Forest: a pgf/Ti*k*Z-based package for drawing linguistic trees

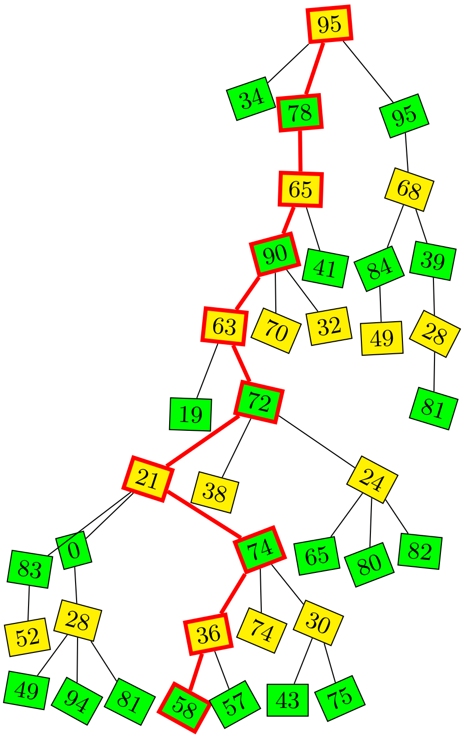
v1.03

Sašo Živanović[*]

January 28, 2013

Abstract

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

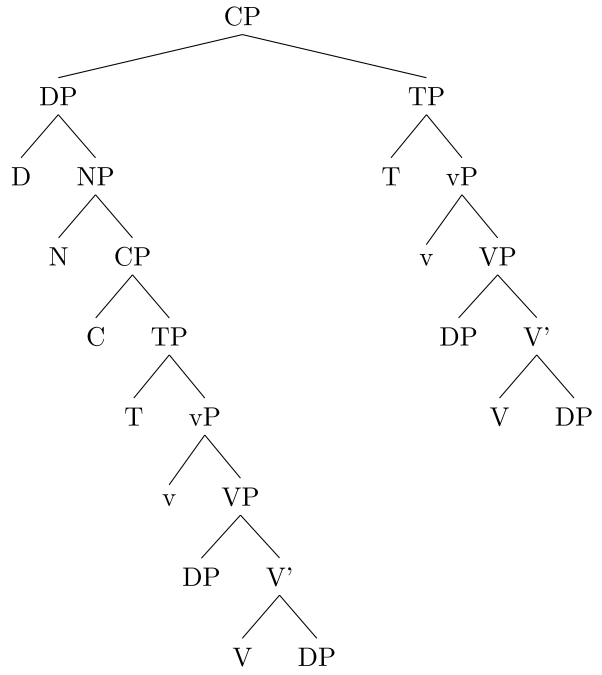
[*] e-mail: saso.zivanovic@guest.arnes.si; web: http://spj.ff.uni-lj.si/zivanovic/

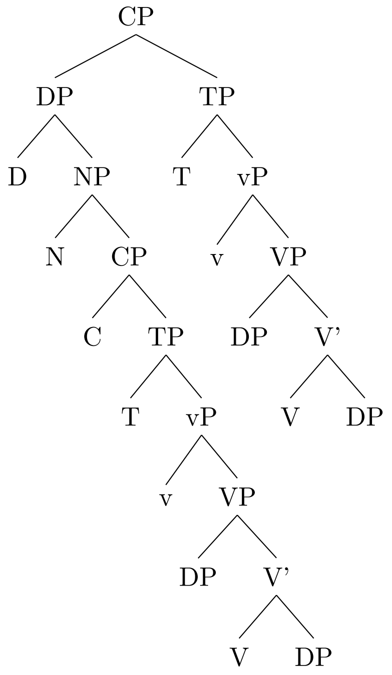
Contents

Part I
User's Guide

1 Introduction

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





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

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!

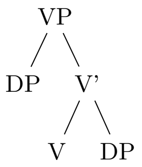
Forest is available at CTAN, and I have also started a style repository at GitHub.

2 Tutorial

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

2.1 Basic usage

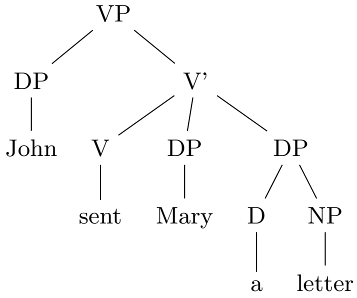
A tree is input by enclosing its specification in a `forest` environment. The tree is encoded by *the bracket syntax*: every node is enclosed in square brackets; the children of a node are given within its brackets, after its content.



```
\begin{forest}
  [VP
    [DP]
    [V’
      [V]
      [DP]
    ]
  ]
\end{forest}
```

(2)

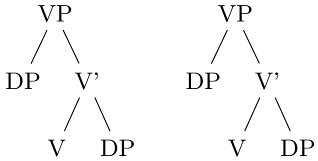
Binary trees are nice, but not the only thing this package can draw. Note that by default, the children are vertically centered with respect to their parent, i.e. the parent is vertically aligned with the midpoint between the first and the last child.



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

(3)

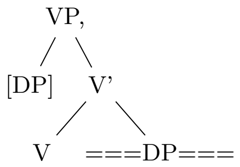
Spaces around brackets are ignored — format your code as you desire!



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

(4)

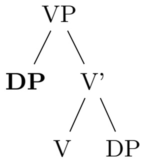
If you need a square bracket as part of a node’s content, use braces. The same is true for the other characters which have a special meaning in the Forest package: comma `,` and equality sign `=`.



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

(5)

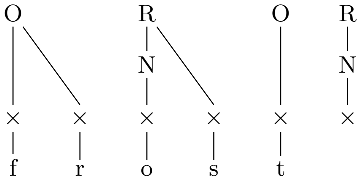
Macros in a node specification will be expanded when the node is drawn — you can freely use formatting commands inside nodes!



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

(6)

All the examples given above produced top-down trees with centered children. The other sections of this manual explain how various properties of a tree can be changed, making it possible to typeset radically different-looking trees. However, you don’t have to learn everything about this package to profit from its power. Using styles, you can draw predefined types of trees with ease. For example, a phonologist can use the `GP1` style from §4 to easily typeset (Government Phonology) phonological representations. The style is applied simply by writing its name before the first (opening) bracket of the tree.



```
\begin{forest} GP1 [
  [O[x[f]][x[r]]]
  [R[N[x[o]]][x[s]]]
  [O[x[t]]]
  [R[N[x]]]
]\end{forest}
```
(8)

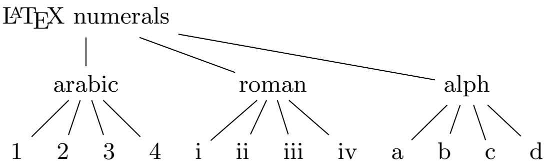
Of course, someone needs to develop the style — you, me, your local TEXnician . . . Furtunately, designing styles is not very difficult once you know your FOREST options. If you write one, please contribute!

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

2.2 Options

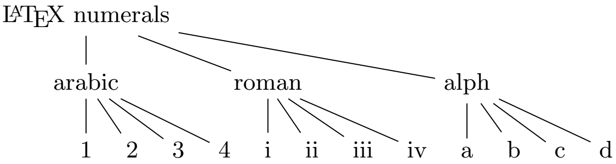
A node can be given various options, which control various properties of the node and the tree. For example, at the end of section 2.1, we have seen that the GP1 style vertically aligns the parent with the first child. This is achieved by setting option calign (for *c*hild-*align*ment) to first (child).

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



```
\begin{forest}
  [\LaTeX\ numerals, calign=first
    [arabic[1][2][3][4]]
    [roman[i][ii][iii][iv]]
    [alph[a][b][c][d]]
  ]
\end{forest}
```
(9)

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:



```
\begin{forest}
  [\LaTeX\ numerals,
      for tree={calign=first}
    [arabic[1][2][3][4]]
    [roman[i][ii][iii][iv]]
    [alph[a][b][c][d]]
  ]
\end{forest}
```
(10)

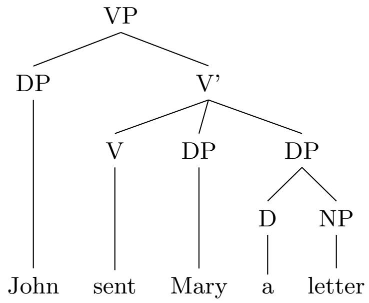
The value of propagator for tree is the option string that we want to process. This option string is propagated to all the nodes in the subtree[1] rooted in the current node (i.e. the node where for tree was given), including the node itself. (Propagator for descendants is just like for tree, only that it excludes the node itself. There are many other for . . . 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 2, §16.11]; this is the default) or a compass direction [see 2, §16.5.1]. (Note: the parent anchor determines where the edge from the child will arrive to this node, not where the node's edge to its parent will start!)

Option tier is what makes the skeletal points × in example (8) align horizontally although they occur at different levels in the logical structure of the tree. Using option tier is very simple: just set

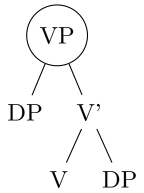
[1] It might be more precise to call this option for subtree . . . but this name at least saves some typing.

tier=tier name at all the nodes that you want to align horizontally. Any tier name will do, as long as the tier names of different tiers are different ... (Yes, you can have multiple tiers!)



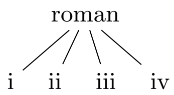
```
\begin{forest}                                                    (11)
  [VP, for tree={parent anchor=south, child anchor=north}
    [DP[John,tier=word]]
    [V'
      [V[sent,tier=word]]
      [DP[Mary,tier=word]]
      [DP[D[a,tier=word]][NP[letter,tier=word]]]
    ]
  ]
\end{forest}
```

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



```
\begin{forest}                                                    (12)
  [VP,circle,draw
    [DP][V'[V][DP]]
  ]
\end{forest}
```

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: × (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 LaTeX convert it to the other formats. We'll start with the easiest case of roman numerals: to get them, we can use the (plain) TeX command \romannumeral. To change the content of the node, we use option content. When specifying its new value, we can use #1 to insert the current content.[3]



```
\begin{forest}                                                    (13)
  [roman, delay={for children={content=\romannumeral#1}}
    [1][2][3][4]
  ]
\end{forest}
```

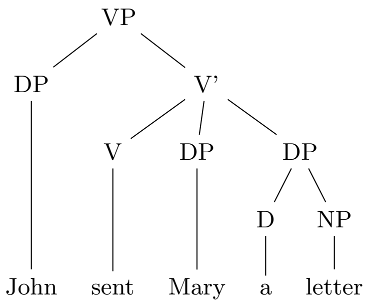
This example introduces another option: delay. Without it, the example wouldn't work: we would get arabic numerals. This is so because of the order in which the options are processed. The processing proceeds through the tree in a depth-first, parent-first fashion (first the parent is processed, and then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

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

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

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

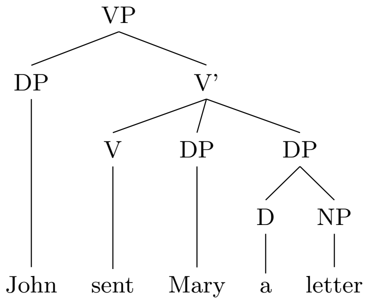
We are now ready to learn about simple conditionals. Every node option has the corresponding `if ...` and `where ...` keys. `if` ⟨*option*⟩=⟨*value*⟩⟨*true options*⟩⟨*false options*⟩ checks whether the value of ⟨*option*⟩ equals ⟨*value*⟩. If so, ⟨*true options*⟩ are processed, otherwise ⟨*false options*⟩. 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`.[4]



(14)

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

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.)



(15)

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

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.

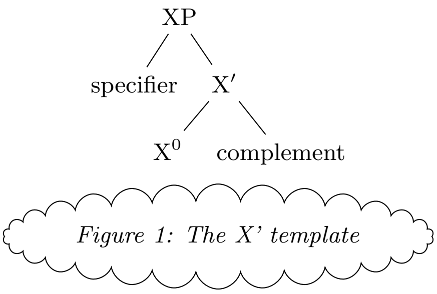
```
\forestset{
  sn edges/.style={for tree={parent anchor=south, child anchor=north}},
  background tree/.style={for tree={
              text opacity=0.2,draw opacity=0.2,edge={draw opacity=0.2}}}
}
```

You might have noticed that the last two examples contain options (actually, keys) even before the first opening bracket, contradicting 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

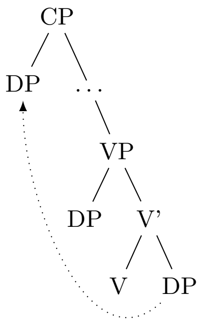
The tree can be decorated (think movement arrows) with arbitrary Ti*k*Z code.

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



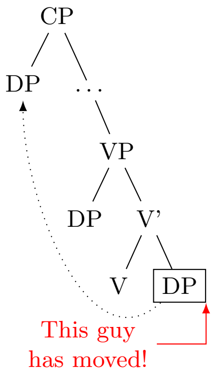
```
\begin{forest}                                                    (16)
  [XP
    [specifier]
    [X$'$
      [X$^0$]
      [complement]
    ]
  ]
  \node at (current bounding box.south)
    [below=1ex,draw,cloud,aspect=6,cloud puffs=30]
    {\emph{Figure 1: The X' template}};
\end{forest}
```

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 Ti*k*Z name using the `name` option, and then use this name in Ti*k*Z code as any other (Ti*k*Z) node name.



```
\begin{forest}                                                    (17)
  [CP
    [DP,name=spec CP]
    [\dots
      [,phantom]
      [VP
        [DP]
        [V'
          [V]
          [DP,name=object]]]]]
  \draw[->,dotted] (object) to[out=south west,in=south] (spec CP);
\end{forest}
```

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



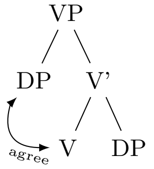
```
\begin{forest}                                                    (18)
  [CP
    [DP,name=spec CP]
    [\dots
      [,phantom]
      [VP
        [DP]
        [V'
          [V]
          [DP,draw] {
            \draw[->,dotted] () to[out=south west,in=south] (spec CP);
            \draw[<-,red] (.south east)--++(0em,-4ex)--++(-2em,0pt)
                 node[anchor=east,align=center]{This guy\\has moved!};
              }
        ]]]]
\end{forest}
```

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

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

A node walk is a concise[5] 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.

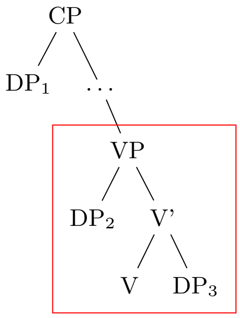
To see the node walk in action, consider the following examples. In the first example, the agree arrow connects the V node, specified simply as `()`, since the Ti*k*Z code follows `[V]`, and the DP node, which is described as “a sister of V’s parent”: `!us` = up + sibling.



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

(19)

The second example uses Ti*k*Z’s fitting library to compute the smallest rectangle containing node VP, its first child (DP$_2$) and its last grandchild (DP$_3$). The example also illustrates that the Ti*k*Z code can be specified via the “normal” option syntax, i.e. as a value to option `tikz`.[6]



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

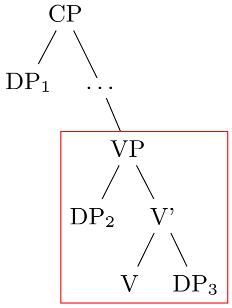
(21)

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*⟩.[7] The system thus computes and stores the positions of children using a coordinate

[5] Actually, Forest distinguishes two kinds of steps in node walks: long and short steps. This section introduces only short steps. See §3.5.1.

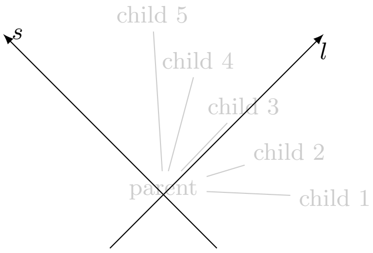
[6] Actually, there’s a simpler way to do this: use `fit to tree`!



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

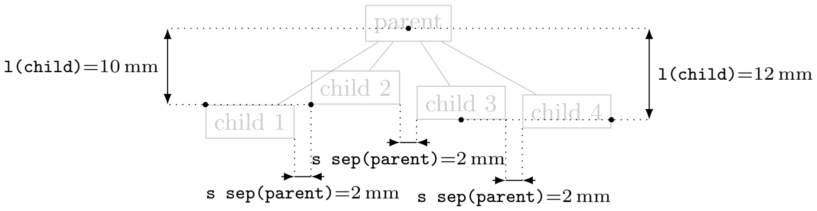
[7] The direction can be specified either in degrees (following the standard mathematical convention that 0 degrees is to the right, and that degrees increase counter-clockwise) or by the compass directions: `east`, `north east`, `north`, etc.

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 *l*evel, s for *s*ibling. The example shows the ls-coordinate system for a node with grow=45.



```
\begin{forest} background tree                                    (22)
  [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 [2, §16.5.1].) In the example below, positions of the anchors are shown by dots: observe that anchors of nodes with the same l are aligned and that the distances between the anchors of the children and the parent are as specified in the code.[8]



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

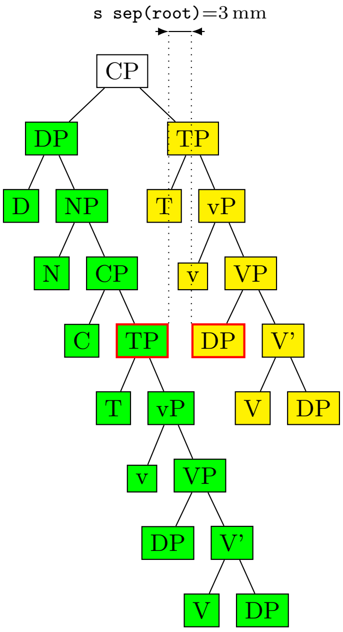
```
\newcommand\measurexdistance[5][####1]{\measurexorydistance{#2}{#3}{#4}{#5}{\x}{-|}{(5pt,0)}{#1}}
\newcommand\measureydistance[5][####1]{\measurexorydistance{#2}{#3}{#4}{#5}{\y}{|-}{(0,5pt)}{#1}}
\tikzset{dimension/.style={<->,>=latex,thin,every rectangle node/.style={midway,font=\scriptsize}},
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

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

(24)

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

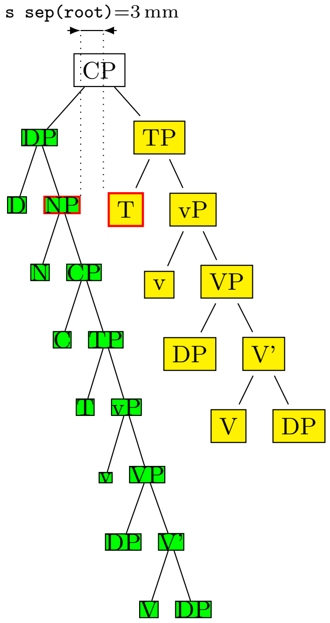
A fuller story is that s sep does not control the s-distance between two siblings, but rather the distance between the subtrees rooted in the siblings. When the green and the yellow child of the white node are s-positioned in the example below, the horizontal distance between the green and the yellow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



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

(25)

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

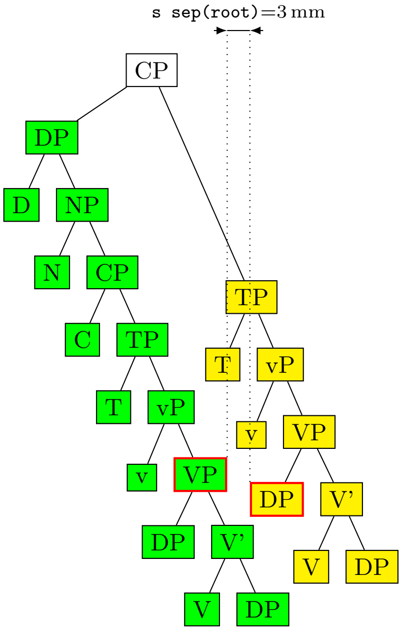


(26)

```
\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.

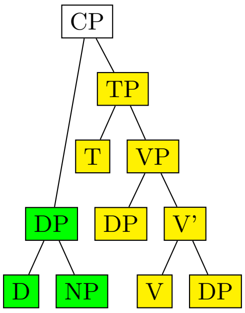
Let's play a bit and change the `l` of the root of the yellow subtree. Below, we set the vertical distance of the yellow TP to its parent to 3 cm: and the yellow submarine sinks diagonally … Now, the closest nodes are the higher yellow DP and the green VP.



(27)

```
\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.)

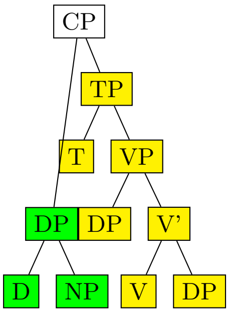


```
\begin{forest}
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D][NP]]
    [TP,for tree={fill=yellow}
      [T][VP[DP][V'[V][DP]]]]
  ]
\end{forest}
```

(28)

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

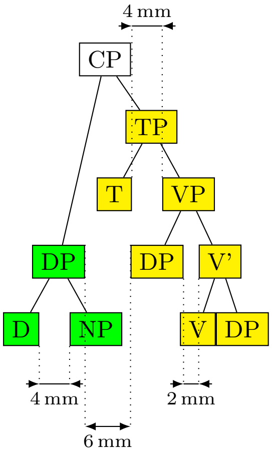
Let's now play with option `s sep`.



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

(29)

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



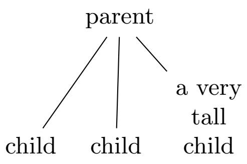
```
\begin{forest}
  for tree={s sep=(3-level)*2mm}
  [CP, for tree=draw
    [DP, for tree={fill=green},l*=3
      [D][NP]]
    [TP,for tree={fill=yellow}
      [T][VP[DP][V'[V][DP]]]]
  ]
  \measurexdistance{(!11.south east)}{(!12.south west)}{+(0,-5mm)}{below}
  \path(md2)-|coordinate(md)(!221.south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

(30)

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

As you have probably noticed, this example also demostrated that we can compute the value of an option using an (arbitrarily complex) formula. This is thanks to PGF's module `pgfmath`. FOREST provides an interface to `pgfmath` by defining `pgfmath` functions for every node option, and some other information, like the level we have used above, the number of children n children, the sequential number of the child n, etc. For details, see §3.6.

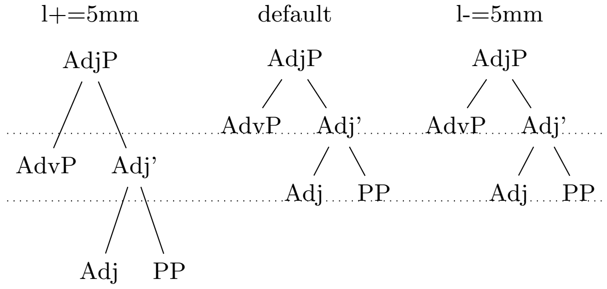
The final separation parameter is l sep. It determines the minimal separation of a node from its descendants. It the value of l is too small, then *all* the children (and thus their subtrees) are pushed away from the parent (by increasing their ls), so that the distance between the node's and each child's subtree boundary is at least l sep. The initial l can be too small for two reasons: either some child is too high, or the parent is too deep. The first problem is easier to see: we force the situation using a bottom-aligned multiline node. (Multiline nodes can be easily created using \\ as a line-separator. However, you must first specify the horizontal alignment using option align (see §3.3.1). Bottom vertical alignment is achieved by setting base=bottom; the default, unlike in Ti*k*Z, is base=top).

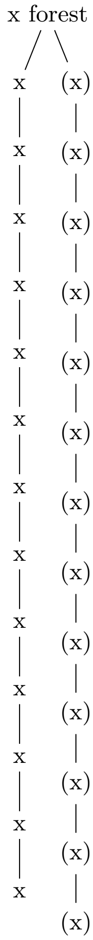


```
\begin{forest}
  [parent
    [child]
    [child]
    [a very\\tall\\child, align=center, base=bottom]
  ]
\end{forest}
```

(31)

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





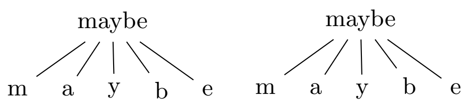
(32)

```
\begin{forest}
  [,phantom,for children={l sep=1ex,fit=band,
    for=1{edge’=,l=0},baseline}
    [{l+=5mm},for descendants/.pgfmath=content
      [AdjP[AdvP][Adj’[Adj][PP]]]]
    [default
      [AdjP[AdvP][Adj’[Adj][PP]]]]
    [{l-=5mm},for descendants/.pgfmath=content
      [AdjP[AdvP][Adj’[Adj][PP]]]]
  ]
  \path (current bounding box.west)|-coordinate(l1)(!212.base);
  \path (current bounding box.west)|-coordinate(l2)(!2121.base);
  \path (current bounding box.east)|-coordinate(r1)(!212.base);
  \path (current bounding box.east)|-coordinate(r2)(!2121.base);
  \draw[dotted] (l1)--(r1) (l2)--(r2);
\end{forest}
\hspace{4cm}
\raisebox{0pt}[\height][0pt]{\begin{forest}
  [x forest, baseline
    [x[x[x[x[x[x[x[x[x[x[x[x[x]]]]]]]]]]]]]
    [(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)[(x)]]]]]]]]]]]]]
  ]
\end{forest}}
```

2.4.1 The defaults, or the hairy details of vertical alignment

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

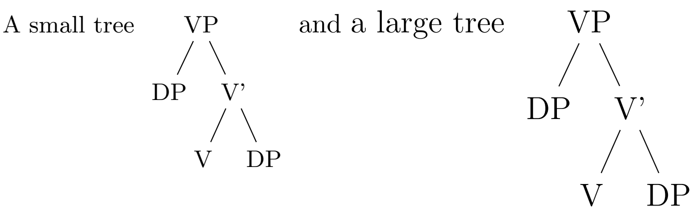
Let us start with the base alignment: Ti*k*Z's default is to anchor the nodes at their center, while Forest, given the usual content of nodes in linguistic representations, rather anchors them at the base [2, §16.5.1]. The difference is particularly clear for a "phonological" representation:



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

(33)

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



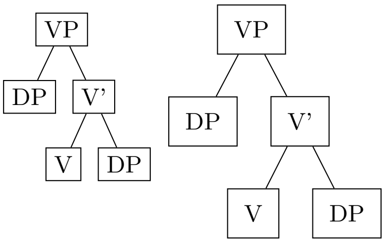
```
\hbox{\small A small tree
\begin{forest} baseline
  [VP[DP][V'[V][DP]]]
\end{forest}
\normalsize and
\large
a large tree
\begin{forest} baseline
  [VP[DP][V'[V][DP]]]
\end{forest}}
```

(34)

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

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

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



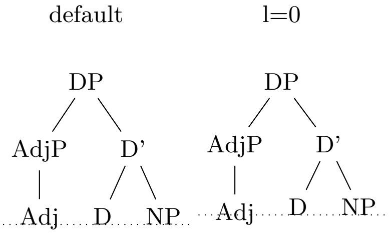
```
\begin{forest} baseline,for tree=draw
  [VP[DP][V'[V][DP]]]
\end{forest}
\pgfkeys{/pgf/inner sep=0.6666em}
\begin{forest} baseline,for tree=draw
  [VP[DP][V'[V][DP]]]
\end{forest}
```

(35)

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

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

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).

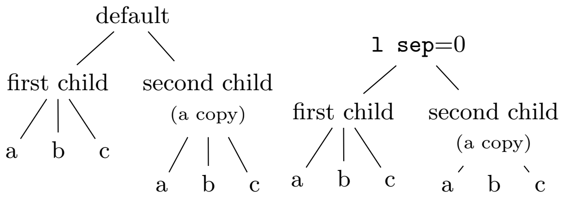


(36)

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

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 Adj*j*P, is pushed lower than the other nodes on level 2. If the content of the nodes is variable enough (various heights and depths), the cumulative effect can be quite strong, see the right tree of example (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 . . .



(37)

```
\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}
```

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

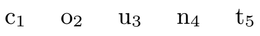
By default, Forest sets the standard node to be a node containing letters d and j. Linguistic representations consist mainly of letters, and in the TEX's default Computer Modern font, d is the highest letter (not character!), and j the deepest, so this decision guarantees that trees containing only letters will look nice. If the tree contains many parentheses, like the right tree of example (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 `l` by saying `l*=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.)

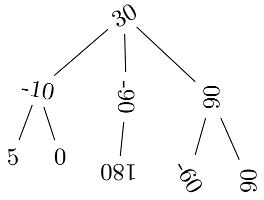


(38)

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

However, only options of the current node can be accessed using `\forestoption`. To access option values of other nodes, Forest's extensions to the PGF's mathematical library `pgfmath`, documented in [2, 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 (Ti*k*Z) option `rotate` is a full-fledged `pgfmath` expression yielding an integer in the range from -30 to 30. Similiarly, `l+` adds a random float in the $[-5, 5]$ range to the current value of `l`.

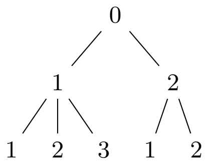
Example (30) demonstrated that information about the node, like the node's level, can be accessed within `pgfmath` expressions. All options are accessible in this way, i.e. every option has a corresponding `pgfmath` function. For example, we could rotate the node based on its content:



(39)

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

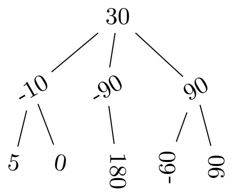
All numeric, dimensional and boolean options of Forest automatically pass the given value through `pgfmath`. If you need pass the value through `pgfmath` for a string option, use the `.pgfmath` handler. The following example sets the node's content to its child sequence number (the root has child sequence number 0).



(40)

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

As mentioned above, using `pgfmath` it is possible to access options of non-current nodes. This is achieved by providing the option function with a ⟨*relative node name*⟩ (see §3.5) argument.[9] In the next example, we rotate the node based on the content of its parent.



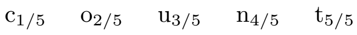
(41)

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

[9] 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 $n = 2, \ldots, 8$) 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.

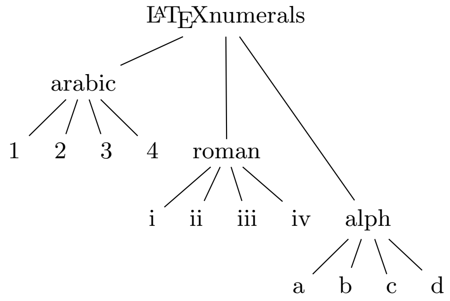


```
\begin{forest} [,phantom,delay={for descendants={
                content/.wrap 3 pgfmath args=
                {#1$_{#2/#3}$}{content}{n}{n_children("!u")}}}
  [c][o][u][n][t]]
\end{forest}
```

(42)

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

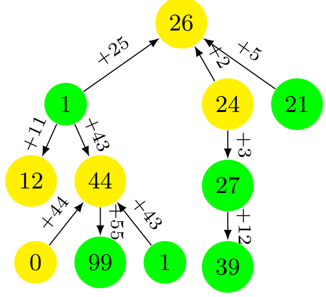
As another example, let's make the numerals example (9) a bit fancier. The numeral type is read off the parent's content and used to construct the appropriate control sequence (`\@arabic`, `\@roman` and `\@alph`). (Also, the numbers are not specified in content anymore: we simply read the sequence number `n`. And, to save some horizontal space for the code, each child of the root is pushed further down.)



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

(43)

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.

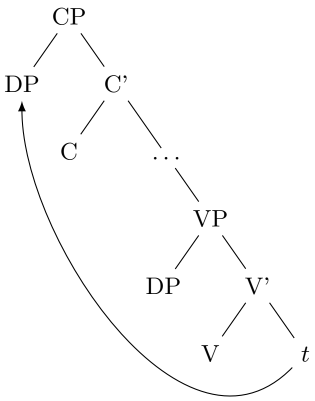


```
\pgfmathsetseed{314159}
\begin{forest}
  before typesetting nodes={
    for descendants={
      if={content()>content("!u")}{edge=->}{
        if={content()<content("!u")}{edge=<-}{}},
        edge label/.wrap pgfmath arg=
          {node[midway,above,sloped,font=\scriptsize]{+#1}}
          {int(abs(content()-content("!u")))}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

(44)

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

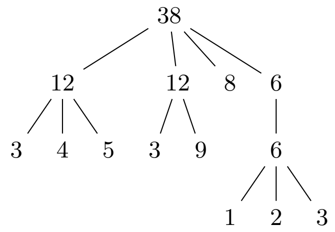
##1: #1 is the argument of the style move, i.e. the given node walk, while ##1 is the original option value (in this case, content).



(45)

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

In the following example, the content of the branching nodes is computed by Forest: a branching node is a sum of its children. 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.



(46)

```
\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}
```

2.6 Externalization

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

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

To use Forest's externalization facilities, say:[10]

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

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

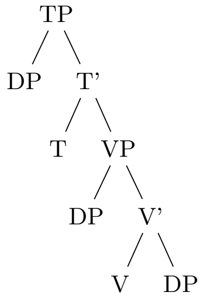
Ti*k*Z's externalization library promises a \label inside the externalized graphics to work out-of-box, while \ref inside the externalized graphics should work only if the externalization is run manually or by make [2, §32.4.1]. A bit surprisingly perhaps, the situation is roughly reversed in Forest. \ref inside the externalized graphics will work out-of-box. \label inside the externalized graphics will not work at

[10] 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 Ti*k*Z'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:[11]



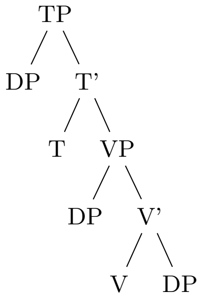
```
\bracketset{action character=@}
\def\XP#1#2#3{#1P[#2][#1'[#1][#3]]}
\begin{forest}
  [@\XP T{DP}{@\XP V{DP}{DP}}]
\end{forest}
```

(47)

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.

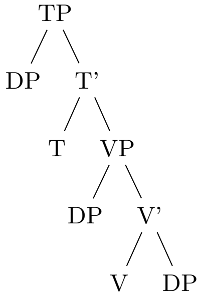
Let us continue with the description of the expansion control facilities of the bracket parser. The expandable token following the action character is expanded only once. Thus, if one defined macro `\VP` in terms of the general `\XP` and tried to use it in the same fashion as `\XP` above, he would fail.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



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

(48)

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.

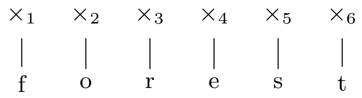


```
\bracketset{action character=@}
\def\XP#1#2#3{#1P[#2][#1'[#1][#3]]}
\def\VP#1#2{\XP V{#1}{#2}}
\begin{forest} @+
  [\XP T{DP}{\VP{DP}{DP}}]
\end{forest}
```

(49)

[11] 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 TEX's macro expansion. There is one final action command, `@@`, which yields control to the user code and expects it to call `\bracketResume` to resume parsing. This is useful to e.g. implement automatic node enumeration:



(50)

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

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

3 Reference

3.1 Environments

environment `\begin{forest}`⟨*tree*⟩`\end{forest}`

`\Forest`[*]{⟨*tree*⟩}

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

3.2 The bracket representation

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

⟨*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.

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

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

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` [2, §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.[12]

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 TeX code. ("TeX code" includes ⟨*value*⟩ expressions!).

  In the context of `\edef` or PGF's handler `.expanded` [2, §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.

[12] 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 TeX, 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*⟩ [1, 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*⟩ [1, 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 [2, §62.1];
2. use the key ⟨*option*⟩`'=`⟨*value*⟩ to invoke a normal TEX assignment.

The two above-mentioned speed-up tricks work for the augmented assignments as well. The keys for the second, TEX-only trick are: ⟨*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` [2, §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:[13]

`\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.

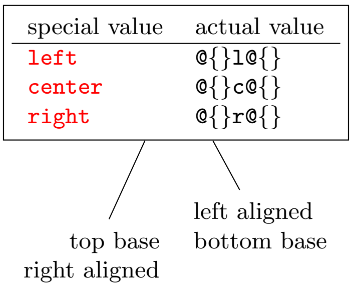
3.3.1 Node appearance

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 separated by `\\` and the columns (if any) by `&`, as usual.

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



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

```
\begin{forest} l sep+=2ex                                        (51)
  [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}
```

[13] `\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.

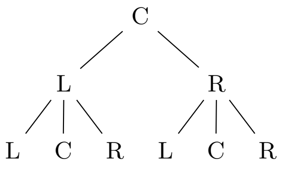
option `base`=⟨*toks: vertical alignment*⟩ `t`

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

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

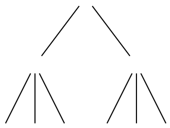
option `content`=⟨*autowrapped toks*⟩ The content of the node. `{}`

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



```
\begin{forest}                                                    (52)
  delay={for tree={
      if n=1{content=L}
           {if n'=1{content=R}
                    {content=C}}}}
  [[[] [] []] [[] [] []]]
\end{forest}
```

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



```
\begin{forest}                                                    (53)
  for tree={
      if n=1{content=L}
           {if n'=1{content=R}
                    {content=C}}}
  [[[] [] []] [[] [] []]]
\end{forest}
```

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

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

This is a fairly low level option, but sometimes you might still want to change its value. If you do so, take care of what is expanded when. 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}){\foresteoption{content format}};
```

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

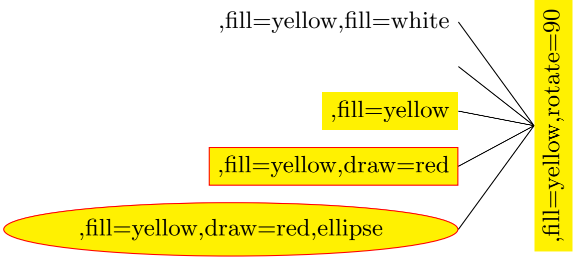
Important: the value of this option is first expanded using `\edef` and only then executed. Note that in its default value, `content format` is fully expanded using `\foresteoption`: this is necessary for complex content formats, such as `tabular` environments.

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

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 Ti*k*Z as options to the Ti*k*Z `node` operation [2, §16].

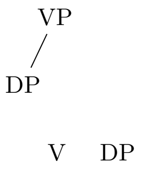
This option is rarely manipulated manually: almost all options unknown to Forest are automatically appended to `node options`. Exceptions are (i) `label` and `pin`, which require special attention in order to work; and (ii) `anchor`, which is saved in order to retain the information about the selected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



```
\begin{forest}
  for descendants={anchor=east,child anchor=east},
  grow=west,anchor=north,parent anchor=north,
  l sep=1cm,
  for tree={fill=yellow},where={n()>3}{draw=red}{},
  delay={for tree={content/.pgfmath=node_options}}
  [root,rotate=90,
    [,fill=white]
    [,node options']
    []
    []
    [,node options={ellipse}]
  ]
\end{forest}
```
(54)

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`.)



```
\begin{forest}
  [VP[DP][V',phantom[V][DP]]]
\end{forest}
```
(55)

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:* Ti*k*Z *anchor name*⟩ base

This is essentially a Ti*k*Z option [see 2, §16.5.1] — it is passed to Ti*k*Z 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 Ti*k*Z 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 first|last|center. 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.

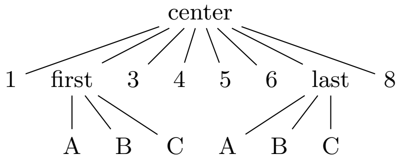
calign=first is an abbreviation for calign=child,calign child=1.

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}
```

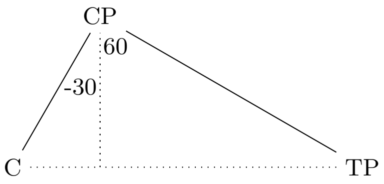
(56)

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}
```

(57)

`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`

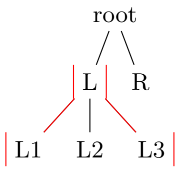
`calign with current` s-aligns the node anchors of the current node and its parent. This key is an abbreviation for:
`for parent/.wrap pgfmath arg={calign=child,calign primary child=##1}{n}`.

`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`

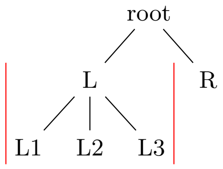
This option sets the type of the (s-)boundary that will be computed for the subtree rooted in the node, thereby determining how it will be packed into the subtree rooted in the node's parent. There are three choices:[14]

- `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.

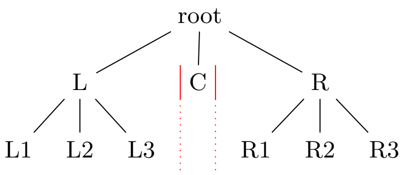


```
\begin{forest}                                    (60)
  delay={for tree={name/.pgfmath=content}}
  [root
    [L,fit=rectangle,
       show boundary
      [L1] [L2] [L3]]
    [R]
  ]
\end{forest}
```

[14] 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.



(61)

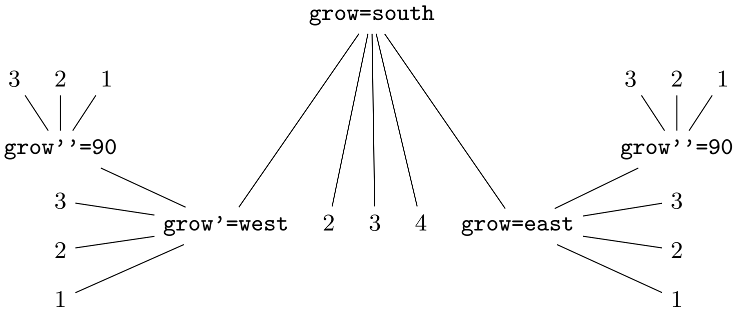
```
\begin{forest}
  delay={for tree={name/.pgfmath=content}}
  [root
    [L[L1][L2][L3]]
    [C,fit=band]
    [R[R1][R2][R3]]
  ]
  \draw[thin,red]
    (C.south west)--(C.north west)
    (C.north east)--(C.south east);
  \draw[thin,red,dotted]
    (C.south west)--+(0,-1)
    (C.south east)--+(0,-1);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

option grow=⟨*count*⟩ The direction of the tree's growth at the node. 270

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

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

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



(62)

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

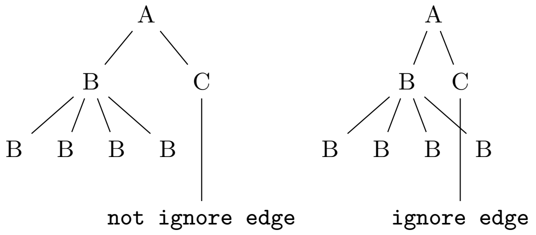
option ignore=⟨*boolean*⟩ false

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.)



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

(63)

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

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

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

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

- The value of l at the beginning of stage pack determines the minimal l-distance between the anchors of the node and its parent. Thus, changing l before packing will influence this process. (During packing, l can be increased due to parent's l sep, tier alignment, or calign 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

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

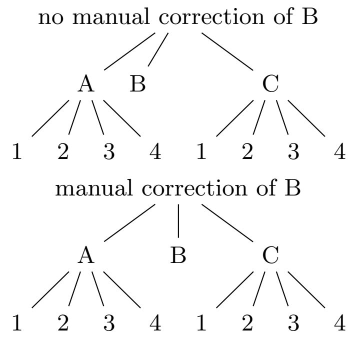
Note that despite the similar name, the semantics of l sep and s sep are quite different.

option reversed=⟨*boolean*⟩ false

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

option 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=⟨*dimen*⟩

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

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

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

option tier=⟨*toks*⟩ {}

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

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

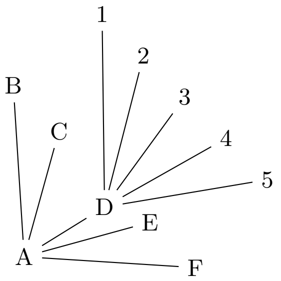
Tier alignment also does not work well with `calign=fixed (edge) angles`, because these child alignment methods may change the l-position of the children. When this might happen, Forest will yield a warning.

option x=⟨*dimen*⟩

option y=⟨*dimen*⟩

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

The values of `x` and `y` are computed in stage `compute xy`. It only makes sense to inspect and change them (for manual adjustments) afterwards (normally, in the `before drawing tree` hook, see §3.3.7.)



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

(65)

3.3.3 Edges

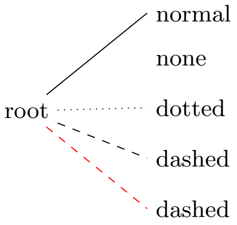
These options determine the shape and position of the edge from a node to its parent. They apply at stage `draw tree`.

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

option `edge`=⟨*keylist*⟩ `draw`

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

Also see key `no edge`.



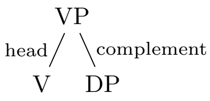
```
\begin{forest} for tree={grow'=0,l=2cm,anchor=west,child anchor=west},
  [root
    [normal]
    [none,no edge]
    [dotted,edge=dotted]
    [dashed,edge=dashed]
    [dashed,edge={dashed,red}]
  ]
\end{forest}
```

(66)

option `edge label`=⟨*toks:* Ti*k*Z *code*⟩ `{}`

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

The packing mechanism is not sensitive to edge labels.



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

(67)

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

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

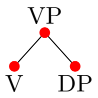
When setting this option, the values of options `edge` and `edge label` can be used in the edge path specification to include the values of options `edge` and `edge node`. Furthermore, two generic anchors, `parent anchor` and `child anchor`, are defined, to facilitate access to options `parent anchor` and `child anchor` from the Ti*k*Z code.

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

option `parent anchor`=⟨*toks:* Ti*k*Z *anchor*⟩ (Information also applies to option `child anchor`.) `{}`

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

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



```
\begin{forest}                                                        (68)
  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}
```

The empty value (which is the default) is interpreted as in Ti*k*Z: 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`=⟨*count*⟩) The internal id of the node.

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

option `max x`=⟨*dimen*⟩

option `max y`=⟨*dimen*⟩

option `min x`=⟨*dimen*⟩

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

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

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

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

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

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

3.3.5 Miscellaneous

`afterthought`=⟨*toks*⟩ Provides the afterthought explicitely.

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

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

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

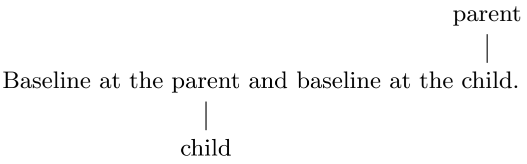
Aliases can be used as the ⟨*forest node name*⟩ part of a relative node name and as the argument to the `name` step of a node walk. The latter includes the usage as the argument of the `for name` propagator.

Technically speaking, Forest alias is *not* a Ti*k*Z alias! However, you can still use it as a "node name" in Ti*k*Z coordinates, since Forest hacks Ti*k*Z's implicit node coordinate system to accept relative node names; see §3.5.2.

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

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

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



Baseline at the parent and baseline at the child.

```
{\tikzexternaldisable                                    (69)
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}.}
```

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

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

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

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

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

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 TEX 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 Ti*k*Z node to the current node's subtree.

This key should be used like /tikz/fit of the Ti*k*Z's fitting library [see 2, §34]: as an option to Ti*k*Z*'s* node operation, the obvious restriction being that fit to 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:* Ti*k*Z *node*⟩ The current node is labelled by a Ti*k*Z node.

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

option `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 Ti*k*Z 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:* Ti*k*Z *node*⟩ The current node gets a pin, see [2, §16.10].

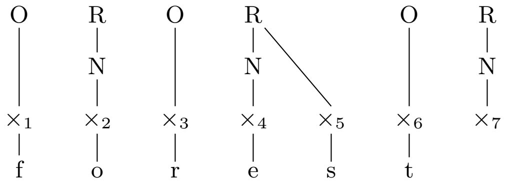
The technical details are the same as for `label`.

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

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

`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}
```
(70)

`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:* Ti*k*Z *code*⟩ "Decorations." `{}`

The code given as the value of this option will be included in the `tikzpicture` environment used to draw the tree. 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 ⟨*keylist*⟩ 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 ⟨*keylist*⟩ 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 ⟨*keylist*⟩ to other node(s) in the tree. (`for` and `for` ⟨*step*⟩ always pass the ⟨*keylist*⟩ to a single node.)

propagator `for`=⟨*node walk*⟩⟨*keylist*⟩ Processes ⟨*keylist*⟩ in the context of the final node in the ⟨*node walk*⟩ starting at the current node.

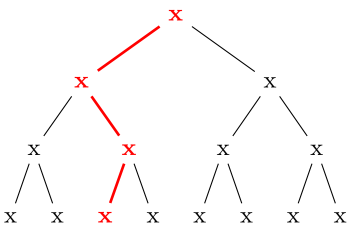
key prefix `for` ⟨*step*⟩=⟨*keylist*⟩ Walks a single-step node-walk ⟨*step*⟩ from the current node and passes the given ⟨*keylist*⟩ to the final (i.e. second) node.

⟨*step*⟩ must be a long node walk step; see §3.5.1. `for` ⟨*step*⟩=⟨*keylist*⟩ is equivalent to `for`=⟨*step*⟩keylist.

Examples: `for parent={l sep+=3mm}`, `for n=2{circle,draw}`.

propagator `for ancestors`=⟨*keylist*⟩

propagator `for ancestors'`=⟨*keylist*⟩ Passes the ⟨*keylist*⟩ to itself, too.



(71)

```
\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}
```

propagator `for all next`=⟨*keylist*⟩ Passes the ⟨*keylist*⟩ to all the following siblings.

propagator `for all previous`=⟨*keylist*⟩ Passes the ⟨*keylist*⟩ to all the preceding siblings.

propagator `for children`=⟨*keylist*⟩

propagator `for descendants`=⟨*keylist*⟩

propagator `for tree`=⟨*keylist*⟩

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`=⟨*pgfmath condition*⟩⟨*true keylist*⟩⟨*false keylist*⟩

If ⟨*pgfmath condition*⟩ evaluates to `true` (non-zero), ⟨*true keylist*⟩ is processed (in the context of the current node); otherwise, ⟨*false keylist*⟩ is processed.

For a detailed description of `pgfmath` expressions, see [2, part VI]. (In short: write the usual mathematical expressions.)

key prefix `if ⟨option⟩=⟨value⟩⟨true keylist⟩⟨false keylist⟩`

A simple conditional is defined for every ⟨*option*⟩: if ⟨*value*⟩ equals the value of the option at the current node, ⟨*true keylist*⟩ is executed; otherwise, ⟨*false keylist*⟩.

propagator `where=⟨value⟩⟨true keylist⟩⟨false keylist⟩`

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

key prefix `where ⟨option⟩=⟨value⟩⟨true keylist⟩⟨false keylist⟩`

Executes simple conditional `if ⟨option⟩` for every node in the current subtree.

key prefix `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*⟩.

This conditional is defined only for ⟨*toks*⟩ options, see §3.3.

key prefix `where in ⟨toks option⟩=⟨toks⟩⟨true keylist⟩⟨false keylist⟩`

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.

This conditional is defined only for ⟨*toks*⟩ 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=⟨keylist⟩` Defers the processing of the ⟨*keylist*⟩ until the next cycle.

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

propagator `if have delayed=⟨true keylist⟩⟨false keylist⟩` If any options were delayed in the current cycle (more precisely, up to the point of the execution of this key), process ⟨*true keylist*⟩, otherwise process ⟨*false keylist*⟩. (`delay n` will trigger "true" for the intermediate cycles.)

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

propagator `before packing=⟨keylist⟩` Defers the processing of the ⟨*keylist*⟩ to until just before the nodes are packed.

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

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

Other propagators

`repeat=⟨number⟩⟨keylist⟩` The ⟨*keylist*⟩ is processed ⟨*number*⟩ times.

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

`process keylist`=⟨*keylist option name*⟩ Processes the keylist saved in option ⟨*keylist option name*⟩ 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 ⟨*keylist option name*⟩ 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 in 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:

⟨*empty*⟩: uses the last (non-integrated, i.e. created/removed/replaced) node.

⟨*node*⟩: a new node is created using the given bracket representation (the node may contain children, i.e. a tree may be specified), and used as the argument to the key.

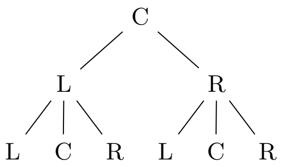
The bracket representation must be enclosed in brackets, which will usually be enclosed in braces to prevent them being parsed while parsing the "host tree."

⟨*relative node name*⟩: the node ⟨*relative node name*⟩ resolves to will be used.

Here is the list of dynamic tree keys:

dynamic tree `append`=⟨*empty*⟩ | `[`⟨*node*⟩`]` | ⟨*relative node name*⟩

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



```
\begin{forest}                                          (72)
  before typesetting nodes={for tree={
    if n=1{content=L}
         {if n'=1{content=R}
                  {content=C}}}}
  [,repeat=2{append={[
    ,repeat=3{append={[]}}
  ]}}]
\end{forest}
```

dynamic tree `create=[⟨node⟩]`

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

dynamic tree `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.

It 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.

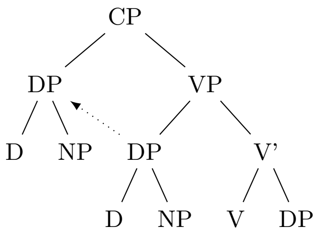
dynamic tree `append''`, `insert after''`, `insert before''`, `prepend''`, `replace by''`

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**=⟨*empty*⟩ | ⟨*macro definition*⟩ ⟨*empty*⟩

Defines a template for constructing the `name` of the copy from the name of the original. ⟨*macro definition*⟩ should be either empty (then, the `name` is constructed from the `id`, as usual), or an expandable macro taking one argument (the name of the original).

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



```
\begin{forest}                                              (73)
  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}
```

A dynamic tree operation is made in two steps:

- If the argument is given by a ⟨*node*⟩ argument, the new node is created immediately, i.e. while the dynamic tree key is being processed. Any options of the new node are implicitely `delay`ed.
- The requested changes in the tree structure are actually made between the cycles of keylist processing.

→ Such a two-stage approach is employed because changing the tree structure during the dynamic tree key processing would lead to an unmanageable order of keylist processing.

→ A consequence of this approach is that nested dynamic tree keys take several cycles to complete. Therefore, be careful when using `delay` and dynamic tree keys simultaneously: in such a case, it is often safer to use `before typesetting nodes` instead of `delay`, see example (72).

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

3.4 Handlers

handler **.pgfmath**=⟨*pgfmath expression*⟩

The result is the evaluation of ⟨*pgfmath expression*⟩ in the context of the current node.

handler **.wrap value**=⟨*macro definition*⟩

The result is the (single) expansion of the given ⟨*macro definition*⟩. The defined macro takes one parameter. The current value of the handled option will be passed as that parameter.

handler **.wrap *n* pgfmath args**=⟨*macro definition*⟩⟨*arg* 1⟩. . . ⟨*arg n*⟩

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

handler **.wrap pgfmath arg**=⟨*macro definition*⟩⟨*arg*⟩

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

3.5 Relative node names

⟨*relative node name*⟩=[⟨*forest node name*⟩][!⟨*node walk*⟩]

⟨*relative node name*⟩ refers to the Forest node at the end of the ⟨*node walk*⟩ starting at node named ⟨*forest node name*⟩. If ⟨*forest node name*⟩ is omitted, the walk starts at the current node. If ⟨*node walk*⟩ is omitted, the "walk" ends at the start node. (Thus, an empty ⟨*relative node name*⟩ 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[15]

⟨*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 nth child; counting starts at 1 (not 0)

⟨*step*⟩ `n'`=n the nth 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

[15] 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, Ti*k*Z's implicit node coordinate system [2, §13.2.3] is hacked to accept relative node names.[16].

The explicit forest coordinate system is called simply forest and used like this: (forest cs:⟨*forest cs spec*⟩); see [2, §13.2.5]. ⟨*forest cs spec*⟩ is a keylist; the following keys are accepted.

forest cs name=⟨*node name*⟩ The node with the given name becomed the current node. The resulting point is its (node) anchor.

forest cs id=⟨*node id*⟩ The node with the given name becomed 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*⟩].

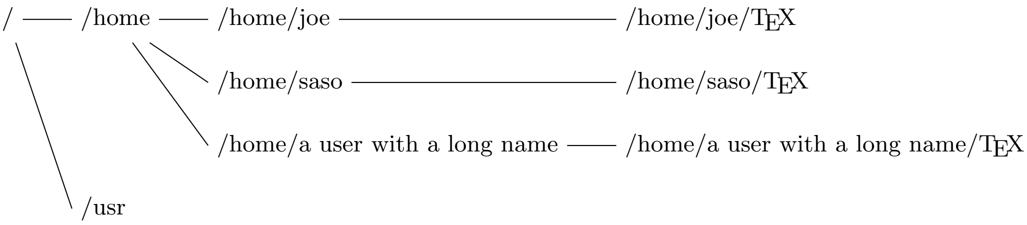
3.6 New pgfmath functions

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().[17]

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.

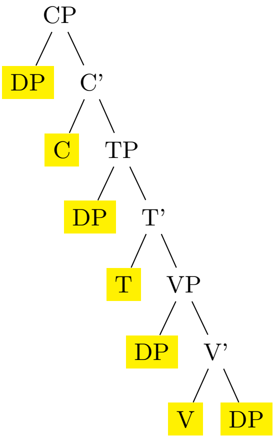
Some random notes on pgfmath: (i) &&, || and ! are boolean "and", "or" and "not", respectively. (ii) The equality operator (for numbers and dimensions) is ==, *not* =. And some examples:


/
/home
/home/joe
/home/joe/TEX
/home/saso
/home/saso/TEX
/home/a user with a long name
/home/a user with a long name/TEX
/usr

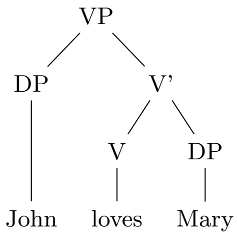
[16] Actually, the hack can be switched on and off on the fly, using \ifforesttikzcshack.

[17] 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}                                                          (74)
  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}                                                          (75)
  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}                                                          (76)
  where n children=0{tier=word,
    if={instr("!P",content("!u"))}{no edge,
      tikz={\draw (!.north west)--
      (!.north east)--(!u.south)--cycle;
    }}{}
  }{},
  [VP[DP[John]][V'[V[loves]][DP[Mary]]]]
\end{forest}
```

3.7 Standard node

\forestStandardNode⟨*node*⟩⟨*environment fingerprint*⟩⟨*calibration procedure*⟩⟨*exported options*⟩

This macro defines the current *standard node*. The standard node declares some options as *exported*. When a new node is created, the values of the exported options are initialized from the standard node. At the beginning of every forest environment, it is checked whether the *environment fingerprint* of the standard node has changed. If it did, the standard node is *calibrated*, adjusting the values of exported options. The *raison d'etre* for such a system is given in §2.4.1.

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

The ⟨*environment fingerprint*⟩ must be an expandable macro definition. It's expansion should change whenever the calibration is necessary.

⟨*calibration procedure*⟩ is a keylist (processed in the /forest path) which calculates the values of exported options.

⟨*exported options*⟩ is a comma-separated list of exported options.

This is how the default standard node is created:

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

3.8 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. Ti*k*Z's externalization library is automatically loaded if necessary.

external/optimize Parallels /tikz/external/optimize: if true (the default), the processing of non-current trees is skipped during the embedded compilation.

external/context If the expansion of the macro stored in this option changes, the tree is recompiled.

external/depends on macro=⟨*cs*⟩ Adds the definition of macro ⟨*cs*⟩ to external/context. Thus, if the definition of ⟨*cs*⟩ is changed, the tree will be recompiled.

Forest respects or is compatible with several (not all) keys and commands of Ti*k*Z's externalization library. In particular, the following keys and commands might be useful; see [2, §32].

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

Forest does not disturbe the externalization of non-Forest pictures. (At least it shouldn't ...)

The main auxiliary file for externalization has suffix .for. The externalized pictures have suffices -forest-*n* (their prefix can be set by /tikz/external/prefix, e.g. to a subdirectory). Information on all trees that were ever externalized in the document (even if they were changed or deleted) is kept. If you need a "clean" .for file, delete it and recompile. Deleting -forest-*n*.pdf will result in recompilation of a specific tree.

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

3.9 Package options

package option `external=true|false` false

Enable/disable externalization, see §3.8.

package option `tikzcshack=true|false` true

Enable/disable the hack into Ti*k*Z's implicite coordinate syntax hacked, see §3.5.

package option `tikzinstallkeys=true|false` true

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

4 Gallery

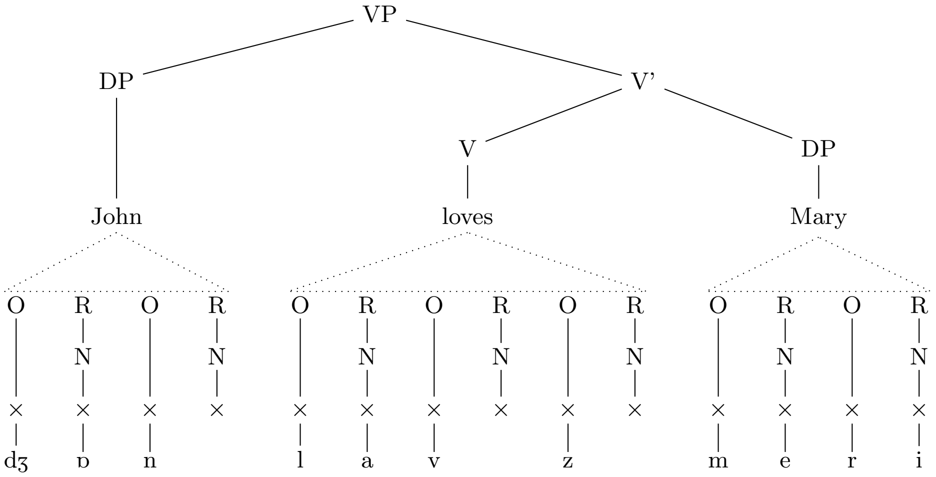
4.1 Styles

`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!

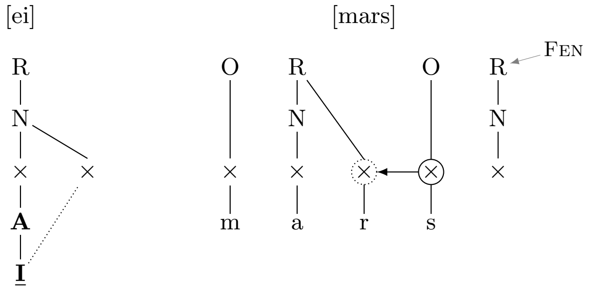
```
\newbox\standardnodestrutbox
\setbox\standardnodestrutbox=\hbox to 0pt{\phantom{\forestOve{standard node}{content}}}
\def\standardnodestrut{\copy\standardnodestrutbox}
\forestset{
  GP1/.style 2 args={
    for n={1}{baseline},
    s sep=0pt, l sep=0pt,
    for descendants={
      l sep=0pt, l={#1},
      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}}},
  el/.style={content=\textsc{\textbf{##1}}},
  head/.style={content=\textsc{\textbf{\underline{##1}}}}
}
```

An example of an “embedded” `GP1` style:

```
\begin{forest}                                                    (77)
  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.

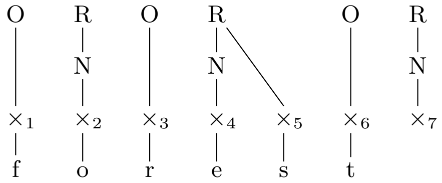


```
\begin{forest}[,phantom,s sep=1cm                                 (78)
  [{[ei]}, GP1
    [R[N[x[A,el[I,head,associate=N]]][x]]
  ]
  [{[mars]}, GP1
    [O[x[m]]]
    [R[N[x[a]]][x,encircle,densely dotted[r]]]
    [O[x,encircle,govern=<[s]]]
    [R,fen[N[x]]]
  ]
]\end{forest}
```

rlap and llap The FOREST versions of TEX's `\rlap` and `\llap`: the "content" added by these styles will influence neither the packing algorithm nor the anchor positions.

(79)

```
\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.

(80)

```
\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
```

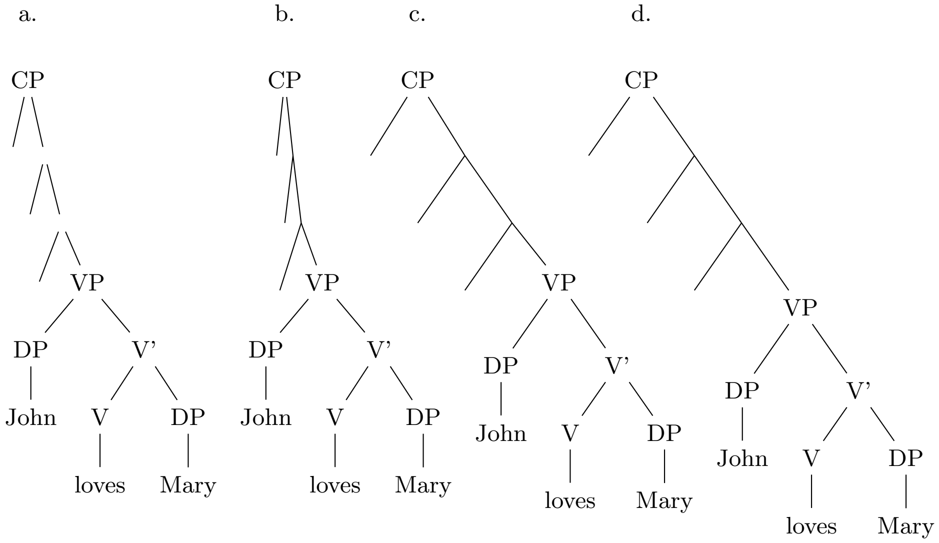
nice empty nodes We often need empty nodes: tree (a) shows how they look like by default: ugly.

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).

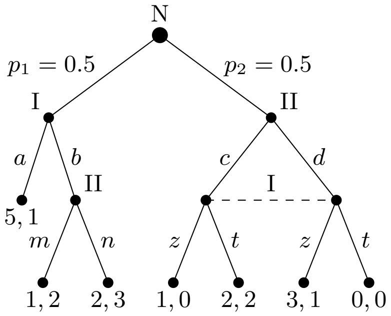
However, at the transitions from empty to non-empty nodes, tree (d) above seems to zigzag (although the base points of the spine nodes are perfectly in line), and the edge to the empty node left to VP seems too long (it reaches to the level of VP's base, while we'd prefer it to stop at the same level 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{                                                                                    (81)
  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,                                                                    %(a)
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
    [CP, delay={where content={}{shape=coordinate}{}}                       %(b)
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
    [CP, for tree={calign=fixed angles},                                      %(c)
         delay={where content={}{shape=coordinate}{}}
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
    [CP, nice empty nodes                                                   %(d)
      [][[][[][VP[DP[John]][V'[V[loves]][DP[Mary]]]]]]]
  ]
\end{forest}
```



4.2 Examples

The following example was inspired by a question on TEX Stackexchange: How to change the level distance in tikz-qtree for one level only?. The question is about `tikz-qtree`: how to adjust the level distance for the first level only, in order to avoid first-level labels crossing the parent–child edge. While this example solves the problem (by manually shifting the offending labels; see `elo` below), it does more: the preamble is setup so that inputing the tree is very easy.



```
\def\getfirst#1;#2\endget{#1}                                                              (82)
\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.

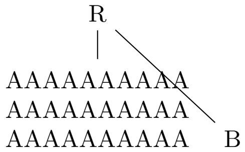
System requirements This package requires LaTeX and eTeX. If you use something else: sorry.

The requirement for LaTeX might be dropped in the future, when I get some time and energy for a code-cleanup (read: to remedy the consequences of my bad programming practices and general disorganization).

The requirement for eTeX will probably stay. If nothing else, Forest is heavy on boxes: every node requires its own . . . and consequently, I have freely used eTeX constructs in the code . . .

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

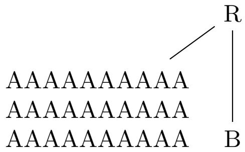
Edges cutting through sibling nodes In the following example, the R–B edge crosses the AAA node, although `ignore edge` is set to the default `false`.



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

(83)

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

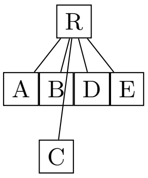


```
\begin{forest}
  calign=last
  [R[AAAAAAAAAA\\AAAAAAAAAA\\AAAAAAAAAA,align=center,base=bottom][B]]
\end{forest}
```

(84)

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

Orphans If the `l` coordinates of adjacent children are too different (as a result of manual adjustment or tier alignment), the packing algorithm might have nothing so say about the desired distance between them: in this sense, node C below is an "orphan."

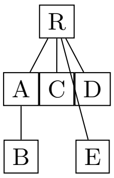


```
\begin{forest}
  for tree={s sep=0,draw},
  [R[A][B][C,l*=2][D][E]]
\end{forest}
```

(85)

To prevent orphans from ending up just anywhere, I have decided to vertically align them with their preceding sibling — although I'm not certain that's really the best solution. In other words, you can rely that the sequence of s-coordinates of siblings is non-decreasing.

The decision also incluences 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

v1.03 (2013/01/28)

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

v1.02 (2013/01/20)

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

v1.01 (2012/11/14)

- Compatibility with the `standalone` package: temporarily disable the effect of `standalone`'s package option `tikz` while typesetting nodes.
- Require at least the [2010/08/21] (v2.0) release of package `etoolbox`.
- Require version [2010/10/13] (v2.10, rcs-revision 1.76) of PGF/Ti*k*Z. Future compatibility: adjust to the change of the "not yet positioned" node name (2.10 `@` → 2.10-csv `PGFINTERNAL`).
- Add this changelog.

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

Acknowledgements Many thanks to the people who have reported bugs! In the chronological order: Markus Pöchtrager, Timothy Dozat, Ignasi Furio.[18]

[18] 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.

```
\ProvidesPackage{forest}[2013/01/28 v1.03 Drawing (linguistic) trees]

\RequirePackage{tikz}[2010/10/13]
\usetikzlibrary{shapes}
\usetikzlibrary{fit}
\usetikzlibrary{calc}
\usepgflibrary{intersections}

\RequirePackage{pgfopts}
\RequirePackage{etoolbox}[2010/08/21]
\RequirePackage{environ}

%\usepackage[trace]{trace-pgfkeys}
```

`/forest` is the root of the key hierarchy.

```
\pgfkeys{/forest/.is family}
\def\forestset#1{\pgfqkeys{/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.

```
\long\def\forest@original@pgfkeysdefnargs@#1#2#3#4{%
  \ifcase#2\relax
  \pgfkeyssetvalue{#1/.@args}{}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8##9}%
  \else
  \pgfkeys@error{\string\pgfkeysdefnargs: expected  <= 9 arguments, got #2}%
  \fi
  \pgfkeysgetvalue{#1/.@args}\pgfkeys@tempargs
  \def\pgfkeys@temp{\expandafter#4\csname pgfk@#1/.@@body\endcsname}%
  \expandafter\pgfkeys@temp\pgfkeys@tempargs{#3}%
  % eliminate the \pgfeov at the end such that TeX gobbles spaces
  % by using
  % \pgfkeysdef{#1}{\pgfkeysvalueof{#1/.@@body}##1}
  % (with expansion of ’#1’):
  \edef\pgfkeys@tempargs{\noexpand\pgfkeysvalueof{#1/.@@body}}%
  \def\pgfkeys@temp{\pgfkeysdef{#1}}%
  \expandafter\pgfkeys@temp\expandafter{\pgfkeys@tempargs##1}%
  \pgfkeyssetvalue{#1/.@body}{#3}%
}

\long\def\forest@patched@pgfkeysdefnargs@#1#2#3#4{%
  \ifcase#2\relax
  \pgfkeyssetvalue{#1/.@args}{}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
  %%%%% removed:
  %%%%% \or
  %%%%% \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8}%
  \or
  \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8##9}%
  \else
  \pgfkeys@error{\string\pgfkeysdefnargs: expected  <= 9 arguments, got #2}%
  \fi
  \pgfkeysgetvalue{#1/.@args}\pgfkeys@tempargs
  \def\pgfkeys@temp{\expandafter#4\csname pgfk@#1/.@@body\endcsname}%
  \expandafter\pgfkeys@temp\pgfkeys@tempargs{#3}%
  % eliminate the \pgfeov at the end such that TeX gobbles spaces
  % by using
  % \pgfkeysdef{#1}{\pgfkeysvalueof{#1/.@@body}##1}
  % (with expansion of ’#1’):
  \edef\pgfkeys@tempargs{\noexpand\pgfkeysvalueof{#1/.@@body}}%
  \def\pgfkeys@temp{\pgfkeysdef{#1}}%
  \expandafter\pgfkeys@temp\expandafter{\pgfkeys@tempargs##1}%
  \pgfkeyssetvalue{#1/.@body}{#3}%
}
\ifx\pgfkeysdefnargs@\forest@original@pgfkeysdefnargs@
  \let\pgfkeysdefnargs@\forest@patched@pgfkeysdefnargs@
\fi
```

Minor: a leaking space in the very first line.

```
\def\forest@original@pgfpointintersectionoflines#1#2#3#4{%
  {
    %
    % Compute orthogonal vector to #1--#2
    %
    \pgf@process{#2}%
    \pgf@xa=\pgf@x%
    \pgf@ya=\pgf@y%
```

```
\pgf@process{#1}%
\advance\pgf@xa by-\pgf@x%
\advance\pgf@ya by-\pgf@y%
\pgf@ya=-\pgf@ya%
% Normalise a bit
\c@pgf@counta=\pgf@xa%
\ifnum\c@pgf@counta<0\relax%
  \c@pgf@counta=-\c@pgf@counta\relax%
\fi%
\c@pgf@countb=\pgf@ya%
\ifnum\c@pgf@countb<0\relax%
  \c@pgf@countb=-\c@pgf@countb\relax%
\fi%
\advance\c@pgf@counta by\c@pgf@countb\relax%
\divide\c@pgf@counta by 65536\relax%
\ifnum\c@pgf@counta>0\relax%
  \divide\pgf@xa by\c@pgf@counta\relax%
  \divide\pgf@ya by\c@pgf@counta\relax%
\fi%
%
% Compute projection
%
\pgf@xc=\pgf@sys@tonumber{\pgf@ya}\pgf@x%
\advance\pgf@xc by\pgf@sys@tonumber{\pgf@xa}\pgf@y%
%
% The orthogonal vector is (\pgf@ya,\pgf@xa)
%
%
% Compute orthogonal vector to #3--#4
%
\pgf@process{#4}%
\pgf@xb=\pgf@x%
\pgf@yb=\pgf@y%
\pgf@process{#3}%
\advance\pgf@xb by-\pgf@x%
\advance\pgf@yb by-\pgf@y%
\pgf@yb=-\pgf@yb%
% Normalise a bit
\c@pgf@counta=\pgf@xb%
\ifnum\c@pgf@counta<0\relax%
  \c@pgf@counta=-\c@pgf@counta\relax%
\fi%
\c@pgf@countb=\pgf@yb%
\ifnum\c@pgf@countb<0\relax%
  \c@pgf@countb=-\c@pgf@countb\relax%
\fi%
\advance\c@pgf@counta by\c@pgf@countb\relax%
\divide\c@pgf@counta by 65536\relax%
\ifnum\c@pgf@counta>0\relax%
  \divide\pgf@xb by\c@pgf@counta\relax%
  \divide\pgf@yb by\c@pgf@counta\relax%
\fi%
%
% Compute projection
%
\pgf@yc=\pgf@sys@tonumber{\pgf@yb}\pgf@x%
\advance\pgf@yc by\pgf@sys@tonumber{\pgf@xb}\pgf@y%
%
% The orthogonal vector is (\pgf@yb,\pgf@xb)
%
% Setup transformation matrx (this is just to use the matrix
    % inversion)
    %
    \pgfsettransform{{\pgf@sys@tonumber\pgf@ya}{\pgf@sys@tonumber\pgf@yb}{\pgf@sys@tonumber\pgf@xa}{\pgf@sys@
    \pgftransforminvert%
    \pgf@process{\pgfpointtransformed{\pgfpoint{\pgf@xc}{\pgf@yc}}}%
  }%
}
\def\forest@patched@pgfpointintersectionoflines#1#2#3#4{%
  {% added the percent sign in this line
    %
    % Compute orthogonal vector to #1--#2
    %
    \pgf@process{#2}%
    \pgf@xa=\pgf@x%
    \pgf@ya=\pgf@y%
    \pgf@process{#1}%
    \advance\pgf@xa by-\pgf@x%
    \advance\pgf@ya by-\pgf@y%
    \pgf@ya=-\pgf@ya%
    % Normalise a bit
    \c@pgf@counta=\pgf@xa%
    \ifnum\c@pgf@counta<0\relax%
      \c@pgf@counta=-\c@pgf@counta\relax%
    \fi%
    \c@pgf@countb=\pgf@ya%
    \ifnum\c@pgf@countb<0\relax%
      \c@pgf@countb=-\c@pgf@countb\relax%
    \fi%
    \advance\c@pgf@counta by\c@pgf@countb\relax%
    \divide\c@pgf@counta by 65536\relax%
    \ifnum\c@pgf@counta>0\relax%
      \divide\pgf@xa by\c@pgf@counta\relax%
      \divide\pgf@ya by\c@pgf@counta\relax%
    \fi%
    %
    % Compute projection
    %
    \pgf@xc=\pgf@sys@tonumber{\pgf@ya}\pgf@x%
    \advance\pgf@xc by\pgf@sys@tonumber{\pgf@xa}\pgf@y%
    %
    % The orthogonal vector is (\pgf@ya,\pgf@xa)
    %
    %
    % Compute orthogonal vector to #3--#4
    %
    \pgf@process{#4}%
    \pgf@xb=\pgf@x%
    \pgf@yb=\pgf@y%
    \pgf@process{#3}%
    \advance\pgf@xb by-\pgf@x%
    \advance\pgf@yb by-\pgf@y%
    \pgf@yb=-\pgf@yb%
    % Normalise a bit
    \c@pgf@counta=\pgf@xb%
    \ifnum\c@pgf@counta<0\relax%
      \c@pgf@counta=-\c@pgf@counta\relax%
    \fi%
    \c@pgf@countb=\pgf@yb%
    \ifnum\c@pgf@countb<0\relax%
      \c@pgf@countb=-\c@pgf@countb\relax%
    \fi%
    \advance\c@pgf@counta by\c@pgf@countb\relax%
    \divide\c@pgf@counta by 65536\relax%
    \ifnum\c@pgf@counta>0\relax%
      \divide\pgf@xb by\c@pgf@counta\relax%
      \divide\pgf@yb by\c@pgf@counta\relax%
    \fi%
    %
    % Compute projection
    %
    \pgf@yc=\pgf@sys@tonumber{\pgf@yb}\pgf@x%
    \advance\pgf@yc by\pgf@sys@tonumber{\pgf@xb}\pgf@y%
    %
    % The orthogonal vector is (\pgf@yb,\pgf@xb)
    %
    % Setup transformation matrx (this is just to use the matrix
    % inversion)
    %
    \pgfsettransform{{\pgf@sys@tonumber\pgf@ya}{\pgf@sys@tonumber\pgf@yb}{\pgf@sys@tonumber\pgf@xa}{\pgf@sys@
    \pgftransforminvert%
    \pgf@process{\pgfpointtransformed{\pgfpoint{\pgf@xc}{\pgf@yc}}}%
  }%
}

\ifx\pgfpointintersectionoflines\forest@original@pgfpointintersectionoflines
  \let\pgfpointintersectionoflines\forest@patched@pgfpointintersectionoflines
\fi

 % hah: hacking forest --- it depends on some details of PGF implementation
\def\forest@pgf@notyetpositioned{not yet positionedPGFINTERNAL}%
\expandafter\ifstrequal\expandafter{\pgfversion}{2.10}{%
  \def\forest@pgf@notyetpositioned{not yet positioned@}%
}{}
```

8 Utilities

Escaping \ifs.

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

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

```
\def\newloop#1{%
  \count@=\escapechar
  \escapechar=-1
  \expandafter\newloop@parse@loopname\string#1\newloop@end
  \escapechar=\count@
}%
{\lccode`7=`l \lccode`8=`o \lccode`9=`p
  \lowercase{\gdef\newloop@parse@loopname#17889#2\newloop@end{%
      \edef\newloop@marshal{%
        \noexpand\csdef{#1loop#2}####1\expandafter\noexpand\csname #1repeat#2\endcsname{%
          \noexpand\csdef{#1iterate#2}{####1\relax\noexpand\expandafter\expandafter\noexpand\csname#1iterate#
          \expandafter\noexpand\csname#1iterate#2\endcsname
          \let\expandafter\noexpand\csname#1iterate#2\endcsname\relax
        }%
      }%
      \newloop@marshal
    }%
  }%
}%
```

Additional loops (for embedding).

```
\newloop\forest@loop
\newloop\forest@loopa
\newloop\forest@loopb
\newloop\forest@loopc
\newloop\forest@sort@loop
\newloop\forest@sort@loopA
```

New counters, dimens, ifs.

```
\newdimen\forest@temp@dimen
\newcount\forest@temp@count
\newcount\forest@n
\newif\ifforest@temp
\newcount\forest@temp@global@count
```

Appending and prepending to token lists.

```
\def\apptotoks#1#2{\expandafter#1\expandafter{\the#1#2}}
\long\def\lapptotoks#1#2{\expandafter#1\expandafter{\the#1#2}}
\def\eapptotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter#1\expandafter\expandafter\expandaf
\def\pretotoks#1#2{\toks@={#2}\expandafter\expandafter\expandafter#1\expandafter\expandafter\expandafter{\exp
\def\epretotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter#1\expandafter\expandafter\expandaf
\def\gapptotoks#1#2{\expandafter\global\expandafter#1\expandafter{\the#1#2}}
\def\xapptotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter\global\expandafter\expandafter\exp
\def\gpretotoks#1#2{\toks@={#2}\expandafter\expandafter\expandafter\global\expandafter\expandafter\expandafte
\def\xpretotoks#1#2{\edef\pot@temp{#2}\expandafter\expandafter\expandafter\global\expandafter\expandafter\exp
```

Expanding number arguments.

```
\def\expandnumberarg#1#2{\expandafter#1\expandafter{\number#2}}
\def\expandtwonumberargs#1#2#3{%
  \expandafter\expandtwonumberargs@\expandafter#1\expandafter{\number#3}{#2}}
\def\expandtwonumberargs@#1#2#3{%
  \expandafter#1\expandafter{\number#3}{#2}}
\def\expandthreenumberargs#1#2#3#4{%
  \expandafter\expandthreenumberargs@\expandafter#1\expandafter{\number#4}{#2}{#3}}
\def\expandthreenumberargs@#1#2#3#4{%
  \expandafter\expandthreenumberargs@@\expandafter#1\expandafter{\number#4}{#2}{#3}}
\def\expandthreenumberargs@@#1#2#3#4{%
  \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.

```
\def\forest@convert@others@to@underscores#1#2{%
  \def\forest@cotu@result{}%
  \forest@cotu#1\forest@end
  \let#2\forest@cotu@result
}
\def\forest@cotu{%
  \futurelet\forest@cotu@nextchar\forest@cotu@checkforspace
}
\def\forest@cotu@checkforspace{%
  \expandafter\ifx\space\forest@cotu@nextchar
    \let\forest@cotu@next\forest@cotu@havespace
  \else
    \let\forest@cotu@next\forest@cotu@nospace
  \fi
  \forest@cotu@next
}
\def\forest@cotu@havespace#1{%
  \appto\forest@cotu@result{_}%
  \forest@cotu#1%
}
\def\forest@cotu@nospace{%
```

```
  \ifx\forest@cotu@nextchar\forest@end
    \@escapeif\@gobble
  \else
    \@escapeif\forest@cotu@nospaceB
  \fi
}
\def\forest@cotu@nospaceB{%
  \ifcat\forest@cotu@nextchar a%
    \let\forest@cotu@next\forest@cotu@have@alphanum
  \else
    \ifcat\forest@cotu@nextchar 0%
      \let\forest@cotu@next\forest@cotu@have@alphanum
    \else
      \let\forest@cotu@next\forest@cotu@haveother
    \fi
  \fi
  \forest@cotu@next
}
\def\forest@cotu@have@alphanum#1{%
  \appto\forest@cotu@result{#1}%
  \forest@cotu
}
\def\forest@cotu@haveother#1{%
  \appto\forest@cotu@result{_}%
  \forest@cotu
}
```

Additional list macros.

```
\def\forest@listedel#1#2{% #1 = list, #2 = item
  \edef\forest@marshal{\noexpand\forest@listdel\noexpand#1{#2}}%
  \forest@marshal
}
\def\forest@listcsdel#1#2{%
  \expandafter\forest@listdel\csname #1\endcsname{#2}%
}
\def\forest@listcsedel#1#2{%
  \expandafter\forest@listedel\csname #1\endcsname{#2}%
}
\edef\forest@restorelistsepcatcode{\noexpand\catcode`|\the\catcode`|\relax}%
\catcode`\|=3
\gdef\forest@listdel#1#2{%
  \def\forest@listedel@A##1|#2|##2\forest@END{%
    \forest@listedel@B##1|##2\forest@END%|
  }%
  \def\forest@listedel@B|##1\forest@END{%|
    \def#1{##1}%
  }%
  \expandafter\forest@listedel@A\expandafter|#1\forest@END%|
}
\forest@restorelistsepcatcode
```

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

```
\def\forest@strip@braces#1{%
  \forest@strip@braces@A#1\forest@strip@braces@preend\forest@strip@braces@end
}
\def\forest@strip@braces@A#1#2\forest@strip@braces@end{%
  #1\ifx\forest@strip@braces@preend#2\else\@escapeif{\forest@strip@braces@A#2\forest@strip@braces@end}\fi
}
```

8.1 Sorting

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

The user should prepare the data in an arbitrarily encoded array,[19] 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.

```
\def\forest@sort#1#2#3#4#5{%
  \let\forest@sort@cmp#1\relax
  \let\forest@sort@let#2\relax
  \let\forest@sort@direction#3\relax
  \forest@@sort{#4}{#5}%
}
```

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.

```
\def\forest@quicksort@minarraylength{10000}
\def\forest@@sort#1#2{%
  \ifnum#1<#2\relax\@escapeif{%
    \forest@sort@m=#2
    \advance\forest@sort@m -#1
    \ifnum\forest@sort@m>\forest@quicksort@minarraylength\relax\@escapeif{%
      \forest@quicksort{#1}{#2}%
    }\else\@escapeif{%
      \forest@insertionsort{#1}{#2}%
    }\fi
  }\fi
}
```

Various counters and macros needed by the sorting algorithms.

```
\newcount\forest@sort@m\newcount\forest@sort@k\newcount\forest@sort@p
\def\forest@sort@ascending{>}
\def\forest@sort@descending{<}
\def\forest@sort@cmp{%
  \PackageError{sort}{You must define forest@sort@cmp function before calling
    sort}{The macro must take two arguments, indices of the array
    elements to be compared, and return '=' if the elements are equal
    and '>'/'<' if the first is greater /less than the secong element.}%
}
\def\forest@sort@cmp@gt{\def\forest@sort@cmp@result{>}}
\def\forest@sort@cmp@lt{\def\forest@sort@cmp@result{<}}
\def\forest@sort@cmp@eq{\def\forest@sort@cmp@result{=}}
\def\forest@sort@let{%
  \PackageError{sort}{You must define forest@sort@let function before calling
    sort}{The macro must take two arguments, indices of the array:
    element 2 must be copied onto element 1.}%
}
```

Quick sort macro (adapted from laansort).

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

[19] 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`).

```
  \forest@sort@m=#2
  \advance\forest@sort@m -#1
  \ifodd\forest@sort@m\relax\advance\forest@sort@m1 \fi
  \divide\forest@sort@m 2
  \advance\forest@sort@m #1
```

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

```
  \forest@sort@cmp{#1}{#2}%
  \if\forest@sort@cmp@result=%
    \forest@sort@p=#1
  \else
    \if\forest@sort@cmp@result>%
      \forest@sort@p=#1\relax
    \else
      \forest@sort@p=#2\relax
    \fi
    \forest@sort@cmp{\the\forest@sort@p}{\the\forest@sort@m}%
    \if\forest@sort@cmp@result<%
    \else
      \forest@sort@p=\the\forest@sort@m
    \fi
  \fi
```

Exchange the pivot and the first element.

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

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

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

Loop through the list.

```
  \forest@sort@k=#1\relax
  \forest@sort@loop
  \ifnum\forest@sort@k<#2\relax
    \advance\forest@sort@k 1
```

Compare the pivot and the current element.

```
    \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.

```
    \ifx\forest@sort@direction\forest@sort@cmp@result
      \advance\forest@sort@m 1
      \forest@sort@xch{\the\forest@sort@m}{\the\forest@sort@k}
    \fi
  \forest@sort@repeat
```

Move the pivot element into its final position.

```
  \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.

```
  \forest@sort@k=\forest@sort@m
  \advance\forest@sort@k -1
  \advance\forest@sort@m 1
  \edef\forest@sort@marshal{%
    \noexpand\forest@@sort{#1}{\the\forest@sort@k}%
    \noexpand\forest@@sort{\the\forest@sort@m}{#2}%
  }%
  \forest@sort@marshal
}
% We defines the item-exchange macro in terms of the (user-provided)
% array let macro.
%    \begin{macrocode}
\def\forest@sort@xch#1#2{%
  \forest@sort@let{aux}{#1}%
  \forest@sort@let{#1}{#2}%
  \forest@sort@let{#2}{aux}%
}
```

Insertion sort.

```
\def\forest@insertionsort#1#2{%
  \forest@sort@m=#1
  \edef\forest@insertionsort@low{#1}%
  \forest@sort@loopA
  \ifnum\forest@sort@m<#2
    \advance\forest@sort@m 1
    \forest@insertionsort@Qbody
  \forest@sort@repeatA
}
\newif\ifforest@insertionsort@loop
\def\forest@insertionsort@Qbody{%
  \forest@sort@let{aux}{\the\forest@sort@m}%
  \forest@sort@k\forest@sort@m
  \advance\forest@sort@k -1
  \forest@insertionsort@looptrue
  \forest@sort@loop
  \ifforest@insertionsort@loop
    \forest@insertionsort@qbody
  \forest@sort@repeat
  \advance\forest@sort@k 1
  \forest@sort@let{\the\forest@sort@k}{aux}%
}
\def\forest@insertionsort@qbody{%
  \forest@sort@cmp{\the\forest@sort@k}{aux}%
  \ifx\forest@sort@direction\forest@sort@cmp@result\relax
    \forest@sort@p=\forest@sort@k
    \advance\forest@sort@p 1
    \forest@sort@let{\the\forest@sort@p}{\the\forest@sort@k}%
    \advance\forest@sort@k -1
    \ifnum\forest@sort@k<\forest@insertionsort@low\relax
      \forest@insertionsort@loopfalse
    \fi
  \else
    \forest@insertionsort@loopfalse
  \fi
}
```

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

Compare numbers.

```
\def\forest@sort@cmpnumcs#1#2{%
  \ifnum\csname#1\endcsname>\csname#2\endcsname\relax
   \forest@sort@cmp@gt
 \else
   \ifnum\csname#1\endcsname<\csname#2\endcsname\relax
     \forest@sort@cmp@lt
   \else
     \forest@sort@cmp@eq
   \fi
 \fi
}
```

Compare dimensions.

```
\def\forest@sort@cmpdimcs#1#2{%
  \ifdim\csname#1\endcsname>\csname#2\endcsname\relax
   \forest@sort@cmp@gt
 \else
   \ifdim\csname#1\endcsname<\csname#2\endcsname\relax
     \forest@sort@cmp@lt
   \else
     \forest@sort@cmp@eq
   \fi
 \fi
}
```

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

```
\def\forest@sort@cmptwodimcs#1#2#3#4{%
  \ifdim\csname#1\endcsname>\csname#3\endcsname\relax
   \forest@sort@cmp@gt
 \else
   \ifdim\csname#1\endcsname<\csname#3\endcsname\relax
     \forest@sort@cmp@lt
   \else
     \ifdim\csname#2\endcsname>\csname#4\endcsname\relax
       \forest@sort@cmp@gt
     \else
       \ifdim\csname#2\endcsname<\csname#4\endcsname\relax
         \forest@sort@cmp@lt
       \else
         \forest@sort@cmp@eq
       \fi
     \fi
   \fi
 \fi
}
```

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).

```
\def\forest@reversearray#1#2#3{%
  \let\forest@sort@let#1%
  \c@pgf@countc=#2
  \c@pgf@countd=#3
  \advance\c@pgf@countd -1
  \forest@loopa
  \ifnum\c@pgf@countc<\c@pgf@countd\relax
    \forest@sort@xch{\the\c@pgf@countc}{\the\c@pgf@countd}%
    \advance\c@pgf@countc 1
    \advance\c@pgf@countd -1
  \forest@repeata
}
```

9 The bracket representation parser

9.1 The user interface macros

Settings.

```
\def\bracketset#1{\pgfqkeys{/bracket}{#1}}%
\bracketset{%
  /bracket/.is family,
  /handlers/.let/.style={\pgfkeyscurrentpath/.code={\let#1##1}},
  opening bracket/.let=\bracket@openingBracket,
  closing bracket/.let=\bracket@closingBracket,
  action character/.let=\bracket@actionCharacter,
  opening bracket=[,
  closing bracket=],
  action character,
  new node/.code n args={3}{% #1=preamble, #2=node spec, #3=cs receiving the id
    \forest@node@new#3%
    \forestOset{#3}{given options}{content'=#2}%
    \ifblank{#1}{}{%
      \forestOpreto{#3}{given options}{#1,}%
    }%
  },
  set afterthought/.code 2 args={% #1=node id, #2=afterthought
    \ifblank{#2}{}{\forestOappto{#1}{given options}{,afterthought={#2}}}%
  }
}
```

\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.)

```
\newtoks\bracket@content
\newtoks\bracket@afterthought
\def\bracketParse#1#2={%
  \def\bracketEndParsingHook{#1}%
  \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 afterhoughts.)

```
  \bracket@content={}%
  \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.

```
  \let\bracket@state\bracket@state@starting
  \bracket@ignorespacestrue
```

By default, don't expand anything.

```
  \bracket@expandtokensfalse
```

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

```
  \def\bracket@parentNode{0}%
  \def\bracket@rootNode{0}%
  \def\bracket@newNode{0}%
  \def\bracket@afterthoughtNode{0}%
```

Finally, we start the parser.

```
  \bracket@Parse
}
```

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.

```
\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.

```
\def\bracket@Parse{%
  \futurelet\bracket@next@token\bracket@Parse@checkForSpace
}
\def\bracket@Parse@checkForSpace{%
  \expandafter\ifx\space\bracket@next@token\@escapeif{%
```

```
    \ifbracket@ignorespaces\else
      \bracket@haveSpacetrue
    \fi
    \expandafter\bracket@Parse\romannumeral-`0%
  }\else\@escapeif{%
    \bracket@Parse@maybeexpand
  }\fi
}
```

We either fully expand the next token (using a popular TEXnical trick . . .) or don't expand it at all, depending on the state of `\ifbracket@expandtokens`.

```
\newif\ifbracket@expandtokens
\def\bracket@Parse@maybeexpand{%
  \ifbracket@expandtokens\@escapeif{%
    \expandafter\bracket@Parse@peekAhead\romannumeral-`0%
  }\else\@escapeif{%
    \bracket@Parse@peekAhead
  }\fi
}
```

We then look ahead to see what's coming.

```
\def\bracket@Parse@peekAhead{%
  \futurelet\bracket@next@token\bracket@Parse@checkForTeXGroup
}
```

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

```
\def\bracket@Parse@checkForTeXGroup{%
  \ifx\bracket@next@token\bgroup%
    \@escapeif{\bracket@Parse@appendGroup}%
  \else
    \@escapeif{\bracket@Parse@token}%
  \fi
}
```

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.

```
\long\def\bracket@Parse@token#1{%
  \ifx#1\bracket@openingBracket
    \@escapeif{\bracket@Parse@openingBracketFound}%
  \else
    \@escapeif{%
      \ifx#1\bracket@closingBracket
        \@escapeif{\bracket@Parse@closingBracketFound}%
      \else
        \@escapeif{%
          \ifx#1\bracket@actionCharacter
            \@escapeif{\futurelet\bracket@next@token\bracket@Parse@actionCharacterFound}%
          \else
            \@escapeif{\bracket@Parse@appendToken#1}%
          \fi
        }%
      \fi
    }%
  \fi
}
```

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

```
\newif\ifbracket@haveSpace
\newif\ifbracket@ignorespaces
\def\bracket@Parse@appendSpace{%
```

```
  \ifbracket@haveSpace
    \ifcase\bracket@state\relax
      \eapptotoks\bracket@content\space
    \or
      \eapptotoks\bracket@afterthought\space
    \or
      \eapptotoks\bracket@afterthought\space
    \fi
    \bracket@haveSpacefalse
  \fi
}
\long\def\bracket@Parse@appendToken#1{%
  \bracket@Parse@appendSpace
  \ifcase\bracket@state\relax
    \lapptotoks\bracket@content{#1}%
  \or
    \lapptotoks\bracket@afterthought{#1}%
  \or
    \lapptotoks\bracket@afterthought{#1}%
  \fi
  \bracket@ignorespacesfalse
  \bracket@Parse
}
\def\bracket@Parse@appendGroup#1{%
  \bracket@Parse@appendSpace
  \ifcase\bracket@state\relax
    \apptotoks\bracket@content{{#1}}%
  \or
    \apptotoks\bracket@afterthought{{#1}}%
  \or
    \apptotoks\bracket@afterthought{{#1}}%
  \fi
  \bracket@ignorespacesfalse
  \bracket@Parse
}
```

Declare states.

```
\def\bracket@state@inContent{0}
\def\bracket@state@inAfterthought{1}
\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.)

```
\def\bracket@Parse@openingBracketFound{%
  \bracket@haveSpacefalse
  \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.

```
    \@escapeif{%
      \bracket@createNode
      \ifnum\bracket@parentNode=0 \else
        \forest@node@Append{\bracket@parentNode}{\bracket@newNode}%
      \fi
      \let\bracket@parentNode\bracket@newNode
      \bracket@Parse
    }%
  \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.

```
    \@escapeif{%
      \bracket@addAfterthought
      \let\bracket@state\bracket@state@inContent
      \bracket@Parse
    }%
  \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.

```
    \@escapeif{%
      \let\bracket@state\bracket@state@inContent
      \bracket@Parse
    }%
  \fi
}
\def\bracket@Parse@closingBracketFound{%
  \bracket@haveSpacefalse
  \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.

```
    \@escapeif{%
      \bracket@createNode
      \ifnum\bracket@parentNode=0
        \@escapeif\bracketEndParsingHook
      \else
        \@escapeif{%
          \let\bracket@afterthoughtNode\bracket@newNode
          \let\bracket@state\bracket@state@inAfterthought
          \forest@node@Append{\bracket@parentNode}{\bracket@newNode}%
          \bracket@Parse
        }%
      \fi
    }%
  \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.

```
    \@escapeif{%
      \bracket@addAfterthought
      \let\bracket@afterthoughtNode\bracket@parentNode
      \edef\bracket@parentNode{\forestOve{\bracket@parentNode}{@parent}}%
      \ifnum\bracket@parentNode=0
        \expandafter\bracketEndParsingHook
      \else
        \expandafter\bracket@Parse
      \fi
    }%
  \else % starting
```

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

```
    \PackageError{forest}{You're attempting to start a bracket representation
      with a closing bracket}{}%
  \fi
}
```

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

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

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

```
  \ifx\bracket@next@token\bgroup\@escapeif{%
    \bracket@Parse@action@expandgroup
  }\else\@escapeif{%
    \bracket@Parse@action@notagroup
  }\fi
}
\def\bracket@Parse@action@expandgroup#1{%
  \edef\bracket@Parse@action@expandgroup@macro{#1}%
  \expandafter\bracket@Parse\bracket@Parse@action@expandgroup@macro
}
\let\bracket@action@fullyexpandCharacter+
\let\bracket@action@dontexpandCharacter-
\let\bracket@action@executeCharacter!
\def\bracket@Parse@action@notagroup#1{%
```

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

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

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

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

Inhibit expansion of the next token.

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

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

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

Anything else will be expanded once.

```
          \expandafter\bracket@Parse#1%
        }\fi
      }\fi
    }\fi
  }\fi
}
```

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.

```
\def\bracket@createNode{%
  \ifnum\bracket@rootNode=0
    % root node
    \bracketset{new node/.expanded=%
      {\the\bracket@afterthought}%
      {\the\bracket@content}%
      \noexpand\bracket@newNode
    }%
    \bracket@afterthought={}%
    \let\bracket@rootNode\bracket@newNode
    \expandafter\let\bracket@saveRootNodeTo\bracket@newNode
  \else
    % other nodes
    \bracketset{new node/.expanded=%
      {}%
      {\the\bracket@content}%
      \noexpand\bracket@newNode
    }%
  \fi
  \bracket@content={}%
}
```

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

```
\def\bracket@addAfterthought{%
  \bracketset{%
    set afterthought/.expanded={\bracket@afterthoughtNode}{\the\bracket@afterthought}%
  }%
  \bracket@afterthought={}%
}
```

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.

```
 % full expansion expands precisely to the value
\def\forestov#1{\expandafter\expandafter\expandafter\expandonce
  \pgfkeysvalueof{/forest/@node/\forest@cn/#1}}
 % full expansion expands all the way
\def\forestove#1{\pgfkeysvalueof{/forest/@node/\forest@cn/#1}}
 % full expansion expands to the cs holding the value
\def\forestom#1{\expandafter\expandonce\expandafter{\pgfkeysvalueof{/forest/@node/\forest@cn/#1}}}\def\forest
\def\forestoget#1#2{\pgfkeysgetvalue{/forest/@node/\forest@cn/#1}{#2}}
\def\forestolet#1#2{\pgfkeyslet{/forest/@node/\forest@cn/#1}{#2}}
\def\forestoset#1#2{\pgfkeyssetvalue{/forest/@node/\forest@cn/#1}{#2}}
\def\forestoeset#1#2{%
  \edef\forest@option@temp{%
    \noexpand\pgfkeyssetvalue{/forest/@node/\forest@cn/#1}{#2}%
  }\forest@option@temp
}
\def\forestoappto#1#2{%
  \forestoeset{#1}{\forestov{#1}\unexpanded{#2}}%
}
\def\forestoifdefined#1#2#3{%
  \pgfkeysifdefined{/forest/@node/\forest@cn/#1}{#2}{#3}%
}
```

User macros for retrieving node options of the current node.

```
\let\forestoption\forestov
\let\foresteoption\forestove
```

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

```
\def\forestOv#1#2{\expandafter\expandafter\expandafter\expandonce
  \pgfkeysvalueof{/forest/@node/#1/#2}}
\def\forestOve#1#2{\pgfkeysvalueof{/forest/@node/#1/#2}}
 % full expansion expands to the cs holding the value
```

```
\def\forestOm#1#2{\expandafter\expandonce\expandafter{\pgfkeysvalueof{/forest/@node/#1/#2}}}
\def\forestOget#1#2#3{\pgfkeysgetvalue{/forest/@node/#1/#2}{#3}}
\def\forestOget#1#2#3{\pgfkeysgetvalue{/forest/@node/#1/#2}{#3}}
\def\forestOlet#1#2#3{\pgfkeyslet{/forest/@node/#1/#2}{#3}}
\def\forestOset#1#2#3{\pgfkeyssetvalue{/forest/@node/#1/#2}{#3}}
\def\forestOeset#1#2#3{%
  \edef\forestoption@temp{%
    \noexpand\pgfkeyssetvalue{/forest/@node/#1/#2}{#3}%
  }\forestoption@temp
}
\def\forestOappto#1#2#3{%
  \forestOeset{#1}{#2}{\forestOv{#1}{#2}\unexpanded{#3}}%
}
\def\forestOeappto#1#2#3{%
  \forestOeset{#1}{#2}{\forestOv{#1}{#2}#3}%
}
\def\forestOpreto#1#2#3{%
  \forestOeset{#1}{#2}{\unexpanded{#3}\forestOv{#1}{#2}}%
}
\def\forestOepreto#1#2#3{%
  \forestOeset{#1}{#2}{#3\forestOv{#1}{#2}}%
}
\def\forestOifdefined#1#2#3#4{%
  \pgfkeysifdefined{/forest/@node/#1/#2}{#3}{#4}%
}
\def\forestOletO#1#2#3#4{% option #2 of node #1 <-- option #4 of node #3
  \forestOget{#3}{#4}\forestoption@temp
  \forestOlet{#1}{#2}\forestoption@temp}
\def\forestOleto#1#2#3{%
  \forestoget{#3}\forestoption@temp
  \forestOlet{#1}{#2}\forestoption@temp}
\def\forestoletO#1#2#3{%
  \forestOget{#2}{#3}\forestoption@temp
  \forestolet{#1}\forestoption@temp}
\def\forestoleto#1#2{%
  \forestoget{#2}\forestoption@temp
  \forestolet{#1}\forestoption@temp}
```

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

```
\def\forest@node@init{%
  \forestoset{@parent}{0}%
  \forestoset{@previous}{0}% previous sibling
  \forestoset{@next}{0}%     next sibling
  \forestoset{@first}{0}%  primary child
  \forestoset{@last}{0}%   last child
}
\def\forestoinit#1{%
  \pgfkeysgetvalue{/forest/#1}\forestoinit@temp
  \forestolet{#1}\forestoinit@temp
}
\newcount\forest@node@maxid
\def\forest@node@new#1{% #1 = cs receiving the new node id
  \advance\forest@node@maxid1
  \forest@fornode{\the\forest@node@maxid}{%
    \forest@node@init
    \forest@node@setname{node@\forest@cn}%
    \forest@initializefromstandardnode
    \edef#1{\forest@cn}%
  }%
}
\let\forestoinit@orig\forestoinit
```

```
\def\forest@node@copy#1#2{% #1=from node id, cs receiving the new node id
  \advance\forest@node@maxid1
  \def\forestoinit##1{\forestoletO{##1}{#1}{##1}}%
  \forest@fornode{\the\forest@node@maxid}{%
    \forest@node@init
    \forest@node@setname{\forest@copy@name@template{\forestOve{#1}{name}}}%
    \edef#2{\forest@cn}%
  }%
  \let\forestoinit\forestoinit@orig
}
\forestset{
  copy name template/.code={\def\forest@copy@name@template##1{#1}},
  copy name template/.default={node@\the\forest@node@maxid},
  copy name template
}
\def\forest@tree@copy#1#2{% #1=from node id, #2=cs receiving the new node id
  \forest@node@copy{#1}\forest@node@copy@temp@id
  \forest@fornode{\forest@node@copy@temp@id}{%
    \expandafter\forest@tree@copy@\expandafter{\forest@node@copy@temp@id}{#1}%
    \edef#2{\forest@cn}%
  }%
}
\def\forest@tree@copy@#1#2{%
  \forest@node@Foreachchild{#2}{%
    \expandafter\forest@tree@copy\expandafter{\forest@cn}\forest@node@copy@temp@childid
    \forest@node@Append{#1}{\forest@node@copy@temp@childid}%
  }%
}
```

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.

```
\def\forest@cn{0}
\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.

```
\def\forest@node@append#1{\expandtwonumberargs\forest@node@Append{\forest@cn}{#1}}
\def\forest@node@prepend#1{\expandtwonumberargs\forest@node@Insertafter{\forest@cn}{#1}{0}}
\def\forest@node@insertafter#1#2{%
  \expandthreenumberargs\forest@node@Insertafter{\forest@cn}{#1}{#2}}
\def\forest@node@insertbefore#1#2{%
  \expandthreenumberargs\forest@node@Insertafter{\forest@cn}{#1}{\forestOve{#2}{@previous}}%
}
\def\forest@node@remove{\expandnumberarg\forest@node@Remove{\forest@cn}}
\def\forest@node@Append#1#2{\expandtwonumberargs\forest@node@Append@{#1}{#2}}
\def\forest@node@Prepend#1#2{\expandtwonumberargs\forest@node@Insertafter{#1}{#2}{0}}
\def\forest@node@Insertafter#1#2#3{% #2 is inserted after #3
  \expandthreenumberargs\forest@node@Insertafter@{#1}{#2}{#3}%
}
\def\forest@node@Insertbefore#1#2#3{% #2 is inserted before #3
  \expandthreenumberargs\forest@node@Insertafter{#1}{#2}{\forestOve{#3}{@previous}}%
}
\def\forest@node@Remove#1{\expandnumberarg\forest@node@Remove@{#1}}
\def\forest@node@Insertafter@#1#2#3{%
  \ifnum\forestOve{#2}{@parent}=0
  \else
```

```
    \PackageError{forest}{Insertafter(#1,#2,#3):
      node #2 already has a parent (\forestOve{#2}{@parent})}{}%
  \fi
  \ifnum#3=0
  \else
    \ifnum#1=\forestOve{#3}{@parent}
    \else
      \PackageError{forest}{Insertafter(#1,#2,#3): node #1 is not the parent of the
            intended sibling #3 (with parent \forestOve{#3}{@parent})}{}%
    \fi
  \fi
  \forestOeset{#2}{@parent}{#1}%
  \forestOeset{#2}{@previous}{#3}%
  \ifnum#3=0
    \forestOget{#1}{@first}\forest@node@temp
    \forestOeset{#1}{@first}{#2}%
  \else
    \forestOget{#3}{@next}\forest@node@temp
    \forestOeset{#3}{@next}{#2}%
  \fi
  \forestOeset{#2}{@next}{\forest@node@temp}%
  \ifnum\forest@node@temp=0
    \forestOeset{#1}{@last}{#2}%
  \else
    \forestOeset{\forest@node@temp}{@previous}{#2}%
  \fi
}
\def\forest@node@Append@#1#2{%
  \ifnum\forestOve{#2}{@parent}=0
  \else
    \PackageError{forest}{Append(#1,#2):
      node #2 already has a parent (\forestOve{#2}{@parent})}{}%
  \fi
  \forestOeset{#2}{@parent}{#1}%
  \forestOget{#1}{@last}\forest@node@temp
  \forestOeset{#1}{@last}{#2}%
  \forestOeset{#2}{@previous}{\forest@node@temp}%
  \ifnum\forest@node@temp=0
    \forestOeset{#1}{@first}{#2}%
  \else
    \forestOeset{\forest@node@temp}{@next}{#2}%
  \fi
}
\def\forest@node@Remove@#1{%
  \forestOget{#1}{@parent}\forest@node@temp@parent
  \ifnum\forest@node@temp@parent=0
  \else
    \forestOget{#1}{@previous}\forest@node@temp@previous
    \forestOget{#1}{@next}\forest@node@temp@next
    \ifnum\forest@node@temp@previous=0
      \forestOeset{\forest@node@temp@parent}{@first}{\forest@node@temp@next}%
    \else
      \forestOeset{\forest@node@temp@previous}{@next}{\forest@node@temp@next}%
    \fi
    \ifnum\forest@node@temp@next=0
      \forestOeset{\forest@node@temp@parent}{@last}{\forest@node@temp@previous}%
    \else
      \forestOeset{\forest@node@temp@next}{@previous}{\forest@node@temp@previous}%
    \fi
    \forestOset{#1}{@parent}{0}%
    \forestOset{#1}{@previous}{0}%
    \forestOset{#1}{@next}{0}%
  \fi
}
```

Looping methods.

```
\def\forest@forthis#1{%
  \edef\forest@node@marshal{\unexpanded{#1}\def\noexpand\forest@cn}%
  \expandafter\forest@node@marshal\expandafter{\forest@cn}%
}
\def\forest@fornode#1#2{%
  \edef\forest@node@marshal{\edef\noexpand\forest@cn{#1}\unexpanded{#2}\def\noexpand\forest@cn}%
  \expandafter\forest@node@marshal\expandafter{\forest@cn}%
}
\def\forest@fornode@ifexists#1#2{%
  \edef\forest@node@temp{#1}%
  \ifnum\forest@node@temp=0
  \else
    \@escapeif{\expandnumberarg\forest@fornode{\forest@node@temp}{#2}}%
  \fi
}
\def\forest@node@foreachchild#1{\forest@node@Foreachchild{\forest@cn}{#1}}
\def\forest@node@Foreachchild#1#2{%
  \forest@fornode{\forestOve{#1}{@first}}{\forest@node@@forselfandfollowingsiblings{#2}}%
}
\def\forest@node@@forselfandfollowingsiblings#1{%
  \ifnum\forest@cn=0
  \else
    \forest@forthis{#1}%
    \@escapeif{%
      \edef\forest@cn{\forestove{@next}}%
      \forest@node@@forselfandfollowingsiblings{#1}%
    }%
  \fi
}
\def\forest@node@foreach#1{\forest@node@Foreach{\forest@cn}{#1}}
\def\forest@node@Foreach#1#2{%
  \forest@fornode{#1}{\forest@node@@foreach{#2}}%
}
\def\forest@node@@foreach#1{%
  \forest@forthis{#1}%
  \ifnum\forestove{@first}=0
  \else\@escapeif{%
      \edef\forest@cn{\forestove{@first}}%
      \forest@node@@forselfandfollowingsiblings{\forest@node@@foreach{#1}}%
    }%
  \fi
}
\def\forest@node@foreachdescendant#1{\forest@node@Foreachdescendant{\forest@cn}{#1}}
\def\forest@node@Foreachdescendant#1#2{%
  \forest@node@Foreachchild{#1}{%
    \forest@node@foreach{#2}%
  }%
}
```

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

```
\def\forest@node@Compute@numeric@ts@info@#1{%
  \forest@node@Foreach{#1}{\forest@node@@compute@numeric@ts@info}%
  \ifnum\forestOve{#1}{@parent}=0
  \else
    \fornode{#1}{\forest@node@@compute@numeric@ts@info@nbar}%
  \fi
  \forest@node@Foreachdescendant{#1}{\forest@node@@compute@numeric@ts@info@nbar}%
}
\def\forest@node@@compute@numeric@ts@info{%
  \forestoset{n children}{0}%
  %
  \edef\forest@node@temp{\forestove{@previous}}%
  \ifnum\forest@node@temp=0
    \forestoset{n}{1}%
  \else
    \forestoeset{n}{\number\numexpr\forestOve{\forest@node@temp}{n}+1}%
  \fi
  %
  \edef\forest@node@temp{\forestove{@parent}}%
  \ifnum\forest@node@temp=0
    \forestoset{n}{0}%
    \forestoset{n'}{0}%
    \forestoset{level}{0}%
  \else
    \forestOeset{\forest@node@temp}{n children}{%
      \number\numexpr\forestOve{\forest@node@temp}{n children}+1%
    }%
    \forestoeset{level}{%
      \number\numexpr\forestOve{\forest@node@temp}{level}+1%
    }%
  \fi
}
\def\forest@node@@compute@numeric@ts@info@nbar{%
  \forestoeset{n'}{\number\numexpr\forestOve{\forestove{@parent}}{n children}-\forestove{n}+1}%
}
\def\forest@node@compute@numeric@ts@info#1{%
  \expandnumberarg\forest@node@Compute@numeric@ts@info@{\forest@cn}%
}
\def\forest@node@Compute@numeric@ts@info#1{%
  \expandnumberarg\forest@node@Compute@numeric@ts@info@{#1}%
}
```

Tree structure queries.

```
\def\forest@node@rootid{%
  \expandnumberarg\forest@node@Rootid{\forest@cn}%
}
\def\forest@node@Rootid#1{% #1=node
  \ifnum\forestOve{#1}{@parent}=0
    #1%
  \else
    \@escapeif{\expandnumberarg\forest@node@Rootid{\forestOve{#1}{@parent}}}%
  \fi
}
\def\forest@node@nthchildid#1{% #1=n
  \ifnum#1<1
    0%
  \else
    \expandnumberarg\forest@node@nthchildid@{\number\forestove{@first}}{#1}%
  \fi
}
\def\forest@node@nthchildid@#1#2{%
  \ifnum#1=0
    0%
  \else
    \ifnum#2>1
      \@escapeifif{\expandtwonumberargs
        \forest@node@nthchildid@{\forestOve{#1}{@next}}{\numexpr#2-1}}%
    \else
```

```
      #1%
    \fi
  \fi
}
\def\forest@node@nbarthchildid#1{% #1=n
  \expandnumberarg\forest@node@nbarthchildid@{\number\forestove{@last}}{#1}%
}
\def\forest@node@nbarthchildid@#1#2{%
  \ifnum#1=0
    0%
  \else
    \ifnum#2>1
      \@escapeifif{\expandtwonumberargs
        \forest@node@nbarthchildid@{\forestOve{#1}{@previous}}{\numexpr#2-1}}%
    \else
      #1%
    \fi
  \fi
}
\def\forest@node@nornbarthchildid#1{%
  \ifnum#1>0
    \forest@node@nthchildid{#1}%
  \else
    \ifnum#1<0
      \forest@node@nbarthchildid{-#1}%
    \else
      \forest@node@nornbarthchildid@error
    \fi
  \fi
}
\def\forest@node@nornbarthchildid@error{%
  \PackageError{forest}{In \string\forest@node@nornbarthchildid, n should !=0}{}%
}
\def\forest@node@previousleafid{%
  \expandnumberarg\forest@node@Previousleafid{\forest@cn}%
}
\def\forest@node@Previousleafid#1{%
  \ifnum\forestOve{#1}{@previous}=0
    \@escapeif{\expandnumberarg\forest@node@previousleafid@Goup{#1}}%
  \else
    \expandnumberarg\forest@node@previousleafid@Godown{\forestOve{#1}{@previous}}%
  \fi
}
\def\forest@node@previousleafid@Goup#1{%
  \ifnum\forestOve{#1}{@parent}=0
    \PackageError{forest}{get previous leaf: this is the first leaf}{}%
  \else
    \@escapeif{\expandnumberarg\forest@node@Previousleafid{\forestOve{#1}{@parent}}}%
  \fi
}
\def\forest@node@previousleafid@Godown#1{%
  \ifnum\forestOve{#1}{@last}=0
    #1%
  \else
    \@escapeif{\expandnumberarg\forest@node@previousleafid@Godown{\forestOve{#1}{@last}}}%
  \fi
}
\def\forest@node@nextleafid{%
  \expandnumberarg\forest@node@Nextleafid{\forest@cn}%
}
\def\forest@node@Nextleafid#1{%
```

```
  \ifnum\forestOve{#1}{@next}=0
    \@escapeif{\expandnumberarg\forest@node@nextleafid@Goup{#1}}%
  \else
    \expandnumberarg\forest@node@nextleafid@Godown{\forestOve{#1}{@next}}%
  \fi
}
\def\forest@node@nextleafid@Goup#1{%
  \ifnum\forestOve{#1}{@parent}=0
    \PackageError{forest}{get next leaf: this is the last leaf}{}%
  \else
    \@escapeif{\expandnumberarg\forest@node@Nextleafid{\forestOve{#1}{@parent}}}%
  \fi
}
\def\forest@node@nextleafid@Godown#1{%
  \ifnum\forestOve{#1}{@first}=0
    #1%
  \else
    \@escapeif{\expandnumberarg\forest@node@nextleafid@Godown{\forestOve{#1}{@first}}}%
  \fi
}
\def\forest@node@linearnextid{%
  \ifnum\forestove{@first}=0
    \expandafter\forest@node@linearnextnotdescendantid
  \else
    \forestove{@first}%
  \fi
}
\def\forest@node@linearnextnotdescendantid{%
  \expandnumberarg\forest@node@Linearnextnotdescendantid{\forest@cn}%
}
\def\forest@node@Linearnextnotdescendantid#1{%
  \ifnum\forestOve{#1}{@next}=0
    \@escapeif{\expandnumberarg\forest@node@Linearnextnotdescendantid{\forestOve{#1}{@parent}}}%
  \else
    \forestOve{#1}{@next}%
  \fi
}
\def\forest@node@linearpreviousid{%
  \ifnum\forestove{@previous}=0
    \forestove{@parent}%
  \else
    \forest@node@previousleafid
  \fi
}
\def\forest@ifancestorof#1{% is the current node an ancestor of #1? Yes: #2, no: #3
  \expandnumberarg\forest@ifancestorof@{\forestOve{#1}{@parent}}%
}
\def\forest@ifancestorof@#1#2#3{%
  \ifnum#1=0
    \def\forest@ifancestorof@next{\@secondoftwo}%
  \else
    \ifnum\forest@cn=#1
      \def\forest@ifancestorof@next{\@firstoftwo}%
    \else
      \def\forest@ifancestorof@next{\expandnumberarg\forest@ifancestorof@{\forestOve{#1}{@parent}}}%
    \fi
  \fi
  \forest@ifancestorof@next{#2}{#3}%
}
```

10.3 Node walk

```
\newloop\forest@nodewalk@loop
\forestset{
  @handlers@save@currentpath/.code={%
    \edef\pgfkeyscurrentkey{\pgfkeyscurrentpath}%
    \let\forest@currentkey\pgfkeyscurrentkey
    \pgfkeys@split@path
    \edef\forest@currentpath{\pgfkeyscurrentpath}%
    \let\forest@currentname\pgfkeyscurrentname
  },
  /handlers/.step 0 args/.style={
    /forest/@handlers@save@currentpath,
    \forest@currentkey/.code={#1\forestset{node walk/every step}},
    /forest/for \forest@currentname/.style/.expanded={%
      for={\forest@currentname}{####1}%
    }
  },
  /handlers/.step 1 arg/.style={%
    /forest/@handlers@save@currentpath,
    \forest@currentkey/.code={#1\forestset{node walk/every step}},
    /forest/for \forest@currentname/.style 2 args/.expanded={%
      for={\forest@currentname=####1}{####2}%
    }
  },
  node walk/.code={%
    \forestset{%
      node walk/before walk,%
      node walk/.cd,
      #1,%
      /forest/.cd,
      node walk/after walk
    }%
  },
  for/.code 2 args={%
    \forest@forthis{%
      \pgfkeysalso{%
        node walk/before walk/.style={},%
        node walk/every step/.style={},%
        node walk/after walk/.style={/forest,if id=0{}{#2}},%
        %node walk/after walk/.style={#2},%
        node walk={#1}%
      }%
    }%
  },
  node walk/.cd,
  before walk/.code={},
  every step/.code={},
  after walk/.code={},
  current/.step 0 args={},
  current/.default=1,
  next/.step 0 args={\edef\forest@cn{\forestove{@next}}},
  next/.default=1,
  previous/.step 0 args={\edef\forest@cn{\forestove{@previous}}},
  previous/.default=1,
  parent/.step 0 args={\edef\forest@cn{\forestove{@parent}}},
  parent/.default=1,
  first/.step 0 args={\edef\forest@cn{\forestove{@first}}},
  first/.default=1,
  last/.step 0 args={\edef\forest@cn{\forestove{@last}}},
  last/.default=1,
  n/.step 1 arg={%
    \def\forest@nodewalk@temp{#1}%
    \ifx\forest@nodewalk@temp\pgfkeysnovalue@text
      \edef\forest@cn{\forestove{@next}}%
    \else
      \edef\forest@cn{\forest@node@nthchildid{#1}}%
    \fi
  },
  n'/.step 1 arg={\edef\forest@cn{\forest@node@nbarthchildid{#1}}},
  sibling/.step 0 args={%
    \edef\forest@cn{%
      \ifnum\forestove{@previous}=0
        \forestove{@next}%
      \else
        \forestove{@previous}%
      \fi
    }%
  },
  previous leaf/.step 0 args={\edef\forest@cn{\forest@node@previousleafid}},
  previous leaf/.default=1,
  next leaf/.step 0 args={\edef\forest@cn{\forest@node@nextleafid}},
  next leaf/.default=1,
  linear next/.step 0 args={\edef\forest@cn{\forest@node@linearnextid}},
  linear previous/.step 0 args={\edef\forest@cn{\forest@node@linearpreviousid}},
  first leaf/.step 0 args={%
    \forest@nodewalk@loop
      \edef\forest@cn{\forestove{@first}}%
    \unless\ifnum\forestove{@first}=0
    \forest@nodewalk@repeat
  },
  last leaf/.step 0 args={%
    \forest@nodewalk@loop
      \edef\forest@cn{\forestove{@last}}%
    \unless\ifnum\forestove{@last}=0
    \forest@nodewalk@repeat
  },
  to tier/.step 1 arg={%
    \def\forest@nodewalk@giventier{#1}%
    \forest@nodewalk@loop
      \forestoget{tier}\forest@nodewalk@tier
    \unless\ifx\forest@nodewalk@tier\forest@nodewalk@giventier
      \forestoget{@parent}\forest@cn
    \forest@nodewalk@repeat
  },
  next on tier/.step 0 args={\forest@nodewalk@nextontier},
  next on tier/.default=1,
  previous on tier/.step 0 args={\forest@nodewalk@previousontier},
  previous on tier/.default=1,
  name/.step 1 arg={\edef\forest@cn{\forest@node@Nametoid{#1}}},
  root/.step 0 args={\edef\forest@cn{\forest@node@rootid}},
  root'/.step 0 args={\edef\forest@cn{\forest@root}},
  id/.step 1 arg={\edef\forest@cn{#1}},
  % maybe it's not wise to have short-step sequences and names potentially clashing
  % .unknown/.code={%
  %   \forest@node@Ifnamedefined{\pgfkeyscurrentname}%
  %     {\pgfkeysalso{name=\pgfkeyscurrentname}}%
  %     {\expandafter\forest@nodewalk@shortsteps\pgfkeyscurrentname\forest@nodewalk@endshortsteps}%
  % },
  .unknown/.code={%
    \expandafter\forest@nodewalk@shortsteps\pgfkeyscurrentname\forest@nodewalk@endshortsteps
  },
  node walk/.style={/forest/node walk={#1}},
  trip/.code={\forest@forthis{\pgfkeysalso{#1}}},
  group/.code={\forest@go{#1}\forestset{node walk/every step}},
  % repeat is taken later from /forest/repeat
  p/.style={previous=1},
  %n/.style={next=1}, % defined in "long" n
  u/.style={parent=1},
  s/.style={sibling},
  c/.style={current=1},
  r/.style={root},
  P/.style={previous leaf=1},
  N/.style={next leaf=1},
  F/.style={first leaf=1},
  L/.style={last leaf=1},
  >/.style={next on tier=1},
  </.style={previous on tier=1},
  1/.style={n=1},
  2/.style={n=2},
  3/.style={n=3},
  4/.style={n=4},
  5/.style={n=5},
  6/.style={n=6},
  7/.style={n=7},
  8/.style={n=8},
  9/.style={n=9},
  l/.style={last=1},
  %{...} is short for group={...}
}
\def\forest@nodewalk@nextontier{%
  \forestoget{tier}\forest@nodewalk@giventier
  \edef\forest@cn{\forest@node@linearnextnotdescendantid}%
  \forest@nodewalk@loop
    \forestoget{tier}\forest@nodewalk@tier
  \unless\ifx\forest@nodewalk@tier\forest@nodewalk@giventier
    \edef\forest@cn{\forest@node@linearnextid}%
  \forest@nodewalk@repeat
}
\def\forest@nodewalk@previousontier{%
  \forestoget{tier}\forest@nodewalk@giventier
  \forest@nodewalk@loop
    \edef\forest@cn{\forest@node@linearpreviousid}%
    \forestoget{tier}\forest@nodewalk@tier
  \unless\ifx\forest@nodewalk@tier\forest@nodewalk@giventier
  \forest@nodewalk@repeat
}
\def\forest@nodewalk@shortsteps{%
  \futurelet\forest@nodewalk@nexttoken\forest@nodewalk@shortsteps@
}
\def\forest@nodewalk@shortsteps@#1{%
  \ifx\forest@nodewalk@nexttoken\forest@nodewalk@endshortsteps
  \else
    \ifx\forest@nodewalk@nexttoken\bgroup
      \pgfkeysalso{group=#1}%
      \@escapeifif\forest@nodewalk@shortsteps
    \else
      \pgfkeysalso{#1}%
      \@escapeifif\forest@nodewalk@shortsteps
    \fi
  \fi
}
\def\forest@go#1{%
```

```
  {%
    \forestset{%
      node walk/before walk/.code={},%
      node walk/every step/.code={},%
      node walk/after walk/.code={},%
      node walk={#1}%
    }%
    \expandafter
  }%
  \expandafter\def\expandafter\forest@cn\expandafter{\forest@cn}%
}
```

10.4 Node options

10.4.1 Option-declaration mechanism

Common code for declaring options.

```
\def\forest@declarehandler#1#2#3{%#1=handler for specific type,#2=option name,#3=default value
  \pgfkeyssetvalue{/forest/#2}{#3}%
  \appto\forest@node@init{\forestoinit{#2}}%
  \forest@convert@others@to@underscores{#2}\forest@pgfmathoptionname
  \edef\forest@marshal{%
    \noexpand#1{/forest/#2}{/forest}{#2}{\forest@pgfmathoptionname}%
  }\forest@marshal
}
\def\forest@def@with@pgfeov#1#2{% \pgfeov mustn't occur in the arg of the .code handler!!!
  \long\def#1##1\pgfeov{#2}%
}
```

Option-declaration handlers.

```
\newtoks\forest@temp@toks
\def\forest@declaretoks@handler#1#2#3#4{%
  \forest@declaretoks@handler@A{#1}{#2}{#3}{#4}{}%
}
\def\forest@declarekeylist@handler#1#2#3#4{%
  \forest@declaretoks@handler@A{#1}{#2}{#3}{#4}{,}%
  \pgfkeysgetvalue{#1/.@cmd}\forest@temp
  \pgfkeyslet{#1'/.@cmd}\forest@temp
  \pgfkeyssetvalue{#1'/option@name}{#3}%
  \pgfkeysgetvalue{#1+/.@cmd}\forest@temp
  \pgfkeyslet{#1/.@cmd}\forest@temp
 }
\def\forest@declaretoks@handler@A#1#2#3#4#5{% #1=key,#2=path,#3=name,#4=pgfmathname,#5=infix
  \pgfkeysalso{%
    #1/.code={\forestOset{\forest@setter@node}{#3}{##1}},
    #1+/.code={\forestOappto{\forest@setter@node}{#3}{#5##1}},
    #1-/.code={\forestOpreto{\forest@setter@node}{#3}{##1#5}},
    #2/if #3/.code n args={3}{%
      \forestoget{#3}\forest@temp@option@value
      \edef\forest@temp@compared@value{\unexpanded{##1}}%
      \ifx\forest@temp@option@value\forest@temp@compared@value
        \pgfkeysalso{##2}%
      \else
        \pgfkeysalso{##3}%
      \fi
    },
    #2/if in #3/.code n args={3}{%
      \forestoget{#3}\forest@temp@option@value
      \edef\forest@temp@compared@value{\unexpanded{##1}}%
      \expandafter\expandafter\expandafter\pgfutil@in@\expandafter\expandafter\expandafter{\expandafter\fores
      \ifpgfutil@in@
        \pgfkeysalso{##2}%
      \else
        \pgfkeysalso{##3}%
      \fi
    },
    #2/where #3/.style n args={3}{for tree={#2/if #3={##1}{##2}{##3}}},
    #2/where in #3/.style n args={3}{for tree={#2/if in #3={##1}{##2}{##3}}}
  }%
  \pgfkeyssetvalue{#1/option@name}{#3}%
  \pgfkeyssetvalue{#1+/option@name}{#3}%
  \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@toks{##1}{#3}}%
}
\def\forest@declareautowrappedtoks@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname,#5=infix
  \forest@declaretoks@handler{#1}{#2}{#3}{#4}%
  \pgfkeysgetvalue{#1/.@cmd}\forest@temp
  \pgfkeyslet{#1'/.@cmd}\forest@temp
  \pgfkeysalso{#1/.style={#1'/.wrap value={##1}}}%
  \pgfkeyssetvalue{#1'/option@name}{#3}%
  \pgfkeysgetvalue{#1+/.@cmd}\forest@temp
  \pgfkeyslet{#1+'/.@cmd}\forest@temp
  \pgfkeysalso{#1+/.style={#1+'/.wrap value={##1}}}%
  \pgfkeyssetvalue{#1+'/option@name}{#3}%
  \pgfkeysgetvalue{#1-/.@cmd}\forest@temp
  \pgfkeyslet{#1-'/.@cmd}\forest@temp
  \pgfkeysalso{#1-/.style={#1-'/.wrap value={##1}}}%
  \pgfkeyssetvalue{#1-'/option@name}{#3}%
}
\def\forest@declarereadonlydimen@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
  \pgfkeysalso{%
    #2/if #3/.code n args={3}{%
      \forestoget{#3}\forest@temp@option@value
      \ifdim\forest@temp@option@value=##1\relax
        \pgfkeysalso{##2}%
      \else
        \pgfkeysalso{##3}%
      \fi
    },
    #2/where #3/.style n args={3}{for tree={#2/if #3={##1}{##2}{##3}}},
  }%
  \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@dimen{##1}{#3}}%
}
\def\forest@declaredimen@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
  \forest@declarereadonlydimen@handler{#1}{#2}{#3}{#4}%
  \pgfkeysalso{%
    #1/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \forestOlet{\forest@setter@node}{#3}\forest@temp
    },
    #1+/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \pgfutil@tempdima=\forestove{#3}
      \advance\pgfutil@tempdima\forest@temp\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
    #1-/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \pgfutil@tempdima=\forestove{#3}
      \advance\pgfutil@tempdima-\forest@temp\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
    #1*/.style={%
      #1={#4()*(##1)}%
    },
    #1:/.style={%
      #1={#4()/(##1)}%
    },
    #1'/.code={%
      \pgfutil@tempdima=##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
    #1'+/.code={%
      \pgfutil@tempdima=\forestove{#3}\relax
      \advance\pgfutil@tempdima##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
    #1'-/.code={%
      \pgfutil@tempdima=\forestove{#3}\relax
      \advance\pgfutil@tempdima-##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
    #1'*/.style={%
      \pgfutil@tempdima=\forestove{#3}\relax
      \multiply\pgfutil@tempdima##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
    #1':/.style={%
      \pgfutil@tempdima=\forestove{#3}\relax
      \divide\pgfutil@tempdima##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\pgfutil@tempdima}%
    },
  }%
  \pgfkeyssetvalue{#1/option@name}{#3}%
  \pgfkeyssetvalue{#1+/option@name}{#3}%
  \pgfkeyssetvalue{#1-/option@name}{#3}%
  \pgfkeyssetvalue{#1*/option@name}{#3}%
  \pgfkeyssetvalue{#1:/option@name}{#3}%
  \pgfkeyssetvalue{#1'/option@name}{#3}%
  \pgfkeyssetvalue{#1'+/option@name}{#3}%
  \pgfkeyssetvalue{#1'-/option@name}{#3}%
  \pgfkeyssetvalue{#1'*/option@name}{#3}%
  \pgfkeyssetvalue{#1':/option@name}{#3}%
}
\def\forest@declarereadonlycount@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
  \pgfkeysalso{
    #2/if #3/.code n args={3}{%
      \forestoget{#3}\forest@temp@option@value
      \ifnum\forest@temp@option@value=##1\relax
        \pgfkeysalso{##2}%
      \else
        \pgfkeysalso{##3}%
      \fi
    },
    #2/where #3/.style n args={3}{for tree={#2/if #3={##1}{##2}{##3}}},
  }%
  \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@count{##1}{#3}}%
}
\def\forest@declarecount@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
  \forest@declarereadonlycount@handler{#1}{#2}{#3}{#4}%
  \pgfkeysalso{
    #1/.code={%
      \pgfmathtruncatemacro\forest@temp{##1}%
      \forestOlet{\forest@setter@node}{#3}\forest@temp
    },
    #1+/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \c@pgf@counta=\forestove{#3}\relax
      \advance\c@pgf@counta\forest@temp\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1-/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \c@pgf@counta=\forestove{#3}\relax
      \advance\c@pgf@counta-\forest@temp\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1*/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \c@pgf@counta=\forestove{#3}\relax
      \multiply\c@pgf@counta\forest@temp\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1:/.code={%
      \pgfmathsetlengthmacro\forest@temp{##1}%
      \c@pgf@counta=\forestove{#3}\relax
      \divide\c@pgf@counta\forest@temp\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1’/.code={%
      \c@pgf@counta=##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1’+/.code={%
      \c@pgf@counta=\forestove{#3}\relax
      \advance\c@pgf@counta##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1’-/.code={%
      \c@pgf@counta=\forestove{#3}\relax
      \advance\c@pgf@counta-##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1’*/.style={%
      \c@pgf@counta=\forestove{#3}\relax
      \multiply\c@pgf@counta##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
    #1’:/.style={%
      \c@pgf@counta=\forestove{#3}\relax
      \divide\c@pgf@counta##1\relax
      \forestOeset{\forest@setter@node}{#3}{\the\c@pgf@counta}%
    },
  }%
  \pgfkeyssetvalue{#1/option@name}{#3}%
  \pgfkeyssetvalue{#1+/option@name}{#3}%
  \pgfkeyssetvalue{#1-/option@name}{#3}%
  \pgfkeyssetvalue{#1*/option@name}{#3}%
  \pgfkeyssetvalue{#1:/option@name}{#3}%
  \pgfkeyssetvalue{#1’/option@name}{#3}%
  \pgfkeyssetvalue{#1’+/option@name}{#3}%
  \pgfkeyssetvalue{#1’-/option@name}{#3}%
  \pgfkeyssetvalue{#1’*/option@name}{#3}%
  \pgfkeyssetvalue{#1’:/option@name}{#3}%
}
\def\forest@declareboolean@handler#1#2#3#4{% #1=key,#2=path,#3=name,#4=pgfmathname
```

```
  \pgfkeysalso{%
    #1/.code={%
      \ifstrequal{##1}{1}{%
        \forestOset{\forest@setter@node}{#3}{1}%
      }{%
        \pgfmathifthenelse{##1}{1}{0}%
        \forestOlet{\forest@setter@node}{#3}\pgfmathresult
      }%
    },
    #1/.default=1,
    #2/not #3/.code={\forestOset{\forest@setter@node}{#3}{0}},
    #2/if #3/.code 2 args={%
      \forestoget{#3}\forest@temp@option@value
      \ifnum\forest@temp@option@value=1
        \pgfkeysalso{##1}%
      \else
        \pgfkeysalso{##2}%
      \fi
    },
    #2/where #3/.style 2 args={for tree={#2/if #3={##1}{##2}}}
  }%
  \pgfkeyssetvalue{#1/option@name}{#3}%
  \pgfmathdeclarefunction{#4}{1}{\forest@pgfmathhelper@attribute@count{##1}{#3}}%
}
\pgfkeys{/forest,
  declare toks/.code 2 args={%
    \forest@declarehandler\forest@declaretoks@handler{#1}{#2}%
  },
  declare autowrapped toks/.code 2 args={%
    \forest@declarehandler\forest@declareautowrappedtoks@handler{#1}{#2}%
  },
  declare keylist/.code 2 args={%
    \forest@declarehandler\forest@declarekeylist@handler{#1}{#2}%
  },
  declare readonly dimen/.code={%
    \forest@declarehandler\forest@declarereadonlydimen@handler{#1}{}%
  },
  declare dimen/.code 2 args={%
    \forest@declarehandler\forest@declaredimen@handler{#1}{#2}%
  },
  declare readonly count/.code={%
    \forest@declarehandler\forest@declarereadonlycount@handler{#1}{}%
  },
  declare count/.code 2 args={%
    \forest@declarehandler\forest@declarecount@handler{#1}{#2}%
  },
  declare boolean/.code 2 args={%
    \forest@declarehandler\forest@declareboolean@handler{#1}{#2}%
  },
  /handlers/.pgfmath/.code={%
    \pgfmathparse{#1}%
    \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\pgfmathresult}%
  },
  /handlers/.wrap value/.code={%
    \edef\forest@handlers@wrap@currentpath{\pgfkeyscurrentpath}%
    \pgfkeysgetvalue{\forest@handlers@wrap@currentpath/option@name}\forest@currentoptionname
    \expandafter\forestoget\expandafter{\forest@currentoptionname}\forest@option@value
    \forest@def@with@pgfeov\forest@wrap@code{#1}%
    \expandafter\edef\expandafter\forest@wrapped@value\expandafter{\expandafter\expandonce\expandafter{\expan
    \pgfkeysalso{\forest@handlers@wrap@currentpath/.expand once=\forest@wrapped@value}%
  },
  /handlers/.wrap pgfmath arg/.code 2 args={%
    \pgfmathparse{#2}\let\forest@wrap@arg@i\pgfmathresult
    \edef\forest@wrap@args{{\expandonce\forest@wrap@arg@i}}%
    \def\forest@wrap@code##1{#1}%
    \expandafter\expandafter\expandafter\forest@temp@toks\expandafter\expandafter\expandafter{\expandafter\fo
    \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\the\forest@temp@toks}%
  },
  /handlers/.wrap 2 pgfmath args/.code n args={3}{%
    \pgfmathparse{#2}\let\forest@wrap@arg@i\pgfmathresult
    \pgfmathparse{#3}\let\forest@wrap@arg@ii\pgfmathresult
    \edef\forest@wrap@args{{\expandonce\forest@wrap@arg@i}{\expandonce\forest@wrap@arg@ii}}%
    \def\forest@wrap@code##1##2{#1}%
    \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\forest@wrapped\expandafter\e
    \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\forest@wrapped}%
  },
  /handlers/.wrap 3 pgfmath args/.code n args={4}{%
    \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{}{}{}{}{}{3}%
    \forest@wrap@n@pgfmath@do{#1}{3}},
  /handlers/.wrap 4 pgfmath args/.code n args={5}{%
    \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{}{}{}{}{4}%
    \forest@wrap@n@pgfmath@do{#1}{4}},
  /handlers/.wrap 5 pgfmath args/.code n args={6}{%
    \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{}{}{}{5}%
    \forest@wrap@n@pgfmath@do{#1}{5}},
  /handlers/.wrap 6 pgfmath args/.code n args={7}{%
    \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{#7}{}{}{6}%
    \forest@wrap@n@pgfmath@do{#1}{6}},
  /handlers/.wrap 7 pgfmath args/.code n args={8}{%
    \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{#7}{#8}{}{7}%
    \forest@wrap@n@pgfmath@do{#1}{7}},
  /handlers/.wrap 8 pgfmath args/.code n args={9}{%
    \forest@wrap@n@pgfmath@args{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}{8}%
    \forest@wrap@n@pgfmath@do{#1}{8}},
}
\def\forest@wrap@n@pgfmath@args#1#2#3#4#5#6#7#8#9{%
  \pgfmathparse{#1}\let\forest@wrap@arg@i\pgfmathresult
  \ifnum#9>1 \pgfmathparse{#2}\let\forest@wrap@arg@ii\pgfmathresult\fi
  \ifnum#9>2 \pgfmathparse{#3}\let\forest@wrap@arg@iii\pgfmathresult\fi
  \ifnum#9>3 \pgfmathparse{#4}\let\forest@wrap@arg@iv\pgfmathresult\fi
  \ifnum#9>4 \pgfmathparse{#5}\let\forest@wrap@arg@v\pgfmathresult\fi
  \ifnum#9>5 \pgfmathparse{#6}\let\forest@wrap@arg@vi\pgfmathresult\fi
  \ifnum#9>6 \pgfmathparse{#7}\let\forest@wrap@arg@vii\pgfmathresult\fi
  \ifnum#9>7 \pgfmathparse{#8}\let\forest@wrap@arg@viii\pgfmathresult\fi
  \edef\forest@wrap@args{%
    {\expandonce\forest@wrap@arg@i}
    \ifnum#9>1 {\expandonce\forest@wrap@arg@ii}\fi
    \ifnum#9>2 {\expandonce\forest@wrap@arg@iii}\fi
    \ifnum#9>3 {\expandonce\forest@wrap@arg@iv}\fi
    \ifnum#9>4 {\expandonce\forest@wrap@arg@v}\fi
    \ifnum#9>5 {\expandonce\forest@wrap@arg@vi}\fi
    \ifnum#9>6 {\expandonce\forest@wrap@arg@vii}\fi
    \ifnum#9>7 {\expandonce\forest@wrap@arg@viii}\fi
  }%
}
\def\forest@wrap@n@pgfmath@do#1#2{%
  \ifcase#2\relax
  \or\def\forest@wrap@code##1{#1}%
  \or\def\forest@wrap@code##1##2{#1}%
  \or\def\forest@wrap@code##1##2##3{#1}%
  \or\def\forest@wrap@code##1##2##3##4{#1}%
  \or\def\forest@wrap@code##1##2##3##4##5{#1}%
  \or\def\forest@wrap@code##1##2##3##4##5##6{#1}%
  \or\def\forest@wrap@code##1##2##3##4##5##6##7{#1}%
  \or\def\forest@wrap@code##1##2##3##4##5##6##7##8{#1}%
  \fi
  \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\forest@wrapped\expandafter\exp
  \pgfkeysalso{\pgfkeyscurrentpath/.expand once=\forest@wrapped}%
}
```

10.4.2 Declaring options

```
\def\forest@node@setname#1{%
  \forestoeset{name}{#1}%
  \csedef{forest@id@of@#1}{\forest@cn}%
}
\def\forest@node@Nametoid#1{% #1 = name
  \csname forest@id@of@#1\endcsname
}
\def\forest@node@Ifnamedefined#1{% #1 = name, #2=true,#3=false
  \ifcsname forest@id@of@#1\endcsname
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
}
\def\forest@node@setalias#1{%
  \csedef{forest@id@of@#1}{\forest@cn}%
}
\def\forest@node@Setalias#1#2{%
  \csedef{forest@id@of@#2}{#1}%
}
\forestset{
  TeX/.code={#1},
  TeX'/.code={\appto\forest@externalize@loadimages{#1}#1},
  TeX''/.code={\appto\forest@externalize@loadimages{#1}},
  declare toks={name}{},
  name/.code={% override the default setter
    \forest@node@setname{#1}%
  },
  alias/.code={\forest@node@setalias{#1}},
  begin draw/.code={\begin{tikzpicture}},
  end draw/.code={\end{tikzpicture}},
  begin forest/.code={},
  end forest/.code={},
  declare autowrapped toks={content}{},
  declare count={grow}{270},
  TeX={% a hack for grow-reversed connection, and compass-based grow specification
    \pgfkeysgetvalue{/forest/grow/.@cmd}\forest@temp
    \pgfkeyslet{/forest/grow@@/.@cmd}\forest@temp
  },
  grow/.style={grow@={#1},reversed=0},
  grow'/.style={grow@={#1},reversed=1},
  grow''/.style={grow@={#1}},
  grow@/.is choice,
  grow@/east/.style={/forest/grow@@=0},
  grow@/north east/.style={/forest/grow@@=45},
  grow@/north/.style={/forest/grow@@=90},
  grow@/north west/.style={/forest/grow@@=135},
  grow@/west/.style={/forest/grow@@=180},
  grow@/south west/.style={/forest/grow@@=225},
  grow@/south/.style={/forest/grow@@=270},
  grow@/south east/.style={/forest/grow@@=315},
  grow@/.unknown/.code={\let\forest@temp@grow\pgfkeyscurrentname
    \pgfkeysalso{/forest/grow@@/.expand once=\forest@temp@grow}},
  declare boolean={reversed}{0},
  declare toks={parent anchor}{},
  declare toks={child anchor}{},
  declare toks={anchor}{base},
  declare toks={calign}{midpoint},
  TeX={%
    \pgfkeysgetvalue{/forest/calign/.@cmd}\forest@temp
    \pgfkeyslet{/forest/calign'/.@cmd}\forest@temp
  },
  calign/.is choice,
  calign/child/.style={calign'=child},
  calign/first/.style={calign'=child,calign primary child=1},
  calign/last/.style={calign'=child,calign primary child=-1},
  calign with current/.style={for parent/.wrap pgfmath arg={calign=child,calign primary child=##1}{n}},
  calign with current edge/.style={for parent/.wrap pgfmath arg={calign=child edge,calign primary child=##1}{
  calign/child edge/.style={calign'=child edge},
  calign/midpoint/.style={calign'=midpoint},
  calign/center/.style={calign'=midpoint,calign primary child=1,calign secondary child=-1},
  calign/edge midpoint/.style={calign'=edge midpoint},
  calign/fixed angles/.style={calign'=fixed angles},
  calign/fixed edge angles/.style={calign'=fixed edge angles},
  calign/.unknown/.code={\PackageError{forest}{unknown calign '\pgfkeyscurrentname'}{}},
  declare count={calign primary child}{1},
  declare count={calign secondary child}{-1},
  declare count={calign primary angle}{-35},
  declare count={calign secondary angle}{35},
  calign child/.style={calign primary child={#1}},
  calign angle/.style={calign primary angle={-#1},calign secondary angle={#1}},
  declare toks={tier}{},
  declare toks={fit}{tight},
  declare boolean={ignore}{0},
  declare boolean={ignore edge}{0},
  no edge/.style={edge'={},ignore edge},
  declare keylist={edge}{draw},
  declare toks={edge path}{%
    \noexpand\path[\forestoption{edge}]%
    (\forestOve{\forestove{@parent}}{name}.parent anchor)--(\forestove{name}.child anchor)\forestoption{edge
  triangle/.style={edge path={%
      \noexpand\path[\forestoption{edge}]%
      (\forestove{name}.north east)--(\forestOve{\forestove{@parent}}{name}.south)--(\forestove{name}.north w
  declare toks={edge label}{},
  declare boolean={phantom}{0},
  baseline/.style={alias={forest@baseline@node}},
  declare readonly count={n},
  declare readonly count={n'},
  declare readonly count={n children},
  declare readonly count={level},
  declare dimen=x{},
  declare dimen=y{},
  declare dimen={s}{0pt},
  declare dimen={l}{6ex}, % just in case: should be set by the calibration
  declare dimen={s sep}{0.6666em},
  declare dimen={l sep}{1ex},  % just in case: calibration!
  declare keylist={node options}{},
  declare toks={tikz}{},
  afterthought/.style={tikz+={#1}},
  label/.style={tikz={\path[late options={%
        name=\forestoption{name},label={#1}}];}},
  pin/.style={tikz={\path[late options={%
```

```
      name=\forestoption{name},pin={#1}}];}},
declare toks={content format}{\forestoption{content}},
math content/.style={content format={\ensuremath{\forestoption{content}}}},
declare toks={node format}{%
  \noexpand\node
  [\forestoption{node options},anchor=\forestoption{anchor}]%
  (\forestoption{name})%
  {\foresteoption{content format}};%
},
tabular@environment/.style={content format={%
  \noexpand\begin{tabular}[\forestoption{base}]{\forestoption{align}}%
    \forestoption{content}%
   \noexpand\end{tabular}%
}},
declare toks={align}{},
TeX={\pgfkeysgetvalue{/forest/align/.@cmd}\forest@temp
  \pgfkeyslet{/forest/align'/.@cmd}\forest@temp},
align/.is choice,
align/.unknown/.code={%
  \edef\forest@marshal{%
    \noexpand\pgfkeysalso{%
      align'={\pgfkeyscurrentname},%
      tabular@environment
    }%
  }\forest@marshal
},
align/center/.style={align'={@{}c@{}},tabular@environment},
align/left/.style={align'={@{}l@{}},tabular@environment},
align/right/.style={align'={@{}r@{}},tabular@environment},
declare toks={base}{t},
TeX={\pgfkeysgetvalue{/forest/base/.@cmd}\forest@temp
  \pgfkeyslet{/forest/base'/.@cmd}\forest@temp},
base/.is choice,
base/top/.style={base'=t},
base/bottom/.style={base'=b},
base/.unknown/.style={base'/.expand once=\pgfkeyscurrentname},
.unknown/.code={%
  \expandafter\pgfutil@in@\expandafter.\expandafter{\pgfkeyscurrentname}%
  \ifpgfutil@in@
    \expandafter\forest@relatednode@option@setter\pgfkeyscurrentname=#1\forest@END
  \else
    \edef\forest@marshal{%
      \noexpand\pgfkeysalso{node options={\pgfkeyscurrentname=\unexpanded{#1}}}%
    }\forest@marshal
  \fi
},
get node boundary/.code={%
  \forestoget{boundary}\forest@node@boundary
  \def#1{}%
  \forest@extendpath#1\forest@node@boundary{\pgfpoint{\forestove{x}}{\forestove{y}}}%
},
% get min l tree boundary/.code={%
%   \forest@get@tree@boundary{negative}{\the\numexpr\forestove{grow}-90\relax}#1},
% get max l tree boundary/.code={%
%   \forest@get@tree@boundary{positive}{\the\numexpr\forestove{grow}-90\relax}#1},
get min s tree boundary/.code={%
  \forest@get@tree@boundary{negative}{\forestove{grow}}#1},
get max s tree boundary/.code={%
  \forest@get@tree@boundary{positive}{\forestove{grow}}#1},
fit to tree/.code={%
  \pgfkeysalso{%
```

```
   /forest/get min s tree boundary=\forest@temp@negative@boundary,
   /forest/get max s tree boundary=\forest@temp@positive@boundary
  }%
  \edef\forest@temp@boundary{\expandonce{\forest@temp@negative@boundary}\expandonce{\forest@temp@positive@b
  \forest@path@getboundingrectangle@xy\forest@temp@boundary
  \pgfkeysalso{inner sep=0,fit/.expanded={(\the\pgf@xa,\the\pgf@ya)(\the\pgf@xb,\the\pgf@yb)}}%
 },
 use as bounding box/.style={%
  before drawing tree={
   tikz+/.expanded={%
    \noexpand\pgfresetboundingbox
    \noexpand\useasboundingbox
    ($(.anchor)+(\forestoption{min x},\forestoption{min y})$)
    rectangle
    ($(.anchor)+(\forestoption{max x},\forestoption{max y})$)
    ;
   }
  }
 },
 use as bounding box'/.style={%
  before drawing tree={
   tikz+/.expanded={%
    \noexpand\pgfresetboundingbox
    \noexpand\useasboundingbox
    ($(.anchor)+(\forestoption{min x}+\pgfkeysvalueof{/pgf/outer xsep}/2+\pgfkeysvalueof{/pgf/inner xsep}
    rectangle
    ($(.anchor)+(\forestoption{max x}-\pgfkeysvalueof{/pgf/outer xsep}/2-\pgfkeysvalueof{/pgf/inner xsep}
    ;
   }
  }
 },
}%
\def\forest@get@tree@boundary#1#2#3{%#1=pos/neg,#2=grow,#3=receiving cs
 \def#3{}%
 \forest@node@getedge{#1}{#2}\forest@temp@boundary
 \forest@extendpath#3\forest@temp@boundary{\pgfpoint{\forestove{x}}{\forestove{y}}}%
}
\def\forest@setter@node{\forest@cn}%
\def\forest@relatednode@option@setter#1.#2=#3\forest@END{%
 \forest@forthis{%
  \forest@nameandgo{#1}%
  \let\forest@setter@node\forest@cn
 }%
 \pgfkeysalso{#2={#3}}%
 \def\forest@setter@node{\forest@cn}%
}%
```

10.4.3 Option propagation

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

```
\forestset{
 for tree/.code={\forest@node@foreach{\pgfkeysalso{#1}}},
 if/.code n args={3}{%
  \pgfmathparse{#1}%
  \ifnum\pgfmathresult=0 \pgfkeysalso{#3}\else\pgfkeysalso{#2}\fi
 },
 where/.style n args={3}{for tree={if={#1}{#2}{#3}}},
 for descendants/.code={\forest@node@foreachdescendant{\pgfkeysalso{#1}}},
 for all next/.style={for next={#1,for all next={#1}}},
 for all previous/.style={for previous={#1,for all previous={#1}}},
 for siblings/.style={for all previous={#1},for all next={#1}},
 for ancestors/.style={for parent={#1,for ancestors={#1}}},
 for ancestors'/.style={#1,for ancestors={#1}},
 for children/.code={\forest@node@foreachchild{\pgfkeysalso{#1}}},
 for c-commanded={for sibling={for tree={#1}}},
 for c-commanders={for sibling={#1},for parent={for c-commanders={#1}}}
}
```

A bit of complication to allow for nested repeats without TEX groups.

```
\newcount\forest@repeat@key@depth
\forestset{%
  repeat/.code 2 args={%
    \advance\forest@repeat@key@depth1
    \pgfmathparse{int(#1)}%
    \csedef{forest@repeat@key@\the\forest@repeat@key@depth}{\pgfmathresult}%
    \expandafter\newloop\csname forest@repeat@key@loop@\the\forest@repeat@key@depth\endcsname
    \def\forest@marshal{%
      \csname forest@repeat@key@loop@\the\forest@repeat@key@depth\endcsname
        \forest@temp@count=\csname forest@repeat@key@\the\forest@repeat@key@depth\endcsname\relax
      \ifnum\forest@temp@count>0
        \advance\forest@temp@count-1
        \csedef{forest@repeat@key@\the\forest@repeat@key@depth}{\the\forest@temp@count}%
        \pgfkeysalso{#2}%
    }%
    \expandafter\forest@marshal\csname forest@repeat@key@repeat@\the\forest@repeat@key@depth\endcsname
    \advance\forest@repeat@key@depth-1
  },
}
\pgfkeysgetvalue{/forest/repeat/.@cmd}\forest@temp
\pgfkeyslet{/forest/node walk/repeat/.@cmd}\forest@temp
%
```

10.4.4 pgfmath extensions

```
\pgfmathdeclarefunction{strequal}{2}{%
  \ifstrequal{#1}{#2}{\def\pgfmathresult{1}}{\def\pgfmathresult{0}}%
}
\pgfmathdeclarefunction{instr}{2}{%
  \pgfutil@in@{#1}{#2}%
  \ifpgfutil@in@\def\pgfmathresult{1}\else\def\pgfmathresult{0}\fi
}
\pgfmathdeclarefunction{strcat}{...}{%
  \edef\pgfmathresult{\forest@strip@braces{#1}}%
}
\def\forest@pgfmathhelper@attribute@toks#1#2{%
  \forest@forthis{%
    \forest@nameandgo{#1}%
    \forestoget{#2}\pgfmathresult
  }%
}
\def\forest@pgfmathhelper@attribute@dimen#1#2{%
  \forest@forthis{%
    \forest@nameandgo{#1}%
    \forestoget{#2}\forest@temp
    \pgfmathparse{+\forest@temp}%
  }%
}
\def\forest@pgfmathhelper@attribute@count#1#2{%
  \forest@forthis{%
    \forest@nameandgo{#1}%
    \forestoget{#2}\forest@temp
    \pgfmathtruncatemacro\pgfmathresult{\forest@temp}%
  }%
}
\pgfmathdeclarefunction{id}{1}{%
  \forest@forthis{%
    \forest@nameandgo{#1}%
    \let\pgfmathresult\forest@cn
  }%
}
\forestset{%
  if id/.code n args={3}{%
    \ifnum#1=\forest@cn\relax
      \pgfkeysalso{#2}%
    \else
      \pgfkeysalso{#3}%
    \fi
  },
  where id/.style n args={3}{for tree={if id={#1}{#2}{#3}}}
}
```

10.5 Dynamic tree

```
\def\forest@last@node{0}
\def\forest@nodehandleby@name@nodewalk@or@bracket#1{%
  \ifx\pgfkeysnovalue#1%
    \edef\forest@last@node{\forest@node@Nametoid{forest@last@node}}%
  \else
    \forest@nodehandleby@nnb@checkfirst#1\forest@END
  \fi
}
\def\forest@nodehandleby@nnb@checkfirst#1#2\forest@END{%
  \ifx[#1%]
    \forest@create@node{#1#2}%
  \else
    \forest@forthis{%
      \forest@nameandgo{#1#2}%
      \let\forest@last@node\forest@cn
    }%
  \fi
}
\def\forest@create@node#1{% #1=bracket representation
  \bracketParse{\forest@create@collectafterthought}%
               \forest@last@node=#1\forest@end@create@node
}
\def\forest@create@collectafterthought#1\forest@end@create@node{%
  \forestOletO{\forest@last@node}{delay}{\forest@last@node}{given options}%
  \forestOset{\forest@last@node}{given options}{}%
  \forestOeappto{\forest@last@node}{delay}{,\unexpanded{#1}}%
}
\def\forest@create@collectafterthought#1\forest@end@create@node{%
  \forest@node@Foreach{\forest@last@node}{%
    \forestoleto{delay}{given options}%
    \forestoset{given options}{}%
  }%
  \forestOeappto{\forest@last@node}{delay}{,\unexpanded{#1}}%
}
\def\forest@remove@node#1{%
  \forest@node@Remove{#1}%
}
\def\forest@append@node#1#2{%
  \forest@node@Remove{#2}%
  \forest@node@Append{#1}{#2}%
```

```
}
\def\forest@prepend@node#1#2{%
  \forest@node@Remove{#2}%
  \forest@node@Prepend{#1}{#2}%
}
\def\forest@insertafter@node#1#2{%
  \forest@node@Remove{#2}%
  \forest@node@Insertafter{\forestOve{#1}{@parent}}{#2}{#1}%
}
\def\forest@insertbefore@node#1#2{%
  \forest@node@Remove{#2}%
  \forest@node@Insertbefore{\forestOve{#1}{@parent}}{#2}{#1}%
}
\def\forest@appto@do@dynamics#1#2{%
   \forest@nodehandleby@name@nodewalk@or@bracket{#2}%
   \ifcase\forest@dynamics@copyhow\relax\or
     \forest@tree@copy{\forest@last@node}\forest@last@node
   \or
     \forest@node@copy{\forest@last@node}\forest@last@node
   \fi
   \forest@node@Ifnamedefined{forest@last@node}{%
     \forestOepreto{\forest@last@node}{delay}
       {for id={\forest@node@Nametoid{forest@last@node}}{alias=forest@last@node},}%
     }{}%
   \forest@havedelayedoptionstrue
   \edef\forest@marshal{%
     \noexpand\apptotoks\noexpand\forest@do@dynamics{%
       \noexpand#1{\forest@cn}{\forest@last@node}}%
   }\forest@marshal
}
\forestset{%
  create/.code={\forest@create@node{#1}},
  append/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@dynamics\forest@append@node{#1}},
  prepend/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@dynamics\forest@prepend@node{#1}},
  insert after/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@dynamics\forest@insertafter@node{#1}},
  insert before/.code={\def\forest@dynamics@copyhow{0}\forest@appto@do@dynamics\forest@insertbefore@node{#1}}
  append'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@dynamics\forest@append@node{#1}},
  prepend'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@dynamics\forest@prepend@node{#1}},
  insert after'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@dynamics\forest@insertafter@node{#1}},
  insert before'/.code={\def\forest@dynamics@copyhow{1}\forest@appto@do@dynamics\forest@insertbefore@node{#1}
  append''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@dynamics\forest@append@node{#1}},
  prepend''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@dynamics\forest@prepend@node{#1}},
  insert after''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@dynamics\forest@insertafter@node{#1}}
  insert before''/.code={\def\forest@dynamics@copyhow{2}\forest@appto@do@dynamics\forest@insertbefore@node{#1
  remove/.code={%
    \pgfkeysalso{alias=forest@last@node}%
    \expandafter\apptotoks\expandafter\forest@do@dynamics\expandafter{%
      \expandafter\forest@remove@node\expandafter{\forest@cn}}%
  },
  set root/.code={%
    \forest@nodehandleby@name@nodewalk@or@bracket{#1}%
    \edef\forest@marshal{%
      \noexpand\apptotoks\noexpand\forest@do@dynamics{%
        \def\noexpand\forest@root{\forest@last@node}%
      }%
    }\forest@marshal
  },
  replace by/.code={\forest@replaceby@code{#1}{insert after}},
  replace by'/.code={\forest@replaceby@code{#1}{insert after'}},
  replace by''/.code={\forest@replaceby@code{#1}{insert after''}},
}
```

```
\def\forest@replaceby@code#1#2{%#1=node spec,#2=insert after[’][’]
  \ifnum\forestove{@parent}=0
    \pgfkeysalso{set root={#1}}%
  \else
    \pgfkeysalso{alias=forest@last@node,#2={#1}}%
    \eapptotoks\forest@do@dynamics{%
      \noexpand\ifnum\noexpand\forestOve{\forest@cn}{@parent}=\forestove{@parent}
        \noexpand\forest@remove@node{\forest@cn}%
      \noexpand\fi
    }%
  \fi
}
```

11 Stages

```
\forestset{
  stages/.style={
    process keylist=before typesetting nodes,
    typeset nodes stage,
    process keylist=before packing,
    pack stage,
    process keylist=before computing xy,
    compute xy stage,
    process keylist=before drawing tree,
    draw tree stage,
  },
  typeset nodes stage/.style={for root’=typeset nodes},
  pack stage/.style={for root’=pack},
  compute xy stage/.style={for root’=compute xy},
  draw tree stage/.style={for root’=draw tree},
  process keylist/.code={\forest@process@hook@keylist{#1}},
  declare keylist={given options}{},
  declare keylist={before typesetting nodes}{},
  declare keylist={before packing}{},
  declare keylist={before computing xy}{},
  declare keylist={before drawing tree}{},
  declare keylist={delay}{},
  delay/.append code={\forest@havedelayedoptionstrue},
  delay n/.style 2 args={if={#1==0}{#2}{delay@n={#1}{#2}}},
  delay@n/.style 2 args={
    if={#1==1}{delay={#2}}{delay={delay@n/.wrap pgfmath arg={{##1}{#2}}{#1-1}}}
  },
  if have delayed/.code 2 args={%
    \ifforest@havedelayedoptions\pgfkeysalso{#1}\else\pgfkeysalso{#2}\fi
  },
  typeset nodes/.code={%
    \forest@drawtree@preservenodeboxes@false
    \forest@node@foreach{\forest@node@typeset}},
  typeset nodes’/.code={%
    \forest@drawtree@preservenodeboxes@true
    \forest@node@foreach{\forest@node@typeset}},
  typeset node/.code={%
    \forest@drawtree@preservenodeboxes@false
    \forest@node@typeset
  },
  pack/.code={\forest@pack},
  pack’/.code={\forest@pack@onlythisnode},
  compute xy/.code={\forest@node@computeabsolutepositions},
  draw tree box/.store in=\forest@drawtreebox,
  draw tree box,
  draw tree/.code={%
```

```
    \forest@drawtree@preservenodeboxes@false
    \forest@node@drawtree
  },
  draw tree'/.code={%
    \forest@drawtree@preservenodeboxes@true
    \forest@node@drawtree
  },
}
\newtoks\forest@do@dynamics
\newif\ifforest@havedelayedoptions
\def\forest@process@hook@keylist#1{%
  \forest@loopa
    \forest@havedelayedoptionsfalse
    \forest@do@dynamics={}%
    \forest@fornode{\forest@root}{\forest@process@hook@keylist@{#1}}%
    \expandafter\ifstrempty\expandafter{\the\forest@do@dynamics}{}{%
      \the\forest@do@dynamics
      \forest@node@Compute@numeric@ts@info{\forest@root}%
      \forest@havedelayedoptionstrue
    }%
  \ifforest@havedelayedoptions
    \forest@node@Foreach{\forest@root}{%
      \forestoget{delay}\forest@temp@delayed
      \forestolet{#1}\forest@temp@delayed
      \forestoset{delay}{}%
    }%
  \forest@repeata
}
\def\forest@process@hook@keylist@#1{%
  \forest@node@foreach{%
    \forestoget{#1}\forest@temp@keys
    \ifdefvoid\forest@temp@keys{}{%
      \forestoset{#1}{}%
      \expandafter\forestset\expandafter{\forest@temp@keys}%
    }%
  }%
}
```

11.1 Typesetting nodes

```
\def\forest@node@typeset{%
  \let\forest@next\forest@node@typeset@
  \forestoifdefined{box}{%
    \ifforest@drawtree@preservenodeboxes@
      \let\forest@next\relax
    \fi
  }{%
    \locbox\forest@temp@box
    \forestolet{box}\forest@temp@box
  }%
  \def\forest@node@typeset@restore{}%
  \ifdefined\ifsa@tikz\forest@standalone@hack\fi
  \forest@next
  \forest@node@typeset@restore
}
\def\forest@standalone@hack{%
  \ifsa@tikz
    \let\forest@standalone@tikzpicture\tikzpicture
    \let\forest@standalone@endtikzpicture\endtikzpicture
    \let\tikzpicture\sa@orig@tikzpicture
    \let\endtikzpicture\sa@orig@endtikzpicture
    \def\forest@node@typeset@restore{%
      \let\tikzpicture\forest@standalone@tikzpicture
      \let\endtikzpicture\forest@standalone@endtikzpicture
    }%
  \fi
}
\newbox\forest@box
\def\forest@node@typeset@{%
  \forestoget{name}\forest@nodename
  \edef\forest@temp@nodeformat{\forestove{node format}}%
  \gdef\forest@smuggle{}%
  \setbox0=\hbox{%
    \begin{tikzpicture}%
      \pgfpositionnodelater{\forest@positionnodelater@save}%
      \forest@temp@nodeformat
      \pgfinterruptpath
      \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{forestcomputenodeboundary}%
      \endpgfinterruptpath
      %\forest@compute@node@boundary\forest@temp
      %\xappto\forest@smuggle{\noexpand\forestoset{boundary}{\expandonce\forest@temp}}%
      \if\relax\forestove{parent anchor}\relax
        \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{center}%
      \else
        \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{\forestove{parent anchor}}%
      \fi
      \xappto\forest@smuggle{%
        \noexpand\forestoset{parent@anchor}{%
          \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
          \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
      \if\relax\forestove{child anchor}\relax
        \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{center}%
      \else
        \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{\forestove{child anchor}}%
      \fi
      \xappto\forest@smuggle{%
        \noexpand\forestoeset{child@anchor}{%
          \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
          \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
      \if\relax\forestove{anchor}\relax
        \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{center}%
      \else
        \pgfpointanchor{\forest@pgf@notyetpositioned\forest@nodename}{\forestove{anchor}}%
      \fi
      \xappto\forest@smuggle{%
        \noexpand\forestoeset{@anchor}{%
          \noexpand\noexpand\noexpand\pgf@x=\the\pgf@x\relax
          \noexpand\noexpand\noexpand\pgf@y=\the\pgf@y\relax}}%
    \end{tikzpicture}%
  }%
  \setbox\forestove{box}=\box\forest@box % smuggle the box
  \forestolet{boundary}\forest@global@boundary
  \forest@smuggle % ... and the rest
}
\forestset{
  declare readonly dimen={min x},
  declare readonly dimen={min y},
  declare readonly dimen={max x},
  declare readonly dimen={max y},
}
\def\forest@patch@enormouscoordinateboxbounds@plus#1{%
  \expandafter\ifstrequal\expandafter{#1}{16000.0pt}{\def#1{0.0pt}}{}%
```

```
}
\def\forest@patch@enormouscoordinateboxbounds@minus#1{%
  \expandafter\ifstrequal\expandafter{#1}{-16000.0pt}{\def#1{0.0pt}}{}%
}
\def\forest@positionnodelater@save{%
  \global\setbox\forest@box=\box\pgfpositionnodelaterbox
  \xappto\forest@smuggle{\noexpand\forestoset{later@name}{\pgfpositionnodelatername}}%
  % a bug in pgf? ---well, here's a patch
  \forest@patch@enormouscoordinateboxbounds@plus\pgfpositionnodelaterminx
  \forest@patch@enormouscoordinateboxbounds@plus\pgfpositionnodelaterminy
  \forest@patch@enormouscoordinateboxbounds@minus\pgfpositionnodelatermaxx
  \forest@patch@enormouscoordinateboxbounds@minus\pgfpositionnodelatermaxy
  % end of patch
  \xappto\forest@smuggle{\noexpand\forestoset{min x}{\pgfpositionnodelaterminx}}%
  \xappto\forest@smuggle{\noexpand\forestoset{min y}{\pgfpositionnodelaterminy}}%
  \xappto\forest@smuggle{\noexpand\forestoset{max x}{\pgfpositionnodelatermaxx}}%
  \xappto\forest@smuggle{\noexpand\forestoset{max y}{\pgfpositionnodelatermaxy}}%
}
\def\forest@node@forest@positionnodelater@restore{%
  \ifforest@drawtree@preservenodeboxes@
    \let\forest@boxorcopy\copy
  \else
    \let\forest@boxorcopy\box
  \fi
  \forestoget{box}\forest@temp
  \setbox\pgfpositionnodelaterbox=\forest@boxorcopy\forest@temp
  \edef\pgfpositionnodelatername{\forestove{later@name}}%
  \edef\pgfpositionnodelaterminx{\forestove{min x}}%
  \edef\pgfpositionnodelaterminy{\forestove{min y}}%
  \edef\pgfpositionnodelatermaxx{\forestove{max x}}%
  \edef\pgfpositionnodelatermaxy{\forestove{max y}}%
}
```

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.

```
\def\forest@pack{%
  \forest@pack@computetiers
  \forest@pack@computegrowthuniformity
  \forest@@pack
}
\def\forest@@pack{%
  \ifnum\forestove{n children}>0
    \ifnum\forestove{uniform growth}>0
      \forest@pack@level@uniform
      \forest@pack@aligntiers@ofsubtree
      \forest@pack@sibling@uniform@recursive
    \else
      \forest@node@foreachchild{\forest@@pack}%
      \forest@pack@level@nonuniform
      \forest@pack@aligntiers
      \forest@pack@sibling@uniform@applyreversed
    \fi
  \fi
}
\def\forest@pack@onlythisnode{%
  \ifnum\forestove{n children}>0
    \forest@pack@computetiers
      \forest@pack@level@nonuniform
      \forest@pack@aligntiers
      \forest@pack@sibling@uniform@applyreversed
  \fi
}
```

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

```
\def\forest@pack@computegrowthuniformity{%
  \forest@node@foreachchild{\forest@pack@computegrowthuniformity}%
  \edef\forest@pack@cgu@uniformity{%
    \ifnum\forestove{n children}=0
    2\else 1\fi
  }%
  \forestoget{grow}\forest@pack@cgu@parentgrow
  \forest@node@foreachchild{%
    \ifnum\forestove{uniform growth}=0
      \def\forest@pack@cgu@uniformity{0}%
    \else
      \ifnum\forestove{uniform growth}=1
        \ifnum\forestove{grow}=\forest@pack@cgu@parentgrow\relax\else
          \def\forest@pack@cgu@uniformity{0}%
        \fi
      \fi
    \fi
  }%
  \forestolet{uniform growth}\forest@pack@cgu@uniformity
}
```

Pack children in the level dimension in a uniform tree.

```
\def\forest@pack@level@uniform{%
  \let\forest@plu@minchildl\relax
  \forestoget{grow}\forest@plu@grow
  \forest@node@foreachchild{%
    \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
    \advance\pgf@xa\forestove{l}\relax
    \ifx\forest@plu@minchildl\relax
      \edef\forest@plu@minchildl{\the\pgf@xa}%
    \else
      \ifdim\pgf@xa<\forest@plu@minchildl\relax
        \edef\forest@plu@minchildl{\the\pgf@xa}%
      \fi
    \fi
  }%
  \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
  \pgfutil@tempdima=\pgf@xb\relax
  \advance\pgfutil@tempdima -\forest@plu@minchildl\relax
  \advance\pgfutil@tempdima \forestove{l sep}\relax
  \ifdim\pgfutil@tempdima>0pt
    \forest@node@foreachchild{%
      \forestoeset{l}{\the\dimexpr\forestove{l}+\the\pgfutil@tempdima}%
    }%
  \fi
  \forest@node@foreachchild{%
    \ifnum\forestove{n children}>0
      \forest@pack@level@uniform
    \fi
  }%
}
```

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

```
\def\forest@pack@level@nonuniform{%
  \let\forest@plu@minchildl\relax
  \forestoget{grow}\forest@plu@grow
  \forest@node@foreachchild{%
    \forest@node@getedge{negative}{\forest@plu@grow}{\forest@plnu@negativechildedge}%
    \forest@node@getedge{positive}{\forest@plu@grow}{\forest@plnu@positivechildedge}%
    \def\forest@plnu@childedge{\forest@plnu@negativechildedge\forest@plnu@positivechildedge}%
    \forest@path@getboundingrectangle@ls\forest@plnu@childedge{\forest@plu@grow}%
    \advance\pgf@xa\forestove{l}\relax
    \ifx\forest@plu@minchildl\relax
      \edef\forest@plu@minchildl{\the\pgf@xa}%
    \else
      \ifdim\pgf@xa<\forest@plu@minchildl\relax
        \edef\forest@plu@minchildl{\the\pgf@xa}%
      \fi
    \fi
  }%
  \forest@node@getboundingrectangle@ls{\forest@plu@grow}%
  \pgfutil@tempdima=\pgf@xb\relax
  \advance\pgfutil@tempdima -\forest@plu@minchildl\relax
  \advance\pgfutil@tempdima \forestove{l sep}\relax
  \ifdim\pgfutil@tempdima>0pt
    \forest@node@foreachchild{%
      \forestoeset{l}{\the\dimexpr\the\pgfutil@tempdima+\forestove{l}}%
    }%
  \fi
}
```

Align tiers.

```
\def\forest@pack@aligntiers{%
  \forestoget{grow}\forest@temp@parentgrow
  \forestoget{@tiers}\forest@temp@tiers
  \forlistloop\forest@pack@aligntier@\forest@temp@tiers
}
\def\forest@pack@aligntiers@ofsubtree{%
  \forest@node@foreach{\forest@pack@aligntiers}%
}
\def\forest@pack@aligntiers@computeabsl{%
  \forestoleto{abs@l}{l}%
  \forest@node@foreachdescendant{\forest@pack@aligntiers@computeabsl@}%
}
\def\forest@pack@aligntiers@computeabsl@{%
  \forestoeset{abs@l}{\the\dimexpr\forestove{l}+\forestOve{\forestove{@parent}}{abs@l}}%
}
\def\forest@pack@aligntier@#1{%
  \forest@pack@aligntiers@computeabsl
  \pgfutil@tempdima=-\maxdimen\relax
  \def\forest@temp@currenttier{#1}%
  \forest@node@foreach{%
    \forestoget{tier}\forest@temp@tier
    \ifx\forest@temp@currenttier\forest@temp@tier
      \ifdim\pgfutil@tempdima<\forestove{abs@l}\relax
        \pgfutil@tempdima=\forestove{abs@l}\relax
      \fi
    \fi
  }%
  \ifdim\pgfutil@tempdima=-\maxdimen\relax\else
    \forest@node@foreach{%
      \forestoget{tier}\forest@temp@tier
      \ifx\forest@temp@currenttier\forest@temp@tier
        \forestoeset{l}{\the\dimexpr\pgfutil@tempdima-\forestove{abs@l}+\forestove{l}}%
      \fi
    }%
  \fi
}
```

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

```
\def\forest@pack@sibling@uniform@recursive{%
  \forest@node@foreachchild{\forest@pack@sibling@uniform@recursive}%
  \forest@pack@sibling@uniform@applyreversed
}
```

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

```
\def\forest@pack@sibling@uniform@applyreversed{%
  \ifnum\forestove{n children}>1
    \ifnum\forestove{reversed}=0
      \pack@sibling@uniform@main{first}{last}{next}{previous}%
    \else
      \pack@sibling@uniform@main{last}{first}{previous}{next}%
    \fi
  \fi
}
```

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

```
\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.

```
  \forestoget{@#1}\forest@child
  \edef\forest@temp{%
    \noexpand\forest@fornode{\forestove{@#1}}{%
      \noexpand\forest@node@getedge
        {positive}
        {\forestove{grow}}
        \noexpand\forest@temp@edge
    }%
  }\forest@temp
  \forest@pack@pgfpoint@childsposition\forest@child
  \let\forest@previous@positive@edge\pgfutil@empty
  \forest@extendpath\forest@previous@positive@edge\forest@temp@edge{}%
  \forestOget{\forest@child}{@#3}\forest@child
```

Loop until the current child is the null node.

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

Get the negative edge of the child.

```
    \edef\forest@temp{%
      \noexpand\forest@fornode{\forest@child}{%
        \noexpand\forest@node@getedge
          {negative}
          {\forestove{grow}}
          \noexpand\forest@temp@edge
        }%
    }\forest@temp
```

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

```
    \forest@pack@pgfpoint@childsposition\forest@child
```

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

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

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

```
    \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!)

```
    \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`.

```
    \ifx\forest@csdistance\relax
      %\forestOeset{\forest@child}{s}{\forest@previous@child@s}%
    \else
      \advance\pgfutil@tempdimb-\forest@csdistance\relax
      \advance\pgfutil@tempdimb\forestove{s sep}\relax
      \forestOeset{\forest@child}{s}{\the\dimexpr\forestove{s}-\forest@csdistance+\forestove{s sep}}%
    \fi
```

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

```
    \ifdim\forestOve{\forest@child}{s}<\forest@previous@child@s\relax
      \forestOeset{\forest@child}{s}{\forest@previous@child@s}%
    \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.

```
    \forestOget{\forest@child}{s}\forest@child@s
    \edef\forest@previous@child@s{\forest@child@s}%
    \edef\forest@temp{%
      \noexpand\forest@fornode{\forest@child}{%
        \noexpand\forest@node@getedge
          {positive}
          {\forestove{grow}}
          \noexpand\forest@temp@edge
        }%
    }\forest@temp
    \forest@pack@pgfpoint@childsposition\forest@child
    \forest@extendpath\forest@previous@positive@edge\forest@temp@edge{}%
    \forest@getpositivetightedgeofpath\forest@previous@positive@edge\forest@previous@positive@edge
    \forestOget{\forest@child}{@#3}\forest@child
  \forest@repeatb
```

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

```
  \csname forest@calign@\forestove{calign}\endcsname
}
```

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

```
\def\forest@pack@pgfpoint@childsposition#1{%
  {%
    \pgftransformreset
    \pgftransformrotate{\forestove{grow}}%
    \forest@fornode{#1}{%
      \pgfpointtransformed{\pgfqpoint{\forestove{l}}{\forestove{s}}}%
    }%
  }%
}
```

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

```
\def\forest@pack@pgfpoint@positioningrow#1{%
  {%
    \pgftransformreset
    \pgftransformrotate{#1}%
    \pgfpointtransformed{\pgfqpoint{\forestove{l}}{\forestove{s}}}%
  }%
}
```

Child alignment.

```
\def\forest@calign@s@shift#1{%
  \pgfutil@tempdima=#1\relax
  \forest@node@foreachchild{%
    \forestoeset{s}{\the\dimexpr\forestove{s}+\pgfutil@tempdima}%
  }%
}
\def\forest@calign@child{%
  \forest@calign@s@shift{-\forestOve{\forest@node@nornbarthchildid{\forestove{calign primary child}}}{s}}%
}
\csdef{forest@calign@child edge}{%
  {%
    \edef\forest@temp@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
    \pgftransformreset
    \pgftransformrotate{\forestove{grow}}%
    \pgfpointtransformed{\pgfqpoint{\forestOve{\forest@temp@child}{l}}{\forestOve{\forest@temp@child}{s}}}%
    \pgf@xa=\pgf@x\relax\pgf@ya=\pgf@y\relax
    \forestOve{\forest@temp@child}{child@anchor}%
    \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
    \forestove{parent@anchor}%
    \advance\pgf@xa-\pgf@x\relax\advance\pgf@ya-\pgf@y\relax
    \edef\forest@marshal{%
      \noexpand\pgftransformreset
      \noexpand\pgftransformrotate{-\forestove{grow}}%
      \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
    }\forest@marshal
  }%
  \forest@calign@s@shift{\the\dimexpr-\the\pgf@y}%
}
\csdef{forest@calign@midpoint}{%
  \forest@calign@s@shift{\the\dimexpr 0pt -%
    (\forestOve{\forest@node@nornbarthchildid{\forestove{calign primary child}}}{s}%
    +\forestOve{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}{s}%
    )/2\relax
  }%
}
\csdef{forest@calign@edge midpoint}{%
  {%
    \edef\forest@temp@firstchild{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
    \edef\forest@temp@secondchild{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
    \pgftransformreset
    \pgftransformrotate{\forestove{grow}}%
    \pgfpointtransformed{\pgfqpoint{\forestOve{\forest@temp@firstchild}{l}}{\forestOve{\forest@temp@firstchil
    \pgf@xa=\pgf@x\relax\pgf@ya=\pgf@y\relax
    \forestOve{\forest@temp@firstchild}{child@anchor}%
    \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
    \edef\forest@marshal{%
      \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\forestOve{\forest@temp@secondchild}{l}}{\forestOve{\
    }\forest@marshal
    \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
    \forestOve{\forest@temp@secondchild}{child@anchor}%
    \advance\pgf@xa\pgf@x\relax\advance\pgf@ya\pgf@y\relax
    \divide\pgf@xa2 \divide\pgf@ya2
    \edef\forest@marshal{%
      \noexpand\pgftransformreset
      \noexpand\pgftransformrotate{-\forestove{grow}}%
      \noexpand\pgfpointtransformed{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
    }\forest@marshal
  }%
  \forest@calign@s@shift{\the\dimexpr-\the\pgf@y}%
}
```

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.

```
\csdef{forest@calign@fixed angles}{%
  \edef\forest@ca@first@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
  \edef\forest@ca@second@child{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
  \ifnum\forestove{reversed}=1
    \let\forest@temp\forest@ca@first@child
    \let\forest@ca@first@child\forest@ca@second@child
    \let\forest@ca@second@child\forest@temp
  \fi
  \forestOget{\forest@ca@first@child}{l}\forest@ca@first@l
  \forestOget{\forest@ca@second@child}{l}\forest@ca@second@l
  \pgfmathsetlengthmacro\forest@ca@desired@s@distance{%
    tan(\forestove{calign secondary angle})*\forest@ca@second@l
    -tan(\forestove{calign primary angle})*\forest@ca@first@l
  }%
  \forestOget{\forest@ca@first@child}{s}\forest@ca@first@s
  \forestOget{\forest@ca@second@child}{s}\forest@ca@second@s
  \pgfmathsetlengthmacro\forest@ca@actual@s@distance{%
    \forest@ca@second@s-\forest@ca@first@s}%
  \ifdim\forest@ca@desired@s@distance>\forest@ca@actual@s@distance\relax
    \ifdim\forest@ca@actual@s@distance=0pt
      \pgfmathsetlength\pgfutil@tempdima{tan(\forestove{calign primary angle})*\forest@ca@second@l}%
      \pgfmathsetlength\pgfutil@tempdimb{\forest@ca@desired@s@distance/(\forestove{n children}-1)}%
      \forest@node@foreachchild{%
        \forestoeset{s}{\the\pgfutil@tempdima}%
        \advance\pgfutil@tempdima\pgfutil@tempdimb
      }%
      \def\forest@calign@anchor{0pt}%
    \else
      \pgfmathsetmacro\forest@ca@ratio{%
        \forest@ca@desired@s@distance/\forest@ca@actual@s@distance}%
      \forest@node@foreachchild{%
        \pgfmathsetlengthmacro\forest@temp{\forest@ca@ratio*\forestove{s}}%
        \forestolet{s}\forest@temp
      }%
      \pgfmathsetlengthmacro\forest@calign@anchor{%
        -tan(\forestove{calign primary angle})*\forest@ca@first@l}%
    \fi
  \else
    \ifdim\forest@ca@desired@s@distance<\forest@ca@actual@s@distance\relax
      \pgfmathsetlengthmacro\forest@ca@ratio{%
        \forest@ca@actual@s@distance/\forest@ca@desired@s@distance}%
      \forest@node@foreachchild{%
        \pgfmathsetlengthmacro\forest@temp{\forest@ca@ratio*\forestove{l}}%
        \forestolet{l}\forest@temp
      }%
      \forestOget{\forest@ca@first@child}{l}\forest@ca@first@l
      \pgfmathsetlengthmacro\forest@calign@anchor{%
        -tan(\forestove{calign primary angle})*\forest@ca@first@l}%
    \fi
  \fi
  \forest@calign@s@shift{-\forest@calign@anchor}%
}
\csdef{forest@calign@fixed edge angles}{%
  \edef\forest@ca@first@child{\forest@node@nornbarthchildid{\forestove{calign primary child}}}%
  \edef\forest@ca@second@child{\forest@node@nornbarthchildid{\forestove{calign secondary child}}}%
```

```
\ifnum\forestove{reversed}=1
  \let\forest@temp\forest@ca@first@child
  \let\forest@ca@first@child\forest@ca@second@child
  \let\forest@ca@second@child\forest@temp
\fi
\forestOget{\forest@ca@first@child}{l}\forest@ca@first@l
\forestOget{\forest@ca@second@child}{l}\forest@ca@second@l
\forestoget{parent@anchor}\forest@ca@parent@anchor
\forest@ca@parent@anchor
\edef\forest@ca@parent@anchor@s{\the\pgf@x}%
\edef\forest@ca@parent@anchor@l{\the\pgf@y}%
\forestOget{\forest@ca@first@child}{child@anchor}\forest@ca@first@child@anchor
\forest@ca@first@child@anchor
\edef\forest@ca@first@child@anchor@s{\the\pgf@x}%
\edef\forest@ca@first@child@anchor@l{\the\pgf@y}%
\forestOget{\forest@ca@second@child}{child@anchor}\forest@ca@second@child@anchor
\forest@ca@second@child@anchor
\edef\forest@ca@second@child@anchor@s{\the\pgf@x}%
\edef\forest@ca@second@child@anchor@l{\the\pgf@y}%
\pgfmathsetlengthmacro\forest@ca@desired@second@edge@s{tan(\forestove{calign secondary angle})*%
  (\forest@ca@second@l-\forest@ca@second@child@anchor@l+\forest@ca@parent@anchor@l)}%
\pgfmathsetlengthmacro\forest@ca@desired@first@edge@s{tan(\forestove{calign primary angle})*%
  (\forest@ca@first@l-\forest@ca@first@child@anchor@l+\forest@ca@parent@anchor@l)}
\pgfmathsetlengthmacro\forest@ca@desired@s@distance{\forest@ca@desired@second@edge@s-\forest@ca@desired@fir
\forestOget{\forest@ca@first@child}{s}\forest@ca@first@s
\forestOget{\forest@ca@second@child}{s}\forest@ca@second@s
\pgfmathsetlengthmacro\forest@ca@actual@s@distance{%
  \forest@ca@second@s+\forest@ca@second@child@anchor@s
  -\forest@ca@first@s-\forest@ca@first@child@anchor@s}%
\ifdim\forest@ca@desired@s@distance>\forest@ca@actual@s@distance\relax
  \ifdim\forest@ca@actual@s@distance=0pt
    \forestoget{n children}\forest@temp@n@children
    \forest@node@foreachchild{%
      \forestoget{child@anchor}\forest@temp@child@anchor
      \forest@temp@child@anchor
      \edef\forest@temp@child@anchor@s{\the\pgf@x}%
      \pgfmathsetlengthmacro\forest@temp{%
        \forest@ca@desired@first@edge@s+(\forestove{n}-1)*\forest@ca@desired@s@distance/(\forest@temp@n@chi
      \forestolet{s}\forest@temp
    }%
    \def\forest@calign@anchor{0pt}%
  \else
    \pgfmathsetmacro\forest@ca@ratio{%
      \forest@ca@desired@s@distance/\forest@ca@actual@s@distance}%
    \forest@node@foreachchild{%
      \forestoget{child@anchor}\forest@temp@child@anchor
      \forest@temp@child@anchor
      \edef\forest@temp@child@anchor@s{\the\pgf@x}%
      \pgfmathsetlengthmacro\forest@temp{%
        \forest@ca@ratio*(%
          \forestove{s}-\forest@ca@first@s
          +\forest@temp@child@anchor@s-\forest@ca@first@child@anchor@s)%
        +\forest@ca@first@s
        +\forest@ca@first@child@anchor@s-\forest@temp@child@anchor@s}%
      \forestolet{s}\forest@temp
    }%
    \pgfmathsetlengthmacro\forest@calign@anchor{%
      -tan(\forestove{calign primary angle})*(\forest@ca@first@l-\forest@ca@first@child@anchor@l+\forest@ca
      +\forest@ca@first@child@anchor@s-\forest@ca@parent@anchor@s
    }%
  \fi
  \else
    \ifdim\forest@ca@desired@s@distance<\forest@ca@actual@s@distance\relax
      \pgfmathsetlengthmacro\forest@ca@ratio{%
        \forest@ca@actual@s@distance/\forest@ca@desired@s@distance}%
      \forest@node@foreachchild{%
        \forestoget{child@anchor}\forest@temp@child@anchor
        \forest@temp@child@anchor
        \edef\forest@temp@child@anchor@l{\the\pgf@y}%
        \pgfmathsetlengthmacro\forest@temp{%
          \forest@ca@ratio*(%
            \forestove{l}+\forest@ca@parent@anchor@l-\forest@temp@child@anchor@l)
            -\forest@ca@parent@anchor@l+\forest@temp@child@anchor@l}%
        \forestolet{l}\forest@temp
      }%
      \forestOget{\forest@ca@first@child}{l}\forest@ca@first@l
      \pgfmathsetlengthmacro\forest@calign@anchor{%
        -tan(\forestove{calign primary angle})*(\forest@ca@first@l+\forest@ca@parent@anchor@l-\forest@temp@ch
        +\forest@ca@first@child@anchor@s-\forest@ca@parent@anchor@s
      }%
    \fi
  \fi
  \forest@calign@s@shift{-\forest@calign@anchor}%
}
```

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.

```
\def\forest@node@getedge#1#2#3{%
  \forestoget{#1@edge@#2}#3%
  \ifx#3\relax
    \forest@node@foreachchild{%
      \forest@node@getedge{#1}{#2}{\forest@temp@edge}%
    }%
    \forest@forthis{\forest@node@getedges{#2}}%
    \forestoget{#1@edge@#2}#3%
  \fi
}
```

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.

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

Run the computation in a TEX group.

```
  %{%
```

Setup the grow line.

```
    \forest@setupgrowline{#1}%
```

Get the edge of the node itself.

```
    \ifnum\forestove{ignore}=0
      \forestoget{boundary}\forest@node@boundary
    \else
      \def\forest@node@boundary{}%
    \fi
    \csname forest@getboth\forestove{fit}edgesofpath\endcsname
        \forest@node@boundary\forest@negative@node@edge\forest@positive@node@edge
    \forestolet{negative@edge@#1}\forest@negative@node@edge
    \forestolet{positive@edge@#1}\forest@positive@node@edge
```

Add the edges of the children.

```
    \get@edges@merge{negative}{#1}%
    \get@edges@merge{positive}{#1}%
  %}%
```

```
}
```

Merge the #1 (=negative or positive) edge of the node with #1 edges of the children. #2 = grow angle.

```
\def\get@edges@merge#1#2{%
  \ifnum\forestove{n children}>0
    \forestoget{#1@edge@#2}\forest@node@edge
```

Remember the node's parent anchor and add it to the path (for breaking).

```
    \forestove{parent@anchor}%
    \edef\forest@getedge@pa@l{\the\pgf@x}%
    \edef\forest@getedge@pa@s{\the\pgf@y}%
    \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).

```
    \pgftransformreset
    \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.

```
    \def\forest@all@edges{}%
    \forest@node@foreachchild{%
      \forestoget{#1@edge@#2}\forest@temp@edge
      \pgfpointtransformed{\pgfqpoint{\forestove{l}}{\forestove{s}}}%
      \forest@extendpath\forest@node@edge\forest@temp@edge{}%
      \ifnum\forestove{ignore edge}=0
        \pgfpointadd
          {\pgfpointtransformed{\pgfqpoint{\forestove{l}}{\forestove{s}}}}%
          {\forestove{child@anchor}}%
        \pgfgetlastxy{\forest@getedge@ca@l}{\forest@getedge@ca@s}%
        \eappto\forest@all@edges{%
          \noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@pa@l}{\forest@getedge@pa@s}%
          \noexpand\pgfsyssoftpath@linetotoken{\forest@getedge@ca@l}{\forest@getedge@ca@s}%
        }%
        % this deals with potential overlap of the edges:
        \eappto\forest@node@edge{\noexpand\pgfsyssoftpath@movetotoken{\forest@getedge@ca@l}{\forest@getedge@c
      \fi
    }%
    \ifdefempty{\forest@all@edges}{}{%
      \pgfintersectionofpaths{\pgfsetpath\forest@all@edges}{\pgfsetpath\forest@node@edge}%
      \def\forest@edgenode@intersections{}%
      \forest@merge@intersectionloop
      \eappto\forest@node@edge{\expandonce{\forest@all@edges}\expandonce{\forest@edgenode@intersections}}%
    }%
```

Process the path into an edge and store the edge.

```
    \csname forest@get#1\forestove{fit}edgeofpath\endcsname\forest@node@edge\forest@node@edge
    \forestolet{#1@edge@#2}\forest@node@edge
  \fi
}
\newloop\forest@merge@loop
\def\forest@merge@intersectionloop{%
  \c@pgf@counta=0
  \forest@merge@loop
  \ifnum\c@pgf@counta<\pgfintersectionsolutions\relax
    \advance\c@pgf@counta1
    \pgfpointintersectionsolution{\the\c@pgf@counta}%
    \eappto\forest@edgenode@intersections{\noexpand\pgfsyssoftpath@movetotoken
      {\the\pgf@x}{\the\pgf@y}}%
  \forest@merge@repeat
}
```

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

```
\def\forest@node@getboundingrectangle@ls#1{%
  \forestoget{boundary}\forest@node@boundary
  \forest@path@getboundingrectangle@ls\forest@node@boundary{#1}%
}
```

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).

```
\def\forest@pgfpathtransformed#1{%
  \forest@save@pgfsyssoftpath@tokendefs
  \let\pgfsyssoftpath@movetotoken\forest@pgfpathtransformed@moveto
  \let\pgfsyssoftpath@linetotoken\forest@pgfpathtransformed@lineto
  \pgfsyssoftpath@setcurrentpath\pgfutil@empty
  #1%
  \forest@restore@pgfsyssoftpath@tokendefs
}
\def\forest@pgfpathtransformed@moveto#1#2{%
  \forest@pgfpathtransformed@op\pgfsyssoftpath@moveto{#1}{#2}%
}
\def\forest@pgfpathtransformed@lineto#1#2{%
  \forest@pgfpathtransformed@op\pgfsyssoftpath@lineto{#1}{#2}%
}
\def\forest@pgfpathtransformed@op#1#2#3{%
  \pgfpointtransformed{\pgfqpoint{#2}{#3}}%
  \edef\forest@temp{%
    \noexpand#1{\the\pgf@x}{\the\pgf@y}%
  }%
  \forest@temp
}
```

11.2.1 Tiers

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

```
\def\forest@pack@computetiers{%
  {%
    \forest@pack@tiers@getalltiersinsubtree
    \forest@pack@tiers@computetierhierarchy
    \forest@pack@tiers@findcontainers
    \forest@pack@tiers@raisecontainers
    \forest@pack@tiers@computeprocessingorder
    \gdef\forest@smuggle{}%
    \forest@pack@tiers@write
  }%
  \forest@node@foreach{\forestoset{@tiers}{}}%
  \forest@smuggle
}
```

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

```
\def\forest@pack@tiers@getalltiersinsubtree{%
  \ifnum\forestove{n children}>0
    \forest@node@foreachchild{\forest@pack@tiers@getalltiersinsubtree}%
  \fi
  \forestoget{tier}\forest@temp@mytier
  \def\forest@temp@mytiers{}%
  \ifdefempty\forest@temp@mytier{}{%
    \listeadd\forest@temp@mytiers\forest@temp@mytier
  }%
  \ifnum\forestove{n children}>0
    \forest@node@foreachchild{%
```

```
      \forestoget{tiers}\forest@temp@tiers
      \forlistloop\forest@pack@tiers@forhandlerA\forest@temp@tiers
    }%
  \fi
  \forestolet{tiers}\forest@temp@mytiers
}
\def\forest@pack@tiers@forhandlerA#1{%
  \ifinlist{#1}\forest@temp@mytiers{}{%
    \listeadd\forest@temp@mytiers{#1}%
  }%
}
```

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.

```
\def\forest@pack@tiers@computetierhierarchy{%
  \def\forest@tiers@ancestors{}%
  \forestoget{tiers}\forest@temp@mytiers
  \forlistloop\forest@pack@tiers@cth@init\forest@temp@mytiers
  \forest@pack@tiers@computetierhierarchy@
}
\def\forest@pack@tiers@cth@init#1{%
  \csdef{forest@tiers@higher@#1}{}%
  \csdef{forest@tiers@lower@#1}{}%
}
\def\forest@pack@tiers@computetierhierarchy@{%
  \forestoget{tier}\forest@temp@mytier
  \ifdefempty\forest@temp@mytier{}{%
    \forlistloop\forest@pack@tiers@forhandlerB\forest@tiers@ancestors
    \listeadd\forest@tiers@ancestors\forest@temp@mytier
  }%
  \forest@node@foreachchild{%
    \forest@pack@tiers@computetierhierarchy@
  }%
  \forestoget{tier}\forest@temp@mytier
  \ifdefempty\forest@temp@mytier{}{%
    \forest@listedel\forest@tiers@ancestors\forest@temp@mytier
  }%
}
\def\forest@pack@tiers@forhandlerB#1{%
  \def\forest@temp@tier{#1}%
  \ifx\forest@temp@tier\forest@temp@mytier
    \PackageError{forest}{Circular tier hierarchy (tier \forest@temp@mytier)}{}%
  \fi
  \ifinlistcs{#1}{forest@tiers@higher@\forest@temp@mytier}{}{%
    \listcsadd{forest@tiers@higher@\forest@temp@mytier}{#1}}%
  \xifinlistcs\forest@temp@mytier{forest@tiers@lower@#1}{}{%
    \listcseadd{forest@tiers@lower@#1}{\forest@temp@mytier}}%
}
\def\forest@pack@tiers@findcontainers{%
  \forestoget{tiers}\forest@temp@tiers
  \forlistloop\forest@pack@tiers@findcontainer\forest@temp@tiers
}
\def\forest@pack@tiers@findcontainer#1{%
  \def\forest@temp@tier{#1}%
  \forestoget{tier}\forest@temp@mytier
  \ifx\forest@temp@tier\forest@temp@mytier
    \csedef{forest@tiers@container@#1}{\forest@cn}%
  \else\@escapeif{%
    \forest@pack@tiers@findcontainerA{#1}%
  }\fi%
}
```

```
\def\forest@pack@tiers@findcontainerA#1{%
  \c@pgf@counta=0
  \forest@node@foreachchild{%
    \forestoget{tiers}\forest@temp@tiers
    \ifinlist{#1}\forest@temp@tiers{%
      \advance\c@pgf@counta 1
      \let\forest@temp@child\forest@cn
    }{}%
  }%
  \ifnum\c@pgf@counta>1
    \csedef{forest@tiers@container@#1}{\forest@cn}%
  \else\@escapeif{% surely =1
    \forest@fornode{\forest@temp@child}{%
      \forest@pack@tiers@findcontainer{#1}%
    }%
  }\fi
}
\def\forest@pack@tiers@raisecontainers{%
  \forestoget{tiers}\forest@temp@mytiers
  \forlistloop\forest@pack@tiers@rc@forhandlerA\forest@temp@mytiers
}
\def\forest@pack@tiers@rc@forhandlerA#1{%
  \edef\forest@tiers@temptier{#1}%
  \letcs\forest@tiers@containernodeoftier{forest@tiers@container@#1}%
  \letcs\forest@temp@lowertiers{forest@tiers@lower@#1}%
  \forlistloop\forest@pack@tiers@rc@forhandlerB\forest@temp@lowertiers
}
\def\forest@pack@tiers@rc@forhandlerB#1{%
  \letcs\forest@tiers@containernodeoflowertier{forest@tiers@container@#1}%
  \forestOget{\forest@tiers@containernodeoflowertier}{content}\lowercontent
  \forestOget{\forest@tiers@containernodeoftier}{content}\uppercontent
  \forest@fornode{\forest@tiers@containernodeoflowertier}{%
    \forest@ifancestorof
      {\forest@tiers@containernodeoftier}
      {\csletcs{forest@tiers@container@\forest@tiers@temptier}{forest@tiers@container@#1}}%
      {}%
  }%
}
\def\forest@pack@tiers@computeprocessingorder{%
  \def\forest@tiers@processingorder{}%
  \forestoget{tiers}\forest@tiers@cpo@tierstodo
  \forest@loopa
    \ifdefempty\forest@tiers@cpo@tierstodo{\forest@tempfalse}{\forest@temptrue}%
  \ifforest@temp
    \def\forest@tiers@cpo@tiersremaining{}%
    \def\forest@tiers@cpo@tiersindependent{}%
    \forlistloop\forest@pack@tiers@cpo@forhandlerA\forest@tiers@cpo@tierstodo
    \ifdefempty\forest@tiers@cpo@tiersindependent{%
      \PackageError{forest}{Circular tiers!}{}}{}%
    \forlistloop\forest@pack@tiers@cpo@forhandlerB\forest@tiers@cpo@tiersremaining
    \let\forest@tiers@cpo@tierstodo\forest@tiers@cpo@tiersremaining
  \forest@repeata
}
\def\forest@pack@tiers@cpo@forhandlerA#1{%
  \ifcsempty{forest@tiers@higher@#1}{%
    \listadd\forest@tiers@cpo@tiersindependent{#1}%
    \listadd\forest@tiers@processingorder{#1}%
  }{%
    \listadd\forest@tiers@cpo@tiersremaining{#1}%
  }%
}
```

```
\def\forest@pack@tiers@cpo@forhandlerB#1{%
  \def\forest@pack@tiers@cpo@aremainingtier{#1}%
  \forlistloop\forest@pack@tiers@cpo@forhandlerC\forest@tiers@cpo@tiersindependent
}
\def\forest@pack@tiers@cpo@forhandlerC#1{%
  \ifinlistcs{#1}{forest@tiers@higher@\forest@pack@tiers@cpo@aremainingtier}{%
    \forest@listcsdel{forest@tiers@higher@\forest@pack@tiers@cpo@aremainingtier}{#1}%
  }{}%
}
\def\forest@pack@tiers@write{%
  \forlistloop\forest@pack@tiers@write@forhandler\forest@tiers@processingorder
}
\def\forest@pack@tiers@write@forhandler#1{%
  \forest@fornode{\csname forest@tiers@container@#1\endcsname}{%
    \forest@pack@tiers@check{#1}%
  }%
  \xappto\forest@smuggle{%
    \noexpand\listadd
      \forestOm{\csname forest@tiers@container@#1\endcsname}{@tiers}%
      {#1}%
  }%
}
 % checks if the tier is compatible with growth changes and calign=node/edge angle
\def\forest@pack@tiers@check#1{%
  \def\forest@temp@currenttier{#1}%
  \forest@node@foreachdescendant{%
    \ifnum\forestove{grow}=\forestOve{\forestove{@parent}}{grow}
    \else
      \forest@pack@tiers@check@grow
    \fi
    \ifnum\forestove{n children}>1
      \forestoget{calign}\forest@temp
      \ifx\forest@temp\forest@pack@tiers@check@nodeangle
        \forest@pack@tiers@check@calign
      \fi
      \ifx\forest@temp\forest@pack@tiers@check@edgeangle
        \forest@pack@tiers@check@calign
      \fi
    \fi
  }%
}
\def\forest@pack@tiers@check@nodeangle{node angle}%
\def\forest@pack@tiers@check@edgeangle{edge angle}%
\def\forest@pack@tiers@check@grow{%
  \forestoget{content}\forest@temp@content
  \let\forest@temp@currentnode\forest@cn
  \forest@node@foreachdescendant{%
    \forestoget{tier}\forest@temp
    \ifx\forest@temp@currenttier\forest@temp
      \forest@pack@tiers@check@grow@error
    \fi
  }%
}
\def\forest@pack@tiers@check@grow@error{%
  \PackageError{forest}{Tree growth direction changes in node \forest@temp@currentnode\space
    (content: \forest@temp@content), while tier '\forest@temp' is specified for nodes both
    out- and inside the subtree rooted in node \forest@temp@currentnode.  This will not work.}{}%
}
\def\forest@pack@tiers@check@calign{%
  \forest@node@foreachchild{%
    \forestoget{tier}\forest@temp
    \ifx\forest@temp@currenttier\forest@temp
      \forest@pack@tiers@check@calign@warning
    \fi
  }%
}
\def\forest@pack@tiers@check@calign@warning{%
  \PackageWarning{forest}{Potential option conflict: node \forestove{@parent} (content:
    '\forestOve{\forestove{@parent}}{content}') was given 'calign=\forestove{calign}', while its
    child \forest@cn\space (content: '\forestove{content}') was given 'tier=\forestove{tier}'.
    The parent's 'calign' will only work if the child was the lowest node on its tier before the
    alignment.}{}
}
```

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.

```
\pgfdeclaregenericanchor{forestcomputenodeboundary}{%
  \letcs\forest@temp@boundary@macro{forest@compute@node@boundary@#1}%
  \ifcsname forest@compute@node@boundary@#1\endcsname
    \csname forest@compute@node@boundary@#1\endcsname
  \else
    \forest@compute@node@boundary@rectangle
  \fi
  \pgfsyssoftpath@getcurrentpath\forest@temp
  \global\let\forest@global@boundary\forest@temp
}
\def\forest@mt#1{%
  \expandafter\pgfpointanchor\expandafter{\pgfreferencednodename}{#1}%
  \pgfsyssoftpath@moveto{\the\pgf@x}{\the\pgf@y}%
}%
\def\forest@lt#1{%
  \expandafter\pgfpointanchor\expandafter{\pgfreferencednodename}{#1}%
  \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
}%
\def\forest@compute@node@boundary@coordinate{%
  \forest@mt{center}%
}
\def\forest@compute@node@boundary@circle{%
  \forest@mt{east}%
  \forest@lt{north east}%
  \forest@lt{north}%
  \forest@lt{north west}%
  \forest@lt{west}%
  \forest@lt{south west}%
  \forest@lt{south}%
  \forest@lt{south east}%
  \forest@lt{east}%
}
\def\forest@compute@node@boundary@rectangle{%
  \forest@mt{south west}%
  \forest@lt{south east}%
  \forest@lt{north east}%
  \forest@lt{north west}%
  \forest@lt{south west}%
}
\def\forest@compute@node@boundary@diamond{%
  \forest@mt{east}%
  \forest@lt{north}%
  \forest@lt{west}%
  \forest@lt{south}%
  \forest@lt{east}%
}
\let\forest@compute@node@boundary@ellipse\forest@compute@node@boundary@circle
\def\forest@compute@node@boundary@trapezium{%
  \forest@mt{top right corner}%
  \forest@lt{top left corner}%
  \forest@lt{bottom left corner}%
  \forest@lt{bottom right corner}%
  \forest@lt{top right corner}%
}
\def\forest@compute@node@boundary@semicircle{%
  \forest@mt{arc start}%
  \forest@lt{north}%
  \forest@lt{east}%
  \forest@lt{north east}%
  \forest@lt{apex}%
  \forest@lt{north west}%
  \forest@lt{west}%
  \forest@lt{arc end}%
  \forest@lt{arc start}%
}
\newloop\forest@computenodeboundary@loop
\csdef{forest@compute@node@boundary@regular polygon}{%
  \forest@mt{corner 1}%
  \c@pgf@counta=\sides\relax
  \forest@computenodeboundary@loop
  \ifnum\c@pgf@counta>0
    \forest@lt{corner \the\c@pgf@counta}%
    \advance\c@pgf@counta-1
  \forest@computenodeboundary@repeat
}%
\def\forest@compute@node@boundary@star{%
  \forest@mt{outer point 1}%
  \c@pgf@counta=\totalstarpoints\relax
  \divide\c@pgf@counta2
  \forest@computenodeboundary@loop
  \ifnum\c@pgf@counta>0
    \forest@lt{inner point \the\c@pgf@counta}%
    \forest@lt{outer point \the\c@pgf@counta}%
    \advance\c@pgf@counta-1
  \forest@computenodeboundary@repeat
}%
\csdef{forest@compute@node@boundary@isosceles triangle}{%
  \forest@mt{apex}%
  \forest@lt{left corner}%
  \forest@lt{right corner}%
  \forest@lt{apex}%
}
\def\forest@compute@node@boundary@kite{%
  \forest@mt{upper vertex}%
  \forest@lt{left vertex}%
  \forest@lt{lower vertex}%
  \forest@lt{right vertex}%
  \forest@lt{upper vertex}%
}
\def\forest@compute@node@boundary@dart{%
  \forest@mt{tip}%
  \forest@lt{left tail}%
  \forest@lt{tail center}%
  \forest@lt{right tail}%
  \forest@lt{tip}%
}
\csdef{forest@compute@node@boundary@circular sector}{%
  \forest@mt{sector center}%
  \forest@lt{arc start}%
  \forest@lt{arc center}%
  \forest@lt{arc end}%
  \forest@lt{sector center}%
}
\def\forest@compute@node@boundary@cylinder{%
  \forest@mt{top}%
  \forest@lt{after top}%
  \forest@lt{before bottom}%
  \forest@lt{bottom}%
  \forest@lt{after bottom}%
  \forest@lt{before top}%
  \forest@lt{top}%
}
\cslet{forest@compute@node@boundary@forbidden sign}\forest@compute@node@boundary@circle
\cslet{forest@compute@node@boundary@magnifying glass}\forest@compute@node@boundary@circle
\def\forest@compute@node@boundary@cloud{%
  \getradii
  \forest@mt{puff 1}%
  \c@pgf@counta=\puffs\relax
  \forest@computenodeboundary@loop
  \ifnum\c@pgf@counta>0
    \forest@lt{puff \the\c@pgf@counta}%
    \advance\c@pgf@counta-1
  \forest@computenodeboundary@repeat
}
\def\forest@compute@node@boundary@starburst{
  \calculatestarburstpoints
  \forest@mt{outer point 1}%
  \c@pgf@counta=\totalpoints\relax
  \divide\c@pgf@counta2
  \forest@computenodeboundary@loop
  \ifnum\c@pgf@counta>0
    \forest@lt{inner point \the\c@pgf@counta}%
    \forest@lt{outer point \the\c@pgf@counta}%
    \advance\c@pgf@counta-1
  \forest@computenodeboundary@repeat
}%
\def\forest@compute@node@boundary@signal{%
  \forest@mt{east}%
  \forest@lt{south east}%
  \forest@lt{south west}%
  \forest@lt{west}%
  \forest@lt{north west}%
  \forest@lt{north east}%
  \forest@lt{east}%
}
\def\forest@compute@node@boundary@tape{%
  \forest@mt{north east}%
  \forest@lt{60}%
  \forest@lt{north}%
  \forest@lt{120}%
  \forest@lt{north west}%
  \forest@lt{south west}%
  \forest@lt{240}%
  \forest@lt{south}%
  \forest@lt{310}%
  \forest@lt{south east}%
  \forest@lt{north east}%
}
\csdef{forest@compute@node@boundary@single arrow}{%
  \forest@mt{tip}%
  \forest@lt{after tip}%
  \forest@lt{after head}%
  \forest@lt{before tail}%
  \forest@lt{after tail}%
  \forest@lt{before head}%
  \forest@lt{before tip}%
  \forest@lt{tip}%
}
\csdef{forest@compute@node@boundary@double arrow}{%
  \forest@mt{tip 1}%
  \forest@lt{after tip 1}%
  \forest@lt{after head 1}%
  \forest@lt{before head 2}%
  \forest@lt{before tip 2}%
  \forest@mt{tip 2}%
  \forest@lt{after tip 2}%
  \forest@lt{after head 2}%
  \forest@lt{before head 1}%
  \forest@lt{before tip 1}%
  \forest@lt{tip 1}%
}
\csdef{forest@compute@node@boundary@arrow box}{%
  \forest@mt{before north arrow}%
  \forest@lt{before north arrow head}%
  \forest@lt{before north arrow tip}%
  \forest@lt{north arrow tip}%
  \forest@lt{after north arrow tip}%
  \forest@lt{after north arrow head}%
  \forest@lt{after north arrow}%
  \forest@lt{north east}%
  \forest@lt{before east arrow}%
  \forest@lt{before east arrow head}%
  \forest@lt{before east arrow tip}%
  \forest@lt{east arrow tip}%
  \forest@lt{after east arrow tip}%
  \forest@lt{after east arrow head}%
  \forest@lt{after east arrow}%
  \forest@lt{south east}%
  \forest@lt{before south arrow}%
  \forest@lt{before south arrow head}%
  \forest@lt{before south arrow tip}%
  \forest@lt{south arrow tip}%
  \forest@lt{after south arrow tip}%
  \forest@lt{after south arrow head}%
  \forest@lt{after south arrow}%
  \forest@lt{south west}%
  \forest@lt{before west arrow}%
  \forest@lt{before west arrow head}%
  \forest@lt{before west arrow tip}%
  \forest@lt{west arrow tip}%
  \forest@lt{after west arrow tip}%
  \forest@lt{after west arrow head}%
  \forest@lt{after west arrow}%
  \forest@lt{north west}%
  \forest@lt{before north arrow}%
}
```

```
\cslet{forest@compute@node@boundary@circle split}\forest@compute@node@boundary@circle
\cslet{forest@compute@node@boundary@circle solidus}\forest@compute@node@boundary@circle
\cslet{forest@compute@node@boundary@ellipse split}\forest@compute@node@boundary@ellipse
\cslet{forest@compute@node@boundary@rectangle split}\forest@compute@node@boundary@rectangle
\def\forest@compute@node@boundary@@callout{%
  \beforecalloutpointer
  \pgfsyssoftpath@moveto{\the\pgf@x}{\the\pgf@y}%
  \calloutpointeranchor
  \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
  \aftercalloutpointer
  \pgfsyssoftpath@lineto{\the\pgf@x}{\the\pgf@y}%
}
\csdef{forest@compute@node@boundary@rectangle callout}{%
  \forest@compute@node@boundary@rectangle
  \rectanglecalloutpoints
  \forest@compute@node@boundary@@callout
}
\csdef{forest@compute@node@boundary@ellipse callout}{%
  \forest@compute@node@boundary@ellipse
  \ellipsecalloutpoints
  \forest@compute@node@boundary@@callout
}
\csdef{forest@compute@node@boundary@cloud callout}{%
  \forest@compute@node@boundary@cloud
  % at least a first approx...
  \forest@mt{center}%
  \forest@lt{pointer}%
}%
\csdef{forest@compute@node@boundary@cross out}{%
  \forest@mt{south east}%
  \forest@lt{north west}%
  \forest@mt{south west}%
  \forest@lt{north east}%
}%
\csdef{forest@compute@node@boundary@strike out}{%
  \forest@mt{north east}%
  \forest@lt{south west}%
}%
\cslet{forest@compute@node@boundary@rounded rectangle}\forest@compute@node@boundary@rectangle
\csdef{forest@compute@node@boundary@chamfered rectangle}{%
  \forest@mt{before south west}%
  \forest@mt{after south west}%
  \forest@lt{before south east}%
  \forest@lt{after south east}%
  \forest@lt{before north east}%
  \forest@lt{after north east}%
  \forest@lt{before north west}%
  \forest@lt{after north west}%
  \forest@lt{before south west}%
}%
```

11.3 Compute absolute positions

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

```
\def\forest@node@computeabsolutepositions{%
  \forestoset{x}{0pt}%
  \forestoset{y}{0pt}%
  \edef\forest@marshal{%
    \noexpand\forest@node@foreachchild{%
```

```
      \noexpand\forest@node@computeabsolutepositions@{0pt}{0pt}{\forestove{grow}}%
    }%
  }\forest@marshal
}
\def\forest@node@computeabsolutepositions@#1#2#3{%
  \pgfpointadd{\pgfpoint{#1}{#2}}{%
    \pgfpointadd{\pgfpolar{#3}{\forestove{l}}}{\pgfpolar{90 + #3}{\forestove{s}}}}%
  \pgfgetlastxy\forest@temp@x\forest@temp@y
  \forestolet{x}\forest@temp@x
  \forestolet{y}\forest@temp@y
  \edef\forest@marshal{%
    \noexpand\forest@node@foreachchild{%
      \noexpand\forest@node@computeabsolutepositions@{\forest@temp@x}{\forest@temp@y}{\forestove{grow}}%
    }%
  }\forest@marshal
}
```

11.4 Drawing the tree

```
\newif\ifforest@drawtree@preservenodeboxes@
\def\forest@node@drawtree{%
  \expandafter\ifstrequal\expandafter{\forest@drawtreebox}{\pgfkeysnovalue}{%
    \let\forest@drawtree@beginbox\relax
    \let\forest@drawtree@endbox\relax
  }{%
    \edef\forest@drawtree@beginbox{\global\setbox\forest@drawtreebox=\hbox\bgroup}%
    \let\forest@drawtree@endbox\egroup
  }%
  \ifforest@external@
    \ifforest@externalize@tree@
      \forest@temptrue
    \else
      \tikzifexternalizing{%
        \ifforest@was@tikzexternalwasenable
          \forest@temptrue
          \pgfkeys{/tikz/external/optimize=false}%
          \let\forest@drawtree@beginbox\relax
          \let\forest@drawtree@endbox\relax
        \else
          \forest@tempfalse
        \fi
      }{%
        \forest@tempfalse
      }%
    \fi
    \ifforest@temp
      \advance\forest@externalize@inner@n 1
      \edef\forest@externalize@filename{%
        \tikzexternalrealjob-forest-\forest@externalize@outer@n
        \ifnum\forest@externalize@inner@n=0 \else.\the\forest@externalize@inner@n\fi}%
      \expandafter\tikzsetnextfilename\expandafter{\forest@externalize@filename}%
      \tikzexternalenable
      \pgfkeysalso{/tikz/external/remake next,/tikz/external/export next}%
    \fi
    \ifforest@externalize@tree@
      \typeout{forest: Invoking a recursive call to generate the external picture
        ’\forest@externalize@filename’ for the following context+code:
        ’\expandafter\detokenize\expandafter{\forest@externalize@id}’}%
    \fi
  \fi
  %
  \ifforesttikzcshack
    \let\forest@original@tikz@parse@node\tikz@parse@node
    \let\tikz@parse@node\forest@tikz@parse@node
  \fi
  \forest@drawtree@beginbox
  \pgfkeysalso{/forest/begin draw}%
  \forest@node@drawtree@
  \pgfkeysalso{/forest/end draw}%
  \forest@drawtree@endbox
  \ifforesttikzcshack
    \let\tikz@parse@node\forest@original@tikz@parse@node
  \fi
  %
  \ifforest@external@
    \ifforest@externalize@tree@
      \tikzexternaldisable
      \eappto\forest@externalize@checkimages{%
        \noexpand\forest@includeexternal@check{\forest@externalize@filename}%
      }%
      \expandafter\ifstrequal\expandafter{\forest@drawtreebox}{\pgfkeysnovalue}{%
        \eappto\forest@externalize@loadimages{%
          \noexpand\forest@includeexternal{\forest@externalize@filename}%
        }%
      }{%
        \eappto\forest@externalize@loadimages{%
          \noexpand\forest@includeexternal@box\forest@drawtreebox{\forest@externalize@filename}%
        }%
      }%
    \fi
  \fi
}
\def\forest@node@drawtree@{%
  \forest@node@foreach{\forest@draw@node}%
  \forest@node@Ifnamedefined{forest@baseline@node}{%
    \edef\forest@temp{%
      \noexpand\pgfsetbaselinepointlater{%
        \noexpand\pgfpointanchor
          {\forestOve{\forest@node@Nametoid{forest@baseline@node}}{name}}
          {\forestOve{\forest@node@Nametoid{forest@baseline@node}}{anchor}}
      }%
    }\forest@temp
  }{}%
  \forest@node@foreachdescendant{\forest@draw@edge}%
  \forest@node@foreach{\forest@draw@tikz}%
}
\def\forest@draw@node{%
  \ifnum\forestove{phantom}=0
    \forest@node@forest@positionnodelater@restore
    \ifforest@drawtree@preservenodeboxes@
      \pgfnodealias{forest@temp}{\forestove{later@name}}%
    \fi
    \pgfpositionnodenow{\pgfqpoint{\forestove{x}}{\forestove{y}}}%
    \ifforest@drawtree@preservenodeboxes@
      \pgfnodealias{\forestove{later@name}}{forest@temp}%
    \fi
  \fi
}
\def\forest@draw@edge{%
  \ifnum\forestove{phantom}=0
    \ifnum\forestOve{\forestove{@parent}}{phantom}=0
```

```
      \edef\forest@temp{\forestove{edge path}}%
      \forest@temp
    \fi
  \fi
}
\def\forest@draw@tikz{%
  \forestove{tikz}%
}
```

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

```
\def\forest@tikz@parse@node#1(#2){%
  \pgfutil@in@.{#2}%
  \ifpgfutil@in@
    \expandafter\forest@tikz@parse@node@checkiftikzname@withdot
  \else%
    \expandafter\forest@tikz@parse@node@checkiftikzname@withoutdot
  \fi%
  #1(#2)\forest@end
}
\def\forest@tikz@parse@node@checkiftikzname@withdot#1(#2.#3)\forest@end{%
  \forest@tikz@parse@node@checkiftikzname#1{#2}{.#3}}
\def\forest@tikz@parse@node@checkiftikzname@withoutdot#1(#2)\forest@end{%
  \forest@tikz@parse@node@checkiftikzname#1{#2}{}}
\def\forest@tikz@parse@node@checkiftikzname#1#2#3{%
  \expandafter\ifx\csname pgf@sh@ns@#2\endcsname\relax
    \forest@forthis{%
      \forest@nameandgo{#2}%
      \edef\forest@temp@relativenodename{\forestove{name}}%
    }%
  \else
    \def\forest@temp@relativenodename{#2}%
  \fi
  \expandafter\forest@original@tikz@parse@node\expandafter#1\expandafter(\forest@temp@relativenodename#3)%
}
\def\forest@nameandgo#1{%
  \pgfutil@in@!{#1}%
  \ifpgfutil@in@
    \forest@nameandgo@(#1)%
  \else
    \ifstrempty{#1}{}{\edef\forest@cn{\forest@node@Nametoid{#1}}}%
  \fi
}
\def\forest@nameandgo@(#1!#2){%
  \ifstrempty{#1}{}{\edef\forest@cn{\forest@node@Nametoid{#1}}}%
  \forest@go{#2}%
}
```

`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.

```
\pgfdeclaregenericanchor{parent anchor}{%
  \forest@generic@parent@child@anchor{parent }{#1}}
\pgfdeclaregenericanchor{child anchor}{%
  \forest@generic@parent@child@anchor{child }{#1}}
\pgfdeclaregenericanchor{anchor}{%
  \forest@generic@parent@child@anchor{}{#1}}
\def\forest@generic@parent@child@anchor#1#2{%
  \forestOget{\forest@node@Nametoid{\pgfreferencednodename}}{#1anchor}\forest@temp@parent@anchor
  \ifdefempty\forest@temp@parent@anchor{%
    \pgf@sh@reanchor{#2}{center}%
    \xdef\forest@hack@tikzshapeborder{%
      \noexpand\tikz@shapebordertrue
      \def\noexpand\tikz@shapeborder@name{\pgfreferencednodename}%
    }\aftergroup\forest@hack@tikzshapeborder
  }{%
    \pgf@sh@reanchor{#2}{\forest@temp@parent@anchor}%
  }%
  }
```

12 Geometry

A *α grow line* is a line through the origin at angle α. The following macro sets up the grow line, which can then be used by other code (the change is local to the 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° counter-clockwise.

```
\newdimen\forest@xg
\newdimen\forest@yg
\newdimen\forest@xs
\newdimen\forest@ys
\def\forest@setupgrowline#1{%
  \edef\forest@grow{#1}%
  \pgfpointpolar\forest@grow{1pt}%
  \forest@xg=\pgf@x
  \forest@yg=\pgf@y
  \forest@xs=-\pgf@y
  \forest@ys=\pgf@x
}
```

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`.

```
\def\forest@pgfpointprojectiontogrowline#1{{%
  \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.

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

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

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

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

```
\def\forest@distancetogrowline#1#2{%
  \pgf@process{#2}%
  #1=\pgf@sys@tonumber{\pgf@x}\forest@xs\relax
  \advance#1 by\pgf@sys@tonumber{\pgf@y}\forest@ys\relax
}
```

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[20] path `#1` onto the grow line. The result is an array of tuples (`xo`, `yo`, `xp`, `yp`), where `xo` and `yo` stand for the *o*riginal point, and `xp` and `yp` stand for its *p*rojection. 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

[20] A path is *simple* if it consists of only move-to and line-to operations.

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`.

```
\let\forest@pp@n\relax
\def\forest@projectpathtogrowline#1#2{%
  \edef\forest@pp@prefix{#2}%
  \forest@save@pgfsyssoftpath@tokendefs
  \let\pgfsyssoftpath@movetotoken\forest@projectpath@processpoint
  \let\pgfsyssoftpath@linetotoken\forest@projectpath@processpoint
  \c@pgf@counta=0
  #1%
  \csedef{#2n}{\the\c@pgf@counta}%
  \forest@restore@pgfsyssoftpath@tokendefs
}
```

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

```
\def\forest@projectpath@processpoint#1#2{%
  \pgfqpoint{#1}{#2}%
  \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta xo\endcsname{\the\pgf@x}%
  \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta yo\endcsname{\the\pgf@y}%
  \forest@pgfpointprojectiontogrowline{}%
  \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta xp\endcsname{\the\pgf@x}%
  \expandafter\edef\csname\forest@pp@prefix\the\c@pgf@counta yp\endcsname{\the\pgf@y}%
  \advance\c@pgf@counta 1\relax
}
```

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

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

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

- 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`.[21]

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

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

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

[21] 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.

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

```
    \letcs\forest@xo{#1\the\c@pgf@counta xo}%
    \letcs\forest@yo{#1\the\c@pgf@counta yo}%
    \letcs\forest@xp{#1\the\c@pgf@counta xp}%
    \letcs\forest@yp{#1\the\c@pgf@counta yp}%
    \ifnum\c@pgf@countb<0
      \forest@equaltotolerancefalse
    \else
      \forest@equaltotolerance
        {\pgfqpoint\forest@xp\forest@yp}%
        {\pgfqpoint
          {\csname#2\the\c@pgf@countb x\endcsname}%
          {\csname#2\the\c@pgf@countb y\endcsname}%
        }%
    \fi
    \ifforest@equaltotolerance\else
```

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

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

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

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

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

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

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

Clean-up the raw array item.

```
    \cslet{#1\the\c@pgf@counta xo}\relax
    \cslet{#1\the\c@pgf@counta yo}\relax
    \cslet{#1\the\c@pgf@counta xp}\relax
    \cslet{#1\the\c@pgf@counta yp}\relax
    \advance\c@pgf@counta 1
  \repeat
```

Clean up the raw array length.

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

Store the length of the outer result array.

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

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

```
\def\forest@ppiraw@let#1#2{%
  \csletcs{\forest@ppi@inputprefix#1xo}{\forest@ppi@inputprefix#2xo}%
  \csletcs{\forest@ppi@inputprefix#1yo}{\forest@ppi@inputprefix#2yo}%
  \csletcs{\forest@ppi@inputprefix#1xp}{\forest@ppi@inputprefix#2xp}%
  \csletcs{\forest@ppi@inputprefix#1yp}{\forest@ppi@inputprefix#2yp}%
}
```

Item comparision macro for quicksorting the raw projection data.

```
\def\forest@ppiraw@cmp#1#2{%
  \forest@sort@cmptwodimcs
    {\forest@ppi@inputprefix#1xp}{\forest@ppi@inputprefix#1yp}%
    {\forest@ppi@inputprefix#2xp}{\forest@ppi@inputprefix#2yp}%
}
```

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

```
\def\forest@append@point@to@inner@array#1#2#3{%
  \c@pgf@countc=\csname#3n\endcsname\relax
  \csedef{#3\the\c@pgf@countc x}{#1}%
  \csedef{#3\the\c@pgf@countc y}{#2}%
  \forest@distancetogrowline\pgfutil@tempdima{\pgfqpoint#1#2}%
  \csedef{#3\the\c@pgf@countc d}{\the\pgfutil@tempdima}%
  \advance\c@pgf@countc 1
  \csedef{#3n}{\the\c@pgf@countc}%
}
```

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.

```
\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.

```
  \edef\forest@bp@prefix{#2}%
  \forest@save@pgfsyssoftpath@tokendefs
  \let\pgfsyssoftpath@movetotoken\forest@breakpath@processfirstpoint
  \let\pgfsyssoftpath@linetotoken\forest@breakpath@processfirstpoint
  %\pgfusepath{}% empty the current path. ok?
  #1%
  \forest@restore@pgfsyssoftpath@tokendefs
  \pgfsyssoftpath@getcurrentpath#3%
}
```

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

```
\def\forest@breakpath@processfirstpoint#1#2{%
  \forest@breakpath@processmoveto{#1}{#2}%
  \let\pgfsyssoftpath@movetotoken\forest@breakpath@processmoveto
  \let\pgfsyssoftpath@linetotoken\forest@breakpath@processlineto
}
```

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.

```
\def\forest@breakpath@processmoveto#1#2{%
  \pgfsyssoftpath@moveto{#1}{#2}%
  \def\forest@previous@x{#1}%
  \def\forest@previous@y{#2}%
  \expandafter\let\expandafter\forest@previous@i
    \csname\forest@bp@prefix(#1,#2)\endcsname
  \expandafter\let\expandafter\forest@previous@px
    \csname\forest@bp@prefix\forest@previous@i x\endcsname
  \expandafter\let\expandafter\forest@previous@py
    \csname\forest@bp@prefix\forest@previous@i y\endcsname
}
```

This is the heart of the path-breaking procedure.

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

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

```
  \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.)

```
  \expandafter\let\expandafter\forest@i
    \csname\forest@bp@prefix(#1,#2)\endcsname
  \expandafter\let\expandafter\forest@px
    \csname\forest@bp@prefix\forest@i x\endcsname
  \expandafter\let\expandafter\forest@py
    \csname\forest@bp@prefix\forest@i y\endcsname
```

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

```
  \forest@equaltotolerance
    {\pgfqpoint{\forest@previous@px}{\forest@previous@py}}%
    {\pgfqpoint{\forest@px}{\forest@py}}%
  \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.

```
    \let\forest@breakpath@op\pgfsyssoftpath@moveto
  \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.

```
    \forest@temp@count=\forest@previous@i\relax
    \ifnum\forest@previous@i<\forest@i\relax
      \def\forest@breakpath@step{1}%
      \def\forest@breakpath@test{\forest@temp@count<\forest@i\relax}%
    \else
      \def\forest@breakpath@step{-1}%
      \def\forest@breakpath@test{\forest@temp@count>\forest@i\relax}%
    \fi
```

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

```
    \loop
      \advance\forest@temp@count\forest@breakpath@step\relax
    \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.)

```
      \pgfpointintersectionoflines
        {\pgfqpoint
          {\csname\forest@bp@prefix\the\forest@temp@count x\endcsname}%
          {\csname\forest@bp@prefix\the\forest@temp@count y\endcsname}%
        }%
        {\pgfpointadd
          {\pgfqpoint
            {\csname\forest@bp@prefix\the\forest@temp@count x\endcsname}%
            {\csname\forest@bp@prefix\the\forest@temp@count y\endcsname}%
          }%
          {\pgfqpoint{\forest@xs}{\forest@ys}}%
        }%
        {\pgfqpoint{\forest@previous@x}{\forest@previous@y}}%
        {\pgfqpoint{#1}{#2}}%
```

Break the segment at the intersection.

```
      \pgfgetlastxy\forest@last@x\forest@last@y
      \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
```

Append the breaking point to the inner array for the projection.

```
      \forest@append@point@to@inner@array
        \forest@last@x\forest@last@y
        {\forest@bp@prefix\the\forest@temp@count @}%
```

Cache the projection of the new segment edge.

```
      \csedef{\forest@bp@prefix(\the\pgf@x,\the\pgf@y)}{\the\forest@temp@count}%
    \repeat
  \fi
```

Add the current point.

```
  \forest@breakpath@op{#1}{#2}%
```

Setup new "previous" info: the segment edge, its projection's index, and the projection.

```
  \def\forest@previous@x{#1}%
  \def\forest@previous@y{#2}%
  \let\forest@previous@i\forest@i
  \let\forest@previous@px\forest@px
  \let\forest@previous@py\forest@py
}
```

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.[22] Furthermore imagine that the path is opaque. Then the negative/positive tight edge of the path is the part of the path that is illuminated.

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

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

```
\def\forest@getnegativetightedgeofpath#1#2{%
  \forest@get@onetightedgeofpath#1\forest@sort@ascending#2}
\def\forest@getpositivetightedgeofpath#1#2{%
  \forest@get@onetightedgeofpath#1\forest@sort@descending#2}
\def\forest@get@onetightedgeofpath#1#2#3{%
  {%
    \forest@get@one@tightedgeofpath#1#2\forest@gep@edge
    \global\let\forest@gep@global@edge\forest@gep@edge
  }%
  \let#3\forest@gep@global@edge
}
\def\forest@get@one@tightedgeofpath#1#2#3{%
```

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

```
  \forest@projectpathtogrowline#1{forest@pp@}%
  \forest@sortprojections{forest@pp@}%
  \forest@processprojectioninfo{forest@pp@}{forest@pi@}%
```

Break the path.

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

[22] For the definition of negative/positive side, see forest@distancetogrowline in §12.1

Compile some more useful information.

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

The auxiliary data is set up: do the work!

```
  \forest@gettightedgeofpath@getedge
  \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).

```
  \forest@simplifypath\forest@edge#3%
}
```

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

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

Reverse the order of items in the inner arrays.

```
    \c@pgf@counta=0
    \loop
    \ifnum\c@pgf@counta<\forest@pi@n\relax
      \forest@ppi@deflet{forest@pi@\the\c@pgf@counta @}%
      \forest@reversearray\forest@ppi@let
        {0}%
        {\csname forest@pi@\the\c@pgf@counta @n\endcsname}%
      \advance\c@pgf@counta 1
    \repeat
```

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

```
    \forest@gettightedgeofpath@getedge
    \pgfsyssoftpath@getcurrentpath\forest@edge
    \forest@simplifypath\forest@edge\forest@gep@secondedge
```

Smuggle the results out of the enclosing TEX group.

```
      \global\let\forest@gep@global@firstedge\forest@gep@firstedge
      \global\let\forest@gep@global@secondedge\forest@gep@secondedge
    }%
  \let#2\forest@gep@global@firstedge
  \let#3\forest@gep@global@secondedge
}
```

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

```
\def\forest@sort@inner@arrays#1#2{%
  \c@pgf@counta=0
  \forest@loopa
  \ifnum\c@pgf@counta<\csname#1n\endcsname
    \c@pgf@countb=\csname#1\the\c@pgf@counta @n\endcsname\relax
    \ifnum\c@pgf@countb>1
      \advance\c@pgf@countb -1
      \forest@ppi@deflet{#1\the\c@pgf@counta @}%
      \forest@ppi@defcmp{#1\the\c@pgf@counta @}%
      \forest@sort\forest@ppi@cmp\forest@ppi@let#2{0}{\the\c@pgf@countb}%
    \fi
    \advance\c@pgf@counta 1
  \forest@repeata
}
```

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.

```
\def\forest@ppi@deflet#1{%
```

```
  \edef\forest@ppi@let##1##2{%
    \noexpand\csletcs{#1##1x}{#1##2x}%
    \noexpand\csletcs{#1##1y}{#1##2y}%
    \noexpand\csletcs{#1##1d}{#1##2d}%
  }%
}
```

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.

```
\def\forest@ppi@defcmp#1{%
  \edef\forest@ppi@cmp##1##2{%
    \noexpand\forest@sort@cmpdimcs{#1##1d}{#1##2d}%
  }%
}
```

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`).

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

This macro is just a wrapper to process the path.

```
\def\forest@pathtodict#1#2{%
  \edef\forest@pathtodict@prefix{#2}%
  \forest@save@pgfsyssoftpath@tokendefs
  \let\pgfsyssoftpath@movetotoken\forest@pathtodict@movetoop
  \let\pgfsyssoftpath@linetotoken\forest@pathtodict@linetoop
  \def\forest@pathtodict@subpathstart{}%
  #1%
  \forest@restore@pgfsyssoftpath@tokendefs
}
```

When a move-to operation is encountered:

```
\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.

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

When a line-to operation is encountered:

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

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

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

Mark the segment as existing.

```
  \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.

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

In this macro, the edge is actually computed.

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

Clear the path and the last projection.

```
  \pgfsyssoftpath@setcurrentpath\pgfutil@empty
  \let\forest@last@x\relax
  \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.)

```
  \c@pgf@counta=0
  \forest@temp@count=\forest@pi@n\relax
  \advance\forest@temp@count -1
  \edef\forest@nminusone{\the\forest@temp@count}%
  \forest@loopa
  \ifnum\c@pgf@counta<\forest@nminusone\relax
    \forest@gettightedgeofpath@getedge@loopa
  \forest@repeata
```

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

```
  \ifnum\forest@nminusone<\forest@n\relax\else
    \ifnum\csname forest@pi@\forest@nminusone @n\endcsname>0
      \forest@gettightedgeofpath@maybemoveto{\forest@nminusone}{0}%
    \fi
  \fi
}
```

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.

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

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

```
      \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.

```
      \c@pgf@countb=0
      \forest@loopb
      \ifnum\c@pgf@countb<\csname forest@pi@\the\c@pgf@counta @n\endcsname\relax
        \forest@gettightedgeofpath@getedge@loopb
      \forest@repeatb
    \fi
    \advance\c@pgf@counta 1
}
```

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

```
\def\forest@gettightedgeofpath@getedge@loopb{%
        \c@pgf@countc=0
        \advance\c@pgf@counta 1
        \edef\forest@aplusone{\the\c@pgf@counta}%
        \advance\c@pgf@counta -1
        \forest@loopc
        \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!

```
          \forest@tempfalse
          \expandafter\ifx\csname forest@pi@(%
            \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb x\endcsname,%
            \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb y\endcsname)--(%
            \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname,%
            \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname)%
```

```
      \endcsname\forest@inpath
      \forest@temptrue
     \else
      \expandafter\ifx\csname forest@pi@(%
       \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname,%
       \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname)--(%
       \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb x\endcsname,%
       \csname forest@pi@\the\c@pgf@counta @\the\c@pgf@countb y\endcsname)%
       \endcsname\forest@inpath
       \forest@temptrue
      \fi
     \fi
     \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.

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

Second, create a line to the end point.

```
      \edef\forest@last@x{%
       \csname forest@pi@\forest@aplusone @\the\c@pgf@countc x\endcsname}%
      \edef\forest@last@y{%
       \csname forest@pi@\forest@aplusone @\the\c@pgf@countc y\endcsname}%
      \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
```

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

```
      \c@pgf@countc=\csname forest@pi@\forest@aplusone @n\endcsname
      \c@pgf@countb=\csname forest@pi@\the\c@pgf@counta @n\endcsname
     \fi
     \advance\c@pgf@countc 1
    \forest@repeatc
    \advance\c@pgf@countb 1
}
```

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

```
\def\forest@gettightedgeofpath@maybemoveto#1#2{%
 \forest@temptrue
 \ifx\forest@last@x\relax\else
  \ifdim\forest@last@x=\csname forest@pi@#1@#2x\endcsname\relax
   \ifdim\forest@last@y=\csname forest@pi@#1@#2y\endcsname\relax
    \forest@tempfalse
   \fi
  \fi
 \fi
 \ifforest@temp
  \edef\forest@last@x{\csname forest@pi@#1@#2x\endcsname}%
  \edef\forest@last@y{\csname forest@pi@#1@#2y\endcsname}%
  \pgfsyssoftpath@moveto\forest@last@x\forest@last@y
 \fi
}
```

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

```
\def\forest@simplifypath#1#2{%
 \pgfsyssoftpath@setcurrentpath\pgfutil@empty
 \forest@save@pgfsyssoftpath@tokendefs
 \let\pgfsyssoftpath@movetotoken\forest@simplifypath@moveto
```

```
  \let\pgfsyssoftpath@linetotoken\forest@simplifypath@lineto
  \let\forest@last@x\relax
  \let\forest@last@y\relax
  \let\forest@last@atan\relax
  #1%
  \ifx\forest@last@x\relax\else
    \ifx\forest@last@atan\relax\else
      \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
    \fi
  \fi
  \forest@restore@pgfsyssoftpath@tokendefs
  \pgfsyssoftpath@getcurrentpath#2%
}
```

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.

```
\def\forest@simplifypath@moveto#1#2{%
  \ifx\forest@last@x\relax\else
    \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
  \fi
  \pgfsyssoftpath@moveto{#1}{#2}%
  \def\forest@last@x{#1}%
  \def\forest@last@y{#2}%
  \let\forest@last@atan\relax
}
```

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

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

When a line-to is encountered. . .

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

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

```
    \def\forest@last@x{#1}%
    \def\forest@last@y{#2}%
    \let\forest@last@atan\relax
  \else
```

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

```
    \pgfpointdiff{\pgfqpoint{#1}{#2}}{\pgfqpoint{\forest@last@x}{\forest@last@y}}%
    \ifdim\pgf@x<\pgfintersectiontolerance
      \ifdim-\pgf@x<\pgfintersectiontolerance
        \pgf@x=0pt
      \fi
    \fi
    \csname pgfmathatan2\endcsname{\pgf@x}{\pgf@y}%
    \let\forest@current@atan\pgfmathresult
    \ifx\forest@last@atan\relax
```

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

```
      \def\forest@last@x{#1}%
      \def\forest@last@y{#2}%
      \let\forest@last@atan\forest@current@atan
    \else
```

Otherwise, compare the first and the current slope.

```
      \pgfutil@tempdima=\forest@current@atan pt
      \advance\pgfutil@tempdima -\forest@last@atan pt
      \ifdim\pgfutil@tempdima<0pt\relax
        \multiply\pgfutil@tempdima -1
      \fi
      \ifdim\pgfutil@tempdima<\forest@getedgeofpath@precision\relax
      \else
```

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

```
        \pgfsyssoftpath@lineto\forest@last@x\forest@last@y
        \let\forest@last@atan\forest@current@atan
      \fi
```

In any event, update the last point.

```
      \def\forest@last@x{#1}%
      \def\forest@last@y{#2}%
    \fi
  \fi
}
```

12.4 Get rectangle/band edge

```
\def\forest@getnegativerectangleedgeofpath#1#2{%
  \forest@getnegativerectangleorbandedgeofpath{#1}{#2}{\the\pgf@xb}}
\def\forest@getpositiverectangleedgeofpath#1#2{%
  \forest@getpositiverectangleorbandedgeofpath{#1}{#2}{\the\pgf@xb}}
\def\forest@getbothrectangleedgesofpath#1#2#3{%
  \forest@getbothrectangleorbandedgesofpath{#1}{#2}{#3}{\the\pgf@xb}}
\def\forest@bandlength{5000pt} % something large (ca. 180cm), but still manageable for TeX without producing
\def\forest@getnegativebandedgeofpath#1#2{%
  \forest@getnegativerectangleorbandedgeofpath{#1}{#2}{\forest@bandlength}}
\def\forest@getpositivebandedgeofpath#1#2{%
  \forest@getpositiverectangleorbandedgeofpath{#1}{#2}{\forest@bandlength}}
\def\forest@getbothbandedgesofpath#1#2#3{%
  \forest@getbothrectangleorbandedgesofpath{#1}{#2}{#3}{\forest@bandlength}}
\def\forest@getnegativerectangleorbandedgeofpath#1#2#3{%
  \forest@path@getboundingrectangle@ls#1{\forest@grow}%
  \edef\forest@gre@path{%
    \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@ya}%
    \noexpand\pgfsyssoftpath@linetotoken{#3}{\the\pgf@ya}%
  }%
  {%
    \pgftransformreset
    \pgftransformrotate{\forest@grow}%
    \forest@pgfpathtransformed\forest@gre@path
  }%
  \pgfsyssoftpath@getcurrentpath#2%
}
\def\forest@getpositiverectangleorbandedgeofpath#1#2#3{%
  \forest@path@getboundingrectangle@ls#1{\forest@grow}%
  \edef\forest@gre@path{%
    \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@yb}%
    \noexpand\pgfsyssoftpath@linetotoken{#3}{\the\pgf@yb}%
  }%
  {%
    \pgftransformreset
    \pgftransformrotate{\forest@grow}%
    \forest@pgfpathtransformed\forest@gre@path
  }%
  \pgfsyssoftpath@getcurrentpath#2%
}
\def\forest@getbothrectangleorbandedgesofpath#1#2#3#4{%
  \forest@path@getboundingrectangle@ls#1{\forest@grow}%
  \edef\forest@gre@negpath{%
    \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@ya}%
    \noexpand\pgfsyssoftpath@linetotoken{#4}{\the\pgf@ya}%
  }%
  \edef\forest@gre@pospath{%
    \noexpand\pgfsyssoftpath@movetotoken{\the\pgf@xa}{\the\pgf@yb}%
    \noexpand\pgfsyssoftpath@linetotoken{#4}{\the\pgf@yb}%
  }%
  {%
    \pgftransformreset
    \pgftransformrotate{\forest@grow}%
    \forest@pgfpathtransformed\forest@gre@negpath
  }%
  \pgfsyssoftpath@getcurrentpath#2%
  {%
    \pgftransformreset
    \pgftransformrotate{\forest@grow}%
    \forest@pgfpathtransformed\forest@gre@pospath
  }%
  \pgfsyssoftpath@getcurrentpath#3%
}
```

12.5 Distance between paths

Another crucial part of the package.

```
\def\forest@distance@between@edge@paths#1#2#3{%
  % #1, #2 = (edge) paths
  %
  % project paths
  \forest@projectpathtogrowline#1{forest@p1@}%
  \forest@projectpathtogrowline#2{forest@p2@}%
  % merge projections (the lists are sorted already, because edge
  % paths are |sorted|)
  \forest@dbep@mergeprojections
    {forest@p1@}{forest@p2@}%
    {forest@P1@}{forest@P2@}%
  % process projections
  \forest@processprojectioninfo{forest@P1@}{forest@PI1@}%
  \forest@processprojectioninfo{forest@P2@}{forest@PI2@}%
  % break paths
  \forest@breakpath#1{forest@PI1@}\forest@broken@one
  \forest@breakpath#2{forest@PI2@}\forest@broken@two
  % sort inner arrays ---optimize: it’s enough to find max and min
  \forest@sort@inner@arrays{forest@PI1@}\forest@sort@descending
  \forest@sort@inner@arrays{forest@PI2@}\forest@sort@ascending
  % compute the distance
  \let\forest@distance\relax
  \c@pgf@countc=0
  \loop
  \ifnum\c@pgf@countc<\csname forest@PI1@n\endcsname\relax
    \ifnum\csname forest@PI1@\the\c@pgf@countc @n\endcsname=0 \else
      \ifnum\csname forest@PI2@\the\c@pgf@countc @n\endcsname=0 \else
        \pgfutil@tempdima=\csname forest@PI2@\the\c@pgf@countc @0d\endcsname\relax
        \advance\pgfutil@tempdima -\csname forest@PI1@\the\c@pgf@countc @0d\endcsname\relax
        \ifx\forest@distance\relax
          \edef\forest@distance{\the\pgfutil@tempdima}%
        \else
          \ifdim\pgfutil@tempdima<\forest@distance\relax
            \edef\forest@distance{\the\pgfutil@tempdima}%
          \fi
        \fi
      \fi
    \fi
    \advance\c@pgf@countc 1
  \repeat
  \let#3\forest@distance
}
  % merge projections: we need two projection arrays, both containing
  % projection points from both paths, but each with the original
  % points from only one path
\def\forest@dbep@mergeprojections#1#2#3#4{%
  % TODO: optimize: v bistvu ni treba sortirat, ker je edge path e sortiran
  \forest@sortprojections{#1}%
  \forest@sortprojections{#2}%
  \c@pgf@counta=0
  \c@pgf@countb=0
  \c@pgf@countc=0
  \edef\forest@input@prefix@one{#1}%
  \edef\forest@input@prefix@two{#2}%
  \edef\forest@output@prefix@one{#3}%
  \edef\forest@output@prefix@two{#4}%
  \forest@dbep@mp@iterate
  \csedef{#3n}{\the\c@pgf@countc}%
  \csedef{#4n}{\the\c@pgf@countc}%
}
\def\forest@dbep@mp@iterate{%
  \let\forest@dbep@mp@next\forest@dbep@mp@iterate
  \ifnum\c@pgf@counta<\csname\forest@input@prefix@one n\endcsname\relax
    \ifnum\c@pgf@countb<\csname\forest@input@prefix@two n\endcsname\relax
      \let\forest@dbep@mp@next\forest@dbep@mp@do
    \else
      \let\forest@dbep@mp@next\forest@dbep@mp@iteratefirst
    \fi
  \else
    \ifnum\c@pgf@countb<\csname\forest@input@prefix@two n\endcsname\relax
      \let\forest@dbep@mp@next\forest@dbep@mp@iteratesecond
    \else
      \let\forest@dbep@mp@next\relax
    \fi
  \fi
  \forest@dbep@mp@next
}
\def\forest@dbep@mp@do{%
  \forest@sort@cmptwodimcs%
    {\forest@input@prefix@one\the\c@pgf@counta xp}%
    {\forest@input@prefix@one\the\c@pgf@counta yp}%
    {\forest@input@prefix@two\the\c@pgf@countb xp}%
    {\forest@input@prefix@two\the\c@pgf@countb yp}%
  \if\forest@sort@cmp@result=%
    \forest@dbep@mp@@store@p\forest@input@prefix@one\c@pgf@counta
    \forest@dbep@mp@@store@o\forest@input@prefix@one
        \c@pgf@counta\forest@output@prefix@one
    \forest@dbep@mp@@store@o\forest@input@prefix@two
        \c@pgf@countb\forest@output@prefix@two
    \advance\c@pgf@counta 1
    \advance\c@pgf@countb 1
  \else
    \if\forest@sort@cmp@result>%
      \forest@dbep@mp@@store@p\forest@input@prefix@two\c@pgf@countb
      \forest@dbep@mp@@store@o\forest@input@prefix@two
          \c@pgf@countb\forest@output@prefix@two
      \advance\c@pgf@countb 1
    \else%<
      \forest@dbep@mp@@store@p\forest@input@prefix@one\c@pgf@counta
      \forest@dbep@mp@@store@o\forest@input@prefix@one
          \c@pgf@counta\forest@output@prefix@one
      \advance\c@pgf@counta 1
    \fi
  \fi
  \advance\c@pgf@countc 1
  \forest@dbep@mp@iterate
}
\def\forest@dbep@mp@@store@p#1#2{%
  \csletcs
    {\forest@output@prefix@one\the\c@pgf@countc xp}%
    {#1\the#2xp}%
  \csletcs
    {\forest@output@prefix@one\the\c@pgf@countc yp}%
    {#1\the#2yp}%
  \csletcs
    {\forest@output@prefix@two\the\c@pgf@countc xp}%
    {#1\the#2xp}%
  \csletcs
    {\forest@output@prefix@two\the\c@pgf@countc yp}%
    {#1\the#2yp}%
}
\def\forest@dbep@mp@@store@o#1#2#3{%
  \csletcs{#3\the\c@pgf@countc xo}{#1\the#2xo}%
  \csletcs{#3\the\c@pgf@countc yo}{#1\the#2yo}%
}
\def\forest@dbep@mp@iteratefirst{%
  \forest@dbep@mp@iterateone\forest@input@prefix@one\c@pgf@counta\forest@output@prefix@one
}
\def\forest@dbep@mp@iteratesecond{%
  \forest@dbep@mp@iterateone\forest@input@prefix@two\c@pgf@countb\forest@output@prefix@two
}
\def\forest@dbep@mp@iterateone#1#2#3{%
  \loop
  \ifnum#2<\csname#1n\endcsname\relax
    \forest@dbep@mp@@store@p#1#2%
    \forest@dbep@mp@@store@o#1#2#3%
    \advance\c@pgf@countc 1
    \advance#21
  \repeat
}
```

12.6 Utilities

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

```
\newif\ifforest@equaltotolerance
\def\forest@equaltotolerance#1#2{{%
  \pgfpointdiff{#1}{#2}%
  \ifdim\pgf@x<0pt \multiply\pgf@x -1 \fi
  \ifdim\pgf@y<0pt \multiply\pgf@y -1 \fi
  \global\forest@equaltotolerancefalse
  \ifdim\pgf@x<\pgfintersectiontolerance\relax
    \ifdim\pgf@y<\pgfintersectiontolerance\relax
      \global\forest@equaltotolerancetrue
    \fi
  \fi
}}
```

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

```
\def\forest@save@pgfsyssoftpath@tokendefs{%
  \let\forest@origmovetotoken\pgfsyssoftpath@movetotoken
  \let\forest@origlinetotoken\pgfsyssoftpath@linetotoken
  \let\forest@origcurvetosupportatoken\pgfsyssoftpath@curvetosupportatoken
  \let\forest@origcurvetosupportbtoken\pgfsyssoftpath@curvetosupportbtoken
  \let\forest@origcurvetotoken\pgfsyssoftpath@curvetototoken
  \let\forest@origrectcornertoken\pgfsyssoftpath@rectcornertoken
  \let\forest@origrectsizetoken\pgfsyssoftpath@rectsizetoken
  \let\forest@origclosepathtoken\pgfsyssoftpath@closepathtoken
  \let\pgfsyssoftpath@movetotoken\forest@badtoken
  \let\pgfsyssoftpath@linetotoken\forest@badtoken
  \let\pgfsyssoftpath@curvetosupportatoken\forest@badtoken
  \let\pgfsyssoftpath@curvetosupportbtoken\forest@badtoken
  \let\pgfsyssoftpath@curvetototoken\forest@badtoken
  \let\pgfsyssoftpath@rectcornertoken\forest@badtoken
  \let\pgfsyssoftpath@rectsizetoken\forest@badtoken
  \let\pgfsyssoftpath@closepathtoken\forest@badtoken
}
\def\forest@badtoken{%
  \PackageError{forest}{This token should not be in this path}{}%
}
\def\forest@restore@pgfsyssoftpath@tokendefs{%
  \let\pgfsyssoftpath@movetotoken\forest@origmovetotoken
  \let\pgfsyssoftpath@linetotoken\forest@origlinetotoken
  \let\pgfsyssoftpath@curvetosupportatoken\forest@origcurvetosupportatoken
  \let\pgfsyssoftpath@curvetosupportbtoken\forest@origcurvetosupportbtoken
  \let\pgfsyssoftpath@curvetototoken\forest@origcurvetotoken
  \let\pgfsyssoftpath@rectcornertoken\forest@origrectcornertoken
  \let\pgfsyssoftpath@rectsizetoken\forest@origrectsizetoken
  \let\pgfsyssoftpath@closepathtoken\forest@origclosepathtoken
}
```

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

```
\def\forest@extendpath#1#2#3{%
  \pgf@process{#3}%
  \pgfsyssoftpath@setcurrentpath#1%
  \forest@save@pgfsyssoftpath@tokendefs
  \let\pgfsyssoftpath@movetotoken\forest@extendpath@moveto
  \let\pgfsyssoftpath@linetotoken\forest@extendpath@lineto
  #2%
  \forest@restore@pgfsyssoftpath@tokendefs
  \pgfsyssoftpath@getcurrentpath#1%
}
\def\forest@extendpath@moveto#1#2{%
  \forest@extendpath@do{#1}{#2}\pgfsyssoftpath@moveto
}
\def\forest@extendpath@lineto#1#2{%
  \forest@extendpath@do{#1}{#2}\pgfsyssoftpath@lineto
}
\def\forest@extendpath@do#1#2#3{%
  {%
    \advance\pgf@x #1
    \advance\pgf@y #2
    #3{\the\pgf@x}{\the\pgf@y}%
  }%
}
```

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.)

```
\def\forest@path@getboundingrectangle@ls#1#2{%
  {%
```

```
    \pgftransformreset
    \pgftransformrotate{-(#2)}%
    \forest@pgfpathtransformed#1%
  }%
  \pgfsyssoftpath@getcurrentpath\forest@gbr@rotatedpath
  \forest@path@getboundingrectangle@xy\forest@gbr@rotatedpath
}
\def\forest@path@getboundingrectangle@xy#1{%
  \forest@save@pgfsyssoftpath@tokendefs
  \let\pgfsyssoftpath@movetotoken\forest@gbr@firstpoint
  \let\pgfsyssoftpath@linetotoken\forest@gbr@firstpoint
  #1%
  \forest@restore@pgfsyssoftpath@tokendefs
}
\def\forest@gbr@firstpoint#1#2{%
  \pgf@xa=#1 \pgf@xb=#1 \pgf@ya=#2 \pgf@yb=#2
  \let\pgfsyssoftpath@movetotoken\forest@gbr@point
  \let\pgfsyssoftpath@linetotoken\forest@gbr@point
}
\def\forest@gbr@point#1#2{%
  \ifdim#1<\pgf@xa\relax\pgf@xa=#1 \fi
  \ifdim#1>\pgf@xb\relax\pgf@xb=#1 \fi
  \ifdim#2<\pgf@ya\relax\pgf@ya=#2 \fi
  \ifdim#2>\pgf@yb\relax\pgf@yb=#2 \fi
}
```

13 The outer UI

13.1 Package options

```
\newif\ifforesttikzcshack
\foresttikzcshacktrue
\newif\ifforest@install@keys@to@tikz@path@
\forest@install@keys@to@tikz@path@true
\forestset{package@options/.cd,
  external/.is if=forest@external@,
  tikzcshack/.is if=foresttikzcshack,
  tikzinstallkeys/.is if=forest@install@keys@to@tikz@path@,
}
```

13.2 Externalization

```
\pgfkeys{/forest/external/.cd,
  copy command/.initial={cp "\source" "\target"},
  optimize/.is if=forest@external@optimize@,
  context/.initial={%
    \forestOve{\csname forest@id@of@standard node\endcsname}{environment@formula}},
  depends on macro/.style={context/.append/.expanded={%
      \expandafter\detokenize\expandafter{#1}}},
}
\def\forest@external@copy#1#2{%
  \pgfkeysgetvalue{/forest/external/copy command}\forest@copy@command
  \ifx\forest@copy@command\pgfkeysnovalue\else
    \IfFileExists{#1}{%
      {%
        \def\source{#1}%
        \def\target{#2}%
        \immediate\write18{\forest@copy@command}%
      }%
    }{}%
  \fi
}
\newif\ifforest@external@
\newif\ifforest@external@optimize@
\forest@external@optimize@true
\ProcessPgfPackageOptions{/forest/package@options}
\ifforest@install@keys@to@tikz@path@
  \tikzset{fit to tree/.style={/forest/fit to tree}}
\fi
\ifforest@external@
  \ifdefined\tikzexternal@tikz@replacement\else
    \usetikzlibrary{external}%
  \fi
  \pgfkeys{%
    /tikz/external/failed ref warnings for={},
    /pgf/images/aux in dpth=false,
  }%
  \tikzifexternalizing{}{%
    \forest@external@copy{\jobname.aux}{\jobname.aux.copy}%
  }%
  \AtBeginDocument{%
    \tikzifexternalizing{%
      \IfFileExists{\tikzexternalrealjob.aux.copy}{%
        \makeatletter
        \input \tikzexternalrealjob.aux.copy
        \makeatother
      }{}%
    }{%
      \newwrite\forest@auxout
      \immediate\openout\forest@auxout=\tikzexternalrealjob.for.tmp
    }%
    \IfFileExists{\tikzexternalrealjob.for}{%
      {%
        \makehashother\makeatletter
        \input \tikzexternalrealjob.for
      }%
    }{}%
  }%
  \AtEndDocument{%
    \tikzifexternalizing{}{%
      \immediate\closeout\forest@auxout
      \forest@external@copy{\jobname.for.tmp}{\jobname.for}%
    }%
  }%
\fi
```

13.3 The forest environment

There are three ways to invoke Forest: the environent 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.

```
\newenvironment{forest}{\pgfkeysalso{/forest/begin forest}\Collect@Body\forest@env}{}
\long\def\Forest{\pgfkeysalso{/forest/begin forest}\@ifnextchar*{\forest@nogroup}{\forest@group}}
\def\forest@group#1{{\forest@env{#1}}}
\def\forest@nogroup*#1{\forest@env{#1}}
\newif\ifforest@externalize@tree@
\newif\ifforest@was@tikzexternalwasenable
\long\def\forest@env#1{%
  \let\forest@external@next\forest@begin
  \forest@was@tikzexternalwasenablefalse
  \ifdefined\tikzexternal@tikz@replacement
    \ifx\tikz\tikzexternal@tikz@replacement
```

```
      \forest@was@tikzexternalwasenabletrue
      \tikzexternaldisable
    \fi
  \fi
  \forest@externalize@tree@false
  \ifforest@external@
    \ifforest@was@tikzexternalwasenable
      \tikzifexternalizing{%
        \let\forest@external@next\forest@begin@externalizing
      }{%
        \let\forest@external@next\forest@begin@externalize
      }%
    \fi
  \fi
  \forest@standardnode@calibrate
  \forest@external@next{#1}%
}
```

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

```
\long\def\forest@begin@externalizing#1{%
  \forest@external@setup{#1}%
  \let\forest@external@next\forest@begin
  \forest@externalize@inner@n=-1
  \ifforest@external@optimize@\forest@externalizing@maybeoptimize\fi
  \forest@external@next{#1}%
  \tikzexternalenable
}
\def\forest@externalizing@maybeoptimize{%
  \edef\forest@temp{\tikzexternalrealjob-forest-\forest@externalize@outer@n}%
  \edef\forest@marshal{%
    \noexpand\pgfutil@in@
      {\expandafter\detokenize\expandafter{\forest@temp}.}
      {\expandafter\detokenize\expandafter{\jobname}.}%
  }\forest@marshal
  \ifpgfutil@in@
  \else
    \let\forest@external@next\@gobble
  \fi
}
```

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

```
\long\def\forest@begin@externalize#1{%
  \forest@external@setup{#1}%
  \iftikzexternal@file@isuptodate
    \setbox0=\hbox{%
      \csname forest@externalcheck@\forest@externalize@outer@n\endcsname
    }%
  \fi
  \iftikzexternal@file@isuptodate
    \csname forest@externalload@\forest@externalize@outer@n\endcsname
  \else
    \forest@externalize@tree@true
    \forest@externalize@inner@n=-1
    \forest@begin{#1}%
    \ifcsdef{forest@externalize@@\forest@externalize@id}{}{%
      \immediate\write\forest@auxout{%
        \noexpand\forest@external
        {\forest@externalize@outer@n}%
        {\expandafter\detokenize\expandafter{\forest@externalize@id}}%
        {\expandonce\forest@externalize@checkimages}%
        {\expandonce\forest@externalize@loadimages}%
      }%
    }%
  \fi
  \tikzexternalenable
}
\def\forest@includeexternal@check#1{%
  \tikzsetnextfilename{#1}%
  \tikzexternal@externalizefig@systemcall@uptodatecheck
}
\def\makehashother{\catcode`\#=12}%
\long\def\forest@external@setup#1{%
  % set up \forest@externalize@id and \forest@externalize@outer@n
  % we need to deal with #s correctly (\write doubles them)
  \setbox0=\hbox{\makehashother\makeatletter
    \scantokens{\forest@temp@toks{#1}}\expandafter
  }%
  \expandafter\forest@temp@toks\expandafter{\the\forest@temp@toks}%
  \edef\forest@temp{\pgfkeysvalueof{/forest/external/context}}%
  \edef\forest@externalize@id{%
    \expandafter\detokenize\expandafter{\forest@temp}%
    @@%
    \expandafter\detokenize\expandafter{\the\forest@temp@toks}%
  }%
  \letcs\forest@externalize@outer@n{forest@externalize@@\forest@externalize@id}%
  \ifdefined\forest@externalize@outer@n
    \global\tikzexternal@file@isuptodatetrue
  \else
    \global\advance\forest@externalize@max@outer@n 1
    \edef\forest@externalize@outer@n{\the\forest@externalize@max@outer@n}%
    \global\tikzexternal@file@isuptodatefalse
  \fi
  \def\forest@externalize@loadimages{}%
  \def\forest@externalize@checkimages{}%
}
\newcount\forest@externalize@max@outer@n
\global\forest@externalize@max@outer@n=0
\newcount\forest@externalize@inner@n
```

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

```
\long\def\forest@external#1#2#3#4{% #1=n,#2=context+source code,#3=update check code, #4=load code
  \ifnum\forest@externalize@max@outer@n<#1
    \global\forest@externalize@max@outer@n=#1
  \fi
  \global\csdef{forest@externalize@@\detokenize{#2}}{#1}%
  \global\csdef{forest@externalcheck@#1}{#3}%
  \global\csdef{forest@externalload@#1}{#4}%
  \tikzifexternalizing{}{%
    \immediate\write\forest@auxout{%
      \noexpand\forest@external{#1}%
      {\expandafter\detokenize\expandafter{#2}}%
      {\unexpanded{#3}}%
      {\unexpanded{#4}}%
    }%
  }%
}
```

These two macros include the external picture.

```
\def\forest@includeexternal#1{%
  \edef\forest@temp{\pgfkeysvalueof{/forest/external/context}}%
  \typeout{forest: Including external picture '#1' for forest context+code:
    '\expandafter\detokenize\expandafter{\forest@externalize@id}'}%
  {%
```

```
    %\def\pgf@declaredraftimage##1##2{\def\pgf@image{\hbox{}}}%
    \tikzsetnextfilename{#1}%
    \tikzexternalenable
    \tikz{}%
  }%
}
\def\forest@includeexternal@box#1#2{%
  \global\setbox#1=\hbox{\forest@includeexternal{#2}}%
}
```

This code runs the bracket parser and stage processing.

```
\long\def\forest@begin#1{%
  \iffalse{\fi\forest@parsebracket#1}%
}
\def\forest@parsebracket{%
  \bracketParse{\forest@get@root@afterthought}\forest@root=%
}
\def\forest@get@root@afterthought{%
  \expandafter\forest@get@root@afterthought@\expandafter{\iffalse}\fi
}
\long\def\forest@get@root@afterthought@#1{%
  \ifblank{#1}{}{%
    \forestOeappto{\forest@root}{given options}{,afterthought={\unexpanded{#1}}}%
  }%
  \forest@do
}
\def\forest@do{%
  \forest@node@Compute@numeric@ts@info{\forest@root}%
  \forestset{process keylist=given options}%
  \forestset{stages}%
  \pgfkeysalso{/forest/end forest}%
  \ifforest@was@tikzexternalwasenable
    \tikzexternalenable
  \fi
}
```

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!

```
\def\forest@standardnode@new{%
  \advance\forest@node@maxid1
  \forest@fornode{\the\forest@node@maxid}{%
    \forest@node@init
    \forest@node@setname{standard node}%
  }%
}
\def\forest@standardnode@calibrate{%
  \forest@fornode{\forest@node@Nametoid{standard node}}{%
    \edef\forest@environment{\forestove{environment@formula}}%
    \forestoget{previous@environment}\forest@previous@environment
    \ifx\forest@environment\forest@previous@environment\else
      \forestolet{previous@environment}\forest@environment
      \forest@node@typeset
      \forestoget{calibration@procedure}\forest@temp
      \expandafter\forestset\expandafter{\forest@temp}%
    \fi
  }%
}
```

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.)

```
\def\forestStandardNode[#1]#2#3#4{%
  \let\forest@standardnode@restoretikzexternal\relax
  \ifdefined\tikzexternaldisable
    \ifx\tikz\tikzexternal@tikz@replacement
      \tikzexternaldisable
      \let\forest@standardnode@restoretikzexternal\tikzexternalenable
    \fi
  \fi
  \forest@standardnode@new
  \forest@fornode{\forest@node@Nametoid{standard node}}{%
    \forestset{content=#1}%
    \forestoset{environment@formula}{#2}%
    \edef\forest@temp{\unexpanded{#3}}%
    \forestolet{calibration@procedure}\forest@temp
    \def\forest@calibration@initializing@code{}%
    \pgfqkeys{/forest/initializing@code}{#4}%
    \forestolet{initializing@code}\forest@calibration@initializing@code
    \forest@standardnode@restoretikzexternal
  }
}
\forestset{initializing@code/.unknown/.code={%
    \eappto\forest@calibration@initializing@code{%
      \noexpand\forestOget{\forest@node@Nametoid{standard node}}{\pgfkeyscurrentname}\noexpand\forest@temp
      \noexpand\forestolet{\pgfkeyscurrentname}\noexpand\forest@temp
    }%
  }
}
```

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

```
\def\forest@initializefromstandardnode{%
  \forestOve{\forest@node@Nametoid{standard node}}{initializing@code}%
}
```

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`.)

```
\forestStandardNode[dj]
  {%
    \forestOve{\forest@node@Nametoid{standard node}}{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}
```

13.5 ls coordinate system

```
\pgfqkeys{/forest/@cs}{%
  name/.code={%
    \edef\forest@cn{\forest@node@Nametoid{#1}}%
    \forest@forestcs@resetxy},
  id/.code={%
    \edef\forest@cn{#1}%
    \forest@forestcs@resetxy},
  go/.code={%
    \forest@go{#1}%
    \forest@forestcs@resetxy},
  anchor/.code={\forest@forestcs@anchor{#1}},
  l/.code={%
    \pgfmathsetlengthmacro\forest@forestcs@l{#1}%
    \forest@forestcs@ls
  },
  s/.code={%
    \pgfmathsetlengthmacro\forest@forestcs@s{#1}%
    \forest@forestcs@ls
  },
  .unknown/.code={%
    \expandafter\pgfutil@in@\expandafter.\expandafter{\pgfkeyscurrentname}%
    \ifpgfutil@in@
      \expandafter\forest@forestcs@namegoanchor\pgfkeyscurrentname\forest@end
    \else
      \expandafter\forest@nameandgo\expandafter{\pgfkeyscurrentname}%
      \forest@forestcs@resetxy
    \fi
  }
}
\def\forest@forestcs@resetxy{%
  \ifnum\forest@cn=0
  \else
    \global\pgf@x\forestove{x}%
    \global\pgf@y\forestove{y}%
  \fi
}
\def\forest@forestcs@ls{%
  \ifdefined\forest@forestcs@l
    \ifdefined\forest@forestcs@s
      {%
        \pgftransformreset
        \pgftransformrotate{\forestove{grow}}%
        \pgfpointtransformed{\pgfpoint{\forest@forestcs@l}{\forest@forestcs@s}}%
      }%
      \global\advance\pgf@x\forestove{x}%
      \global\advance\pgf@y\forestove{y}%
    \fi
  \fi
}
\def\forest@forestcs@anchor#1{%
  \edef\forest@marshal{%
    \noexpand\forest@original@tikz@parse@node\relax
    (\forestove{name}\ifx\relax#1\relax\else.\fi#1)%
  }\forest@marshal
}
```

```
\def\forest@forestcs@namegoanchor#1.#2\forest@end{%
  \forest@nameandgo{#1}%
  \forest@forestcs@anchor{#2}%
}
\tikzdeclarecoordinatesystem{forest}{%
  \forest@forthis{%
    \forest@forestcs@resetxy
    \ifdefined\forest@forestcs@l\undef\forest@forestcs@l\fi
    \ifdefined\forest@forestcs@s\undef\forest@forestcs@s\fi
    \pgfqkeys{/forest/@cs}{#1}%
  }%
}
```

References

[1] Donald E. Knuth. *The TeXbook.* Addison-Wesley, 1996.

[2] Till Tantau. *TikZ & PGF, Manual for Version 2.10*, 2007.

Index