack@cgu@parentgrow
2435 \forest@node@foreachchild{%
2436 \ifnum\forestove{uniform growth}=0
2437 \def\forest@pack@cgu@uniformity{0}%
2438 \else
2439 \ifnum\forestove{uniform growth}=1
2440 \ifnum\forestove{grow}=\forest@pack@cgu@parentgrow\relax\else
2441 \def\forest@pack@cgu@uniformity{0}%
2442 \fi
2443 \fi
2444 \fi
2445 }%
2446 \forestolet{uniform growth}\forest@pack@cgu@uniformity
2447 }

```

Pack children in the level dimension in a uniform tree.

```

2448 \def\forest@pack@level@uniform{%
2449 \let\forest@plu@minchildl\relax
2450 \forestoget{grow}\forest@plu@grow
2451 \forest@node@foreachchild{%
2452 \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
2453 \advance\pgf@xa\forestove{1}\relax
2454 \ifx\forest@plu@minchildl\relax
2455 \edef\forest@plu@minchildl{\the\pgf@xa}%
2456 \else
2457 \ifdim\pgf@xa<\forest@plu@minchildl\relax
2458 \edef\forest@plu@minchildl{\the\pgf@xa}%
2459 \fi
2460 \fi
2461 }%
2462 \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
2463 \pgfutil@tempdima=\pgf@xb\relax
2464 \advance\pgfutil@tempdima -\forest@plu@minchildl\relax
2465 \advance\pgfutil@tempdima \forestove{1 sep}\relax
2466 \ifdim\pgfutil@tempdima>0pt
2467 \forest@node@foreachchild{%
2468 \forestoest{1}{\the\dimexpr\forestove{1}+\the\pgfutil@tempdima}%
2469 }%
2470 \fi
2471 \forest@node@foreachchild{%
2472 \ifnum\forestove{n children}>0
2473 \forest@pack@level@uniform
2474 \fi
2475 }%
2476 }

```

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

```

2477 \def\forest@pack@level@nonuniform{%
2478 \let\forest@plu@minchildl\relax
2479 \forestoget{grow}\forest@plu@grow

```

```

2480 \forest@node@foreachchild{%
2481   \forest@node@getedge{negative}{\forest@plu@grow}{\forest@plnu@negativechildedge}%
2482   \forest@node@getedge{positive}{\forest@plu@grow}{\forest@plnu@positivechildedge}%
2483   \def\forest@plnu@childedge{\forest@plnu@negativechildedge\forest@plnu@positivechildedge}%
2484   \forest@path@getboundingrectangle@ls\forest@plnu@childedge{\forest@plu@grow}%
2485   \advance\pgf@xa\forestove{1}\relax
2486   \ifx\forest@plu@minchildl\relax
2487     \edef\forest@plu@minchildl{\the\pgf@xa}%
2488   \else
2489     \ifdim\pgf@xa<\forest@plu@minchildl\relax
2490       \edef\forest@plu@minchildl{\the\pgf@xa}%
2491     \fi
2492   \fi
2493 }%
2494 \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
2495 \pgfutil@tempdima=\pgf@xb\relax
2496 \advance\pgfutil@tempdima-\forest@plu@minchildl\relax
2497 \advance\pgfutil@tempdima \forestove{1 sep}\relax
2498 \ifdim\pgfutil@tempdima>0pt
2499   \forest@node@foreachchild{%
2500     \forestoeset{1}{\the\dimexpr\the\pgfutil@tempdima+\forestove{1}}%
2501   }%
2502 \fi
2503 }

```

Align tiers.

```

2504 \def\forest@pack@aligntiers{%
2505   \forestoget{grow}\forest@temp@parentgrow
2506   \forestoget{@tiers}\forest@temp@tiers
2507   \forlistloop\forest@pack@aligntier@\forest@temp@tiers
2508 }
2509 \def\forest@pack@aligntiers@ofsubtree{%
2510   \forest@node@foreach{\forest@pack@aligntiers}%
2511 }
2512 \def\forest@pack@aligntiers@computeabsl{%
2513   \forestoleto{abs@1}{1}%
2514   \forest@node@foreachdescendant{\forest@pack@aligntiers@computeabsl@}%
2515 }
2516 \def\forest@pack@aligntiers@computeabsl@{%
2517   \forestoeset{abs@1}{\the\dimexpr\forestove{1}+\forestove{\forestove{@parent}}{abs@1}}%
2518 }
2519 \def\forest@pack@aligntier@#1{%
2520   \forest@pack@aligntiers@computeabsl
2521   \pgfutil@tempdima=-\maxdimen\relax
2522   \def\forest@temp@currenttier{#1}%
2523   \forest@node@foreach{%
2524     \forestoget{tier}\forest@temp@tier
2525     \ifx\forest@temp@currenttier\forest@temp@tier
2526       \ifdim\pgfutil@tempdima<\forestove{abs@1}\relax
2527         \pgfutil@tempdima=\forestove{abs@1}\relax
2528       \fi
2529     \fi
2530   }%
2531   \ifdim\pgfutil@tempdima=-\maxdimen\relax\else
2532     \forest@node@foreach{%
2533       \forestoget{tier}\forest@temp@tier
2534       \ifx\forest@temp@currenttier\forest@temp@tier
2535         \forestoeset{1}{\the\dimexpr\pgfutil@tempdima-\forestove{abs@1}+\forestove{1}}%
2536       \fi
2537     }%
2538   \fi

```

2539 }

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

```
2540 \def\forest@pack@sibling@uniform@recursive{%
2541   \forest@node@foreachchild{\forest@pack@sibling@uniform@recursive}%
2542   \forest@pack@sibling@uniform@applyreversed
2543 }
```

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

```
2544 \def\forest@pack@sibling@uniform@applyreversed{%
2545   \ifnum\forestove{n children}>1
2546     \ifnum\forestove{reversed}=0
2547       \pack@sibling@uniform@main{first}{last}{next}{previous}%
2548     \else
2549       \pack@sibling@uniform@main{last}{first}{previous}{next}%
2550     \fi
2551   \else
2552     \ifnum\forestove{n children}=1
```

No need to run packing, but we still need to align the children.

```
2553     \csname forest@calign@\forestove{calign}\endcsname
2554   \fi
2555 \fi
2556 }
```

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

```
2557 \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.

```
2558   \forestoget{@#1}\forest@child
2559   \edef\forest@temp{%
2560     \noexpand\forest@fornode{\forestove{@#1}}{%
2561       \noexpand\forest@node@getedge
2562         {positive}
2563         {\forestove{grow}}
2564       \noexpand\forest@temp@edge
2565     }%
2566   }\forest@temp
2567   \forest@pack@pgfpoint@childposition\forest@child
2568   \let\forest@previous@positive@edge\pgfutil@empty
2569   \forest@extendpath\forest@previous@positive@edge\forest@temp@edge{%
2570   \forest@get{\forest@child}{@#3}\forest@child
```

Loop until the current child is the null node.

```
2571   \edef\forest@previous@child@s{0pt}%
2572   \forest@loopb
2573   \unless\ifnum\forest@child=0
```

Get the negative edge of the child.

```
2574     \edef\forest@temp{%
2575       \noexpand\forest@fornode{\forest@child}{%
2576         \noexpand\forest@node@getedge
2577           {negative}
2578           {\forestove{grow}}
2579         \noexpand\forest@temp@edge
2580       }%
2581     }\forest@temp
```

Set `\pgf@x` and `\pgf@y` to the position of the child (in the coordinate system of this node).

```
2582     \forest@pack@pgfpoint@childposition\forest@child
```

Translate the edge of the child by the child's position.

```

2583 \let\forest@child@negative@edge\pgfutil@empty
2584 \forest@extendpath\forest@child@negative@edge\forest@temp@edge{}%

```

Setup the grow line: the angle is given by this node's `grow` attribute.

```

2585 \forest@setupgrowline{\forestove{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!)

```

2586 \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`.

```

2587 \ifx\forest@csdistance\relax
2588   %\forest@set{\forest@child}{s}{\forest@previous@child@s}%
2589 \else
2590   \advance\pgfutil@tempdimb-\forest@csdistance\relax
2591   \advance\pgfutil@tempdimb\forestove{s sep}\relax
2592   \forest@set{\forest@child}{s}{\the\dimexpr\forestove{s}-\forest@csdistance+\forestove{s sep}}%
2593 \fi

```

Retain monotonicity (is this ok?). (This problem arises when the adjacent children's `l` are too far apart.)

```

2594 \ifdim\forest@ve{\forest@child}{s}<\forest@previous@child@s\relax
2595   \forest@set{\forest@child}{s}{\forest@previous@child@s}%
2596 \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.

```

2597 \forest@get{\forest@child}{s}\forest@child@s
2598 \edef\forest@previous@child@s{\forest@child@s}%
2599 \edef\forest@temp{%
2600   \noexpand\forest@fornode{\forest@child}{%
2601     \noexpand\forest@node@getedge
2602       {positive}
2603       {\forestove{grow}}
2604     \noexpand\forest@temp@edge
2605   }%
2606 }%
2607 \forest@pack@pgfpoint@child@position\forest@child
2608 \forest@extendpath\forest@previous@positive@edge\forest@temp@edge{}%
2609 \forest@getpositivetightedgeofpath\forest@previous@positive@edge\forest@previous@positive@edge
2610 \forest@get{\forest@child}{@#3}\forest@child
2611 \forest@repeatb

```

Shift the position of all children to achieve the desired alignment of the parent and its children.

```

2612 \csname forest@calign@\forestove{calign}\endcsname
2613 }

```

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

```

2614 \def\forest@pack@pgfpoint@child@position#1{%
2615   {%
2616     \pgftransformreset
2617     \pgftransformrotate{\forestove{grow}}%
2618     \forest@fornode{#1}{%
2619       \pgfpointtransformed{\pgfpoint{\forestove{1}}{\forestove{s}}}%
2620     }%
2621   }%
2622 }

```

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

```

2623 \def\forest@pack@pgfpoint@positioningrow#1{%
2624   {%
2625     \pgftransformreset

```



```

2626 \pgftransformrotate{#1}%
2627 \pgfpointransformed{\pgfqpoint{\forestove{1}}{\forestove{s}}}%
2628 }%
2629 }

Child alignment.
2630 \def\forest@calign@s@shift#1{%
2631 \pgfutil@tempdima=#1\relax
2632 \forest@node@foreachchild{%
2633 \forestoeset{s}{\the\dimexpr\forestove{s}+\pgfutil@tempdima}%
2634 }%
2635 }
2636 \def\forest@calign@child{%
2637 \forest@calign@s@shift{-\forestOve{\forest@node@nornbarthchildid{\forestove{calign primary child}}}{s}}%
2638 }
2639 \csdef{forest@calign@child edge}{%
2640 {%
2641 \edef\forest@temp@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2642 \pgftransformreset
2643 \pgftransformrotate{\forestove{grow}}%
2644 \pgfpointransformed{\pgfqpoint{\forestOve{\forest@temp@child}{1}}{\forestOve{\forest@temp@child}{s}}}%
2645 \pgf@xa=\pgf@x\relax\pgf@ya=\pgf@y\relax
2646 \forestOve{\forest@temp@child}{child@anchor}%
2647 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2648 \forestove{parent@anchor}%
2649 \advance\pgf@xa-\pgf@x\relax\advance\pgf@ya-\pgf@y\relax
2650 \edef\forest@marshal{%
2651 \noexpand\pgftransformreset
2652 \noexpand\pgftransformrotate{-\forestove{grow}}%
2653 \noexpand\pgfpointransformed{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
2654 }\forest@marshal
2655 }%
2656 \forest@calign@s@shift{\the\dimexpr-\the\pgf@y}%
2657 }
2658 \csdef{forest@calign@midpoint}{%
2659 \forest@calign@s@shift{\the\dimexpr Opt -%
2660 (\forestOve{\forest@node@nornbarthchildid{\forestove{calign primary child}}}{s}%
2661 +\forestOve{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}{s}%
2662 )/2\relax
2663 }%
2664 }
2665 \csdef{forest@calign@edge midpoint}{%
2666 {%
2667 \edef\forest@temp@firstchild{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2668 \edef\forest@temp@secondchild{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
2669 \pgftransformreset
2670 \pgftransformrotate{\forestove{grow}}%
2671 \pgfpointransformed{\pgfqpoint{\forestOve{\forest@temp@firstchild}{1}}{\forestOve{\forest@temp@firstchild}{s}}}%
2672 \pgf@xa=\pgf@x\relax\pgf@ya=\pgf@y\relax
2673 \forestOve{\forest@temp@firstchild}{child@anchor}%
2674 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2675 \edef\forest@marshal{%
2676 \noexpand\pgfpointransformed{\noexpand\pgfqpoint{\forestOve{\forest@temp@secondchild}{1}}{\forestOve{\forest@temp@secondchild}{s}}}%
2677 }\forest@marshal
2678 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2679 \forestOve{\forest@temp@secondchild}{child@anchor}%
2680 \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
2681 \divide\pgf@xa2 \divide\pgf@ya2
2682 \edef\forest@marshal{%
2683 \noexpand\pgftransformreset
2684 \noexpand\pgftransformrotate{-\forestove{grow}}%

```

```

2685     \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
2686   }\forest@marshal
2687 }%
2688 \forest@calign@s@shift{\the\dimexpr-\the\pgf@y}%
2689 }

```

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.

```

2690 \csdef{forest@calign@fixed angles}{%
2691   \ifnum\forestove{n children}>1
2692     \edef\forest@ca@first@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
2693     \edef\forest@ca@second@child{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
2694     \ifnum\forestove{reversed}=1
2695       \let\forest@temp\forest@ca@first@child
2696       \let\forest@ca@first@child\forest@ca@second@child
2697       \let\forest@ca@second@child\forest@temp
2698     \fi
2699     \forestOget{\forest@ca@first@child}{1}\forest@ca@first@l
2700     \forestOget{\forest@ca@second@child}{1}\forest@ca@second@l
2701     \pgfmathsetlengthmacro\forest@ca@desired@s@distance{%
2702       tan(\forestove{calign secondary angle})*\forest@ca@second@l
2703       -tan(\forestove{calign primary angle})*\forest@ca@first@l
2704     }%
2705     \forestOget{\forest@ca@first@child}{s}\forest@ca@first@s
2706     \forestOget{\forest@ca@second@child}{s}\forest@ca@second@s
2707     \pgfmathsetlengthmacro\forest@ca@actual@s@distance{%
2708       \forest@ca@second@s-\forest@ca@first@s}%
2709     \ifdim\forest@ca@desired@s@distance>\forest@ca@actual@s@distance\relax
2710       \ifdim\forest@ca@actual@s@distance=0pt
2711         \pgfmathsetlength\pgfutil@tempdima{tan(\forestove{calign primary angle})*\forest@ca@second@l}%
2712         \pgfmathsetlength\pgfutil@tempdimb{\forest@ca@desired@s@distance/(\forestove{n children}-1)}%
2713         \forest@node@foreachchild{%
2714           \forestoaset{s}{\the\pgfutil@tempdima}%
2715           \advance\pgfutil@tempdima\pgfutil@tempdimb
2716         }%
2717         \def\forest@calign@anchor{Opt}%
2718       \else
2719         \pgfmathsetmacro\forest@ca@ratio{%
2720           \forest@ca@desired@s@distance/\forest@ca@actual@s@distance}%
2721         \forest@node@foreachchild{%
2722           \pgfmathsetlengthmacro\forest@temp{\forest@ca@ratio*\forestove{s}}%
2723           \forestolet{s}\forest@temp
2724         }%
2725         \pgfmathsetlengthmacro\forest@calign@anchor{%
2726           -tan(\forestove{calign primary angle})*\forest@ca@first@l}%
2727         \fi
2728       \else
2729         \ifdim\forest@ca@desired@s@distance<\forest@ca@actual@s@distance\relax
2730           \pgfmathsetlengthmacro\forest@ca@ratio{%
2731             \forest@ca@actual@s@distance/\forest@ca@desired@s@distance}%
2732           \forest@node@foreachchild{%
2733             \pgfmathsetlengthmacro\forest@temp{\forest@ca@ratio*\forestove{l}}%
2734             \forestolet{l}\forest@temp
2735           }%
2736           \forestOget{\forest@ca@first@child}{1}\forest@ca@first@l
2737           \pgfmathsetlengthmacro\forest@calign@anchor{%
2738             -tan(\forestove{calign primary angle})*\forest@ca@first@l}%
2739           \fi
2740         \fi

```

```

2741 \forest@calign@s@shift{-\forest@calign@anchor}%
2742 \fi
2743 }
2744 \csdef{forest@calign@fixed edge angles}{%
2745 \ifnum\forestove{n children}>1
2746 \edef\forest@ca@first@child{\forest@node@normbarthchildid{\forestove{calign primary child}}}%
2747 \edef\forest@ca@second@child{\forest@node@normbarthchildid{\forestove{calign secondary child}}}%
2748 \ifnum\forestove{reversed}=1
2749 \let\forest@temp\forest@ca@first@child
2750 \let\forest@ca@first@child\forest@ca@second@child
2751 \let\forest@ca@second@child\forest@temp
2752 \fi
2753 \forest@get{\forest@ca@first@child}{1}\forest@ca@first@l
2754 \forest@get{\forest@ca@second@child}{1}\forest@ca@second@l
2755 \forest@get{parent@anchor}\forest@ca@parent@anchor
2756 \forest@ca@parent@anchor
2757 \edef\forest@ca@parent@anchor@s{\the\pgf@x}%
2758 \edef\forest@ca@parent@anchor@l{\the\pgf@y}%
2759 \forest@get{\forest@ca@first@child}{child@anchor}\forest@ca@first@child@anchor
2760 \forest@ca@first@child@anchor
2761 \edef\forest@ca@first@child@anchor@s{\the\pgf@x}%
2762 \edef\forest@ca@first@child@anchor@l{\the\pgf@y}%
2763 \forest@get{\forest@ca@second@child}{child@anchor}\forest@ca@second@child@anchor
2764 \forest@ca@second@child@anchor
2765 \edef\forest@ca@second@child@anchor@s{\the\pgf@x}%
2766 \edef\forest@ca@second@child@anchor@l{\the\pgf@y}%
2767 \pgfmathsetlengthmacro\forest@ca@desired@second@edge@s{\tan(\forestove{calign secondary angle})*%
2768 (\forest@ca@second@l-\forest@ca@second@child@anchor@l+\forest@ca@parent@anchor@l)}%
2769 \pgfmathsetlengthmacro\forest@ca@desired@first@edge@s{\tan(\forestove{calign primary angle})*%
2770 (\forest@ca@first@l-\forest@ca@first@child@anchor@l+\forest@ca@parent@anchor@l)}%
2771 \pgfmathsetlengthmacro\forest@ca@desired@s@distance{\forest@ca@desired@second@edge@s-\forest@ca@desired@f
2772 \forest@get{\forest@ca@first@child}{s}\forest@ca@first@s
2773 \forest@get{\forest@ca@second@child}{s}\forest@ca@second@s
2774 \pgfmathsetlengthmacro\forest@ca@actual@s@distance{%
2775 \forest@ca@second@s+\forest@ca@second@child@anchor@s
2776 -\forest@ca@first@s-\forest@ca@first@child@anchor@s}%
2777 \ifdim\forest@ca@desired@s@distance>\forest@ca@actual@s@distance\relax
2778 \ifdim\forest@ca@actual@s@distance=0pt
2779 \forest@get{n children}\forest@temp@n@children
2780 \forest@node@foreachchild{%
2781 \forest@get{child@anchor}\forest@temp@child@anchor
2782 \forest@temp@child@anchor
2783 \edef\forest@temp@child@anchor@s{\the\pgf@x}%
2784 \pgfmathsetlengthmacro\forest@temp{%
2785 \forest@ca@desired@first@edge@s+(\forestove{n}-1)*\forest@ca@desired@s@distance/(\forest@temp@n@C
2786 \forest@let{s}\forest@temp
2787 }%
2788 \def\forest@calign@anchor{Opt}%
2789 \else
2790 \pgfmathsetmacro\forest@ca@ratio{%
2791 \forest@ca@desired@s@distance/\forest@ca@actual@s@distance}%
2792 \forest@node@foreachchild{%
2793 \forest@get{child@anchor}\forest@temp@child@anchor
2794 \forest@temp@child@anchor
2795 \edef\forest@temp@child@anchor@s{\the\pgf@x}%
2796 \pgfmathsetlengthmacro\forest@temp{%
2797 \forest@ca@ratio*(%
2798 \forestove{s}-\forest@ca@first@s
2799 +\forest@temp@child@anchor@s-\forest@ca@first@child@anchor@s)%
2800 +\forest@ca@first@s
2801 +\forest@ca@first@child@anchor@s-\forest@temp@child@anchor@s}%

```

```

2802     \forestolet{s}\forest@temp
2803   }%
2804   \pgfmathsetlengthmacro\forest@calign@anchor{%
2805     -tan(\forestove{calign primary angle})*(\forest@ca@first@l-\forest@ca@first@child@anchor@l+\forest@
2806     +\forest@ca@first@child@anchor@s-\forest@ca@parent@anchor@s
2807   }%
2808   \fi
2809 \else
2810   \ifdim\forest@ca@desired@s@distance<\forest@ca@actual@s@distance\relax
2811   \pgfmathsetlengthmacro\forest@ca@ratio{%
2812     \forest@ca@actual@s@distance/\forest@ca@desired@s@distance}%
2813   \forest@node@foreachchild{%
2814     \forestoget{child@anchor}\forest@temp@child@anchor
2815     \forest@temp@child@anchor
2816     \edef\forest@temp@child@anchor@l{\the\pgf}%
2817     \pgfmathsetlengthmacro\forest@temp{%
2818       \forest@ca@ratio*(%
2819         \forestove{l}+\forest@ca@parent@anchor@l-\forest@temp@child@anchor@l)
2820         -\forest@ca@parent@anchor@l+\forest@temp@child@anchor@l}%
2821     \forestolet{l}\forest@temp
2822   }%
2823   \forestoget{\forest@ca@first@child}{l}\forest@ca@first@l
2824   \pgfmathsetlengthmacro\forest@calign@anchor{%
2825     -tan(\forestove{calign primary angle})*(\forest@ca@first@l+\forest@ca@parent@anchor@l-\forest@temp@
2826     +\forest@ca@first@child@anchor@s-\forest@ca@parent@anchor@s
2827   }%
2828   \fi
2829   \fi
2830   \forest@calign@s@shift{-\forest@calign@anchor}%
2831 \fi
2832 }

```

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.

```

2833 \def\forest@node@getedge#1#2#3{%
2834   \forestoget{#1@edge@#2}#3%
2835   \ifx#3\relax
2836     \forest@node@foreachchild{%
2837       \forest@node@getedge{#1}{#2}{\forest@temp@edge}%
2838     }%
2839     \forest@forthis{\forest@node@getedges{#2}}%
2840     \forestoget{#1@edge@#2}#3%
2841 \fi
2842 }

```

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.

```

2843 \def\forest@node@getedges#1{%
  Run the computation in a TEX group.
2844   %{}%
  Setup the grow line.
2845   \forest@setupgrowline{#1}%
  Get the edge of the node itself.
2846   \ifnum\forestove{ignore}=0
2847     \forestoget{boundary}\forest@node@boundary
2848   \else
2849     \def\forest@node@boundary{}%
2850 \fi

```

```

2851 \cscname forest@getboth\forestove{fit}edgesofpath\endcsname
2852 \forest@node@boundary\forest@negative@node@edge\forest@positive@node@edge
2853 \forestolet{negative@edge@#1}\forest@negative@node@edge
2854 \forestolet{positive@edge@#1}\forest@positive@node@edge

Add the edges of the children.
2855 \get@edges@merge{negative}{#1}%
2856 \get@edges@merge{positive}{#1}%
2857 }%
2858 }

Merge the #1 (=negative or positive) edge of the node with #1 edges of the children. #2 = grow
angle.
2859 \def\get@edges@merge#1#2{%
2860 \ifnum\forestove{n children}>0
2861 \forestoget{#1@edge@#2}\forest@node@edge

Remember the node's parent anchor and add it to the path (for breaking).
2862 \forestove{parent@anchor}%
2863 \edef\forest@getedge@pa@l{\the\pgf@x}%
2864 \edef\forest@getedge@pa@s{\the\pgf@y}%
2865 \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).
2866 \pgftransformreset
2867 \pgftransformrotate{\forestove{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.
2868 \def\forest@all@edges{%
2869 \forest@node@foreachchild{%
2870 \forestoget{#1@edge@#2}\forest@temp@edge
2871 \pgfpointransformed{\pgfqpoint{\forestove{1}}{\forestove{s}}}%
2872 \forest@extendpath\forest@node@edge\forest@temp@edge}%
2873 \ifnum\forestove{ignore edge}=0
2874 \pgfpointhead
2875 {\pgfpointransformed{\pgfqpoint{\forestove{1}}{\forestove{s}}}%
2876 {\forestove{child@anchor}}%
2877 \pgfgetlastxy{\forest@getedge@ca@l}{\forest@getedge@ca@s}%
2878 \eappto\forest@all@edges{%
2879 \noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@pa@l}{\forest@getedge@pa@s}%
2880 \noexpand\pgfsyssoftpath@linetotoken{\forest@getedge@ca@l}{\forest@getedge@ca@s}%
2881 }%
2882 % this deals with potential overlap of the edges:
2883 \eappto\forest@node@edge{\noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@ca@l}{\forest@getedge@ca@s}}
2884 \fi
2885 }%
2886 \ifdefempty{\forest@all@edges}{-}{%
2887 \pgfintersectionofpaths{\pgfsetpath\forest@all@edges}{\pgfsetpath\forest@node@edge}%
2888 \def\forest@edgenode@intersections{%
2889 \forest@merge@intersectionloop
2890 \eappto\forest@node@edge{\expandonce{\forest@all@edges}\expandonce{\forest@edgenode@intersections}}%
2891 }%

Process the path into an edge and store the edge.
2892 \cscname forest@get#1\forestove{fit}edgeofpath\endcsname\forest@node@edge\forest@node@edge
2893 \forestolet{#1@edge@#2}\forest@node@edge
2894 \fi
2895 }
2896 \newloop\forest@merge@loop

```

```

2897 \def\forest@merge@intersectionloop{%
2898   \c@pgf@counta=0
2899   \forest@merge@loop
2900   \ifnum\c@pgf@counta<\pgfintersectionsolutions\relax
2901     \advance\c@pgf@counta1
2902     \pgfpointintersectionsolution{\the\c@pgf@counta}%
2903     \eappto\forest@edgenode@intersections{\noexpand\pgfsyssoftpath@movetotoken
2904       {\the\pgf@x}{\the\pgf@y}}%
2905   \forest@merge@repeat
2906 }

```

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

```

2907 \def\forest@node@getboundingrectangle@ls#1{%
2908   \forestoget{boundary}\forest@node@boundary
2909   \forest@path@getboundingrectangle@ls\forest@node@boundary{#1}%
2910 }

```

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).

```

2911 \def\forest@pgfpathtransformed#1{%
2912   \forest@save@pgfsyssoftpath@tokendefs
2913   \let\pgfsyssoftpath@movetotoken\forest@pgfpathtransformed@moveto
2914   \let\pgfsyssoftpath@linetotoken\forest@pgfpathtransformed@lineto
2915   \pgfsyssoftpath@setcurrentpath\pgfutil@empty
2916   #1%
2917   \forest@restore@pgfsyssoftpath@tokendefs
2918 }
2919 \def\forest@pgfpathtransformed@moveto#1#2{%
2920   \forest@pgfpathtransformed@op\pgfsyssoftpath@moveto{#1}{#2}%
2921 }
2922 \def\forest@pgfpathtransformed@lineto#1#2{%
2923   \forest@pgfpathtransformed@op\pgfsyssoftpath@lineto{#1}{#2}%
2924 }
2925 \def\forest@pgfpathtransformed@op#1#2#3{%
2926   \pgfpointtransformed{\pgfqpoint{#2}{#3}}%
2927   \edef\forest@temp{%
2928     \noexpand#1{\the\pgf@x}{\the\pgf@y}%
2929   }%
2930   \forest@temp
2931 }

```

### 11.2.1 Tiers

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

```

2932 \def\forest@pack@computetiers{%
2933   {%
2934     \forest@pack@tiers@getalltiersinsubtree
2935     \forest@pack@tiers@computetierhierarchy
2936     \forest@pack@tiers@findcontainers
2937     \forest@pack@tiers@raisecontainers
2938     \forest@pack@tiers@computeprocessingorder
2939     \gdef\forest@smuggle{}%
2940     \forest@pack@tiers@write
2941   }%
2942   \forest@node@foreach{\forestoset{@tiers}{}}%
2943   \forest@smuggle
2944 }

```

Puts all tiers contained in the subtree into attribute tiers.

```

2945 \def\forest@pack@tiers@getalltiersinsubtree{%
2946   \ifnum\forestove{n children}>0
2947     \forest@node@foreachchild{\forest@pack@tiers@getalltiersinsubtree}%

```

```

2948 \fi
2949 \forestoget{tier}\forest@temp@mytier
2950 \def\forest@temp@mytiers{}%
2951 \ifdefempty\forest@temp@mytier{}-%
2952 \listead\forest@temp@mytiers\forest@temp@mytier
2953 }%
2954 \ifnum\forestove{n children}>0
2955 \forest@node@foreachchild{%
2956 \forestoget{tiers}\forest@temp@tiers
2957 \forlistloop\forest@pack@tiers@forhandlerA\forest@temp@tiers
2958 }%
2959 \fi
2960 \forestolet{tiers}\forest@temp@mytiers
2961 }
2962 \def\forest@pack@tiers@forhandlerA#1{%
2963 \ifinlist{#1}\forest@temp@mytiers{}-%
2964 \listead\forest@temp@mytiers{#1}%
2965 }%
2966 }

```

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.

```

2967 \def\forest@pack@tiers@computetierhierarchy{%
2968 \def\forest@tiers@ancestors{}%
2969 \forestoget{tiers}\forest@temp@mytiers
2970 \forlistloop\forest@pack@tiers@cth@init\forest@temp@mytiers
2971 \forest@pack@tiers@computetierhierarchy@
2972 }
2973 \def\forest@pack@tiers@cth@init#1{%
2974 \csdef{forest@tiers@higher@#1}{}%
2975 \csdef{forest@tiers@lower@#1}{}%
2976 }
2977 \def\forest@pack@tiers@computetierhierarchy@{%
2978 \forestoget{tier}\forest@temp@mytier
2979 \ifdefempty\forest@temp@mytier{}-%
2980 \forlistloop\forest@pack@tiers@forhandlerB\forest@tiers@ancestors
2981 \listead\forest@tiers@ancestors\forest@temp@mytier
2982 }%
2983 \forest@node@foreachchild{%
2984 \forest@pack@tiers@computetierhierarchy@
2985 }%
2986 \forestoget{tier}\forest@temp@mytier
2987 \ifdefempty\forest@temp@mytier{}-%
2988 \forest@listedel\forest@tiers@ancestors\forest@temp@mytier
2989 }%
2990 }
2991 \def\forest@pack@tiers@forhandlerB#1{%
2992 \def\forest@temp@tier{#1}%
2993 \ifx\forest@temp@tier\forest@temp@mytier
2994 \PackageError{forest}{Circular tier hierarchy (tier \forest@temp@mytier)}{}%
2995 \fi
2996 \ifinlistcs{#1}{forest@tiers@higher@\forest@temp@mytier}{}-%
2997 \listcsadd{forest@tiers@higher@\forest@temp@mytier}{#1}%
2998 \xifinlistcs\forest@temp@mytier{forest@tiers@lower@#1}{}-%
2999 \listcseadd{forest@tiers@lower@#1}{\forest@temp@mytier}%
3000 }
3001 \def\forest@pack@tiers@findcontainers{%
3002 \forestoget{tiers}\forest@temp@tiers
3003 \forlistloop\forest@pack@tiers@findcontainer\forest@temp@tiers
3004 }
3005 \def\forest@pack@tiers@findcontainer#1{%

```

```

3006 \def\forest@temp@tier{#1}%
3007 \forestoget{tier}\forest@temp@mytier
3008 \ifx\forest@temp@tier\forest@temp@mytier
3009 \csedef{forest@tiers@container@#1}{\forest@cn}%
3010 \else\@escapeif{%
3011 \forest@pack@tiers@findcontainerA{#1}%
3012 }\fi%
3013 }
3014 \def\forest@pack@tiers@findcontainerA#1{%
3015 \c@pgf@counta=0
3016 \forest@node@foreachchild{%
3017 \forestoget{tiers}\forest@temp@tiers
3018 \ifinlist{#1}\forest@temp@tiers{%
3019 \advance\c@pgf@counta 1
3020 \let\forest@temp@child\forest@cn
3021 }{}}%
3022 }%
3023 \ifnum\c@pgf@counta>1
3024 \csedef{forest@tiers@container@#1}{\forest@cn}%
3025 \else\@escapeif{% surely =1
3026 \forest@fornode{\forest@temp@child}{%
3027 \forest@pack@tiers@findcontainer{#1}%
3028 }%
3029 }\fi
3030 }
3031 \def\forest@pack@tiers@raisecontainers{%
3032 \forestoget{tiers}\forest@temp@mytiers
3033 \forlistloop\forest@pack@tiers@rc@forhandlerA\forest@temp@mytiers
3034 }
3035 \def\forest@pack@tiers@rc@forhandlerA#1{%
3036 \edef\forest@tiers@temptier{#1}%
3037 \letcs\forest@tiers@containernodeoftier{forest@tiers@container@#1}%
3038 \letcs\forest@temp@lowertiers{forest@tiers@lower@#1}%
3039 \forlistloop\forest@pack@tiers@rc@forhandlerB\forest@temp@lowertiers
3040 }
3041 \def\forest@pack@tiers@rc@forhandlerB#1{%
3042 \letcs\forest@tiers@containernodeoflowertier{forest@tiers@container@#1}%
3043 \forest@get{\forest@tiers@containernodeoflowertier}{content}\lowercontent
3044 \forest@get{\forest@tiers@containernodeoftier}{content}\uppercontent
3045 \forest@fornode{\forest@tiers@containernodeoflowertier}{%
3046 \forest@ifancestorof
3047 {\forest@tiers@containernodeoftier}
3048 {\csletcs{forest@tiers@container@\forest@tiers@temptier}{forest@tiers@container@#1}}%
3049 }%
3050 }%
3051 }
3052 \def\forest@pack@tiers@computeprocessingorder{%
3053 \def\forest@tiers@processingorder{}%
3054 \forestoget{tiers}\forest@tiers@cpo@tierstodo
3055 \forest@loopa
3056 \ifdefempty\forest@tiers@cpo@tierstodo{\forest@tempfalse}{\forest@temptrue}%
3057 \ifforest@temp
3058 \def\forest@tiers@cpo@tiersremaining{}%
3059 \def\forest@tiers@cpo@tiersindependent{}%
3060 \forlistloop\forest@pack@tiers@cpo@forhandlerA\forest@tiers@cpo@tierstodo
3061 \ifdefempty\forest@tiers@cpo@tiersindependent{%
3062 \PackageError{forest}{Circular tiers!}{}}{}%
3063 \forlistloop\forest@pack@tiers@cpo@forhandlerB\forest@tiers@cpo@tiersremaining
3064 \let\forest@tiers@cpo@tierstodo\forest@tiers@cpo@tiersremaining
3065 \forest@repeata
3066 }

```



```

3067 \def\forest@pack@tiers@cpo@forhandlerA#1{%
3068   \ifcempty{forest@tiers@higher@#1}{%
3069     \listadd\forest@tiers@cpo@tiersindependent{#1}%
3070     \listadd\forest@tiers@processingorder{#1}%
3071   }{%
3072     \listadd\forest@tiers@cpo@tiersremaining{#1}%
3073   }%
3074 }
3075 \def\forest@pack@tiers@cpo@forhandlerB#1{%
3076   \def\forest@pack@tiers@cpo@aremainingtier{#1}%
3077   \forlistloop\forest@pack@tiers@cpo@forhandlerC\forest@tiers@cpo@tiersindependent
3078 }
3079 \def\forest@pack@tiers@cpo@forhandlerC#1{%
3080   \ifinlistcs{#1}{forest@tiers@higher@\forest@pack@tiers@cpo@aremainingtier}{%
3081     \forest@listcsdel{forest@tiers@higher@\forest@pack@tiers@cpo@aremainingtier}{#1}%
3082   }{%
3083 }
3084 \def\forest@pack@tiers@write{%
3085   \forlistloop\forest@pack@tiers@write@forhandler\forest@tiers@processingorder
3086 }
3087 \def\forest@pack@tiers@write@forhandler#1{%
3088   \forest@fornode{\csname forest@tiers@container@#1\endcsname}{%
3089     \forest@pack@tiers@check{#1}%
3090   }%
3091   \xappto\forest@smuggle{%
3092     \noexpand\listadd
3093     \forest@om{\csname forest@tiers@container@#1\endcsname}{@tiers}%
3094     {#1}%
3095   }%
3096 }
3097 % checks if the tier is compatible with growth changes and calign=node/edge angle
3098 \def\forest@pack@tiers@check#1{%
3099   \def\forest@temp@currenttier{#1}%
3100   \forest@node@foreachdescendant{%
3101     \ifnum\forest@ove{grow}=\forest@ove{\forest@ove{@parent}}{grow}
3102     \else
3103       \forest@pack@tiers@check@grow
3104     \fi
3105     \ifnum\forest@ove{n children}>1
3106       \forest@toget{calign}\forest@temp
3107       \ifx\forest@temp\forest@pack@tiers@check@nodeangle
3108         \forest@pack@tiers@check@calign
3109       \fi
3110       \ifx\forest@temp\forest@pack@tiers@check@edgeangle
3111         \forest@pack@tiers@check@calign
3112       \fi
3113     \fi
3114   }%
3115 }
3116 \def\forest@pack@tiers@check@nodeangle{node angle}%
3117 \def\forest@pack@tiers@check@edgeangle{edge angle}%
3118 \def\forest@pack@tiers@check@grow{%
3119   \forest@toget{content}\forest@temp@content
3120   \let\forest@temp@currentnode\forest@cn
3121   \forest@node@foreachdescendant{%
3122     \forest@toget{tier}\forest@temp
3123     \ifx\forest@temp@currenttier\forest@temp
3124       \forest@pack@tiers@check@grow@error
3125     \fi
3126   }%
3127 }

```

```

3128 \def\forest@pack@tiers@check@grow@error{%
3129   \PackageError{forest}{Tree growth direction changes in node \forest@temp@currentnode\space
3130     (content: \forest@temp@content), while tier '\forest@temp' is specified for nodes both
3131     out- and inside the subtree rooted in node \forest@temp@currentnode. This will not work.}{}%
3132 }
3133 \def\forest@pack@tiers@check@calign{%
3134   \forest@node@foreachchild{%
3135     \forestoget{tier}\forest@temp
3136     \ifx\forest@temp@currenttier\forest@temp
3137       \forest@pack@tiers@check@calign@warning
3138     \fi
3139   }%
3140 }
3141 \def\forest@pack@tiers@check@calign@warning{%
3142   \PackageWarning{forest}{Potential option conflict: node \forestove{@parent} (content:
3143     '\forestOve{\forestove{@parent}}{content}') was given 'calign=\forestove{calign}', while its
3144     child \forest@cn\space (content: '\forestove{content}') was given 'tier=\forestove{tier}'.
3145     The parent's 'calign' will only work if the child was the lowest node on its tier before the
3146     alignment.}{%}
3147 }

```

### 11.2.2 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.

```

3148 \pgfdeclaregenericanchor{forestcomputenodeboundary}{%
3149   \letcs\forest@temp@boundary@macro{forest@compute@node@boundary@#1}%
3150   \ifcsname forest@compute@node@boundary@#1\endcsname
3151     \csname forest@compute@node@boundary@#1\endcsname
3152   \else
3153     \forest@compute@node@boundary@rectangle
3154   \fi
3155   \pgfsyssoftpath@getcurrentpath\forest@temp
3156   \global\let\forest@global@boundary\forest@temp
3157 }
3158 \def\forest@mt#1{%
3159   \expandafter\pgfpoinanchor\expandafter{\pgfreferencednodename}{#1}%
3160   \pgfsyssoftpath@moveto{\the\pgf@x}{\the\pgf@y}%
3161 }%
3162 \def\forest@lt#1{%
3163   \expandafter\pgfpoinanchor\expandafter{\pgfreferencednodename}{#1}%
3164   \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
3165 }%
3166 \def\forest@compute@node@boundary@coordinate{%
3167   \forest@mt{center}%
3168 }
3169 \def\forest@compute@node@boundary@circle{%
3170   \forest@mt{east}%
3171   \forest@lt{north east}%
3172   \forest@lt{north}%
3173   \forest@lt{north west}%
3174   \forest@lt{west}%
3175   \forest@lt{south west}%
3176   \forest@lt{south}%
3177   \forest@lt{south east}%
3178   \forest@lt{east}%
3179 }
3180 \def\forest@compute@node@boundary@rectangle{%
3181   \forest@mt{south west}%
3182   \forest@lt{south east}%

```

```

3183 \forest@lt{north east}%
3184 \forest@lt{north west}%
3185 \forest@lt{south west}%
3186 }
3187 \def\forest@compute@node@boundary@diamond{%
3188 \forest@mt{east}%
3189 \forest@lt{north}%
3190 \forest@lt{west}%
3191 \forest@lt{south}%
3192 \forest@lt{east}%
3193 }
3194 \let\forest@compute@node@boundary@ellipse\forest@compute@node@boundary@circle
3195 \def\forest@compute@node@boundary@trapezium{%
3196 \forest@mt{top right corner}%
3197 \forest@lt{top left corner}%
3198 \forest@lt{bottom left corner}%
3199 \forest@lt{bottom right corner}%
3200 \forest@lt{top right corner}%
3201 }
3202 \def\forest@compute@node@boundary@semicircle{%
3203 \forest@mt{arc start}%
3204 \forest@lt{north}%
3205 \forest@lt{east}%
3206 \forest@lt{north east}%
3207 \forest@lt{apex}%
3208 \forest@lt{north west}%
3209 \forest@lt{west}%
3210 \forest@lt{arc end}%
3211 \forest@lt{arc start}%
3212 }
3213 \newloop\forest@computenodeboundary@loop
3214 \csdef{forest@compute@node@boundary@regular polygon}{%
3215 \forest@mt{corner 1}%
3216 \c@pgf@counta=\sides\relax
3217 \forest@computenodeboundary@loop
3218 \ifnum\c@pgf@counta>0
3219 \forest@lt{corner \the\c@pgf@counta}%
3220 \advance\c@pgf@counta-1
3221 \forest@computenodeboundary@repeat
3222 }%
3223 \def\forest@compute@node@boundary@star{%
3224 \forest@mt{outer point 1}%
3225 \c@pgf@counta=\totalstarpoints\relax
3226 \divide\c@pgf@counta2
3227 \forest@computenodeboundary@loop
3228 \ifnum\c@pgf@counta>0
3229 \forest@lt{inner point \the\c@pgf@counta}%
3230 \forest@lt{outer point \the\c@pgf@counta}%
3231 \advance\c@pgf@counta-1
3232 \forest@computenodeboundary@repeat
3233 }%
3234 \csdef{forest@compute@node@boundary@isosceles triangle}{%
3235 \forest@mt{apex}%
3236 \forest@lt{left corner}%
3237 \forest@lt{right corner}%
3238 \forest@lt{apex}%
3239 }
3240 \def\forest@compute@node@boundary@kite{%
3241 \forest@mt{upper vertex}%
3242 \forest@lt{left vertex}%
3243 \forest@lt{lower vertex}%

```

```

3244 \forest@lt{right vertex}%
3245 \forest@lt{upper vertex}%
3246 }
3247 \def\forest@compute@node@boundary@dart{%
3248 \forest@mt{tip}%
3249 \forest@lt{left tail}%
3250 \forest@lt{tail center}%
3251 \forest@lt{right tail}%
3252 \forest@lt{tip}%
3253 }
3254 \csdef{forest@compute@node@boundary@circular sector}{%
3255 \forest@mt{sector center}%
3256 \forest@lt{arc start}%
3257 \forest@lt{arc center}%
3258 \forest@lt{arc end}%
3259 \forest@lt{sector center}%
3260 }
3261 \def\forest@compute@node@boundary@cylinder{%
3262 \forest@mt{top}%
3263 \forest@lt{after top}%
3264 \forest@lt{before bottom}%
3265 \forest@lt{bottom}%
3266 \forest@lt{after bottom}%
3267 \forest@lt{before top}%
3268 \forest@lt{top}%
3269 }
3270 \cslet{forest@compute@node@boundary@forbidden sign}\forest@compute@node@boundary@circle
3271 \cslet{forest@compute@node@boundary@magnifying glass}\forest@compute@node@boundary@circle
3272 \def\forest@compute@node@boundary@cloud{%
3273 \getradii
3274 \forest@mt{puff 1}%
3275 \c@pgf@counta=\puffs\relax
3276 \forest@computenodeboundary@loop
3277 \ifnum\c@pgf@counta>0
3278 \forest@lt{puff \the\c@pgf@counta}%
3279 \advance\c@pgf@counta-1
3280 \forest@computenodeboundary@repeat
3281 }
3282 \def\forest@compute@node@boundary@starburst{
3283 \calculatestarburstpoints
3284 \forest@mt{outer point 1}%
3285 \c@pgf@counta=\totalpoints\relax
3286 \divide\c@pgf@counta2
3287 \forest@computenodeboundary@loop
3288 \ifnum\c@pgf@counta>0
3289 \forest@lt{inner point \the\c@pgf@counta}%
3290 \forest@lt{outer point \the\c@pgf@counta}%
3291 \advance\c@pgf@counta-1
3292 \forest@computenodeboundary@repeat
3293 }%
3294 \def\forest@compute@node@boundary@signal{%
3295 \forest@mt{east}%
3296 \forest@lt{south east}%
3297 \forest@lt{south west}%
3298 \forest@lt{west}%
3299 \forest@lt{north west}%
3300 \forest@lt{north east}%
3301 \forest@lt{east}%
3302 }
3303 \def\forest@compute@node@boundary@tape{%
3304 \forest@mt{north east}%

```

```

3305 \forest@lt{60}%
3306 \forest@lt{north}%
3307 \forest@lt{120}%
3308 \forest@lt{north west}%
3309 \forest@lt{south west}%
3310 \forest@lt{240}%
3311 \forest@lt{south}%
3312 \forest@lt{310}%
3313 \forest@lt{south east}%
3314 \forest@lt{north east}%
3315 }
3316 \csdef{forest@compute@node@boundary@single arrow}{%
3317 \forest@mt{tip}%
3318 \forest@lt{after tip}%
3319 \forest@lt{after head}%
3320 \forest@lt{before tail}%
3321 \forest@lt{after tail}%
3322 \forest@lt{before head}%
3323 \forest@lt{before tip}%
3324 \forest@lt{tip}%
3325 }
3326 \csdef{forest@compute@node@boundary@double arrow}{%
3327 \forest@mt{tip 1}%
3328 \forest@lt{after tip 1}%
3329 \forest@lt{after head 1}%
3330 \forest@lt{before head 2}%
3331 \forest@lt{before tip 2}%
3332 \forest@mt{tip 2}%
3333 \forest@lt{after tip 2}%
3334 \forest@lt{after head 2}%
3335 \forest@lt{before head 1}%
3336 \forest@lt{before tip 1}%
3337 \forest@lt{tip 1}%
3338 }
3339 \csdef{forest@compute@node@boundary@arrow box}{%
3340 \forest@mt{before north arrow}%
3341 \forest@lt{before north arrow head}%
3342 \forest@lt{before north arrow tip}%
3343 \forest@lt{north arrow tip}%
3344 \forest@lt{after north arrow tip}%
3345 \forest@lt{after north arrow head}%
3346 \forest@lt{after north arrow}%
3347 \forest@lt{north east}%
3348 \forest@lt{before east arrow}%
3349 \forest@lt{before east arrow head}%
3350 \forest@lt{before east arrow tip}%
3351 \forest@lt{east arrow tip}%
3352 \forest@lt{after east arrow tip}%
3353 \forest@lt{after east arrow head}%
3354 \forest@lt{after east arrow}%
3355 \forest@lt{south east}%
3356 \forest@lt{before south arrow}%
3357 \forest@lt{before south arrow head}%
3358 \forest@lt{before south arrow tip}%
3359 \forest@lt{south arrow tip}%
3360 \forest@lt{after south arrow tip}%
3361 \forest@lt{after south arrow head}%
3362 \forest@lt{after south arrow}%
3363 \forest@lt{south west}%
3364 \forest@lt{before west arrow}%
3365 \forest@lt{before west arrow head}%

```

```

3366 \forest@lt{before west arrow tip}%
3367 \forest@lt{west arrow tip}%
3368 \forest@lt{after west arrow tip}%
3369 \forest@lt{after west arrow head}%
3370 \forest@lt{after west arrow}%
3371 \forest@lt{north west}%
3372 \forest@lt{before north arrow}%
3373 }
3374 \cslet{forest@compute@node@boundary@circle split}\forest@compute@node@boundary@circle
3375 \cslet{forest@compute@node@boundary@circle solidus}\forest@compute@node@boundary@circle
3376 \cslet{forest@compute@node@boundary@ellipse split}\forest@compute@node@boundary@ellipse
3377 \cslet{forest@compute@node@boundary@rectangle split}\forest@compute@node@boundary@rectangle
3378 \def\forest@compute@node@boundary@@callout{%
3379 \beforecalloutpointer
3380 \pgfsyssoftpath@moveto{\the\pgf@x}{\the\pgf@y}%
3381 \calloutpointeranchor
3382 \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
3383 \aftercalloutpointer
3384 \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
3385 }
3386 \csdef{forest@compute@node@boundary@rectangle callout}{%
3387 \forest@compute@node@boundary@rectangle
3388 \rectanglecalloutpoints
3389 \forest@compute@node@boundary@@callout
3390 }
3391 \csdef{forest@compute@node@boundary@ellipse callout}{%
3392 \forest@compute@node@boundary@ellipse
3393 \ellipsecalloutpoints
3394 \forest@compute@node@boundary@@callout
3395 }
3396 \csdef{forest@compute@node@boundary@cloud callout}{%
3397 \forest@compute@node@boundary@cloud
3398 % at least a first approx...
3399 \forest@mt{center}%
3400 \forest@lt{pointer}%
3401 }%
3402 \csdef{forest@compute@node@boundary@cross out}{%
3403 \forest@mt{south east}%
3404 \forest@lt{north west}%
3405 \forest@mt{south west}%
3406 \forest@lt{north east}%
3407 }%
3408 \csdef{forest@compute@node@boundary@strike out}{%
3409 \forest@mt{north east}%
3410 \forest@lt{south west}%
3411 }%
3412 \csdef{forest@compute@node@boundary@rounded rectangle}{%
3413 \forest@mt{east}%
3414 \forest@lt{north east}%
3415 \forest@lt{north}%
3416 \forest@lt{north west}%
3417 \forest@lt{west}%
3418 \forest@lt{south west}%
3419 \forest@lt{south}%
3420 \forest@lt{south east}%
3421 \forest@lt{east}%
3422 }%
3423 \csdef{forest@compute@node@boundary@chamfered rectangle}{%
3424 \forest@mt{before south west}%
3425 \forest@mt{after south west}%
3426 \forest@lt{before south east}%

```

```

3427 \forest@lt{after south east}%
3428 \forest@lt{before north east}%
3429 \forest@lt{after north east}%
3430 \forest@lt{before north west}%
3431 \forest@lt{after north west}%
3432 \forest@lt{before south west}%
3433 }%

```

### 11.3 Compute absolute positions

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

```

3434 \def\forest@node@computeabsolutepositions{%
3435   \forestoset{x}{0pt}%
3436   \forestoset{y}{0pt}%
3437   \edef\forest@marshal{%
3438     \noexpand\forest@node@foreachchild{%
3439       \noexpand\forest@node@computeabsolutepositions@{0pt}{0pt}{\forestove{grow}}%
3440     }%
3441   }\forest@marshal
3442 }
3443 \def\forest@node@computeabsolutepositions@#1#2#3{%
3444   \pgfpointadd{\pgfpoint{#1}{#2}}{%
3445     \pgfpointadd{\pgfpolar{#3}{\forestove{1}}}{\pgfpolar{90 + #3}{\forestove{s}}}%
3446   \pgfgetlastxy\forest@temp@x\forest@temp@y
3447   \forestolet{x}\forest@temp@x
3448   \forestolet{y}\forest@temp@y
3449   \edef\forest@marshal{%
3450     \noexpand\forest@node@foreachchild{%
3451       \noexpand\forest@node@computeabsolutepositions@{\forest@temp@x}{\forest@temp@y}{\forestove{grow}}%
3452     }%
3453   }\forest@marshal
3454 }

```

### 11.4 Drawing the tree

```

3455 \newif\ifforest@drawtree@preservenodeboxes@
3456 \def\forest@node@drawtree{%
3457   \expandafter\ifstrequal\expandafter{\forest@drawtreebox}{\pgfkeysnovalue}{%
3458     \let\forest@drawtree@beginbox\relax
3459     \let\forest@drawtree@endbox\relax
3460   }{%
3461     \edef\forest@drawtree@beginbox{\global\setbox\forest@drawtreebox=\hbox\bgroup}%
3462     \let\forest@drawtree@endbox\egroup
3463   }%
3464   \ifforest@external@
3465     \ifforest@externalize@tree@
3466     \forest@temptrue
3467   \else
3468     \tikzifexternalizing{%
3469       \ifforest@was@tikzexternalwasenable
3470         \forest@temptrue
3471         \pgfkeys{/tikz/external/optimize=false}%
3472         \let\forest@drawtree@beginbox\relax
3473         \let\forest@drawtree@endbox\relax
3474       \else
3475         \forest@tempfalse
3476       \fi
3477     }{%
3478       \forest@tempfalse

```

```

3479     }%
3480 \fi
3481 \ifforest@temp
3482   \advance\forest@externalize@inner@n 1
3483   \edef\forest@externalize@filename{%
3484     \tikzexternalrealjob-forest-\forest@externalize@outer@n
3485     \ifnum\forest@externalize@inner@n=0 \else.\the\forest@externalize@inner@n\fi}%
3486   \expandafter\tikzsetnextfilename\expandafter{\forest@externalize@filename}%
3487   \tikzexternalenable
3488   \pgfkeysalso{/tikz/external/remake next,/tikz/external/export next}%
3489 \fi
3490 \ifforest@externalize@tree@
3491   \typeout{forest: Invoking a recursive call to generate the external picture
3492     '\forest@externalize@filename' for the following context+code:
3493     '\expandafter\detokenize\expandafter{\forest@externalize@id}'}%
3494 \fi
3495 \fi
3496 %
3497 \ifforesttikzcshack
3498   \let\forest@original@tikz@parse@node\tikz@parse@node
3499   \let\tikz@parse@node\forest@tikz@parse@node
3500 \fi
3501 \pgfkeysgetvalue{/forest/begin draw/.@cmd}\forest@temp@begindraw
3502 \pgfkeysgetvalue{/forest/end draw/.@cmd}\forest@temp@enddraw
3503 \edef\forest@marshal{%
3504   \noexpand\forest@drawtree@beginbox
3505   \expandonce{\forest@temp@begindraw\pgfkeysnovalue\pgfeov}%
3506   \noexpand\forest@node@drawtree@
3507   \expandonce{\forest@temp@enddraw\pgfkeysnovalue\pgfeov}%
3508   \noexpand\forest@drawtree@endbox
3509 }
3510 \ifforesttikzcshack
3511   \let\tikz@parse@node\forest@original@tikz@parse@node
3512 \fi
3513 %
3514 \ifforest@external@
3515   \ifforest@externalize@tree@
3516     \tikzexternaldisable
3517     \eappto\forest@externalize@checkimages{%
3518       \noexpand\forest@includeexternal@check{\forest@externalize@filename}%
3519     }%
3520     \expandafter\ifstrequal\expandafter{\forest@drawtreebox}{\pgfkeysnovalue}{%
3521       \eappto\forest@externalize@loadimages{%
3522         \noexpand\forest@includeexternal{\forest@externalize@filename}%
3523       }%
3524     }{%
3525       \eappto\forest@externalize@loadimages{%
3526         \noexpand\forest@includeexternal@box\forest@drawtreebox{\forest@externalize@filename}%
3527       }%
3528     }%
3529 \fi
3530 \fi
3531 }
3532 \def\forest@node@drawtree@{%
3533 \forest@node@foreach{\forest@draw@node}%
3534 \forest@node@ifnamedefined{forest@baseline@node}{%
3535 \edef\forest@temp{%
3536 \noexpand\pgfsetbaselinepointlater{%
3537 \noexpand\pgfpoinanchor
3538 {\forestOve{\forest@node@Nametoid{forest@baseline@node}}{name}}
3539 {\forestOve{\forest@node@Nametoid{forest@baseline@node}}{anchor}}

```



```

3540     }%
3541   }\forest@temp
3542 }{}%
3543 \forest@node@foreachdescendant{\forest@draw@edge}%
3544 \forest@node@foreach{\forest@draw@tikz}%
3545 }
3546 \def\forest@draw@node{%
3547   \ifnum\forestove{phantom}=0
3548     \forest@node@forest@positionnodelater@restore
3549     \ifforest@drawtree@preservenodeboxes@
3550       \pgfnodealias{forest@temp}{\forestove{later@name}}%
3551     \fi
3552     \pgfpositionnodenow{\pgfqpoint{\forestove{x}}{\forestove{y}}}%
3553     \ifforest@drawtree@preservenodeboxes@
3554       \pgfnodealias{\forestove{later@name}}{forest@temp}%
3555     \fi
3556   \fi
3557 }
3558 \def\forest@draw@edge{%
3559   \ifnum\forestove{phantom}=0
3560     \ifnum\forestOve{\forestove{parent}}{phantom}=0
3561       \edef\forest@temp{\forestove{edge path}}%
3562     \forest@temp
3563   \fi
3564   \fi
3565 }
3566 \def\forest@draw@tikz{%
3567   \forestove{tikz}%
3568 }

```

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

```

3569 \def\forest@tikz@parse@node#1(#2){%
3570   \pgfutil@in@.#2}%
3571   \ifpgfutil@in@
3572     \expandafter\forest@tikz@parse@node@checkiftikzname@withdot
3573   \else%
3574     \expandafter\forest@tikz@parse@node@checkiftikzname@withoutdot
3575   \fi%
3576   #1(#2)\forest@end
3577 }
3578 \def\forest@tikz@parse@node@checkiftikzname@withdot#1(#2.#3)\forest@end{%
3579   \forest@tikz@parse@node@checkiftikzname#1{#2}{.#3}}
3580 \def\forest@tikz@parse@node@checkiftikzname@withoutdot#1(#2)\forest@end{%
3581   \forest@tikz@parse@node@checkiftikzname#1{#2}{}}
3582 \def\forest@tikz@parse@node@checkiftikzname#1#2#3{%
3583   \expandafter\ifx\csname pgf@sh@ns@#2\endcsname\relax
3584     \forest@forthis{%
3585       \forest@nameandgo{#2}%
3586       \edef\forest@temp@relativenodename{\forestove{name}}%
3587     }%
3588   \else
3589     \def\forest@temp@relativenodename{#2}%
3590   \fi
3591   \expandafter\forest@original@tikz@parse@node\expandafter#1\expandafter(\forest@temp@relativenodename#3)%
3592 }
3593 \def\forest@nameandgo#1{%
3594   \pgfutil@in@!{#1}%
3595   \ifpgfutil@in@
3596     \forest@nameandgo@(#1)%
3597   \else
3598     \ifstrempy{#1}{-}{\edef\forest@cn{\forest@node@Nametoid{#1}}}%

```

```

3599 \fi
3600 }
3601 \def\forest@nameandgo@(#1!#2){%
3602 \ifstrempy{#1}{\edef\forest@cn{\forest@node@Nametoid{#1}}}%
3603 \forest@go{#2}%
3604 }

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.
3605 \pgfdeclaregenericanchor{parent anchor}{%
3606 \forest@generic@parent@child@anchor{parent }{#1}}
3607 \pgfdeclaregenericanchor{child anchor}{%
3608 \forest@generic@parent@child@anchor{child }{#1}}
3609 \pgfdeclaregenericanchor{anchor}{%
3610 \forest@generic@parent@child@anchor}{#1}}
3611 \def\forest@generic@parent@child@anchor#1#2{%
3612 \forest@get{\forest@node@Nametoid{\pgfreferencednodename}}{#1anchor}\forest@temp@parent@anchor
3613 \ifdefempty\forest@temp@parent@anchor{%
3614 \pgf@sh@reanchor{#2}{center}%
3615 \xdef\forest@hack@tikzshapeborder{%
3616 \noexpand\tikz@shapebordertrue
3617 \def\noexpand\tikz@shapeborder@name{\pgfreferencednodename}%
3618 }\aftergroup\forest@hack@tikzshapeborder
3619 }{%
3620 \pgf@sh@reanchor{#2}{\forest@temp@parent@anchor}%
3621 }%
3622 }

```

## 12 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.

```

3623 \newdimen\forest@xg
3624 \newdimen\forest@yg
3625 \newdimen\forest@xs
3626 \newdimen\forest@ys
3627 \def\forest@setupgrowline#1{%
3628 \edef\forest@grow{#1}%
3629 \pgfpointpolar\forest@grow{1pt}%
3630 \forest@xg=\pgf@x
3631 \forest@yg=\pgf@y
3632 \forest@xs=-\pgf@y
3633 \forest@ys=\pgf@x
3634 }

```

### 12.1 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`.

```

3635 \def\forest@pgfpointprojectiontogrowline#1{%
3636 \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.

```

3637 \pgfutil@tempdima=\pgf@sys@tonumber{\pgf@x}\forest@xg%
3638 \advance\pgfutil@tempdima by\pgf@sys@tonumber{\pgf@y}\forest@yg%

```

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

```

3639 \global\pgf@x=\pgf@sys@tonumber{\pgfutil@tempdima}\forest@xg%

```

```

3640 \global\pgf@y=\pgf@sys@tonumber{\pgfutil@tempdima}\forest@yg%
3641 }}

```

The following macro calculates the distance of point #2 to the grow line and stores the result in T<sub>E</sub>X-dimension #1. The distance is the scalar product of the point vector and the normalized vector orthogonal to the grow line.

```

3642 \def\forest@distancetogrowline#1#2{%
3643 \pgf@process{#2}%
3644 #1=\pgf@sys@tonumber{\pgf@x}\forest@xs\relax
3645 \advance#1 by\pgf@sys@tonumber{\pgf@y}\forest@ys\relax
3646 }

```

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>20</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`.

```

3647 \let\forest@pp@n\relax
3648 \def\forest@projectpath@tobrowline#1#2{%
3649 \edef\forest@pp@prefix{#2}%
3650 \forest@save@pgfsyssoftpath@token@defs
3651 \let\pgfsyssoftpath@movetotoken\forest@projectpath@processpoint
3652 \let\pgfsyssoftpath@linetotoken\forest@projectpath@processpoint
3653 \c@pgf@counta=0
3654 #1%
3655 \csedef{#2n}{\the\c@pgf@counta}%
3656 \forest@restore@pgfsyssoftpath@token@defs
3657 }

```

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

```

3658 \def\forest@projectpath@processpoint#1#2{%
3659 \pgfqpoint{#1}{#2}%
3660 \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta xo\endcsname{\the\pgf@x}%
3661 \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta yo\endcsname{\the\pgf@y}%
3662 \forest@pgfp@point@projection@tobrowline{}%
3663 \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta xp\endcsname{\the\pgf@x}%
3664 \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta yp\endcsname{\the\pgf@y}%
3665 \advance\c@pgf@counta 1\relax
3666 }

```

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

```

3667 \def\forest@sort@projections#1{%
3668 % todo: optimize in cases when we know that the array is actually a
3669 % merger of sorted arrays; when does this happen? in
3670 % distance_between_paths, and when merging the edges of the parent
3671 % and its children in a uniform growth tree
3672 \edef\forest@ppi@inputprefix{#1}%
3673 \c@pgf@counta=\csname#1n\endcsname\relax
3674 \advance\c@pgf@counta -1
3675 \forest@sort\forest@ppiraw@cmp\forest@ppiraw@let\forest@sort@ascending{0}{\the\c@pgf@counta}%
3676 }

```

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

<sup>20</sup>A path is *simple* if it consists of only move-to and line-to operations.

- Array of projections (prefix #2)
  - 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>21</sup>

```
3677 \def\forest@processprojectioninfo#1#2{%
3678 \edef\forest@ppi@inputprefix{#1}%
```

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

```
3679 \c@pgf@counta=0
3680 \c@pgf@countb=-1
3681 \loop
3682 \ifnum\c@pgf@counta<\csname#1n\endcsname\relax
```

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

```
3683 \letcs\forest@xo{#1\the\c@pgf@counta xo}%
3684 \letcs\forest@yo{#1\the\c@pgf@counta yo}%
3685 \letcs\forest@xp{#1\the\c@pgf@counta xp}%
3686 \letcs\forest@yp{#1\the\c@pgf@counta yp}%
3687 \ifnum\c@pgf@countb<0
3688 \forest@equaltotolerancefalse
3689 \else
3690 \forest@equaltotolerance
3691 {\pgfpoint\forest@xp\forest@yp}%
3692 {\pgfpoint
3693 {\csname#2\the\c@pgf@countb x\endcsname}%
3694 {\csname#2\the\c@pgf@countb y\endcsname}%
3695 }%
3696 \fi
3697 \ifforest@equaltotolerance\else
```

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

```
3698 \advance\c@pgf@countb 1
3699 \cslet{#2\the\c@pgf@countb x}\forest@xp
3700 \cslet{#2\the\c@pgf@countb y}\forest@yp
3701 \csdef{#2\the\c@pgf@countb @n}{0}%
3702 \fi
```

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

```
3703 % todo: this is ugly!
3704 \ifdefined\forest@xo\ifx\forest@xo\relax\else
3705 \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.

```
3706 \forest@append@point@to@inner@array
3707 \forest@xo\forest@yo
3708 {#2\the\c@pgf@countb @}%
```

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

```
3709 \csdef{#2(\forest@xo,\forest@yo)}{\the\c@pgf@countb}%
3710 \fi\fi
3711 \fi\fi
```

---

<sup>21</sup>At first sight, this information could be cached “at the source”: by `forest@pgfpointprojectiontogrowline`. 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.

Clean-up the raw array item.

```
3712 \cslet{#1\the\c@pgf@counta xo}\relax
3713 \cslet{#1\the\c@pgf@counta yo}\relax
3714 \cslet{#1\the\c@pgf@counta xp}\relax
3715 \cslet{#1\the\c@pgf@counta yp}\relax
3716 \advance\c@pgf@counta 1
3717 \repeat
```

Clean up the raw array length.

```
3718 \cslet{#1n}\relax
```

Store the length of the outer result array.

```
3719 \advance\c@pgf@countb 1
3720 \csedef{#2n}{\the\c@pgf@countb}%
3721 }
```

Item-exchange macro for quicksorting the raw projection data. (#1 is copied into #2.)

```
3722 \def\forest@ppiraw@let#1#2{%
3723 \csletcs{\forest@ppi@inputprefix#1xo}{\forest@ppi@inputprefix#2xo}%
3724 \csletcs{\forest@ppi@inputprefix#1yo}{\forest@ppi@inputprefix#2yo}%
3725 \csletcs{\forest@ppi@inputprefix#1xp}{\forest@ppi@inputprefix#2xp}%
3726 \csletcs{\forest@ppi@inputprefix#1yp}{\forest@ppi@inputprefix#2yp}%
3727 }
```

Item comparison macro for quicksorting the raw projection data.

```
3728 \def\forest@ppiraw@cmp#1#2{%
3729 \forest@sort@cmtwodimcs
3730 {\forest@ppi@inputprefix#1xp}{\forest@ppi@inputprefix#1yp}%
3731 {\forest@ppi@inputprefix#2xp}{\forest@ppi@inputprefix#2yp}%
3732 }
```

Append the point (#1,#2) to the (inner) array of points (prefix #3).

```
3733 \def\forest@append@point@to@inner@array#1#2#3{%
3734 \c@pgf@countc=\csname#3n\endcsname\relax
3735 \csedef{#3\the\c@pgf@countc x}{#1}%
3736 \csedef{#3\the\c@pgf@countc y}{#2}%
3737 \forest@distancetogrowline\pgfutil@tempdima{\pgfqpoint#1#2}%
3738 \csedef{#3\the\c@pgf@countc d}{\the\pgfutil@tempdima}%
3739 \advance\c@pgf@countc 1
3740 \csedef{#3n}{\the\c@pgf@countc}%
3741 }
```

## 12.2 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.

```
3742 \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.

```
3743 \edef\forest@bp@prefix{#2}%
3744 \forest@save@pgfsyssoftpath@tokendefs
3745 \let\pgfsyssoftpath@movetotoken\forest@breakpath@processfirstpoint
3746 \let\pgfsyssoftpath@linetotoken\forest@breakpath@processfirstpoint
3747 %\pgfusepath{}% empty the current path. ok?
3748 #1%
```

```

3749 \forest@restore@pgfsyssoftpath@tokendefs
3750 \pgfsyssoftpath@getcurrentpath#3%
3751 }

```

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.)

```

3752 \def\forest@breakpath@processfirstpoint#1#2{%
3753 \forest@breakpath@processmoveto{#1}{#2}%
3754 \let\pgfsyssoftpath@movetotoken\forest@breakpath@processmoveto
3755 \let\pgfsyssoftpath@linetotoken\forest@breakpath@processlineto
3756 }

```

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.

```

3757 \def\forest@breakpath@processmoveto#1#2{%
3758 \pgfsyssoftpath@moveto{#1}{#2}%
3759 \def\forest@previous@x{#1}%
3760 \def\forest@previous@y{#2}%
3761 \expandafter\let\expandafter\forest@previous@i
3762 \csname\forest@bp@prefix(#1,#2)\endcsname
3763 \expandafter\let\expandafter\forest@previous@px
3764 \csname\forest@bp@prefix\forest@previous@i x\endcsname
3765 \expandafter\let\expandafter\forest@previous@py
3766 \csname\forest@bp@prefix\forest@previous@i y\endcsname
3767 }

```

This is the heart of the path-breaking procedure.

```

3768 \def\forest@breakpath@processlineto#1#2{%

```

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

```

3769 \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.)

```

3770 \expandafter\let\expandafter\forest@i
3771 \csname\forest@bp@prefix(#1,#2)\endcsname
3772 \expandafter\let\expandafter\forest@px
3773 \csname\forest@bp@prefix\forest@i x\endcsname
3774 \expandafter\let\expandafter\forest@py
3775 \csname\forest@bp@prefix\forest@i y\endcsname

```

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

```

3776 \forest@equaltotolerance
3777 {\pgfqpoint{\forest@previous@px}{\forest@previous@py}}%
3778 {\pgfqpoint{\forest@px}{\forest@py}}%
3779 \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.

```

3780 \let\forest@breakpath@op\pgfsyssoftpath@moveto
3781 \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.

```

3782 \forest@temp@count=\forest@previous@i\relax
3783 \ifnum\forest@previous@i<\forest@i\relax
3784 \def\forest@breakpath@step{1}%
3785 \def\forest@breakpath@test{\forest@temp@count<\forest@i\relax}%
3786 \else
3787 \def\forest@breakpath@step{-1}%
3788 \def\forest@breakpath@test{\forest@temp@count>\forest@i\relax}%
3789 \fi

```

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

```

3790   \loop
3791     \advance\forest@temp@count\forest@breakpath@step\relax
3792     \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.)
3793   \pgfpointintersectionoflines
3794     {\pgfpoint
3795       {\csname\forest@bp@prefix\the\forest@temp@count x\endcsname}%
3796       {\csname\forest@bp@prefix\the\forest@temp@count y\endcsname}%
3797     }%
3798   {\pgfpointadd
3799     {\pgfpoint
3800       {\csname\forest@bp@prefix\the\forest@temp@count x\endcsname}%
3801       {\csname\forest@bp@prefix\the\forest@temp@count y\endcsname}%
3802     }%
3803     {\pgfpoint{\forest@xs}{\forest@ys}}%
3804   }%
3805   {\pgfpoint{\forest@previous@x}{\forest@previous@y}}%
3806   {\pgfpoint{#1}{#2}}%

Break the segment at the intersection.
3807   \pgfgetlastxy\forest@last@x\forest@last@y
3808   \pgfsyssoftpath@lineto\forest@last@x\forest@last@y

Append the breaking point to the inner array for the projection.
3809   \forest@append@point@to@inner@array
3810   \forest@last@x\forest@last@y
3811   {\forest@bp@prefix\the\forest@temp@count @}%

Cache the projection of the new segment edge.
3812   \csedef{\forest@bp@prefix(\the\pgf@x,\the\pgf@y)}{\the\forest@temp@count}%
3813   \repeat
3814   \fi

Add the current point.
3815   \forest@breakpath@op{#1}{#2}%

Setup new “previous” info: the segment edge, its projection’s index, and the projection.
3816   \def\forest@previous@x{#1}%
3817   \def\forest@previous@y{#2}%
3818   \let\forest@previous@i\forest@i
3819   \let\forest@previous@px\forest@px
3820   \let\forest@previous@py\forest@py
3821 }

```

### 12.3 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>22</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.

<sup>22</sup>For the definition of negative/positive side, see `forest@distancetogrowline` in §12.1

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.

```

3822 \def\forest@getnegativetightedgeofpath#1#2{%
3823   \forest@get@onetightedgeofpath#1\forest@sort@ascending#2}
3824 \def\forest@getpositivetightedgeofpath#1#2{%
3825   \forest@get@onetightedgeofpath#1\forest@sort@descending#2}
3826 \def\forest@get@onetightedgeofpath#1#2#3{%
3827   {%
3828     \forest@get@one@tightedgeofpath#1#2\forest@gep@edge
3829     \global\let\forest@gep@global@edge\forest@gep@edge
3830   }%
3831   \let#3\forest@gep@global@edge
3832 }
3833 \def\forest@get@one@tightedgeofpath#1#2#3{%

```

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

```

3834 \forest@projectpathtogrowline#1{\forest@pp@}%
3835 \forest@sortprojections{\forest@pp@}%
3836 \forest@processprojectioninfo{\forest@pp@}{\forest@pi@}%

```

Break the path.

```

3837 \forest@breakpath#1{\forest@pi@}\forest@brokenpath

```

Compile some more useful information.

```

3838 \forest@sort@inner@arrays{\forest@pi@}#2%
3839 \forest@pathtodict\forest@brokenpath{\forest@pi@}%

```

The auxiliary data is set up: do the work!

```

3840 \forest@gettightedgeofpath@getedge
3841 \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).

```

3842 \forest@simplifypath\forest@edge#3%
3843 }

```

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

```

3844 \def\forest@getbothtightedgesofpath#1#2#3{%
3845   {%
3846     \forest@get@one@tightedgeofpath#1\forest@sort@ascending\forest@gep@firstedge

```

Reverse the order of items in the inner arrays.

```

3847   \c@pgf@counta=0
3848   \loop
3849   \ifnum\c@pgf@counta<\forest@pi@n\relax
3850     \forest@ppi@deflet{\forest@pi@\the\c@pgf@counta @}%
3851     \forest@reversearray\forest@ppi@let
3852     {0}%
3853     {\csname forest@pi@\the\c@pgf@counta @n\endcsname}%
3854     \advance\c@pgf@counta 1
3855   \repeat

```

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

```

3856   \forest@gettightedgeofpath@getedge
3857   \pgfsyssoftpath@getcurrentpath\forest@edge
3858   \forest@simplifypath\forest@edge\forest@gep@secondedge

```

Smuggle the results out of the enclosing  $\TeX$  group.

```

3859   \global\let\forest@gep@global@firstedge\forest@gep@firstedge
3860   \global\let\forest@gep@global@secondedge\forest@gep@secondedge
3861 }%
3862 \let#2\forest@gep@global@firstedge

```



```
3863 \let#3\forest@gep@global@secondedge
3864 }
```

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

```
3865 \def\forest@sort@inner@arrays#1#2{%
3866   \c@pgf@counta=0
3867   \forest@loopa
3868   \ifnum\c@pgf@counta<\csname#1n\endcsname
3869     \c@pgf@countb=\csname#1\the\c@pgf@counta @n\endcsname\relax
3870     \ifnum\c@pgf@countb>1
3871       \advance\c@pgf@countb -1
3872       \forest@ppi@deflet{#1\the\c@pgf@counta @}%
3873       \forest@ppi@defcmp{#1\the\c@pgf@counta @}%
3874       \forest@sort\forest@ppi@cmp\forest@ppi@let#2{0}{\the\c@pgf@countb}%
3875     \fi
3876     \advance\c@pgf@counta 1
3877   \forest@repeata
3878 }
```

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.

```
3879 \def\forest@ppi@deflet#1{%
3880   \edef\forest@ppi@let##1##2{%
3881     \noexpand\csletcs{#1##1x}{#1##2x}%
3882     \noexpand\csletcs{#1##1y}{#1##2y}%
3883     \noexpand\csletcs{#1##1d}{#1##2d}%
3884   }%
3885 }
```

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.

```
3886 \def\forest@ppi@defcmp#1{%
3887   \edef\forest@ppi@cmp##1##2{%
3888     \noexpand\forest@sort@cmpdimcs{#1##1d}{#1##2d}%
3889   }%
3890 }
```

Put path segments into a “segment dictionary”: for each segment of the path from  $(x_1, y_1)$  to  $(x_2, y_2)$

let \forest@(x1,y1)--(x2,y2) be \forest@inpath (which can be anything but \relax).

```
3891 \let\forest@inpath\advance
```

This macro is just a wrapper to process the path.

```
3892 \def\forest@pathtodict#1#2{%
3893   \edef\forest@pathtodict@prefix{#2}%
3894   \forest@save@pgfsyssoftpath@tokendefs
3895   \let\pgfsyssoftpath@movetotoken\forest@pathtodict@movetoop
3896   \let\pgfsyssoftpath@linetotoken\forest@pathtodict@linetoop
3897   \def\forest@pathtodict@subpathstart{ }%
3898   #1%
3899   \forest@restore@pgfsyssoftpath@tokendefs
3900 }
```

When a move-to operation is encountered:

```
3901 \def\forest@pathtodict@movetoop#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.

```
3902   \def\forest@pathtodict@subpathstart{(#1,#2)-}%
3903 }
```

When a line-to operation is encountered:

```
3904 \def\forest@pathtodict@linetoop#1#2{%
```

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

```
3905 \if\relax\forest@pathtodict@subpathstart\relax\else
3906   \let\forest@pathtodict@from\forest@pathtodict@subpathstart
3907   \fi
```

Mark the segment as existing.

```
3908 \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.

```
3909 \def\forest@pathtodict@from{({#1,#2})-}%
3910 \def\forest@pathtodict@subpathstart{}%
3911 }
```

In this macro, the edge is actually computed.

```
3912 \def\forest@gettightedgeofpath@getedge{%
```

Clear the path and the last projection.

```
3913 \pgfsyssoftpath@setcurrentpath\pgfutil@empty
3914 \let\forest@last@x\relax
3915 \let\forest@last@y\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.)

```
3916 \c@pgf@counta=0
3917 \forest@temp@count=\forest@pi@n\relax
3918 \advance\forest@temp@count -1
3919 \edef\forest@nminusone{\the\forest@temp@count}%
3920 \forest@loopa
3921 \ifnum\c@pgf@counta<\forest@nminusone\relax
3922   \forest@gettightedgeofpath@getedge@loopa
3923 \forest@repeata
```

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

```
3924 \ifnum\forest@nminusone<\forest@n\relax\else
3925   \ifnum\csname forest@pi@\forest@nminusone @n\endcsname>0
3926     \forest@gettightedgeofpath@maybemoveto{\forest@nminusone}{0}%
3927   \fi
3928 \fi
3929 }
```

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.

```
3930 \def\forest@gettightedgeofpath@getedge@loopa{%
3931   \ifnum\csname forest@pi@\the\c@pgf@counta @n\endcsname>0
```

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

```
3932   \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.

```
3933   \c@pgf@countb=0
3934   \forest@loopb
3935   \ifnum\c@pgf@countb<\csname forest@pi@\the\c@pgf@counta @n\endcsname\relax
3936     \forest@gettightedgeofpath@getedge@loopb
3937   \forest@repeatb
3938 \fi
3939 \advance\c@pgf@counta 1
3940 }
```

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

```

3941 \def\forest@gettightedgeofpath@getedge@loopb{%
3942     \c@pgf@countc=0
3943     \advance\c@pgf@counta 1
3944     \edef\forest@aplusone{\the\c@pgf@counta}%
3945     \advance\c@pgf@counta -1
3946     \forest@loopc
3947     \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!

```

3948     \forest@tempfalse
3949     \expandafter\ifx\csname forest@pi@(%
3950         \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb x\endcsname,%
3951         \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb y\endcsname)--(%
3952         \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname,%
3953         \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname)%
3954     \endcsname\forest@inpath
3955     \forest@temptrue
3956     \else
3957     \expandafter\ifx\csname forest@pi@(%
3958         \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname,%
3959         \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname)--(%
3960         \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb x\endcsname,%
3961         \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb y\endcsname)%
3962     \endcsname\forest@inpath
3963     \forest@temptrue
3964     \fi
3965     \fi
3966     \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.

```

3967     \forest@gettightedgeofpath@maybemoveto{\the\c@pgf@counta}{\the\c@pgf@countb}%

```

Second, create a line to the end point.

```

3968     \edef\forest@last@x{%
3969         \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname}%
3970     \edef\forest@last@y{%
3971         \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname}%
3972     \pgfsyssoftpath@lineto\forest@last@x\forest@last@y

```

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

```

3973     \c@pgf@countc=\csname forest@pi@\forest@aplusone @n\endcsname
3974     \c@pgf@countb=\csname forest@pi@\the\c@pgf@counta @n\endcsname
3975     \fi
3976     \advance\c@pgf@countc 1
3977     \forest@repeatc
3978     \advance\c@pgf@countb 1
3979 }

```

`\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.

```

3980 \def\forest@gettightedgeofpath@maybemoveto#1#2{%
3981   \forest@temptrue
3982   \ifx\forest@last@x\relax\else
3983     \ifdim\forest@last@x=\csname forest@pi@#1@#2x\endcsname\relax
3984     \ifdim\forest@last@y=\csname forest@pi@#1@#2y\endcsname\relax
3985     \forest@tempfalse
3986   \fi
3987 \fi
3988 \fi
3989 \ifforest@temp
3990 \edef\forest@last@x{\csname forest@pi@#1@#2x\endcsname}%
3991 \edef\forest@last@y{\csname forest@pi@#1@#2y\endcsname}%
3992 \pgfsyssoftpath@moveto\forest@last@x\forest@last@y
3993 \fi
3994 }

```

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

```

3995 \def\forest@simplifypath#1#2{%
3996   \pgfsyssoftpath@setcurrentpath\pgfutil@empty
3997   \forest@save@pgfsyssoftpath@tokendefs
3998   \let\pgfsyssoftpath@movetotoken\forest@simplifypath@moveto
3999   \let\pgfsyssoftpath@linetotoken\forest@simplifypath@lineto
4000   \let\forest@last@x\relax
4001   \let\forest@last@y\relax
4002   \let\forest@last@atan\relax
4003   #1%
4004   \ifx\forest@last@x\relax\else
4005     \ifx\forest@last@atan\relax\else
4006       \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
4007     \fi
4008   \fi
4009   \forest@restore@pgfsyssoftpath@tokendefs
4010   \pgfsyssoftpath@getcurrentpath#2%
4011 }

```

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.

```

4012 \def\forest@simplifypath@moveto#1#2{%
4013   \ifx\forest@last@x\relax\else
4014     \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
4015   \fi
4016   \pgfsyssoftpath@moveto{#1}{#2}%
4017   \def\forest@last@x{#1}%
4018   \def\forest@last@y{#2}%
4019   \let\forest@last@atan\relax
4020 }

```

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

```

4021 \def\forest@getedgeofpath@precision{1pt}

```

When a line-to is encountered...

```

4022 \def\forest@simplifypath@lineto#1#2{%
4023   \ifx\forest@last@x\relax

```

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

```

4024     \def\forest@last@x{#1}%
4025     \def\forest@last@y{#2}%
4026     \let\forest@last@atan\relax
4027   \else

```

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

```

4028   \pgfpointdiff{\pgfpoint{#1}{#2}}{\pgfpoint{\forest@last@x}{\forest@last@y}}%
4029   \ifdim\pgf@x<\pgfintersectiontolerance
4030     \ifdim-\pgf@x<\pgfintersectiontolerance
4031       \pgf@x=0pt
4032     \fi
4033   \fi
4034   \csname pgfmathatan2\endcsname{\pgf@x}{\pgf@y}%
4035   \let\forest@current@atan\pgfmathresult
4036   \ifx\forest@last@atan\relax

```

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

```

4037     \def\forest@last@x{#1}%
4038     \def\forest@last@y{#2}%
4039     \let\forest@last@atan\forest@current@atan
4040   \else

```

Otherwise, compare the first and the current slope.

```

4041     \pgfutil@tempdima=\forest@current@atan pt
4042     \advance\pgfutil@tempdima -\forest@last@atan pt
4043     \ifdim\pgfutil@tempdima<0pt\relax
4044       \multiply\pgfutil@tempdima -1
4045     \fi
4046     \ifdim\pgfutil@tempdima<\forest@getedgeofpath@precision\relax
4047     \else

```

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

```

4048       \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
4049       \let\forest@last@atan\forest@current@atan
4050     \fi

```

In any event, update the last point.

```

4051     \def\forest@last@x{#1}%
4052     \def\forest@last@y{#2}%
4053   \fi
4054 \fi
4055 }

```

## 12.4 Get rectangle/band edge

```

4056 \def\forest@getnegativerectangleedgeofpath#1#2{%
4057   \forest@getnegativerectangleorbandededgeofpath{#1}{#2}{\the\pgf@xb}}
4058 \def\forest@getpositiverectangleedgeofpath#1#2{%
4059   \forest@getpositiverectangleorbandededgeofpath{#1}{#2}{\the\pgf@xb}}
4060 \def\forest@getbothrectangleedgesofpath#1#2#3{%
4061   \forest@getbothrectangleorbandededgesofpath{#1}{#2}{#3}{\the\pgf@xb}}
4062 \def\forest@bandlength{5000pt} % something large (ca. 180cm), but still manageable for TeX without producing
4063 \def\forest@getnegativebandededgeofpath#1#2{%
4064   \forest@getnegativerectangleorbandededgeofpath{#1}{#2}{\forest@bandlength}}
4065 \def\forest@getpositivebandededgeofpath#1#2{%
4066   \forest@getpositiverectangleorbandededgeofpath{#1}{#2}{\forest@bandlength}}
4067 \def\forest@getbothbandededgesofpath#1#2#3{%
4068   \forest@getbothrectangleorbandededgesofpath{#1}{#2}{#3}{\forest@bandlength}}
4069 \def\forest@getnegativerectangleorbandededgeofpath#1#2#3{%
4070   \forest@path@getboundingrectangle@ls#1{\forest@grow}%
4071   \edef\forest@gre@path{%
4072     \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@ya}%
4073     \noexpand\pgfsyssoftpath@linetotoken{#3}{\the\pgf@ya}%
4074   }%
4075   {%
4076     \pgftransformreset

```

```

4077 \pgftransformrotate{\forest@grow}%
4078 \forest@pgfpathtransformed\forest@gre@path
4079 }%
4080 \pgfsyssoftpath@getcurrentpath#2%
4081 }
4082 \def\forest@getpositiverectangleorbandededgeofpath#1#2#3{%
4083 \forest@path@getboundingrectangle@ls#1{\forest@grow}%
4084 \edef\forest@gre@path{%
4085 \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@yb}%
4086 \noexpand\pgfsyssoftpath@linetotoken{#3}{\the\pgf@yb}%
4087 }%
4088 {%
4089 \pgftransformreset
4090 \pgftransformrotate{\forest@grow}%
4091 \forest@pgfpathtransformed\forest@gre@path
4092 }%
4093 \pgfsyssoftpath@getcurrentpath#2%
4094 }
4095 \def\forest@getbothrectangleorbandededgesofpath#1#2#3#4{%
4096 \forest@path@getboundingrectangle@ls#1{\forest@grow}%
4097 \edef\forest@gre@negpath{%
4098 \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@ya}%
4099 \noexpand\pgfsyssoftpath@linetotoken{#4}{\the\pgf@ya}%
4100 }%
4101 \edef\forest@gre@pospath{%
4102 \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@yb}%
4103 \noexpand\pgfsyssoftpath@linetotoken{#4}{\the\pgf@yb}%
4104 }%
4105 {%
4106 \pgftransformreset
4107 \pgftransformrotate{\forest@grow}%
4108 \forest@pgfpathtransformed\forest@gre@negpath
4109 }%
4110 \pgfsyssoftpath@getcurrentpath#2%
4111 {%
4112 \pgftransformreset
4113 \pgftransformrotate{\forest@grow}%
4114 \forest@pgfpathtransformed\forest@gre@pospath
4115 }%
4116 \pgfsyssoftpath@getcurrentpath#3%
4117 }

```

## 12.5 Distance between paths

Another crucial part of the package.

```

4118 \def\forest@distance@between@edge@paths#1#2#3{%
4119 % #1, #2 = (edge) paths
4120 %
4121 % project paths
4122 \forest@projectpathtogrowline#1{\forest@p1@}%
4123 \forest@projectpathtogrowline#2{\forest@p2@}%
4124 % merge projections (the lists are sorted already, because edge
4125 % paths are |sorted|)
4126 \forest@dbep@mergeprojections
4127 {\forest@p1@}{\forest@p2@}%
4128 {\forest@P1@}{\forest@P2@}%
4129 % process projections
4130 \forest@processprojectioninfo{\forest@P1@}{\forest@PI1@}%
4131 \forest@processprojectioninfo{\forest@P2@}{\forest@PI2@}%
4132 % break paths
4133 \forest@breakpath#1{\forest@PI1@}\forest@broken@one

```

```

4134 \forest@breakpath#2{forest@PI2@}\forest@broken@two
4135 % sort inner arrays ---optimize: it's enough to find max and min
4136 \forest@sort@inner@arrays{forest@PI1@}\forest@sort@descending
4137 \forest@sort@inner@arrays{forest@PI2@}\forest@sort@ascending
4138 % compute the distance
4139 \let\forest@distance\relax
4140 \c@pgf@countc=0
4141 \loop
4142 \ifnum\c@pgf@countc<\csname forest@PI1@n\endcsname\relax
4143 \ifnum\csname forest@PI1@\the\c@pgf@countc @n\endcsname=0 \else
4144 \ifnum\csname forest@PI2@\the\c@pgf@countc @n\endcsname=0 \else
4145 \pgfutil@tempdima=\csname forest@PI2@\the\c@pgf@countc @0d\endcsname\relax
4146 \advance\pgfutil@tempdima -\csname forest@PI1@\the\c@pgf@countc @0d\endcsname\relax
4147 \ifx\forest@distance\relax
4148 \edef\forest@distance{\the\pgfutil@tempdima}%
4149 \else
4150 \ifdim\pgfutil@tempdima<\forest@distance\relax
4151 \edef\forest@distance{\the\pgfutil@tempdima}%
4152 \fi
4153 \fi
4154 \fi
4155 \fi
4156 \advance\c@pgf@countc 1
4157 \repeat
4158 \let#3\forest@distance
4159 }
4160 % merge projections: we need two projection arrays, both containing
4161 % projection points from both paths, but each with the original
4162 % points from only one path
4163 \def\forest@dbep@mergeprojections#1#2#3#4{%
4164 % TODO: optimize: v bistvu ni treba sortirat, ker je edge path e sortiran
4165 \forest@sortprojections{#1}%
4166 \forest@sortprojections{#2}%
4167 \c@pgf@counta=0
4168 \c@pgf@countb=0
4169 \c@pgf@countc=0
4170 \edef\forest@input@prefix@one{#1}%
4171 \edef\forest@input@prefix@two{#2}%
4172 \edef\forest@output@prefix@one{#3}%
4173 \edef\forest@output@prefix@two{#4}%
4174 \forest@dbep@mp@iterate
4175 \csedef{#3n}{\the\c@pgf@countc}%
4176 \csedef{#4n}{\the\c@pgf@countc}%
4177 }
4178 \def\forest@dbep@mp@iterate{%
4179 \let\forest@dbep@mp@next\forest@dbep@mp@iterate
4180 \ifnum\c@pgf@counta<\csname\forest@input@prefix@one n\endcsname\relax
4181 \ifnum\c@pgf@countb<\csname\forest@input@prefix@two n\endcsname\relax
4182 \let\forest@dbep@mp@next\forest@dbep@mp@do
4183 \else
4184 \let\forest@dbep@mp@next\forest@dbep@mp@iteratefirst
4185 \fi
4186 \else
4187 \ifnum\c@pgf@countb<\csname\forest@input@prefix@two n\endcsname\relax
4188 \let\forest@dbep@mp@next\forest@dbep@mp@iteratesecond
4189 \else
4190 \let\forest@dbep@mp@next\relax
4191 \fi
4192 \fi
4193 \forest@dbep@mp@next
4194 }

```

```

4195 \def\forest@dbep@mp@do{%
4196   \forest@sort@cmptwodimcs%
4197   {\forest@input@prefix@one\the\c@pgf@counta xp}%
4198   {\forest@input@prefix@one\the\c@pgf@counta yp}%
4199   {\forest@input@prefix@two\the\c@pgf@countb xp}%
4200   {\forest@input@prefix@two\the\c@pgf@countb yp}%
4201 \if\forest@sort@cmp@result=%
4202   \forest@dbep@mp@@store@p\forest@input@prefix@one\c@pgf@counta
4203   \forest@dbep@mp@@store@o\forest@input@prefix@one
4204   \c@pgf@counta\forest@output@prefix@one
4205   \forest@dbep@mp@@store@o\forest@input@prefix@two
4206   \c@pgf@countb\forest@output@prefix@two
4207   \advance\c@pgf@counta 1
4208   \advance\c@pgf@countb 1
4209 \else
4210   \if\forest@sort@cmp@result>%
4211     \forest@dbep@mp@@store@p\forest@input@prefix@two\c@pgf@countb
4212     \forest@dbep@mp@@store@o\forest@input@prefix@two
4213     \c@pgf@countb\forest@output@prefix@two
4214     \advance\c@pgf@countb 1
4215   \else%<
4216     \forest@dbep@mp@@store@p\forest@input@prefix@one\c@pgf@counta
4217     \forest@dbep@mp@@store@o\forest@input@prefix@one
4218     \c@pgf@counta\forest@output@prefix@one
4219     \advance\c@pgf@counta 1
4220   \fi
4221 \fi
4222 \advance\c@pgf@countc 1
4223 \forest@dbep@mp@iterate
4224 }
4225 \def\forest@dbep@mp@@store@p#1#2{%
4226   \csletcs
4227   {\forest@output@prefix@one\the\c@pgf@countc xp}%
4228   {#1\the#2xp}%
4229   \csletcs
4230   {\forest@output@prefix@one\the\c@pgf@countc yp}%
4231   {#1\the#2yp}%
4232   \csletcs
4233   {\forest@output@prefix@two\the\c@pgf@countc xp}%
4234   {#1\the#2xp}%
4235   \csletcs
4236   {\forest@output@prefix@two\the\c@pgf@countc yp}%
4237   {#1\the#2yp}%
4238 }
4239 \def\forest@dbep@mp@@store@o#1#2#3{%
4240   \csletcs{#3\the\c@pgf@countc xo}{#1\the#2xo}%
4241   \csletcs{#3\the\c@pgf@countc yo}{#1\the#2yo}%
4242 }
4243 \def\forest@dbep@mp@iteratefirst{%
4244   \forest@dbep@mp@iterateone\forest@input@prefix@one\c@pgf@counta\forest@output@prefix@one
4245 }
4246 \def\forest@dbep@mp@iteratesecond{%
4247   \forest@dbep@mp@iterateone\forest@input@prefix@two\c@pgf@countb\forest@output@prefix@two
4248 }
4249 \def\forest@dbep@mp@iterateone#1#2#3{%
4250   \loop
4251   \ifnum#2<\csname#1n\endcsname\relax
4252     \forest@dbep@mp@@store@p#1#2%
4253     \forest@dbep@mp@@store@o#1#2#3%
4254     \advance\c@pgf@countc 1
4255     \advance#21

```



```
4256 \repeat
4257 }
```

## 12.6 Utilities

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

```
4258 \newif\ifforest@equaltotolerance
4259 \def\forest@equaltotolerance#1#2{%
4260   \pgfpointdiff{#1}{#2}%
4261   \ifdim\pgf@x<0pt \multiply\pgf@x -1 \fi
4262   \ifdim\pgf@y<0pt \multiply\pgf@y -1 \fi
4263   \global\forest@equaltotolerancefalse
4264   \ifdim\pgf@x<\pgfintersectiontolerance\relax
4265     \ifdim\pgf@y<\pgfintersectiontolerance\relax
4266       \global\forest@equaltolerancetrue
4267     \fi
4268   \fi
4269 }}
```

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

```
4270 \def\forest@save@pgfsyssoftpath@tokendefs{%
4271   \let\forest@origmovetotoken\pgfsyssoftpath@movetotoken
4272   \let\forest@origlinetotoken\pgfsyssoftpath@linetotoken
4273   \let\forest@origcurvetosupportatoken\pgfsyssoftpath@curvetosupportatoken
4274   \let\forest@origcurvetosupportbtoken\pgfsyssoftpath@curvetosupportbtoken
4275   \let\forest@origcurvetotoken\pgfsyssoftpath@curvetototoken
4276   \let\forest@origrectcornertoken\pgfsyssoftpath@rectcornertoken
4277   \let\forest@origrectsizetoken\pgfsyssoftpath@rectsizetoken
4278   \let\forest@origclosepathtoken\pgfsyssoftpath@closepathtoken
4279   \let\pgfsyssoftpath@movetotoken\forest@badtoken
4280   \let\pgfsyssoftpath@linetotoken\forest@badtoken
4281   \let\pgfsyssoftpath@curvetosupportatoken\forest@badtoken
4282   \let\pgfsyssoftpath@curvetosupportbtoken\forest@badtoken
4283   \let\pgfsyssoftpath@curvetototoken\forest@badtoken
4284   \let\pgfsyssoftpath@rectcornertoken\forest@badtoken
4285   \let\pgfsyssoftpath@rectsizetoken\forest@badtoken
4286   \let\pgfsyssoftpath@closepathtoken\forest@badtoken
4287 }
4288 \def\forest@badtoken{%
4289   \PackageError{forest}{This token should not be in this path}{}%
4290 }
4291 \def\forest@restore@pgfsyssoftpath@tokendefs{%
4292   \let\pgfsyssoftpath@movetotoken\forest@origmovetotoken
4293   \let\pgfsyssoftpath@linetotoken\forest@origlinetotoken
4294   \let\pgfsyssoftpath@curvetosupportatoken\forest@origcurvetosupportatoken
4295   \let\pgfsyssoftpath@curvetosupportbtoken\forest@origcurvetosupportbtoken
4296   \let\pgfsyssoftpath@curvetototoken\forest@origcurvetotoken
4297   \let\pgfsyssoftpath@rectcornertoken\forest@origrectcornertoken
4298   \let\pgfsyssoftpath@rectsizetoken\forest@origrectsizetoken
4299   \let\pgfsyssoftpath@closepathtoken\forest@origclosepathtoken
4300 }
```

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

```
4301 \def\forest@extendpath#1#2#3{%
4302   \pgf@process{#3}%
4303   \pgfsyssoftpath@setcurrentpath#1%
4304   \forest@save@pgfsyssoftpath@tokendefs
4305   \let\pgfsyssoftpath@movetotoken\forest@extendpath@moveto
4306   \let\pgfsyssoftpath@linetotoken\forest@extendpath@lineto
4307   #2%
```

```

4308 \forest@restore@pgfsyssoftpath@tokendefs
4309 \pgfsyssoftpath@getcurrentpath#1%
4310 }
4311 \def\forest@extendpath@moveto#1#2{%
4312 \forest@extendpath@do{#1}{#2}\pgfsyssoftpath@moveto
4313 }
4314 \def\forest@extendpath@lineto#1#2{%
4315 \forest@extendpath@do{#1}{#2}\pgfsyssoftpath@lineto
4316 }
4317 \def\forest@extendpath@do#1#2#3{%
4318 {%
4319 \advance\pgf@x #1
4320 \advance\pgf@y #2
4321 #3{\the\pgf@x}{\the\pgf@y}%
4322 }%
4323 }

    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.)
4324 \def\forest@path@getboundingrectangle@ls#1#2{%
4325 {%
4326 \pgftransformreset
4327 \pgftransformrotate{-(#2)}%
4328 \forest@pgfpathtransformed#1%
4329 }%
4330 \pgfsyssoftpath@getcurrentpath\forest@gbr@rotatedpath
4331 \forest@path@getboundingrectangle@xy\forest@gbr@rotatedpath
4332 }
4333 \def\forest@path@getboundingrectangle@xy#1{%
4334 \forest@save@pgfsyssoftpath@tokendefs
4335 \let\pgfsyssoftpath@movetotoken\forest@gbr@firstpoint
4336 \let\pgfsyssoftpath@linetotoken\forest@gbr@firstpoint
4337 #1%
4338 \forest@restore@pgfsyssoftpath@tokendefs
4339 }
4340 \def\forest@gbr@firstpoint#1#2{%
4341 \pgf@xa=#1 \pgf@xb=#1 \pgf@ya=#2 \pgf@yb=#2
4342 \let\pgfsyssoftpath@movetotoken\forest@gbr@point
4343 \let\pgfsyssoftpath@linetotoken\forest@gbr@point
4344 }
4345 \def\forest@gbr@point#1#2{%
4346 \ifdim#1<\pgf@xa\relax\pgf@xa=#1 \fi
4347 \ifdim#1>\pgf@xb\relax\pgf@xb=#1 \fi
4348 \ifdim#2<\pgf@ya\relax\pgf@ya=#2 \fi
4349 \ifdim#2>\pgf@yb\relax\pgf@yb=#2 \fi
4350 }

```

## 13 The outer UI

### 13.1 Package options

```

4351 \newif\ifforesttikzcshack
4352 \foresttikzcshacktrue
4353 \newif\ifforest@install@keys@to@tikz@path@
4354 \forest@install@keys@to@tikz@path@true
4355 \forestset{package@options/.cd,
4356 external/.is if=forest@external@,
4357 tikzcshack/.is if=foresttikzcshack,
4358 tikzinstallkeys/.is if=forest@install@keys@to@tikz@path@,
4359 }

```

## 13.2 Externalization

```
4360 \pgfkeys{/forest/external/.cd,  
4361   copy command/.initial={cp "\source" "\target"},  
4362   optimize/.is if=forest@external@optimize@,  
4363   context/.initial={%  
4364     \forestOve{\csname forest@id@of@standard node\endcsname}{environment@formula}},  
4365   depends on macro/.style={context/.append/.expanded={%  
4366     \expandafter\detokenize\expandafter{#1}}},  
4367 }  
4368 \def\forest@external@copy#1#2{%  
4369   \pgfkeysgetvalue{/forest/external/copy command}\forest@copy@command  
4370   \ifx\forest@copy@command\pgfkeysnovalue\else  
4371     \IfFileExists{#1}{%  
4372       {%  
4373         \def\source{#1}%  
4374         \def\target{#2}%  
4375         \immediate\write18{\forest@copy@command}%  
4376       }%  
4377     }{%  
4378     \fi  
4379 }  
4380 \newif\ifforest@external@  
4381 \newif\ifforest@external@optimize@  
4382 \forest@external@optimize@true  
4383 \ProcessPgfPackageOptions{/forest/package@options}  
4384 \ifforest@install@keys@to@tikz@path@  
4385 \tikzset{fit to tree/.style={/forest/fit to tree}}  
4386 \fi  
4387 \ifforest@external@  
4388   \ifdefined\tikzexternal@tikz@replacement\else  
4389     \usetikzlibrary{external}%  
4390   \fi  
4391   \pgfkeys{%  
4392     /tikz/external/failed ref warnings for={},  
4393     /pgf/images/aux in dpth=false,  
4394   }%  
4395   \tikzifexternalizing{}{%  
4396     \forest@external@copy{\jobname.aux}{\jobname.aux.copy}%  
4397   }%  
4398   \AtBeginDocument{%  
4399     \tikzifexternalizing{%  
4400       \IfFileExists{\tikzexternalrealjob.aux.copy}{%  
4401         \makeatletter  
4402         \input \tikzexternalrealjob.aux.copy  
4403         \makeatother  
4404       }{%  
4405       }{%  
4406         \newwrite\forest@auxout  
4407         \immediate\openout\forest@auxout=\tikzexternalrealjob.for.tmp  
4408       }%  
4409       \IfFileExists{\tikzexternalrealjob.for}{%  
4410         {%  
4411           \makehashother\makeatletter  
4412           \input \tikzexternalrealjob.for  
4413         }%  
4414       }{%  
4415       }%  
4416     \AtEndDocument{%  
4417       \tikzifexternalizing{}{%  
4418         \immediate\closeout\forest@auxout
```

```

4419     \forest@external@copy{\jobname.for.tmp}{\jobname.for}%
4420   }%
4421 }%
4422 \fi

```

### 13.3 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.

```

4423 \newenvironment{forest}{\pgfkeysalso{/forest/begin forest}\Collect@Body\forest@env}{}
4424 \long\def\Forest{\pgfkeysalso{/forest/begin forest}\@ifnextchar*{\forest@nogroup}{\forest@group}}
4425 \def\forest@group#1{\forest@env{#1}}
4426 \def\forest@nogroup*#1{\forest@env{#1}}
4427 \newif\ifforest@externalize@tree@
4428 \newif\ifforest@was@tikzexternalwasenable
4429 \long\def\forest@env#1{%
4430   \let\forest@external@next\forest@begin
4431   \forest@was@tikzexternalwasenablefalse
4432   \ifdefined\tikzexternal@tikz@replacement
4433     \ifx\tikz\tikzexternal@tikz@replacement
4434       \forest@was@tikzexternalwasenabletrue
4435       \tikzexternaldisable
4436     \fi
4437   \fi
4438   \forest@externalize@tree@false
4439   \ifforest@external@
4440     \ifforest@was@tikzexternalwasenable
4441     \tikzifexternalizing{%
4442       \let\forest@external@next\forest@begin@externalizing
4443     }{%
4444       \let\forest@external@next\forest@begin@externalize
4445     }%
4446   \fi
4447   \fi
4448   \forest@standardnode@calibrate
4449   \forest@external@next{#1}%
4450 }

```

We're externalizing, i.e. this code gets executed in the embedded call.

```

4451 \long\def\forest@begin@externalizing#1{%
4452   \forest@external@setup{#1}%
4453   \let\forest@external@next\forest@begin
4454   \forest@externalize@inner@=-1
4455   \ifforest@external@optimize@\forest@externalizing@maybeoptimize\fi
4456   \forest@external@next{#1}%
4457   \tikzexternalenable
4458 }
4459 \def\forest@externalizing@maybeoptimize{%
4460   \edef\forest@temp{\tikzexternalrealjob-forest-\forest@externalize@outer@n}%
4461   \edef\forest@marshal{%
4462     \noexpand\pgfutil@in@
4463     {\expandafter\detokenize\expandafter{\forest@temp}.}
4464     {\expandafter\detokenize\expandafter{\pgfactualjobname}.}%
4465   }\forest@marshal
4466   \ifpgfutil@in@
4467   \else
4468     \let\forest@external@next@gobble
4469   \fi
4470 }

```

Externalization is enabled, we're in the outer process, deciding if the picture is up-to-date.

```

4471 \long\def\forest@begin@externalize#1{%
4472   \forest@external@setup{#1}%
4473   \iftikzexternal@file@isuptodate
4474     \setbox0=\hbox{%
4475       \csname forest@externalcheck@\forest@externalize@outer@n\endcsname
4476     }%
4477   \fi
4478   \iftikzexternal@file@isuptodate
4479     \csname forest@externalload@\forest@externalize@outer@n\endcsname
4480   \else
4481     \forest@externalize@tree@true
4482     \forest@externalize@inner@n=-1
4483     \forest@begin{#1}%
4484     \ifcsdef{forest@externalize@@\forest@externalize@id}{-}{%
4485       \immediate\write\forest@auxout{%
4486         \noexpand\forest@external
4487         {\forest@externalize@outer@n}%
4488         {\expandafter\detokenize\expandafter{\forest@externalize@id}}%
4489         {\expandonce\forest@externalize@checkimages}%
4490         {\expandonce\forest@externalize@loadimages}%
4491       }%
4492     }%
4493   \fi
4494   \tikzexternalenable
4495 }
4496 \def\forest@includeexternal@check#1{%
4497   \tikzsetnextfilename{#1}%
4498   \IfFileExists{\tikzexternal@filenameprefix/#1}{\tikzexternal@file@isuptodatetrue}{\tikzexternal@file@isuptodatefalse}
4499 }
4500 \def\makehashother{\catcode'\# =12}%
4501 \long\def\forest@external@setup#1{%
4502   % set up \forest@externalize@id and \forest@externalize@outer@n
4503   % we need to deal with #s correctly (\write doubles them)
4504   \setbox0=\hbox{\makehashother\makeatletter
4505     \scantokens{\forest@temp@toks{#1}}\expandafter
4506   }%
4507   \expandafter\forest@temp@toks\expandafter{\the\forest@temp@toks}%
4508   \edef\forest@temp{\pgfkeysvalueof{/forest/external/context}}%
4509   \edef\forest@externalize@id{%
4510     \expandafter\detokenize\expandafter{\forest@temp}%
4511     @@%
4512     \expandafter\detokenize\expandafter{\the\forest@temp@toks}%
4513   }%
4514   \letcs\forest@externalize@outer@n{forest@externalize@@\forest@externalize@id}%
4515   \ifdefined\forest@externalize@outer@n
4516     \global\tikzexternal@file@isuptodatetrue
4517   \else
4518     \global\advance\forest@externalize@max@outer@n 1
4519     \edef\forest@externalize@outer@n{\the\forest@externalize@max@outer@n}%
4520     \global\tikzexternal@file@isuptodatefalse
4521   \fi
4522   \def\forest@externalize@loadimages{}%
4523   \def\forest@externalize@checkimages{}%
4524 }
4525 \newcount\forest@externalize@max@outer@n
4526 \global\forest@externalize@max@outer@n=0
4527 \newcount\forest@externalize@inner@n

```

The .for file is a string of calls of this macro.

```

4528 \long\def\forest@external#1#2#3#4{% #1=n,#2=context+source code,#3=update check code, #4=load code

```

```

4529 \ifnum\forest@externalize@max@outer@n<#1
4530   \global\forest@externalize@max@outer@n=#1
4531 \fi
4532 \global\csdef{forest@externalize@@\detokenize{#2}}{#1}%
4533 \global\csdef{forest@externalcheck@#1}{#3}%
4534 \global\csdef{forest@externalload@#1}{#4}%
4535 \tikzifexternalizing{}{%
4536   \immediate\write\forest@auxout{%
4537     \noexpand\forest@external{#1}%
4538     {\expandafter\detokenize\expandafter{#2}}%
4539     {\unexpanded{#3}}%
4540     {\unexpanded{#4}}%
4541   }%
4542 }%
4543 }

```

These two macros include the external picture.

```

4544 \def\forest@includeexternal#1{%
4545   \edef\forest@temp{\pgfkeysvalueof{/forest/external/context}}%
4546   \typeout{forest: Including external picture '#1' for forest context+code:
4547     '\expandafter\detokenize\expandafter{\forest@externalize@id}'}%
4548   {%
4549     %\def\pgf@declaredraftimage##1##2{\def\pgf@image{\hbox{}}}%
4550     \tikzsetnextfilename{#1}%
4551     \tikzexternalenable
4552     \tikz{}%
4553   }%
4554 }
4555 \def\forest@includeexternal@box#1#2{%
4556   \global\setbox#1=\hbox{\forest@includeexternal{#2}}%
4557 }

```

This code runs the bracket parser and stage processing.

```

4558 \long\def\forest@begin#1{%
4559   \iffalse{\fi\forest@parsebracket#1}%
4560 }
4561 \def\forest@parsebracket{%
4562   \bracketParse{\forest@get@root@afterthought}\forest@root=%
4563 }
4564 \def\forest@get@root@afterthought{%
4565   \expandafter\forest@get@root@afterthought@expandafter{\iffalse}\fi
4566 }
4567 \long\def\forest@get@root@afterthought@#1{%
4568   \ifblank{#1}{}{%
4569     \forest@eappto{\forest@root}{given options}{,afterthought={\unexpanded{#1}}}%
4570   }%
4571   \forest@do
4572 }
4573 \def\forest@do{%
4574   \forest@node@Compute@numeric@ts@info{\forest@root}%
4575   \forestset{process keylist=given options}%
4576   \forestset{stages}%
4577   \pgfkeysalso{/forest/end forest}%
4578   \ifforest@was@tikzexternalwasenable
4579     \tikzexternalenable
4580   \fi
4581 }

```

## 13.4 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!

```

4582 \def\forest@standardnode@new{%
4583   \advance\forest@node@maxid1
4584   \forest@fornode{\the\forest@node@maxid}{%
4585     \forest@node@init
4586     \forest@node@setname{standard node}%
4587   }%
4588 }
4589 \def\forest@standardnode@calibrate{%
4590   \forest@fornode{\forest@node@Nametoid{standard node}}{%
4591     \edef\forest@environment{\forestove{environment@formula}}%
4592     \forestoget{previous@environment}\forest@previous@environment
4593     \ifx\forest@environment\forest@previous@environment\else
4594       \forestolet{previous@environment}\forest@environment
4595       \forest@node@typeset
4596       \forestoget{calibration@procedure}\forest@temp
4597       \expandafter\forestset\expandafter{\forest@temp}%
4598     \fi
4599   }%
4600 }

```

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  $\TeX$  group.)

```

4601 \def\forestStandardNode[#1]#2#3#4{%
4602   \let\forest@standardnode@restoretikzexternal\relax
4603   \ifdefined\tikzexternaldisable
4604     \ifx\tikz\tikzexternal@tikz@replacement
4605       \tikzexternaldisable
4606       \let\forest@standardnode@restoretikzexternal\tikzexternalenable
4607     \fi
4608   \fi
4609   \forest@standardnode@new
4610   \forest@fornode{\forest@node@Nametoid{standard node}}{%
4611     \forestset{content=#1}%
4612     \forestoset{environment@formula}{#2}%
4613     \edef\forest@temp{\unexpanded{#3}}%
4614     \forestolet{calibration@procedure}\forest@temp
4615     \def\forest@calibration@initializing@code{%
4616       \pgfqkeys{/forest/initializing@code}{#4}%
4617     \forestolet{initializing@code}\forest@calibration@initializing@code
4618     \forest@standardnode@restoretikzexternal
4619   }
4620 }
4621 \forestset{initializing@code/.unknown/.code={%
4622   \eappto\forest@calibration@initializing@code{%
4623     \noexpand\forestoget{\forest@node@Nametoid{standard node}}{\pgfkeyscurrentname}\noexpand\forest@temp
4624     \noexpand\forestolet{\pgfkeyscurrentname}\noexpand\forest@temp
4625   }%
4626 }
4627 }

```

This macro is called from a new (non-standard) node’s init.

```

4628 \def\forest@initializefromstandardnode{%

```

```

4629 \forestOve{\forest@node@Nametoid{standard node}}{initializing@code}%
4630 }

```

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) `l 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 `l sep`, thus, we derive `l` from `l 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`.)

```

4631 \forestStandardNode[dj]
4632 {%
4633   \forestOve{\forest@node@Nametoid{standard node}}{content},%
4634   \the\ht\strutbox,\the\pgflinewidth,%
4635   \pgfkeysvalueof{/pgf/inner ysep},\pgfkeysvalueof{/pgf/outer ysep},%
4636   \pgfkeysvalueof{/pgf/inner xsep},\pgfkeysvalueof{/pgf/outer xsep}%
4637 }
4638 {
4639   l sep={\the\ht\strutbox+\pgfkeysvalueof{/pgf/inner ysep}},
4640   l={l sep()+abs(max_y()-min_y()+2*\pgfkeysvalueof{/pgf/outer ysep})},
4641   s sep={2*\pgfkeysvalueof{/pgf/inner xsep}}
4642 }
4643 {l sep,l,s sep}

```

### 13.5 ls coordinate system

```

4644 \pgfqkeys{/forest/@cs}{%
4645   name/.code={%
4646     \edef\forest@cn{\forest@node@Nametoid{#1}}%
4647     \forest@forestcs@resetxy},
4648   id/.code={%
4649     \edef\forest@cn{#1}%
4650     \forest@forestcs@resetxy},
4651   go/.code={%
4652     \forest@go{#1}%
4653     \forest@forestcs@resetxy},
4654   anchor/.code={\forest@forestcs@anchor{#1}},
4655   l/.code={%
4656     \pgfmathsetlengthmacro\forest@forestcs@l{#1}%
4657     \forest@forestcs@ls
4658   },
4659   s/.code={%
4660     \pgfmathsetlengthmacro\forest@forestcs@s{#1}%
4661     \forest@forestcs@ls
4662   },
4663   .unknown/.code={%
4664     \expandafter\pgfutil@in@\expandafter.\expandafter{\pgfkeyscurrentname}%
4665     \ifpgfutil@in@
4666       \expandafter\forest@forestcs@namegoanchor\pgfkeyscurrentname\forest@end
4667     \else
4668       \expandafter\forest@nameandgo\expandafter{\pgfkeyscurrentname}%
4669       \forest@forestcs@resetxy
4670     \fi
4671   }
4672 }
4673 \def\forest@forestcs@resetxy{%
4674   \ifnum\forest@cn=0
4675     \else

```



```

4676 \global\pgf@x\forestove{x}%
4677 \global\pgf@y\forestove{y}%
4678 \fi
4679 }
4680 \def\forest@forestcs@ls{%
4681 \ifdefined\forest@forestcs@l
4682 \ifdefined\forest@forestcs@s
4683 {%
4684 \pgftransformreset
4685 \pgftransformrotate{\forestove{grow}}%
4686 \pgfpointransformed{\pgfpoin{\forest@forestcs@l}{\forest@forestcs@s}}%
4687 }%
4688 \global\advance\pgf@x\forestove{x}%
4689 \global\advance\pgf@y\forestove{y}%
4690 \fi
4691 \fi
4692 }
4693 \def\forest@forestcs@anchor#1{%
4694 \edef\forest@marshal{%
4695 \noexpand\forest@original@tikz@parse@node\relax
4696 (\forestove{name}\ifx\relax#1\relax\else.\fi#1)%
4697 }\forest@marshal
4698 }
4699 \def\forest@forestcs@namegoanchor#1.#2\forest@end{%
4700 \forest@nameandgo{#1}%
4701 \forest@forestcs@anchor{#2}%
4702 }
4703 \tikzdeclarecoordinatesystem{forest}{%
4704 \forest@forthis{%
4705 \forest@forestcs@resetxy
4706 \ifdefined\forest@forestcs@l\undef\forest@forestcs@l\fi
4707 \ifdefined\forest@forestcs@s\undef\forest@forestcs@s\fi
4708 \pgfqkeys{/forest/@cs}{#1}%
4709 }%
4710 }

```

# Index

<b>Symbols</b>	
' key suffix	23
'* key suffix	23
'+ key suffix	23
'- key suffix	23
': key suffix	23
* key suffix	12, 13, 23, 29, 53
+ key suffix	1, 17, 23, 23, 48, 51
- key suffix	23, 31
: key suffix	23
< <i>(short step)</i>	43
> <i>(short step)</i>	43
<b>Numbers</b>	
1 <i>(short step)</i>	9, 43
2 <i>(short step)</i>	9, 43
3 <i>(short step)</i>	9, 43
4 <i>(short step)</i>	9, 43
5 <i>(short step)</i>	9, 43
6 <i>(short step)</i>	9, 43
7 <i>(short step)</i>	9, 43
8 <i>(short step)</i>	9, 43
9 <i>(short step)</i>	9, 43
<b>A</b>	
action character	20, 20, 21, 22
afterthought	33, 35
alias	1, 33
align value	
center	16, 24, 53
left	24
right	24
align option	14, 14, 16, 24, 24, 25, 53
anchor forest cs	44
anchor generic anchor	27
anchor option	10, 10, 15, 25, 26, 26, 26, 32, 44, 51
append dynamic tree	1, 39, 39
append' dynamic tree	40
append'' dynamic tree	40
<i>(autowrapped toks)</i> type	23
<b>B</b>	
b base value	25
band fit value	29, 29
base value	
b	25
bottom	14, 25, 53
t	25
top	14, 25
base option	14, 14, 24, 24, 25, 53
baseline	14, 15, 21, 33, 34, 35
before computing xy propagator	30, 37, 38
before drawing tree propagator	1, 31, 31, 37, 38
before packing propagator	37, 38
before typesetting nodes propagator	
	1, 18, 37, 38, 41, 44, 51
begin draw	34, 54
begin forest	34, 54
<i>(boolean)</i> type	23
bottom base value	14, 25, 53
/bracket	20
\bracketset	20, 21, 22
<b>C</b>	
c <i>(short step)</i>	43
calign value	
center	27
child	27
child edge	27
edge midpoint	27
first	5, 5, 27, 53
fixed angles	27
fixed edge angles	27
last	27, 53
midpoint	27
calign option	5,
	5, 19, 27, 27, 28, 30, 31, 39, 44, 51, 53, 53
calign angle	28
calign child	27
calign primary angle option	27, 28, 28
calign primary child option	28, 28
calign secondary angle option	27, 28, 28
calign secondary child option	28
calign with current	28
calign with current edge	28
center align value	16, 24, 53
center calign value	27
child calign value	27
child anchor generic anchor	32, 32
child anchor option	5, 6, 7, 26, 31, 32, 32, 44, 51
child edge calign value	27
closing bracket	22
compute xy stage	29–31, 38, 39
compute xy stage style	38
content option	1, 6, 6, 14, 17–19, 21, 24, 25,
	25, 26, 28, 29, 35, 36, 39, 44, 44, 45, 48, 50, 51
content format option	25, 25, 26
copy name template dynamic tree	40
<i>(count)</i> type	23
create dynamic tree	40
current <i>(step)</i>	29, 42
<b>D</b>	
declare autowrapped toks	24
declare boolean	24
declare count	24
declare dimen	24
declare keylist	24
declare toks	24, 51
delay propagator	6, 6, 7, 17–19, 21, 25,
	26, 28, 29, 35, 36, 37, 39, 39, 41, 44, 45, 50, 51
delay n propagator	37, 37, 54
<i>(dimen)</i> type	23
draw tree stage	26, 31, 34, 38, 39, 39
draw tree box	39, 46
draw tree stage style	38
draw tree' stage	39, 39



	<b>H</b>		<b>L</b>
handler		L <i>(short step)</i> . . . . .	43
.pgfmath . . . . .	1, 14, 17, 17, 19, 26, 28, 29, 41, 44	l <i>(short step)</i> . . . . .	9, 43
.wrap <i>n</i> pgfmath args . . . . .	18, 41, 41, 42	l forest cs . . . . .	44
.wrap 2 pgfmath args . . . . .	18, 44, 51	l option . . . . .	1, 10, 10, 12, 12, 13, 13, 14, 15, 15–17, 18, 26, 27, 27, 29, 29, 30, 31, 32, 38, 39, 44, 44, 48, 51, 53, 53
.wrap 3 pgfmath args . . . . .	18	l sep option . . . . .	14, 15, 16, 16, 24, 26, 30, 30, 44
.wrap pgfmath arg . . . . .	18, 41, 42, 51	label . . . . .	26, 34, 35
.wrap value . . . . .	6, 23, 41, 42	last <i>(step)</i> . . . . .	42
	<b>I</b>	last calign value . . . . .	27, 53
id <i>(step)</i> . . . . .	42	last leaf <i>(step)</i> . . . . .	42
id forest cs . . . . .	44	left align value . . . . .	24
id option . . . . .	33, 41	level option . . . . .	13, 14, 18, 21, 33, 44
if propagator . . . . .	1, 18, 18, 36, 37, 45	linear next <i>(step)</i> . . . . .	42
if key prefix . . . . .	7, 18, 19, 23, 25, 36, 37, 51	linear previous <i>(step)</i> . . . . .	42
if have delayed propagator . . . . .	37, 54		
if in key prefix . . . . .	23, 25, 37, 37	<b>M</b>	
ignore option . . . . .	29	math content style . . . . .	25, 54
ignore edge option . . . . .	29, 29, 32, 33, 53	max x option . . . . .	33
insert after dynamic tree . . . . .	40	max y option . . . . .	33
insert after' dynamic tree . . . . .	40	midpoint calign value . . . . .	27
insert after'' dynamic tree . . . . .	40	min x option . . . . .	33
insert before dynamic tree . . . . .	40	min y option . . . . .	33
insert before' dynamic tree . . . . .	40		
insert before'' dynamic tree . . . . .	40	<b>N</b>	
instr . . . . .	44	N <i>(short step)</i> . . . . .	43
	<b>K</b>	n <i>(short step)</i> . . . . .	9, 43
key		n <i>(step)</i> . . . . .	42
afterthought . . . . .	33, 35	n option . . . . .	14, 17, 18, 18, 21, 25, 33, 33, 39, 51
alias . . . . .	1, 33	n children option . . . . .	7, 7, 14, 18, 19, 33, 45, 51
baseline . . . . .	14, 15, 21, 33, 34, 35	n' <i>(step)</i> . . . . .	42
draw tree box . . . . .	39, 46	n' option . . . . .	25, 33, 39
label . . . . .	26, 34, 35	name <i>(step)</i> . . . . .	33, 42
no edge . . . . .	15, 32, 32, 33, 45, 48, 51	name forest cs . . . . .	44
pin . . . . .	26, 34, 35	name option . . . . .	1, 8, 8, 15, 26, 28, 29, 33, 34, 41
repeat . . . . .	1, 37, 39, 43	new node . . . . .	22
TeX . . . . .	35, 35, 35, 46, 50	next <i>(step)</i> . . . . .	42
TeX' . . . . .	35, 46	next leaf <i>(step)</i> . . . . .	42
TeX'' . . . . .	35, 35, 46	next on tier <i>(step)</i> . . . . .	42
typeset node . . . . .	1, 38	no edge . . . . .	15, 32, 32, 33, 45, 48, 51
use as bounding box . . . . .	35, 35	node format option . . . . .	25, 26, 26
use as bounding box' . . . . .	34, 35	node options option . . . . .	26, 26
key prefix		<i>(node walk)</i> . . . . .	42
for . . . . .	5, 7, 33, 36, 38, 40, 43	node walk . . . . .	35, 42
if . . . . .	7, 18, 19, 23, 25, 36, 37, 51	node walk <i>(step)</i> . . . . .	42
if in . . . . .	23, 25, 37, 37	node walk/after walk . . . . .	35, 42
not . . . . .	24	node walk/before walk . . . . .	35, 42
where . . . . .	7, 18, 23, 37	node walk/every step . . . . .	35
where in . . . . .	23, 29, 37	not key prefix . . . . .	24
key suffix		<b>O</b>	
' . . . . .	23	opening bracket . . . . .	22
'* . . . . .	23	option	
'+ . . . . .	23	align . . . . .	14, 14, 16, 24, 24, 25, 53
'- . . . . .	23	anchor . . . . .	10, 10, 15, 25, 26, 26, 26, 32, 44, 51
': . . . . .	23	base . . . . .	14, 14, 24, 24, 25, 53
* . . . . .	12, 13, 23, 29, 53	calign 5, 5, 19, 27, 27, 28, 30, 31, 39, 44, 51, 53, 53	
+ . . . . .	1, 17, 23, 23, 48, 51	calign primary angle . . . . .	27, 28, 28
- . . . . .	23, 31	calign primary child . . . . .	28, 28
: . . . . .	23	calign secondary angle . . . . .	27, 28, 28
<i>(keylist)</i> type . . . . .	23	calign secondary child . . . . .	28

child anchor	5, 6, 7, 26, 31, 32, 32, 44, 51
content	1, 6, 6, 14, 17–19, 21, 24, 25, 25, 26, 28, 29, 35, 36, 39, 44, 44, 45, 48, 50, 51
content format	25, 25, 26
edge	1, 14, 18, 32, 32, 32, 33, 36
edge label	18, 32, 32, 32
edge path	26, 32, 32, 33
fit	14, 28, 28, 29, 34, 44
grow	9, 10, 26, 27, 29, 29, 30, 44
id	33, 41
ignore	29
ignore edge	29, 29, 32, 33, 53
l	1, 10, 10, 12, 12, 13, 13, 14, 15, 15–17, 18, 26, 27, 27, 29, 29, 30, 31, 32, 38, 39, 44, 44, 48, 51, 53, 53
l sep	14, 15, 16, 16, 24, 26, 30, 30, 44
level	13, 14, 18, 21, 33, 44
max x	33
max y	33
min x	33
min y	33
n	14, 17, 18, 18, 21, 25, 33, 33, 39, 51
n children	7, 7, 14, 18, 19, 33, 45, 51
n'	25, 33, 39
name	1, 8, 8, 15, 26, 28, 29, 33, 34, 41
node format	25, 26, 26
node options	26, 26
parent anchor	5, 6, 7, 26, 32, 32, 32, 51
phantom	8, 9, 12, 14, 17, 18, 21, 26, 26
reversed	29, 30
s	10, 26, 30, 30, 38, 39, 44
s sep	1, 11, 11, 12, 13, 13, 15, 31, 50, 51, 53
tier	5, 6, 6, 7, 7, 31, 35, 44, 45, 48, 50, 53
tikz	8, 9, 9, 10, 19, 35, 39, 45, 48, 51
x	26, 31, 38, 39
y	1, 26, 31, 31, 38, 39
<b>P</b>	
P <i>(short step)</i>	43
p <i>(short step)</i>	9, 43
pack stage	26, 29, 30, 38, 39
pack stage style	38
pack'	39, 54
package option	
external	19, 47
tikzcshack	44, 47
tikzinstallkeys	47
parent <i>(step)</i>	42, 51
parent anchor generic anchor	32, 32
parent anchor option	5, 6, 7, 26, 32, 32, 32, 51
.pgfmath handler	1, 14, 17, 17, 19, 26, 28, 29, 41, 44
phantom option	8, 9, 12, 14, 17, 18, 21, 26, 26
pin	26, 34, 35
prepend dynamic tree	40
prepend' dynamic tree	40
prepend'' dynamic tree	40
previous <i>(step)</i>	42
previous leaf <i>(step)</i>	42
previous on tier <i>(step)</i>	43
process keylist	38, 39
propagator	
before computing xy	30, 37, 38
before drawing tree	1, 31, 31, 37, 38
before packing	37, 38
before typesetting nodes	
	1, 18, 37, 38, 41, 44, 51
delay	6, 6, 7, 17–19, 21, 25, 26, 28, 29, 35, 36, 37, 39, 39, 41, 44, 45, 50, 51
delay n	37, 37, 54
for	1, 14, 21, 29, 36, 36, 51
for all next	36
for all previous	36
for ancestors	36
for ancestors'	1, 36, 36
for children	1, 6, 14, 15, 18, 19, 36, 48, 51
for current	42
for descendants	5, 14, 15, 17, 18, 26, 36, 51
for first	42
for first leaf	42
for id	42
for last	42
for last leaf	42
for linear next	42
for linear previous	42
for n	42
for n'	42
for name	42
for next	42
for next leaf	42
for next on tier	42
for parent	42
for previous	42
for previous leaf	43
for previous on tier	43
for root	43
for root'	43
for sibling	43
for to tier	43
for tree	1, 5, 10–13, 15–18, 18, 19, 23, 25–29, 31, 32, 36, 37, 39, 44, 45, 51, 53
if	1, 18, 18, 36, 37, 45
if have delayed	37, 54
where	1, 7, 18, 18, 21, 26, 35, 37, 45, 48, 50, 51
<b>R</b>	
r <i>(short step)</i>	43
rectangle fit value	28, 28
<i>(relative node name)</i>	17, 18, 22, 39, 41, 44
remove dynamic tree	40
repeat	1, 37, 39, 43
replace by dynamic tree	40
replace by' dynamic tree	40
replace by'' dynamic tree	40
reversed option	29, 30
right align value	24
root <i>(step)</i>	43
root'	38, 40
root' <i>(step)</i>	43
rotate	17, 26

S	
s <i>(short step)</i> .....	9, 43
s forest cs .....	44
s option .....	10, 26, 30, 30, 38, 39, 44
s sep option .....	1, 11, 11, 12, 13, 13, 15, 31, 50, 51, 53
set afterthought .....	22
set root dynamic tree .....	40, 40, 43
<i>(short step)</i>	
< .....	43
> .....	43
1 .....	9, 43
2 .....	9, 43
3 .....	9, 43
4 .....	9, 43
5 .....	9, 43
6 .....	9, 43
7 .....	9, 43
8 .....	9, 43
9 .....	9, 43
c .....	43
F .....	43
L .....	43
l .....	9, 43
N .....	43
n .....	9, 43
P .....	43
p .....	9, 43
r .....	43
s .....	9, 43
u .....	9, 17, 43
sibling <i>(step)</i> .....	43
stage	
compute xy .....	29–31, 38, 39
draw tree .....	26, 31, 34, 38, 39, 39
draw tree' .....	39, 39
pack .....	26, 29, 30, 38, 39
typeset nodes .....	27, 34, 38, 38
typeset nodes' .....	38
stages style .....	38, 39, 54
<i>(step)</i>	
current .....	29, 42
first .....	42
first leaf .....	42
group .....	42
id .....	42
last .....	42
last leaf .....	42
linear next .....	42
linear previous .....	42
n .....	42
n' .....	42
name .....	33, 42
next .....	42
next leaf .....	42
next on tier .....	42
node walk .....	42
parent .....	42, 51
previous .....	42
previous leaf .....	42
previous on tier .....	43
root .....	43
root' .....	43
sibling .....	43
to tier .....	43
trip .....	43
<i>(step)</i> .....	42
strcat .....	44, 51
strequal .....	44, 51
style	
compute xy stage .....	38
draw tree stage .....	38
math content .....	25, 54
pack stage .....	38
stages .....	38, 39, 54
typeset nodes stage .....	38
T	
t base value .....	25
TeX .....	35, 35, 35, 46, 50
TeX' .....	35, 46
TeX'' .....	35, 35, 46
tier option .....	5, 6, 6, 7, 7, 31, 35, 44, 45, 48, 50, 53
tight fit value .....	28, 28
tikz option .....	8, 9, 9, 10, 19, 35, 39, 45, 48, 51
tikzcsack package option .....	44, 47
tikzinstallkeys package option .....	47
to tier <i>(step)</i> .....	43
<i>(toks)</i> type .....	23, 23
top base value .....	14, 25
triangle .....	33
trip <i>(step)</i> .....	43
type	
<i>(autowrapped toks)</i> .....	23
<i>(boolean)</i> .....	23
<i>(count)</i> .....	23
<i>(dimen)</i> .....	23
<i>(keylist)</i> .....	23
<i>(toks)</i> .....	23, 23
typeset node .....	1, 38
typeset nodes stage .....	27, 34, 38, 38
typeset nodes stage style .....	38
typeset nodes' stage .....	38
typesetting nodes .....	24
U	
u <i>(short step)</i> .....	9, 17, 43
use as bounding box .....	35, 35
use as bounding box' .....	34, 35
W	
where propagator .....	
.....	1, 7, 18, 18, 21, 26, 35, 37, 45, 48, 50, 51
where key prefix .....	7, 18, 23, 37
where in key prefix .....	23, 29, 37
.wrap <i>n</i> pgfmath args handler .....	18, 41, 41, 42
.wrap 2 pgfmath args handler .....	18, 44, 51
.wrap 3 pgfmath args handler .....	18
.wrap pgfmath arg handler .....	18, 41, 42, 51
.wrap value handler .....	6, 23, 41, 42
X	
x option .....	26, 31, 38, 39
Y	
y option .....	1, 26, 31, 31, 38, 39