

# Using TikZ for linguistic diagrams

James A. Crippen

University of British Columbia – jcrippen@gmail.com

March 23, 2017

## Abstract

TikZ is a package for L<sup>A</sup>T<sub>E</sub>X that provides a powerful language for specifying graphics. TikZ can easily produce the kinds of trees and diagrams that are used in linguistics. This document provides a friendly introduction with many examples for using TikZ to draw syntactic trees, autosegmental diagrams, and lattices. TikZ is also combined with ExPex to produce numbered examples containing movement arrows, graphical annotations, and small diagrams. The illustrations of TikZ in this document are all oriented toward manuscripts and articles, but because TikZ is maintained by the same people behind the BEAMER package, most if not all of the techniques presented in this document can also be used in slides and posters.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Trees</b>	<b>6</b>
2.1	Styles and modifying option values . . . . .	8
2.2	Hanging annotations . . . . .	11
2.3	Fancy branches . . . . .	12
2.4	Adjusting node sizes . . . . .	12
2.5	Node content . . . . .	16
2.6	Beyond binary: $n$ -ary branches and triangles . . . . .	18
2.7	Arrows, phases, and other lines . . . . .	21
2.8	TikZ and trees . . . . .	31
<b>3</b>	<b>Matrices</b>	<b>34</b>
3.1	Autosegmental diagrams . . . . .	37
3.2	Partial orderings and lattices . . . . .	40
3.3	Matrices in trees . . . . .	43
3.4	Trees in matrices . . . . .	44
<b>4</b>	<b>Numbered examples</b>	<b>46</b>
4.1	Arrows in examples . . . . .	46
4.2	Trees in examples . . . . .	52
4.3	Tables with drawings in examples . . . . .	55

# 1 Introduction

Linguists often need to draw diagrams in their work. Linguistic diagrams prototypically include syntactic trees and autosegmental representations, but many other kinds of diagrams are also frequently encountered such as type lattices, historical sound changes, flowcharts, categorization trees, and partial orderings. Drawing these can be annoying and time consuming.  $\LaTeX$ -using linguists are generally aware that there are solutions out there for drawing diagrams, and one particularly hears `TikZ` mentioned but rarely if ever explained. This document illustrates some basic uses of `TikZ` for constructing linguistic diagrams along with working illustrations and references to the voluminous `TikZ` documentation.

`TikZ` (a recursive acronym for “`TikZ` ist *kein* Zeichenprogramm”) is a package for  $\LaTeX$  created by Till Tantau at the Universität zu Lübeck Institut für Theoretische Informatik, originally for use in his PhD dissertation. `TikZ` is itself based on a lower level package called `PGF` which contains two layers: the System Layer and the Basic Layer. The System Layer is platform-dependent, producing output variously for PDF and DVI $\rightarrow$ PostScript. The Basic Layer is platform-independent, forming a simple abstraction above the System Layer so that users need not know about the platform-dependent details of graphical output. `TikZ` is a ‘front end’ implemented on top of the Basic Layer of `PGF`, providing users a rich language for drawing complex graphics. Alongside `PGF` and `TikZ` are some associated packages that can be used separately, including `pgfkeys` for key-value management, `pgffor` for looping over lists, `pgfcalendar` for manipulating dates, `pgfpages` for page management, and `pgfmath` for somewhat complex mathematical calculations.

In the rest of this introduction I present the conventions for displaying and discussing  $\LaTeX$  code as well as some general and  $\LaTeX$ -specific programming practices. I assume a working knowledge of  $\LaTeX$  but not necessarily plain  $\TeX$ . (I will introduce a few plain  $\TeX$  macros including `\llap` and `\rlap`, `\phantom` and `\vphantom`, `\halign`, `\noalign`, `\hfil`, `\vtop`, and `\offinterlineskip`.) Readers should already know the difference between the preamble and the document body, the command versus environment distinction, and the basic commands and environments used for everyday typesetting. Section 2 discusses basic tree drawing in `TikZ` including binary and  $n$ -ary trees, triangles, annotations, edge decorations, and movement arrows. Section 3 on matrices introduces `TikZ`’s `matrix` drawing and shows how it can be used to form grid-based diagrams (e.g. autosegmental representations) which are difficult to represent as trees. Section 4 illustrates some advanced techniques for numbered examples by combining `Expex` and `TikZ` together, as well as using some advanced features of `Expex` and some plain  $\TeX$  macros.

Examples of  $\LaTeX$  code are presented like the box below. The top half of the box contains code and the bottom half below the dashed line contains the resulting output after compilation with  $\LaTeX$ . The background colour is a 97% grey (i.e. `black!3!white`, see the `xcolor` package for details) to clarify the difference between an explicit white colour and no colour (transparent). These boxes are created with the `tcolorbox` package (based on `TikZ` and partly inspired by `TikZ`’s documentation) along with the `minted` package and the `Pygments` program for automatic syntax colouring.

This is a `\LaTeX` example using `\texttt{tcolorbox}` with `\texttt{minted}` and `\TikZ`. The grey background shows when a `\colorbox{white}{white background}` is different from the ordinary transparent ‘colour’.

This is a `LaTeX` example using `tcolorbox` with `minted` and `TikZ`. The grey background shows when a `white background` is different from the ordinary transparent ‘colour’.

Occasionally we need to distinguish between code that belongs in the preamble before the start of the `\begin{document}` environment and code that should be in the document body between `\begin{document}` and `\end{document}`. Example boxes that refer specifically to one or the other have a title at the top like the next two examples.

#### Preamble

```
\DeclareDocumentCommand \preamblecommandexample {m} {  
  This command defined in the preamble has been given the mandatory argument ‘#1’.  
}
```

#### Document Body

```
\preamblecommandexample{foo}
```

This command defined in the preamble has been given the mandatory argument ‘foo’.

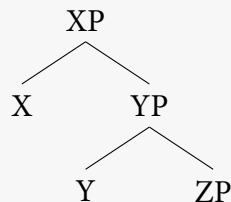
To make `TikZ` available in your document you must `\usepackage{tikz}` in your preamble. This is also a good place to load `TikZ` libraries and specify document-wide styles.

#### Preamble

```
\usepackage{tikz}  
\usetikzlibrary{backgrounds}  
\tikzset{showall/.style={framed, every node/.style=draw}}
```

Many of the examples in this document are specified within `TikZ`’s `tikzpicture` environment, like the small tree in the next box below. We will call such an instance of this environment a ‘`tikzpicture`’, pronounced as something like `/ˈtɪkziˌpɪktʃəɹ/` or `/ˈtɪkzɛdˌpɪktʃəɹ/`. This is a common count noun, pluralized as ‘`tikzpictures`’ with the usual English morphophonology. The example below also illustrates a side-by-side presentation which is used to save space for smaller examples.

```
\begin{tikzpicture}[parent anchor=south,  
  align=center, level distance=2em,  
  anchor=north, sibling distance=4em,  
  child anchor=north]  
\node (top) {XP}  
  child {node {X}}  
  child {node {YP}  
    child {node {Y}}  
    child {node {ZP}}};  
\end{tikzpicture}
```



A `tikzpicture` will usually have an optional argument containing some *key=value* pairs separated by commas. The syntax of these pairs is defined by TikZ's associated `pgfkeys` package. Both *key* and *value* can contain spaces. If a *value* contains commas then it must be enclosed in braces like `{this, example}`. A *value* can be recursive as `{this={example, shows}}`, but this is relatively uncommon even for complex diagrams. Recursive *values* are quite common in `.style={...}` specifications, however.

Long lists of TikZ options are usually necessary but they can be hard to read. In normal usage these options don't have to fit into any particular space constraints because they exist only in the code of a document and are not visible in the output. In this document the options are often compressed down to reduce space usage, but it is usually better in practice to use lots of whitespace so they are easier to quickly scan for differences.

```
\begin{tikzpicture}[%←defensive comment
  baseline=(top.base), % This Long
  align=center,        % column of
  level distance=2em,  % options
  sibling distance=4em, % is easier
  anchor=north,        % to scan
  parent anchor=south, % quickly,
  child anchor=north] % see?
...
\end{tikzpicture}
```

The defensive commenting above is a technique to prevent  $\text{T}_\text{E}\text{X}$  from interpreting a line break as meaningful. Defensive commenting is not actually necessary in many cases, particularly with TikZ which has a sophisticated parser that handles whitespace reliably. But plain  $\text{T}_\text{E}\text{X}$  programs as well as a lot of  $\text{L}_\text{A}\text{T}_\text{E}\text{X}$  code can interpret line breaks as additional space. This is often the source of obscure `over-full hbox` warnings and other subtle problems, and so defensive commenting can be mandatory in such contexts. But too much defensive commenting can be distracting and hard to read, so it should be used judiciously.

A similar technique is defensive bracing where a square bracket appears with a prothetic pair of braces as shown below in some ExPex code. Sometimes a square bracket or other problematic character is instead surrounded parenthetically by braces like `{[ ]}`. In the example below the defensive bracing prevents the initial square bracket from being interpreted as the start of an option list for the `\ex` command.

```
\ex%←defensive comment
%↓defensive bracing
{ }[\textsubscript{CP} [\textsubscript{DP} This ] is
  [\textsubscript{DP} a
    [\textsubscript{NP} bracketed example sentence ]]]
\ex
```

---

(1) `[CP [DP This ] is [DP a [NP bracketed example sentence ]]]`

Note the indentation of bracketed chunks within the example above. The indentation is deeper

the more levels of bracketing are in place. This indentation is purely cosmetic, intended as an aid for human readers of the source code. Like many other programming languages (e.g. C, Lisp, but contra Python)  $\text{\TeX}$  does not care about indentation, so working out good cosmetic indentation is purely for human benefit. Consistent indentation can save countless hours of lost time trying to debug unbalanced parentheses, brackets, and braces, as well as making both code and data (both text and examples) much easier to modify over time. But indentation can also be misleading if it does not actually match the structure. Effort put into good indentation always pays off in the long run.

[[FIXME: Explain 'baseline'. Figure showing different horizontal and vertical measurements of a glyph. Explain font-relative ex and em units versus absolute pt, in, and cm units.]]

[[FIXME: TikZ node diagram showing a rectangular node's points and associated angular degrees.]]

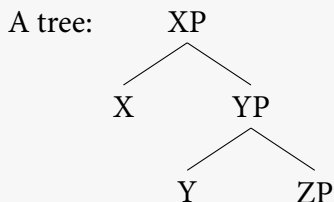
[[FIXME: Discuss xparse.]]

## 2 Trees

There are many linguistic tree drawing packages out there like `tikz-qtrees` and `forest`. These generally provide a familiar bracketed syntax for specifying structure as well as some options for customizing the appearance of trees. TikZ has its own syntax for trees which fits with its more general drawing syntax. The above mentioned tree drawing packages actually use TikZ's trees internally, adding parsers and default options. TikZ easily handles the combination of its tree syntax with other kinds of diagrams in the same `tikzpicture`, making things like arrow drawing, annotation, callouts, subsetting, and even magnification fairly painless. TikZ is much much more than a tree drawing program as attested by the 1000 page manual. Readers interested in going beyond the basic introduction here are encouraged to read the five tutorials at the beginning of the TikZ manual (pp. 28–89) as well as the guidelines on using graphics effectively (pp. 90–96).

In the introduction section there was a very simple tree. It is repeated here with some informative comments and a bit of textual context.

```
A tree:
\begin{tikzpicture}[
  baseline=(top.base), % alignment to baseline of the 'top' node
  align=center,        % centre contents of each node
  level distance=2em,  % 2em between vertical levels
  sibling distance=4em, % 4em between horizontal branches
  parent anchor=south, % connect to the south (bottom centre) of a parent node
  child anchor=north,  % connect to the north (top centre) of a child node
  anchor=north]        % make nodes 'hang' from their north point
% "\node" is TikZ shorthand for "\path node"
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}}};
\end{tikzpicture}
```



The syntax of the tree itself is fairly straightforward. A `\node` named `top` (names are arbitrary) contains the text `'XP'` and has two children. Each child has some content `{...}` which includes another node. The second child contains more children, so the `child {node ...}` structure can be recursive. All of this forms a single TikZ drawing path which ends with a semicolon.

Abstracting from this example, the template `\command ... ;` is a basic TikZ sentence. This `tikzpicture` contains one sentence, but more are possible. With one-sentence `tikzpictures` like this example, it is easy to forget the semicolon at the end. This is such a common mistake that TikZ often suggests this when it encounters an error. Aside from missing semicolons, the other most common mistakes

with TikZ are unbalanced braces {...} and unbalanced brackets [...]. To alleviate this, use an editor like T<sub>E</sub>XShop, T<sub>E</sub>XWorks, vim, or Emacs that highlights opening and closing delimiters.

Most of the magic of TikZ trees is in the options. The `baseline=(top.base)` option above says to align the base point of the node named `top` to the baseline of the surrounding text. This alignment to the surrounding baseline can be seen in that the node ‘XP’ has the bottom of the ‘X’ (its baseline) along the same horizontal line as the bottom of the text ‘A tree:’ that precedes the `tikzpicture`. The reference (`top.base`) selects the point called `base` that belongs to the node named `top`. For more details about nodes and their points see the TikZ manual part III §17 “Nodes and edges”, especially III§17.5 “Positioning Nodes” which shows most of a node’s points.

The `align=center` option says to align the text of each node to the centre of the node. Then `level distance` and `sibling distance` specify the space between vertical levels and horizontal sibling branches, calculated from the anchor point of each node. The `parent anchor` and `child anchor` indicate which points of each node should be used for connecting branches.

The `anchor=north` is often confusing to people learning about TikZ trees. It has the effect of making each node hang from its north point, the top centre of the node. To clarify this, we draw the bounding box of every node by adding the option `every node/.style={draw}`.

A tree:

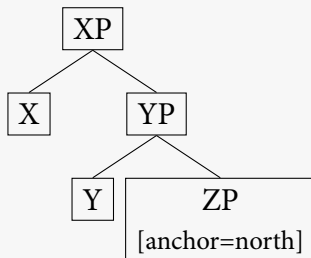
```
\begin{tikzpicture}[every node/.style={draw}, baseline=(top.base),
  level distance=2em, sibling distance=4em, align=center,
  parent anchor=south, child anchor=north, anchor=north]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP\\footnotesize}[anchor=north]}};
\end{tikzpicture}
```

%

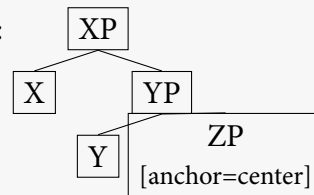
Another tree:

```
\begin{tikzpicture}[every node/.style={draw}, baseline=(top.base),
  level distance=2em, sibling distance=4em, align=center,
  parent anchor=south, child anchor=north, anchor=center] % ← Look here
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP\\footnotesize}[anchor=center]}};
\end{tikzpicture}
```

A tree:



Another tree:



In the second tree above, `anchor=center` makes each level vertically align to the others along its center point instead of its north point. The vertical centre of the ZP node is between the ‘ZP’ line and the ‘[`anchor=center`]’ line, whereas the vertical centre of the Y node is just about where the two arms of the ‘Y’ branch apart. The level distance is still 2em but now this is calculated between the centres of the nodes on each level rather than their tops and bottoms, so the result is that the visual distance between branch levels is smaller.

## 2.1 Styles and modifying option values

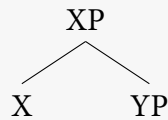
We have been repeating the same options with each `tikzpicture`. It’s easy enough to cut and paste them from one to the next, but it’s tiresome to later read through long lists of options when we need to ensure that they are identical or maybe modify some of them. To relieve the repetition we can define a TikZ style that contains all the relevant options. We then refer to this new style as an option, and the style’s defined list of options will apply wherever it is used.

### Preamble

```
\tikzset{mytree/.style={baseline=(top.base),
                        level distance=2em, sibling distance=4em, align=center,
                        parent anchor=south, child anchor=north, anchor=north}}
```

### Document body

```
\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}};
\end{tikzpicture}
```



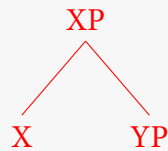
Although styles can be defined anywhere, if a style is going to be used across many `tikzpictures` it is best to define it in the preamble where it is easy to find. Styles can be redefined but this is bad practice because it is easily overlooked and the definition becomes context-dependent. Instead of modifying a style, define a new one with a similar name which includes the previous style. For details on how styles are defined and processed, see the TikZ manual VII§82.4.4 (p. 887) “Defining styles” in the context of key management.

### Preamble

```
\tikzset{myothertree/.style={mytree, level distance=3em, red}}
```

### Document body

```
\begin{tikzpicture}[myothertree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}};
\end{tikzpicture}
```

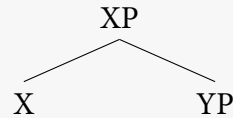


Our new `mytree` style can be used with additional options, even ones that override the style. In the next example below the option `sibling distance=6em` overrides the `sibling distance=4em`



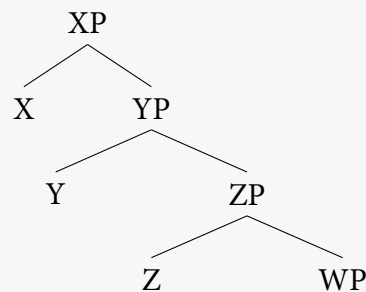
specified above in our `mytree` style definition. The last value assigned to an option takes precedence over any other values. Thus an option can be defined in one style, inherited by another, and then changed in the options list of a `tikzpicture`.

```
\begin{tikzpicture}[mytree,
  sibling distance=6em]
\node (top) {XP}
  child {node {X}}
  child {node {YP}};
\end{tikzpicture}
```



Most option values can also be changed inside of a `child` specification, in which case the new value scopes over everything within (below) that child. The `tikzpicture` below has the `sibling distance` of `4em` supplied by the `mytree` style, but then in the child containing the `YP` node there is a new `[sibling distance=6em]` option specification. Since this is after the `YP` node itself the new value applies only to the children `Y` and `ZP`. Then it recursively applies to all nodes within (below) those nodes.

```
\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    [sibling distance=6em]
    child {node {Y}}
    child {node {ZP}
      child {node {Z}}
      child {node {WP}}}};
\end{tikzpicture}
```



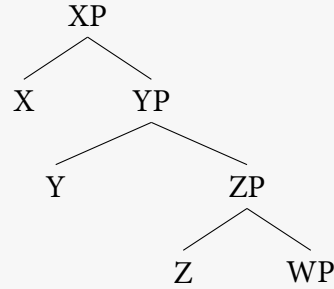
It's not possible to restrict the scope of an option to one level without affecting the levels below. If we want only one level to have a different `sibling distance` then we must respecify the original `sibling distance` for the lower branches. In this case we first say `[sibling distance=6em]` in the `YP` node's definition and then add a `[sibling distance=4em]` option below the `ZP` node which applies to the `Z` and `WP` children. The `4em` value is the same value specified in our `mytree` style so the lower branches will look the same as the ones at the top.<sup>1</sup>

<sup>1</sup>It's technically possible to recover the previous value of `sibling distance`, such as by saving it in a variable before modification, by programmatically walking through the `mytree` options list to find its original definition, or by defining a `\TeX` length register that contains the original value. This is rarely worth the effort, however.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    [sibling distance=6em]
    child {node {Y}}
    child {node {ZP}
      [sibling distance=4em]
      child {node {Z}}
      child {node {WP}}}}};
\end{tikzpicture}

```

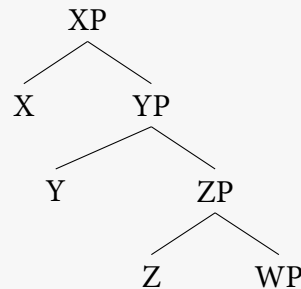


Sometimes we want one branch to be spaced differently from others on the same level. This is done by placing the `sibling distance` option in a single child rather than over several children.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child [sibling distance=6em]
      {node {Y}}
    child {node {ZP}
      child {node {Z}}
      child {node {WP}}}}};
\end{tikzpicture}

```

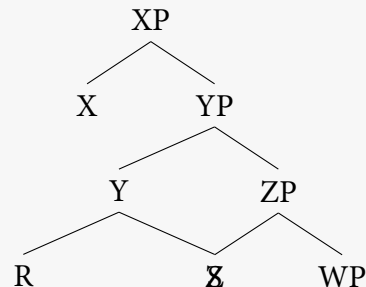


Notice here that the option only scopes over the Y node. If this node had further children they would be affected by the option, but since the ZP and its daughters are not part of the Y node they are unaffected. The next example adds some nodes below Y to show the scope effect. The R and S nodes are each 6 em apart from the centre of the Y node just like the Y node is 6 em from the centre of the YP node.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child [sibling distance=6em]
      {node {Y}
        child {node {R}}
        child {node {S}}}
    child {node {ZP}
      child {node {Z}}
      child {node {WP}}}}};
\end{tikzpicture}

```



This also illustrates that TikZ doesn't care whether your nodes overlap. It puts everything exactly where you tell it to and makes no adjustments behind your back, so finding a good appearance is entirely up to you. TikZ thus follows the YAFIYGI /'jæfi,jigi/ 'You Asked For It, You Got It' approach

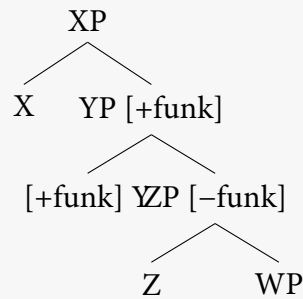
which contrasts with the DWIM /dwim/ ‘Do What I Mean’ approach where software is designed to guess what you ‘really mean’ or ‘don’t actually want’ (compare autocorrect).

TikZ does support algorithmic graph drawing which can do automatic balancing of node layout (see TikZ manual IV§26 “Introduction to Algorithmic Graph Drawing”). But the algorithms for graph balancing are typically designed for computer science, mathematics, and biology so they tend to not match the aesthetic expectations for trees in linguistics. A tree with an unusual aesthetic can be distractingly worse than no tree at all. Furthermore, a well designed tree can subtly emphasize particular theoretical or analytical claims which and this is all but impossible to automate programmatically. Manually adjusted trees are the thus best approach for most linguists, and people who really need automagical trees – e.g. computational linguists – should know enough about graph theory to read the relevant literature and hack the algorithms for their needs.

## 2.2 Hanging annotations

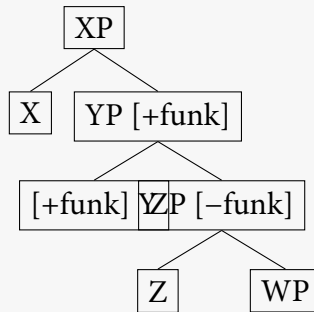
Trees occasionally need stuff hanging around their nodes that aren’t included in calculating the actual size of the node. For example, we might want to indicate a feature like ‘[+funkt]’ to show that a particular node has got the funk. The obvious approach of sticking it in a node looks like crap.

```
\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP [+funkt]}}
    child {node {[+funkt] Y}}
    child {node {ZP [-funkt]}}
      child {node {Z}}
      child {node {WP}}};
\end{tikzpicture}
```



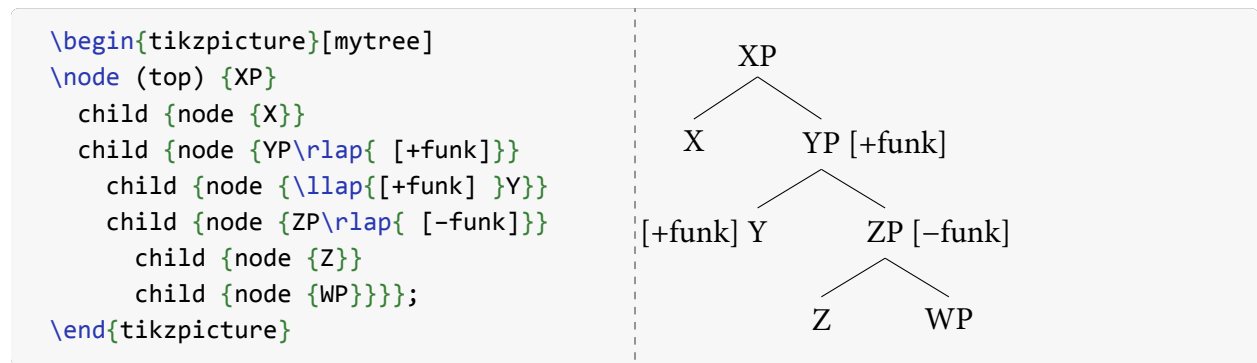
We can see why if we draw the bounding boxes of the nodes. They have grown to include the text we put in them which is not what we really wanted though it is what we said to do.

```
\begin{tikzpicture}[mytree,
  every node/.style={draw}]
\node (top) {XP}
  child {node {X}}
  child {node {YP [+funkt]}}
    child {node {[+funkt] Y}}
    child {node {ZP [-funkt]}}
      child {node {Z}}
      child {node {WP}}};
\end{tikzpicture}
```



To fix this we’ll use some plain TeX commands. The `\rlap{...}` command takes an argument which is printed out in a box of zero width. Saying e.g. `foo\rlap{xxx}bar` will print out ‘xxx’ on to the right of ‘foo’ and on top of ‘bar’ like so: `fooxxxbar`. The `\llap{...}` command similarly prints out its argument in a box of zero width, but this progresses leftward of the point where the command

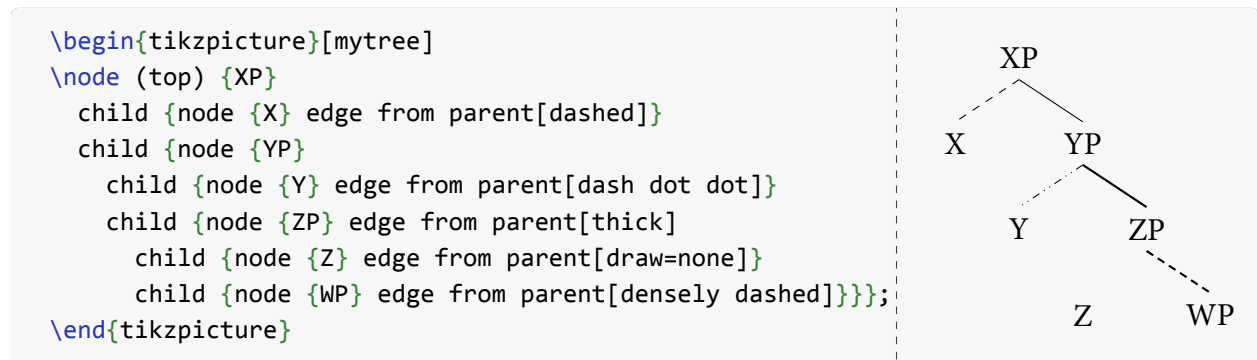
occurs, so `foo\llap{xxx}bar` looks like: `foo`bar. Since TeX puts these in zero-width boxes, when TikZ calculates the sizes of nodes from the text inside them its calculations aren't affected by them.



For more details on `\rlap{...}`, `\llap{...}`, and other related commands, see Alexander Perlis's article "A complement to `\smash`, `\llap`, and `\rlap`" in *TUGboat* vol. 22 no. 4 (2001).<sup>2</sup>

## 2.3 Fancy branches

Sometimes vanilla trees are just too bland and we want to spice things up with some fancy lines. This is a fairly easy task for TikZ which has lots of support for different kinds of lines (see TikZ manual III§15.3 "Drawing a Path"). To modify a particular line of a tree branch (what TikZ calls an 'edge') we add an `edge from parent` option which is some style that TikZ supports for lines.



The first branch from XP to X is dashed which will draw a 50% dash instead of a solid line. The branch from YP to Y is `dash dot dot` which draws a dash followed by two dots and then repeats. The branch from YP to ZP is `thick` which makes the line 0.8pt thick as compared to the default value of 0.4pt (same as `thin`). There is a `line width=...` option which can be used to specify an arbitrary thickness. Notice that the `thick` option is applied not only to the YP to ZP branch but also to the next lower ZP to WP branch since the ZP node also scopes over the WP node. The Z node's branch would also be affected except that it is invisible with `draw=none`.

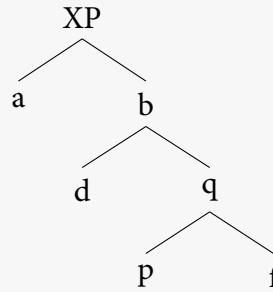
## 2.4 Adjusting node sizes

Our trees so far have always had only uppercase letters in them. Most fonts have all uppercase letters designed to be the same size. But this is not necessarily true for other letters and symbols. Note what

<sup>2</sup>Online at <http://www.tug.org/TUGboat/tb22-4/tb72perlS.pdf>

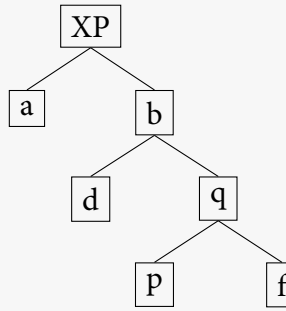
happens to ‘a’ versus ‘b’ below, as well as ‘d’ vs. ‘q’ and ‘p’ vs. ‘f’.

```
\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {a}}
  child {node {b}}
    child {node {d}}
    child {node {q}}
      child {node {p}}
      child {node {f}}}};
\end{tikzpicture}
```



To make this more obvious, we’ll draw the bounding boxes with every `node/.style={draw}`. Some nodes like ‘d’ and ‘q’ are about the same size, but clearly ‘a’ and ‘b’ are not.

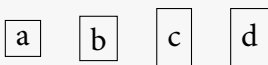
```
\begin{tikzpicture}[mytree,
  every node/.style={draw}]
\node (top) {XP}
  child {node {a}}
  child {node {b}}
    child {node {d}}
    child {node {q}}
      child {node {p}}
      child {node {f}}}};
\end{tikzpicture}
```



Even for the nodes of the same size like ‘p’ and ‘f’ there is still something wrong: the baselines of each node are different. The baseline of ‘p’ is just below the loop of the letter, with its descender extending below the baseline. In contrast, the baseline of ‘f’ is at the bottom of the vertical stem and there is no descender. This can trick us into misreading ‘p’ as ‘P’ among other problems.

We can fix this by adding a `\strut` to each node. Struts are typesetting elements with no horizontal size but with a fixed vertical size, ensuring that a box containing the strut has a minimum height and depth. The `\strut` command inserts a zero width box whose height from the baseline is 70% of the `\baselineskip` value (the height from one baseline to the one on the next line of text) and whose depth below the baseline is 30% of the `\baselineskip`. The example below shows four nodes in a row, the first two without a `\strut` and the second two with a `\strut` added after of each node’s text. The boxes of nodes ‘c’ and ‘d’ are the same size because both nodes expand to completely contain the invisible `\strut`.

```
\begin{tikzpicture}[every node/.style={draw}]
\path node {a} -- (1,0) node {b} -- (2,0) node {c\strut} -- (3,0) node {d\strut};
\end{tikzpicture}
```



Adding a `\strut` to one or two misaligned nodes is a quick and effective solution. But the tree above has more than just one or two problems. Adding a `\strut` to every node in a tree is annoying

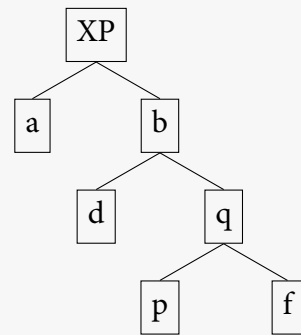
and error prone: we may have quite a few nodes to amend and each time we may accidentally say `\sturt` or the like. An alternative is to explicitly tell TikZ the size that each node should be. The `text height` option defines the distance from the baseline to the top of the node, and the `text depth` likewise gives the distance from the baseline to the bottom of the node. There is also a `text width` for the horizontal size which we don't need here.

```
\begin{tikzpicture}[every node/.style={draw, text height=0.7em, text depth=0.3em}]
\path node {a} -- (1,0) node {b} -- (2,0) node {c} -- (3,0) node {d};
\end{tikzpicture}
```



Applying this to our previous tree, we see the misalignment problem go away.

```
\begin{tikzpicture}[mytree,
                    every node/.style={draw,
                                        text height=0.7em,
                                        text depth=0.3em}]
\node (top) {XP}
  child {node {a}}
  child {node {b}
    child {node {d}}
    child {node {q}
      child {node {p}}
      child {node {f}}}}};
\end{tikzpicture}
```



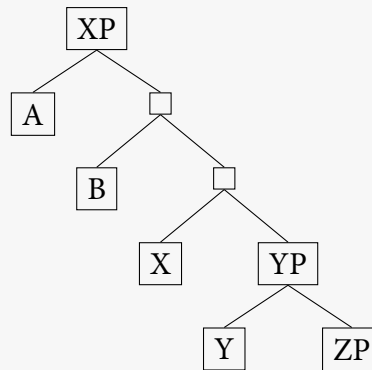
The typical syntactic tree has mostly uppercase letters, with the notable exceptions of *v* and *n* (but not *vP* and *nP*). As such, it's often unnecessary to force the node size with things like `text height` and instead the occasional `\strut` is all that's really needed. Also a forced node size may be too small for an unusual node with a diacritic like 'Å'. So be sure you want to force the node size, rather than blindly doing so in every tree.

All the preceding examples of trees have had a node at every level along the spine of the tree. But people often use trees which have a straight line for the spine so that the nodes have no content. Drawing this kind of tree in TikZ is not straightforward but it is not difficult. If we draw a tree with no node contents the appearance is not what we want.

```

\begin{tikzpicture}[mytree,
  every node/.style={draw}]
\node (top) {XP}
  child {node {A}}
  child {node {}
    child {node {B}}
    child {node {}
      child {node {X}}
      child {node {YP}
        child {node {Y}}
        child {node {ZP}}}}}}};
\end{tikzpicture}

```

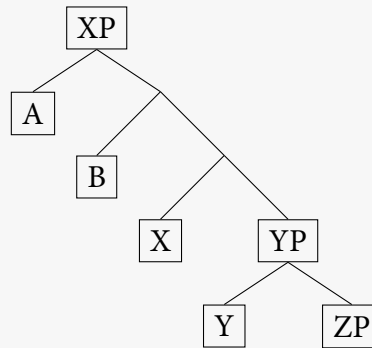


The empty nodes clearly take up some space. We can replace them with coordinate nodes which truly have zero height, width, and depth.

```

\begin{tikzpicture}[mytree,
  every node/.style={draw}]
\node (top) {XP}
  child {node {A}}
  child {coordinate
    child {node {B}}
    child {coordinate
      child {node {X}}
      child {node {YP}
        child {node {Y}}
        child {node {ZP}}}}}}};
\end{tikzpicture}

```

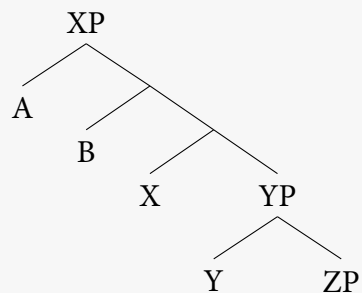


The level distance is calculated from the south of a parent node to the north of a child node. Since the coordinate nodes have zero size their south and north points are identical. This causes the level distance to increase because the difference between north and south is now absent. We can adjust the level distance to fix this, setting it back when we switch to YP.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {A}}
  child {coordinate
    [level distance=1.375em]
    child {node {B}}
    child {coordinate
      child {node {X}}
      child {node {YP}
        [level distance=2em]
        child {node {Y}}
        child {node {ZP}}}}}}};
\end{tikzpicture}

```

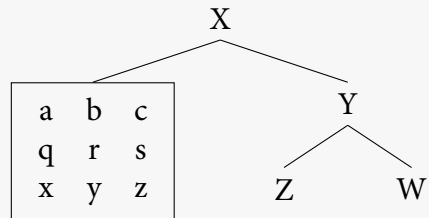


The problem with this solution is that it requires some tweaking of level distances. Consistently using font-relative measurements like `em` and `ex` alleviates some of the problem, but we still have to figure out the difference of distance between the text and coordinate nodes. This solution works well enough for occasional use, but automating the calculation of height differences between nodes and coordinate nodes would be ideal. I leave this issue for future research.

## 2.5 Node content

Nodes of trees generally contain text. So far we have seen very simple examples of text, just short symbols like ‘`XP`’. But nodes can actually contain arbitrary  $\text{\LaTeX}$  code. The following example shows a node that contains a small  $\text{\LaTeX}$  table using the `tabular` environment.

```
\begin{tikzpicture}[mytree]
\node (top) {X}
[sibling distance=8em]
child {node [draw]
{\begin{tabular}{ccc}
a & b & c\\
q & r & s\\
x & y & z\\
\end{tabular}}}
child {node {Y}
[sibling distance=4em]
child {node {Z}}
child {node {W}}};
\end{tikzpicture}
```



Putting a  $\text{\LaTeX}$  tabular environment in a node is obviously possible but there are better ways to achieve the same effect. In section 3 I discuss `TikZ`'s `matrix` node which can be used to construct grid-based alignments such as tables. The example above merely serves to illustrate that `TikZ` nodes can contain relatively complex material, including tables and mathematics.

Some syntactic theories like HPSG and LFG make use of trees that have attribute-value matrices as nodes. There is an `avm` package for typesetting attribute value matrices which we can load in the preamble. The `avm` package is fairly old and makes some assumptions that are incorrect for modern  $\text{\LaTeX}$ , in particular using the deprecated `\sc` instead of modern `\scshape`. We adjust these using the package's `\avmfont` and `\avmvalfont` commands which take the name of a font switching command (i.e. `\scshape` and not `\textsc`).

### Preamble

```
\usepackage{avm}
\avmfont{\scshape}
\avmvalfont{\normalfont}
```

With this code in our preamble we can now stick a simple attribute-value matrix in one of our nodes. The `avm` package actually implements attribute-value matrices as small tables, so it is prone to some of the same problems as with putting `tabular` environments in nodes. In particular, adding branches below an `avm` node requires some fiddling with level distances.

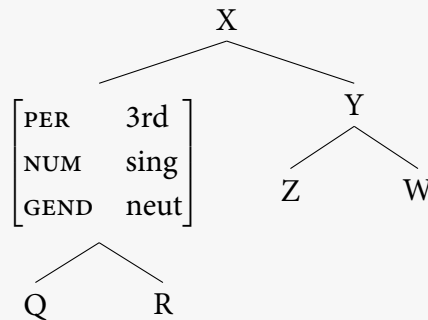


## Document body

```

\begin{tikzpicture}[mytree]
\node (top) {X}
[sibling distance=8em]
child {node
  {\begin{avm}
  \[ per & 3rd \\
  num & sing \\
  gend & neut \]
  \end{avm}}
[sibling distance=4em,
level distance=3.75em]
child {node {Q}}
child {node {R}}}
child {node {Y}
[sibling distance=4em]
child {node {Z}}
child {node {W}}};
\end{tikzpicture}

```



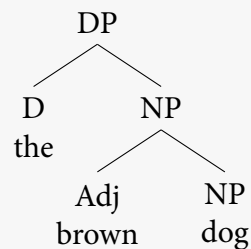
As noted earlier, TikZ has a matrix node type which can be used to make these kinds of structures more cleanly. See section 3 for details.

We haven't yet seen a tree with actual lexical content because so far all the trees have been illustrative nonsense. Trees in the Minimalist Program usually have their lexical content immediately beneath the syntactic category labels. The easiest way to do this is to put a line break in the node contents, either using `\newline` or more commonly `\\`.

```

\begin{tikzpicture}[mytree]
\node (top) {DP}
child {node {D\\the}}
child {node {NP}
  child {node {Adj\\brown}}
  child {node {NP\\dog}}};
\end{tikzpicture}

```

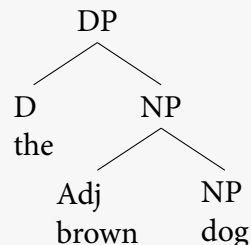


Using the `align` option we can make the text left or right aligned instead of centred, although there is no real reason to do this for normal trees.

```

\begin{tikzpicture}[mytree,
align=left]
\node (top) {DP}
child {node {D\\the}}
child {node {NP}
  child {node {Adj\\brown}}
  child {node {NP\\dog}}};
\end{tikzpicture}

```

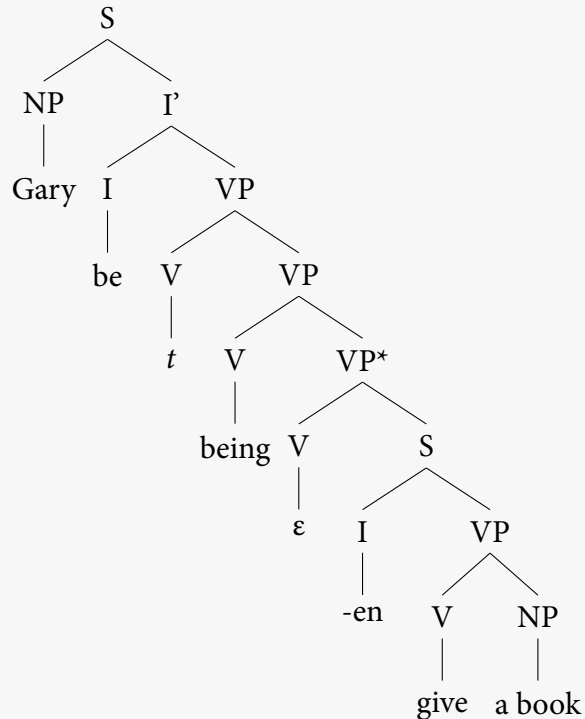


In Government and Binding and older theories the usual practice is to place the lexical content in a unary branch below the syntactic category label. This practice reflects a different approach to lexical insertion, but it also makes trees quite a bit larger. The following example is duplicated from Baker, Johnson, and Roberts’s “Passive arguments raised” (1989, *Ling. Inq.* vol. 20 no. 2).

```

\begin{tikzpicture}[mytree]
\node (top) {S}
  child {node {NP}
    child {node {Gary}}}
  child {node {I'}
    child {node {I}
      child {node {be}}}
    child {node {VP}
      child {node {V}
        child {node {\textit{t}}}}
      child {node {VP}
        child {node {V}
          child {node {being}}}
        child {node {VP*}
          child {node {V}
            child {node {\epsilon}}}
          child {node {S}
            child {node {I}
              child {node {-en}}}
            child {node {VP}
              [sibling distance=3em]
              child {node {V}
                child {node {give}}}
              child {node {NP}
                child {node
                  {a book}}}}}}}}}}}}};
\end{tikzpicture}

```



The tree source code is a bit more complex because it has those extra unary branches. But other than being difficult to read, this kind of branching is not any harder to implement than previous trees.

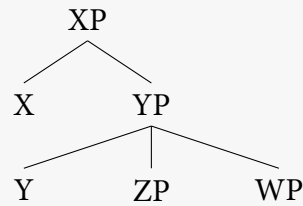
## 2.6 Beyond binary: $n$ -ary branches and triangles

Syntax trees have been largely binary since the 1980s, but there are still occasional situations where more than two branches are necessary. This is trivial in TikZ: simply add more children.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}}
    child {node {WP}}};
\end{tikzpicture}

```

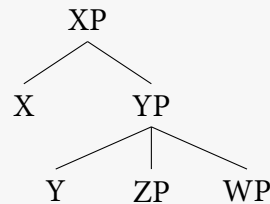


The lowest branches are spaced very far apart. This is because the sibling distance (defined as 4 em in `mytree`) is applied to each pair of siblings, not just to the leftmost and rightmost. Consequently the Y node is 4 em from the ZP node and the ZP node is 4 em from the WP node. To fix this we specify a smaller sibling distance like 3 em.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    [sibling distance=3em]
    child {node {Y}}
    child {node {ZP}}
    child {node {WP}}};
\end{tikzpicture}

```



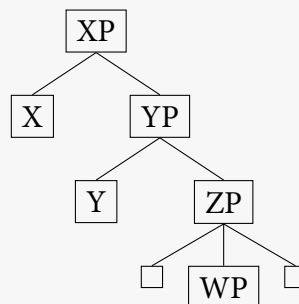
The sibling distance needs to be even smaller for quaternary branches, quinary, etc. But at some point the sibling distance will be smaller than the size of the nodes, in which case the width of the whole set of branches must inevitably be larger. Always plan to spend some quality time adjusting the sibling distance if you need  $n$ -ary branching beyond  $n = 2$ .

Ternary branches are useful for making triangles. TikZ doesn't have a built-in way to form a triangle for part of a tree, but we can make one by some extra drawing. First let's draw a ternary branch that approximates our triangle.

```

\begin{tikzpicture}[mytree,
  every node/.style={draw}]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}
      [sibling distance=2.25em]
      child {node {}}
      child {node {WP}}
      child {node {}}}}};
\end{tikzpicture}

```

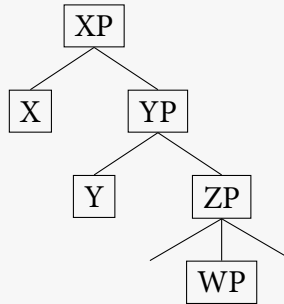


We've used two empty nodes, but we really should use coordinate nodes which have zero size and no content. They still count for position and distance however.

```

\begin{tikzpicture}[mytree,
  every node/.style={draw}]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}
      [sibling distance=2.25em]
      child {coordinate}
      child {node {WP}}
      child {coordinate}}};
\end{tikzpicture}

```

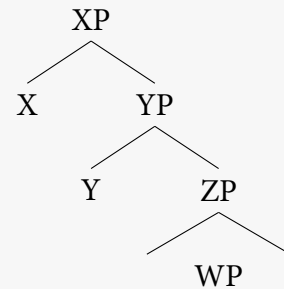


Now we can make the centre branch invisible by using an edge from parent specification.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}
      [sibling distance=2.25em]
      child {coordinate}
      child {node {WP} edge from parent[draw=none]}
      child {coordinate}}};
\end{tikzpicture}

```

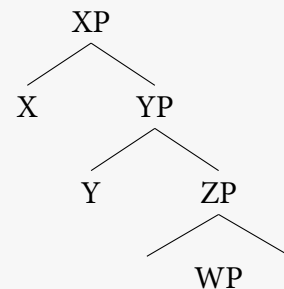


To draw a line across the bottom of that triangle we need some way to refer to each of the coordinate nodes at the end of the branches. We give them names using the (...) syntax.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}
      [sibling distance=2.25em]
      child {coordinate (trileft)}
      child {node {WP} edge from parent[draw=none]}
      child {coordinate (triright)}}};
\end{tikzpicture}

```

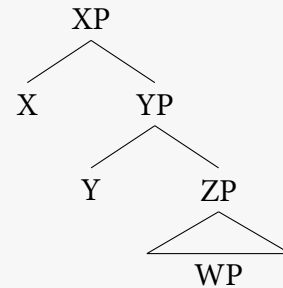


Finally we add an additional drawing statement which draws a path between the two nodes.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}
      [sibling distance=2.25em]
      child {coordinate (trileft)}}
      child {node {WP} edge from parent[draw=none]}
      child {coordinate (triright)}}};
\draw (trileft) -- (triright);
\end{tikzpicture}

```

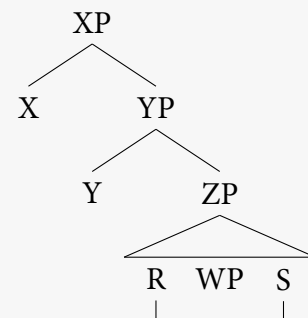


Triangles can be more than just ternary branches. Sometimes we want a couple of separate bits underneath the triangle, especially when we want to refer to some parts below the triangle.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {X}}
  child {node {YP}
    child {node {Y}}
    child {node {ZP}
      [sibling distance=1em]
      child {coordinate (trileft)}}
      child {node (r) {R} edge from parent[draw=none]}
      child {coordinate edge from parent[draw=none]}
      child {node {WP} edge from parent[draw=none]}
      child {coordinate edge from parent[draw=none]}
      child {node (s) {S} edge from parent[draw=none]}
      child {coordinate (triright)}}};
\draw (trileft) -- (triright);
\draw (r.south) -- ++(south:1.5ex) -| (s.south);
\end{tikzpicture}

```



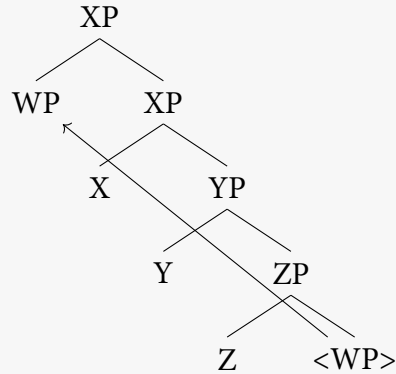
## 2.7 Arrows, phases, and other lines

Trees can be accompanied by other drawings. Probably the most interesting from a syntactician's perspective is the drawing of movement arrows. It's very simple to add a straight line on a tree as shown in the following example.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
% This draws an arrow from wp1 to wp2.
\draw[->] (wp1) -- (wp2);
\end{tikzpicture}

```

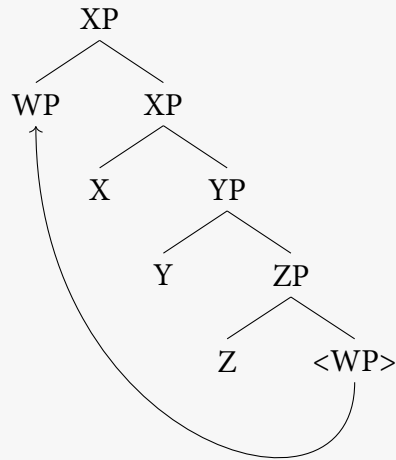


The two nodes containing ‘WP’ are each named (wp1) and (wp2). After the tree there is an additional `\draw` command that draws a line from wp1 to wp2 with an arrowhead at the end. This result is not what any sane syntactician would want for a movement arrow. Movement is usually indicated with a smooth curve from the source to the destination. This can be done in TikZ with a Bézier curve (see TikZ manual part III §14.3 “The Curve-To Operation”) using `.. controls A| and \texttt{\textit{B}} ..|`.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
\draw[->] (wp1.south)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2.south);
\end{tikzpicture}

```



this is some text below the diagram

this is some text below the diagram

Bézier curves have a start point, an end point, and two control points. The control points are positioned relative to the start and end points and cause the line to bend in some direction. The coordinate specifications `+(\theta:r)` for the control points use the polar coordinate system: the first component  $\theta$  is the radial angle and the second component  $r$  is the distance from the origin. The value `+(south:5em)` for the first control point says that this point is on a radial of  $270^\circ$  (`south =  $270^\circ$ , east =  $0^\circ$` ) at a distance of 5 em from the start point wp1. This control point pulls the line downward causing it to bend. Similarly, the second control point is south of wp2 at 10 em, pulling the end of the line a greater amount than at the start. The result is a smooth curve that arcs down

from the bottom of wp1 and swings up to the bottom of wp2.

There is lots of vertical space between the bottom of the tree and the following text. This is because the line and its control points add to the size of the diagram just like the tree does. We fix this with the `overlay` option which, when added to a path, says that the path should be ignored for calculating the bounding box. This option is meant for overlaying diagrams on top of ordinary text (see sec. 4), but it works just as well for our purpose of not changing the bounding box. To make the effect clearer we draw a box around the entirety of the `tikzpicture` with the `framed` option of the `backgrounds` library that we add to the preamble (see TikZ manual V§43 “Background Library”).

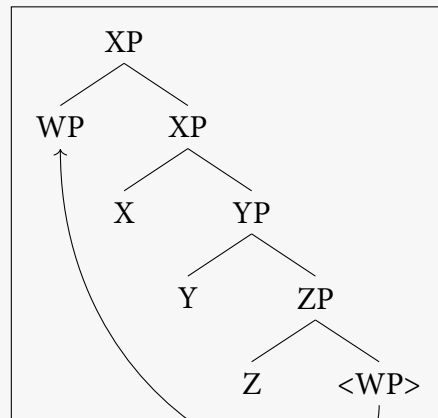
#### Preamble

```
\usetikzlibrary{backgrounds}
```

#### Document body

```
\begin{tikzpicture}[mytree, framed]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
\draw[->, overlay] (wp1)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2);
\end{tikzpicture}
```

this is some text below the diagram



this is some text below the diagram

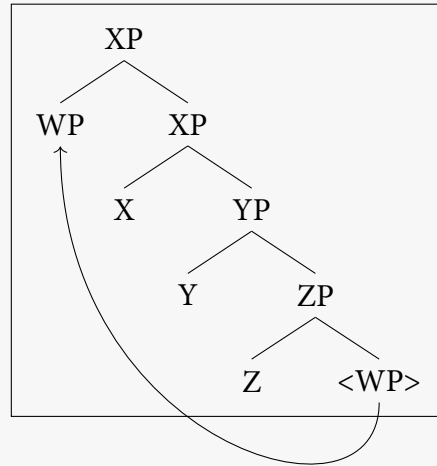
Now the text ‘this is some text below the diagram’ now overlaps with the movement arrow. We can fix this one of two ways: (i) add vertical space with `\vspace` after the `tikzpicture` or (ii) insert something invisible inside the `tikzpicture`. The first solution is simple as shown below.

```

\begin{tikzpicture}[mytree, framed]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
\draw[->, overlay] (wp1)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2);
\end{tikzpicture}
\vspace{2\baselineskip}

```

this is some text below the diagram



this is some text below the diagram

Our first solution is not all that great because it violates the containment of `tikzpictures`. The `\vspace` has to be added after the `\end{tikzpicture}`, and so we now must keep track of both the `tikzpicture` environment and this new appendix. It is very easy to miss this appendix, especially when copying and pasting, and it also means the diagram is no longer just one big box for  $\LaTeX$ .

The other solution modifies the `tikzpicture` itself. We can either add an invisible node to the tree or draw an independent invisible node relative to the bottom of the tree. The first technique is not too hard: define a coordinate node below the lowest node in the tree, adjust its `level` distance to some appropriate length, make it invisible with `[draw=none]` and also make the connecting edge invisible with `edge from parent [draw=none]`. We have done all of this before, so this is simply a novel combination of previous techniques.

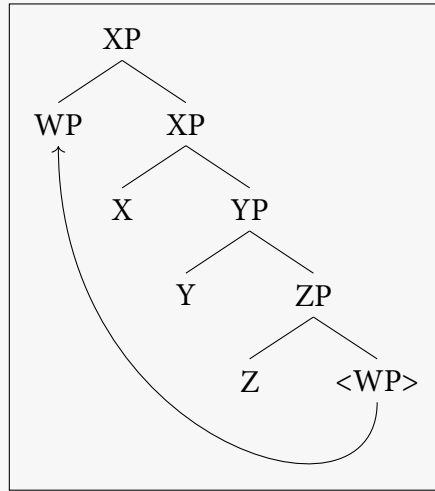


```

\begin{tikzpicture}[mytree, framed]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}
          [level distance=3em]
          child [draw=none]
            {coordinate
              edge from parent
              [draw=none]}]}]}]}
\draw[->, overlay] (wp1)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2);
\end{tikzpicture}

```

this is some text below the diagram



this is some text below the diagram

This works quite well but it has a major drawback in that the tree code is now confusing. Instead of just specifying some tree structure and maybe a few positioning adjustments, we now have one branch with a bunch of configuration cruft to make it invisible. We could leave a comment for readers telling them that all that junk is to resize the bounding box, but a comment wouldn't make it any easier to understand.

We can avoid changing the tree's structure by adding a separate command to specify the node outside of the tree code. To do this we need to position our new node relative to some existing node in the tree, which means loading TikZ's positioning library.

#### Preamble

```
\usetikzlibrary{positioning}
```

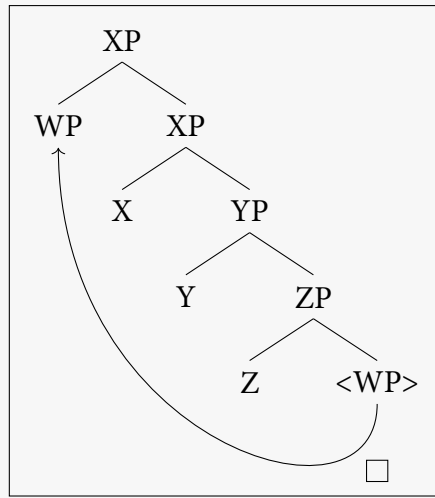
The `positioning` library (see TikZ manual III§17.5.3 (p. 229) “Advanced Placement Options”) provides some sophisticated options for placing nodes relative to other nodes. There is a set of keywords `above`, `below`, `left`, and `right` which take values that can have two parts. The first *shifting part* is an optional dimension or mathematical expression that moves the node in the appropriate direction. For the single keyword options the shifting part is a single value like `left=1em`, but double keywords take two values like `above left=1em and 2em`. The second *of part* contains a coordinate expression, usually the name of some other node, such as `of wp1`. In the following example we use `below=1.75em of wp1` to say that the node should be placed  $1\frac{3}{4}$  em below the `wp1` node. Adding the `draw` option to this node makes it visible so we can fine tune the positioning until we are satisfied.

```

\begin{tikzpicture}[mytree, framed]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
\draw[->, overlay] (wp1)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2);
\node [draw, below=1.75em of wp1] {};
\end{tikzpicture}

```

this is some text below the diagram



this is some text below the diagram

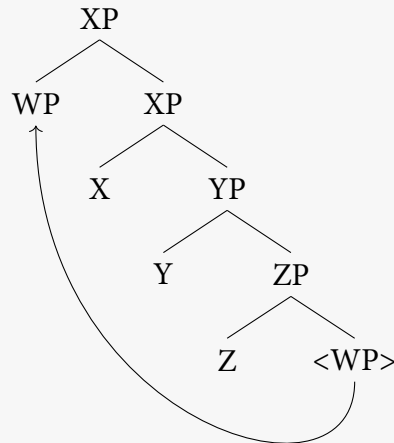
Now if we get rid of the framed and draw options our result looks like what we want.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
\draw[->, overlay] (wp1)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2);
\node [below=1.75em of wp1] {};
\end{tikzpicture}

```

this is some text below the diagram



this is some text below the diagram

There's one last thing to change here, specifically the arrowhead. The arrow uses  $\text{\LaTeX}$ 's default arrowhead shape which is hard to see and rather ugly. TikZ provides quite a few different arrowheads with lots of customization and if you are so inclined you can design your own from scratch (see TikZ manual III§16.5 (p. 202) "Reference: Arrow Tips"). To customize the arrows we need to load the `arrows.meta` library which provides a number of different configurable arrowheads. There's lots that can be done with this library so we're only going to scratch its surface.

Preamble

```
\usetikzlibrary{arrows.meta}
```

I prefer a particular arrowhead style called `Stealth` which has a sharp triangular point and a V-shaped haft:  $\rightarrow$ . I usually define a style called `exarrows` for this in the preamble.

#### Preamble

```
\tikzset{exarrows/.style={semithick,
                           arrows={-Stealth[scale=1, scale length=1,
                                              scale width=1]}}}

```

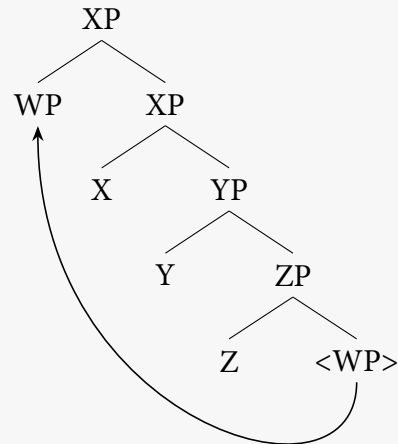
This style says to use a `semithick` (0.6 pt) line instead of the usual `thin` (0.4 pt). The text `arrows={-...}` says to put an arrowhead on the end of the line (compare the options `arrows={<-}` and `arrows={->}`). The arrowhead is called `Stealth` with three specified options for its graphical scale. All three scales are just 1 and so do not change the arrowhead's size, but by specifying them we ensure that the style will override any unexpected scale value supplied elsewhere, and they are also easily customized without having to look up the documentation.

To put this new style into use for our movement arrow we simply change the `\draw` command option of `->` to `exarrows`. Nothing else needs to be modified. Our final result is a very professional looking syntactic tree diagram.

```
\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (wp2) {WP}}
  child {node {XP}
    child {node {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}
        child {node {Z}}
        child {node (wp1) {<WP>}}}}}}};
\draw[exarrows, overlay] (wp1)
  .. controls +(south:5em)
  and +(south:10em) .. (wp2);
\node [below=1.75em of wp1] {};
\end{tikzpicture}

```

This illustrates raising of WP to XP.



This illustrates raising of WP to XP.

The `exarrows` style specifies a plain `semithick` (0.6 pt) line, but obviously other kinds of lines are possible like dashed lines, dotted lines, and double lines (see TikZ manual III§15.3 “Drawing a Path”). More elaborate kinds like zigzag and squiggly lines can be drawn using the `decorations` library (see TikZ manual V§48 “Decoration Library”). Below we load the library in the preamble and then draw a squiggly line by decorating our path with the `snake` decoration.

#### Preamble

```
\usetikzlibrary{decorations}
\usetikzlibrary{decorations.pathmorphing}

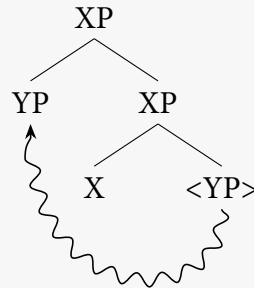
```

Document body

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (yp2) {YP}}
  child {node {XP}
    child {node {X}}
    child {node (yp1) {<YP>}}};
\draw[exarrows, decorate,
  decoration=snake] (yp1.south)
  .. controls +(south:3em)
  and +(south:5.75em) .. (yp2.south);
\end{tikzpicture}

```

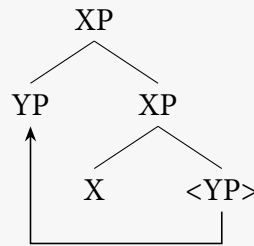


Some people use square lines for movement. These project directly down from the start point, make two 90° turns, and then project up to the end point. TikZ has a special kind of operation to create such combinations of horizontal and vertical lines with a single path (see TikZ manual III§14.2.2 “Horizontal and Vertical Lines”).

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (yp2) {YP}}
  child {node {XP}
    child {node {X}}
    child {node (yp1) {<YP>}}};
\draw[exarrows] (yp1.south)
  -- ++(south:1em)
  -| (yp2.south);
\end{tikzpicture}

```

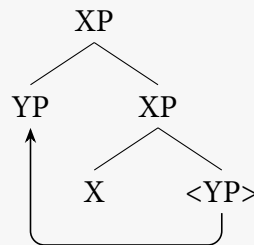


The `\draw` command here once again uses the `exarrows` style, with the path starting at the south point of the `yp1` node. The first specification in this path is `-- ++(south:1em)` which says to draw a straight line `--` to a point based on `yp1` which is south of it by 1 em. From this interstitial point the `-|` command says to draw a horizontal line to the `x` coordinate of `yp2` and then to draw a vertical line from there to `yp2.south`. One spiffy variation of this style uses rounded corners instead of the default sharp corners. The `rounded corners` option has a default value of 4pt.

```

\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node (yp2) {YP}}
  child {node {XP}
    child {node {X}}
    child {node (yp1) {<YP>}}};
\draw[exarrows, rounded corners=1ex]
  (yp1.south) -- ++(south:1em)
  -| (yp2.south);
\end{tikzpicture}

```

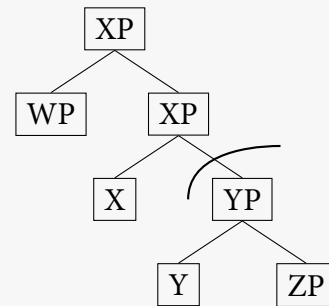


Phases are traditionally indicated with a curved line above the phase head that intersects a tree edge. The curved line is usually placed above the maximal projection of the head. To draw a phase in TikZ we have to specify the start and end points of the phase curve. The best technique is to define the phase line's points relative to other points in the tree so that the phase automatically adapts to changes in the `level distance` and `sibling distance` values as well as size changes from node contents. Below we draw a phase above the YP node on the spine of the tree. The code here requires the `positioning` library to be loaded in the preamble.

```

\begin{tikzpicture}[mytree,
  every node/.style={draw},
  inv/.style={overlay, coordinate}]
\node (top) {XP}
  child {node {WP}}
  child {node {XP}
    child {node {X}}
    child {node (yp) {YP}
      child {node {Y}}
      child {node {ZP}}}}};
\node [inv, left=0.75em of yp] (p1) {};
\node [inv, above right=1em and 0.25em of yp] (p2) {};
\draw [overlay, thick] (p1) to[out=90, in=180] (p2);
\end{tikzpicture}

```



First we draw two coordinate nodes to form the start and end points of the phase boundary. These could be specified in the same sentence as the `\draw` command but separating them out makes it easier to read. The `inv` option is just a style shorthand for `overlay` and `coordinate`, the former meaning that these nodes won't change the size of the diagram and the latter meaning they have zero size. The `left=0.75em of yp` says that the node `p1` is positioned  $\frac{3}{4}$  em to the left of the `yp` node, specifically the left edge of this node (the west point). The positioning option `above right=1em and 0.25em of yp` places the `p2` node 1 em above and  $\frac{1}{4}$  em to the right of the northeast corner of the `yp` node. Calculating the positions of these two nodes has to be done empirically, by repeatedly adjusting and visually inspecting the results.

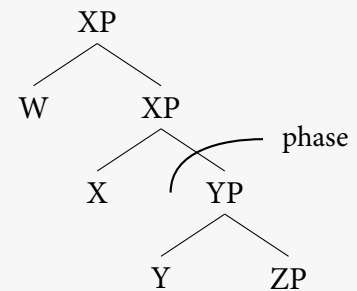
With the two coordinate nodes in place we can draw the actual line for the phase boundary. The `\draw` command has an `overlay` option to again prevent the line from changing the diagram's bounding box, and `thick` makes this line stand out at 0.8 point from the `thin` 0.4 point lines of the tree. The start of the line is the `p1` node and the end of the line is the `p2` node. The `to` path is a special user-defined path operation (see TikZ manual III§14.13 "The To Path Operation"). Among other options, `to`-paths can take out and `in` options that specify the departure and arrival angles of lines. Here we tell the path to depart from the `p1` node at an angle of  $90^\circ$  (the north angle) and to arrive at the `p2` node at an angle of  $180^\circ$  (the west angle).

It is occasionally necessary to add a label to the phase boundary. We can add one by creating a node with some text and positioning it relative to one of the phase boundary's nodes. The example below has a final `\node` which is placed  $\frac{1}{4}$  em to the right of `p2` and contains the text `\small phase` saying to print the word 'phase' in whatever  $\text{\LaTeX}$  has as the `\small` font size.

```

\begin{tikzpicture}[mytree,
  inv/.style={overlay, coordinate}]
\node (top) {XP}
  child {node {W}}
  child {node {XP}
    child {node {X}}
    child {node (yp) {YP}
      child {node {Y}}
      child {node {ZP}}}}};
\node [inv, left=0.75em of yp](p1){};
\node [inv, above right=1em and 0.25em of yp] (p2) {};
\draw [overlay, thick] (p1) to[out=90, in=180] (p2);
\node [right=0.25em of p2] {\small phase};
\end{tikzpicture}

```

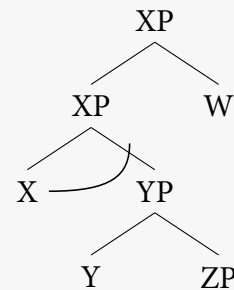


Relatively positioned nodes can also be used to construct complex movement arrows where there is more than one curve in a line. Complex curves are usually frowned upon for GB and Minimalist-style arrows, but LFG sometimes has spaghetti in c-structure to f-structure mapping diagrams. The technique illustrated here uses an intermediate node with separate two-point Bézier curves drawn to it from the start and end points. First we place the intermediate node and draw a curve to it.

```

\begin{tikzpicture}[mytree,
  inv/.style={overlay, coordinate}]
\node (top) {XP}
  child {node (xp1) {XP}
    child {node (x) {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}}}}};
  child {node (w) {W}}};
\node [inv, below right=0.5em and 0.25em of xp1] (i) {};
\draw [semithick] (x.east) .. controls +(east:2em)
  and +(south:1em) .. (i);
\end{tikzpicture}

```

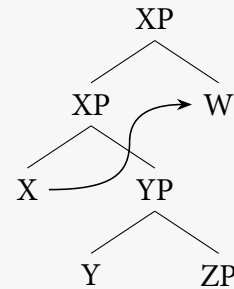


The second `\node` command (the one after the tree) adds an invisible node called `i` at a position  $\frac{1}{2}$  em below and  $\frac{1}{4}$  em to the right of the `xp1` node (the lower of the two XPs). This establishes the intermediate point for our complex curve. Note that even though the `inv` style specifies `coordinate` we still need to have an empty node content with the final `{}` because we have used `\node` and not `\path node`. The `\draw` then adds a line eastward from the `x` node and arriving at the south of the `i` node, with control points adjusting the curvature.

```

\begin{tikzpicture}[mytree,
  inv/.style={overlay, coordinate}]
\node (top) {XP}
  child {node (xp1) {XP}
    child {node (x) {X}}
    child {node {YP}
      child {node {Y}}
      child {node {ZP}}}}
  child {node (w) {W}};
\node [inv, below right=0.5em and 0.25em of xp1] (i) {};
\draw [semithick] (x.east) .. controls +(east:2em)
  and +(south:1em) .. (i);
\draw [exarrows] (i) .. controls +(north:1em)
  and +(west:1em) .. (w.west);
\end{tikzpicture}

```



The second `\draw` command completes our arrow, leaving from the intermediate point `i` and arriving at the west of `w`. Getting the curves to look just right is a matter of adjusting the radial distances and angles of the four control points. An alternative method for drawing complex curves is to use the `smooth` and `tension` options with the `plot` command.<sup>3</sup>

[[FIXME: Other lines like brackets and callouts.]]

[[FIXME: Fitting circles and ellipses around subparts of a tree.]]

## 2.8 TikZ and trees

The trees in this section have all been relatively simple. But more complex trees are generally constructed with the same techniques described above. There is much more that can be done with trees in TikZ and I encourage readers to explore the TikZ documentation as well as discussions about TikZ on StackExchange<sup>4</sup> and the wide variety of examples on T<sub>E</sub>Xample.net.<sup>5</sup>

As a final arboreal illustration, the following two pages provide a moderately complex example of a Christmas tree. Other than the colours, every technique used in this diagram has been presented in the preceding discussion.

<sup>3</sup>See TikZ manual III§22.3 “Plotting Points Given Inline” as well as <http://tex.stackexchange.com/a/33610/2474>.

<sup>4</sup>See <http://tex.stackexchange.com/questions/tagged/tikz-pgf>.

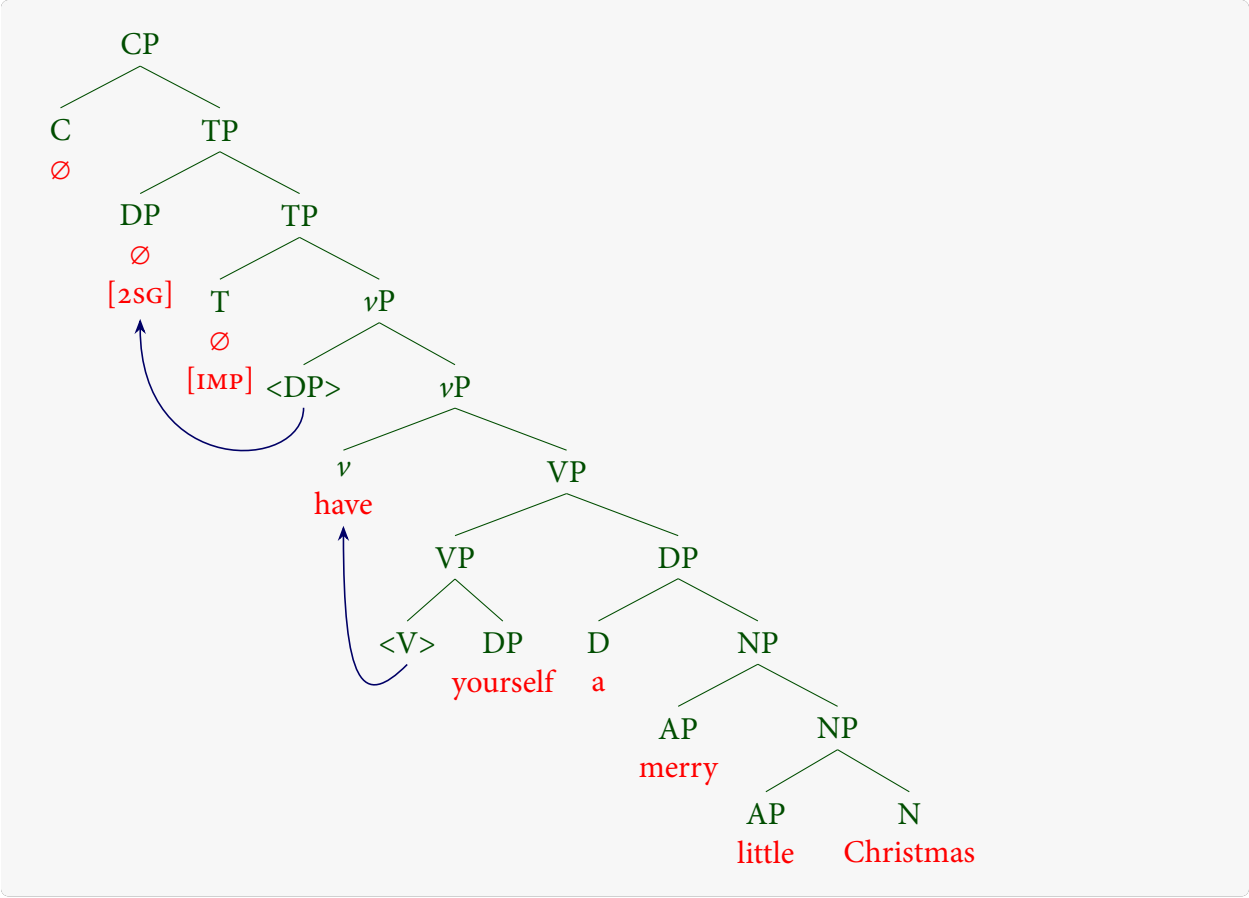
<sup>5</sup>See <http://www.texample.net/tikz/examples/>.

```

\begin{tikzpicture}[mytree, sibling distance=5em,
                  % A suitably coniferous colour. See the xcolor package.
                  color={green!30!black!100}]
\node (top) {CP}
  child {node {C\\textcolor{red}{\emptyset}}}
  child {node {TP}
    child {node (s2) {DP\\textcolor{red}{\emptyset}\\textcolor{red}{[\textsc{2sg}]}}}
    child {node {TP}
      child {node {T\\textcolor{red}{\emptyset}\\textcolor{red}{[\textsc{imp}]}}}
      child {node {\textit{v}P}
        % The <DP> is too tight with T so decrease sibling distance just a touch.
        [sibling distance=4.75em]
        child {node (s1) {<DP>}}
        child {node {\textit{v}P}
          % Here we need lots more sibling distance for the complex VP.
          [sibling distance=7em]
          child {node (v2) {\textit{v}\\textcolor{red}{have}}}
          child {node {VP}
            child {node {VP}
              [sibling distance=3em]
              child {node (v1) {<V>}}
              child {node {DP\\textcolor{red}{yourself}}}
            }
            child {node {DP}
              % Now back to the normal sibling distance.
              [sibling distance=5em]
              child {node {D\\textcolor{red}{a}}}
              child {node {NP}
                child {node {AP\\textcolor{red}{merry}}}
                child {node {NP}
                  % At the bottom, a slightly narrower sibling distance.
                  [sibling distance=4.5em]
                  child {node {AP\\textcolor{red}{little}}}
                  child {node {N\\textcolor{red}{Christmas}}}}}}}}}}}}}}};
\draw [exarrows, color={blue!40!black!100}]% A typical colour for garlands.
  (s1.south)
  .. controls +(south:2em) and +(south:5em) ..
  (s2.south);
\draw [exarrows, color={blue!40!black!100}]% Same colour as above.
  (v1.south)
  % The first control point goes out south west instead of south
  % so that the departure angle is broader, but it still departs
  % from the south point of the v1 node.
  .. controls +(south west:2em) and +(south:5em) ..
  (v2.south);
\end{tikzpicture}

```





### 3 Matrices

A matrix in TikZ is essentially a table, a grid arranged by rows and columns with things in each cell. But since a TikZ matrix is constructed by TikZ rather than made with e.g. L<sup>A</sup>T<sub>E</sub>X's table environment, the result can contain arbitrary TikZ code. Matrices are particularly useful for drawing tree-like diagrams which are too complex to be constructed with simple trees, such as autosegmental diagrams and feature geometry diagrams in phonology. Although the `matrix` command is built into TikZ, there are additional options and code supplied by the `matrix` library so that you generally always want to load this library in the preamble whenever you need to draw a matrix.

#### Preamble

```
\usetikzlibrary{matrix}
```

The example below illustrates a very trivial matrix in TikZ. Unlike L<sup>A</sup>T<sub>E</sub>X tables, TikZ does not need to be told how many columns are going to appear, and alignment of material in each column is done with the usual TikZ options.

```
\begin{tikzpicture}
\matrix {
\node (a) {A}; & \node (b) {B}; & \node (c) {C}; \\
\node (r) {R}; & \node (s) {S}; & \node (t) {T}; \\
\node (x) {X}; & \node (y) {Y}; & \node (z) {Z}; \\
};
\end{tikzpicture}
```

A	B	C
R	S	T
X	Y	Z

Each cell of a TikZ matrix can contain a TikZ statement. Usually this will be a `\node` command with some content. Because the `\node` case is so common, TikZ provides a special option `matrix of nodes` that supplies the usual `\node{...}`; code including the ever-important semicolon.

```
\begin{tikzpicture}
\matrix [matrix of nodes] {
A & B & C \\
R & S & T \\
X & Y & Z \\
};
\end{tikzpicture}
```

A	B	C
R	S	T
X	Y	Z

A TikZ matrix is actually a node in and of itself. If we add the option `draw=red` to the matrix it will appear with a bounding box coloured red. The `\matrix` command is actually shorthand for `\node [matrix]`.

```
\begin{tikzpicture}
\matrix [matrix of nodes, draw=red] {
A & B & C \\
R & S & T \\
X & Y & Z \\
};
\end{tikzpicture}
```

A	B	C
R	S	T
X	Y	Z

Since each cell in a `matrix of nodes` is a node, it is possible to draw lines between them. If a matrix is given a name like `name` then all of the nodes are automatically named `name-row-col` (i.e. `name-x-y`). In the following example we name the matrix `m` and draw lines from cell (1, 1) and cell (3, 1) to cell (2, 3).

```
\begin{tikzpicture}
\matrix (m) [matrix of nodes] {
  A & B & C \\
  R & S & T \\
  X & Y & Z \\
};
\draw (m-1-1) -- (m-2-3);
\draw (m-3-1) -- (m-2-3);
\end{tikzpicture}
```

The diagram shows a 3x3 matrix of nodes. The nodes are arranged in three rows and three columns. The first row contains nodes A, B, and C. The second row contains nodes R, S, and T. The third row contains nodes X, Y, and Z. Two lines are drawn: one from node A (row 1, column 1) to node T (row 2, column 3), and another from node X (row 3, column 1) to node T (row 2, column 3).

Nodes in a matrix with the `matrix of nodes` option can be explicitly named using the `|...|` syntax for specifying options to the node. The content of `|...|` here can be either a node name like `(...)`, an option to the `\node` command like `[...]`, or both. Below we supply some names with `|(...)|` and adjust the `\draw` commands to match.<sup>6</sup>

```
\begin{tikzpicture}
\matrix [matrix of nodes] {
  |(a)| A & B & C \\
  R & S & |(t)| T \\
  |(x)| X & Y & Z \\
};
\draw (a) -- (t);
\draw (x) -- (t);
\end{tikzpicture}
```

The diagram shows a 3x3 matrix of nodes. The nodes are arranged in three rows and three columns. The first row contains nodes |(a)| A, B, and C. The second row contains nodes R, S, and |(t)| T. The third row contains nodes |(x)| X, Y, and Z. Two lines are drawn: one from node |(a)| A (row 1, column 1) to node |(t)| T (row 2, column 3), and another from node |(x)| X (row 3, column 1) to node |(t)| T (row 2, column 3).

TikZ will automatically recognize when an explicit `\node` command (actually a `\path` or a shortcut like `\draw` or `\fill`) has been specified in a `matrix of nodes`. This means that we can mix the succinct `matrix of nodes` syntax with occasional use of the more explicit `\node` syntax.

```
\begin{tikzpicture}
\matrix [matrix of nodes] {
  a & b & \node (c) [circle, inner sep=1pt, draw] {c}; \\
  p & q & r \\
  x & y & z \\
};
\end{tikzpicture}
```

The diagram shows a 3x3 matrix of nodes. The nodes are arranged in three rows and three columns. The first row contains nodes a, b, and a circled node (c). The second row contains nodes p, q, and r. The third row contains nodes x, y, and z.

Nodes in a matrix are just like nodes in a tree, so that the size of a node will vary depending on its contents. This is shown below by drawing the bounding boxes of every node and mixing uppercase and lowercase letters.

<sup>6</sup>The `ltxdoc` class defines a shortcut `|...|` for the `\verb` command. This is incompatible with TikZ's use of `|...|` for matrices and must be disabled by saying `\DeleteShortVerb{|}` somewhere in the preamble.

```

\begin{tikzpicture}[every node/.style={draw}]
\matrix [matrix of nodes] {
  a & b & c \\
  P & Q & R \\
  X & y & z \\
};
\end{tikzpicture}

```

a	b	c
P	Q	R
X	y	z

We can adjust the size of the nodes using the same `text height` and `text width` options that we used in trees. TikZ supplies a shorthand option `nodes` that can be used instead of the much more verbose `every node/.append style={...}` (see TikZ manual III§20.3.3 “Cell Styles and Options”). Beyond this `nodes` option there are other options for modifying specific subsets of a matrix like `every odd row` and `every even column`, `column n` and `row n`, and `execute at empty cell` (cf. `node in empty cell` from the `matrix` library; see V§57.1 p. 647).

```

\begin{tikzpicture}[every node/.style={draw}]
\matrix [matrix of nodes,
  nodes={text height=0.7em, text depth=0.3em}] {
  a & b & c \\
  P & Q & R \\
  X & y & z \\
};
\end{tikzpicture}

```

a	b	c
P	Q	R
X	y	z

As shown by the preceding couple of examples, TikZ automatically smushes the cells together so that the rows and columns have the minimum necessary separation. This can sometimes be inadequate, either too small or too large. The row and column separation distances can be adjusted explicitly with the `row sep` and `column sep` options (see TikZ manual III§20.3.2 “Setting and Adjusting Column and Row Spacing”).

```

\begin{tikzpicture}[every node/.style={draw}]
\matrix [matrix of nodes, row sep=1ex, column sep=1ex] {
  a & b & c \\
  P & Q & R \\
  X & y & z \\
};
\end{tikzpicture}

```

a	b	c
P	Q	R
X	y	z

Attribute-value matrices can be constructed by using the delimiter options that are supplied by the `matrix` library (see TikZ manual V§57.3 “Delimiters”). First we define an `avm` style.

#### Preamble

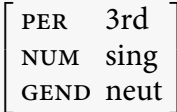
```

\tikzset{avm/.style={left delimiter={[, right delimiter={]},
  inner sep=0.125ex, column sep=0.75ex,
  column 1/.style={font=\scshape},
  nodes={right, text height=0.7em, text depth=0.3em}}}

```

With all of the configuration above in place, we can give a matrix the `avm` option and it magically appears as a nicely formatted attribute-value matrix.

```
\begin{tikzpicture}
\matrix [matrix of nodes, avm] {
per & 3rd \\
num & sing \\
gend & neut \\
};
\end{tikzpicture}
```



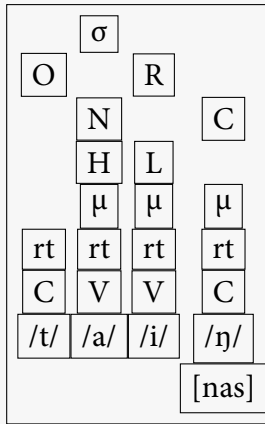
Adding code to draw other things like boxed numerals, indices, type annotations, and spaghetti lines is left as an exercise for the reader. It is possible in principle to implement everything from the `avm` package in `TikZ`, although getting the syntax to be a drop-in replacement will be challenging.

### 3.1 Autosegmental diagrams

Autosegmental diagrams represent phonological structures horizontally arranged in time segments and vertically arranged in tiers. There can be association lines between nodes in each level and between nodes in each tier. Because autosegmental diagrams may include cyclic paths they cannot be easily represented using `TikZ`'s tree syntax. They are however a natural candidate for representation with `TikZ`'s matrices.

We want to design an autosegmental diagram of a hypothetical word /tâiŋ/ which is pronounced as [tâiŋ̃] with tone and nasalization spreading. In addition to the facts in that transcription, we also want to indicate moraicity of the vowels and the coda consonant, as well as the internal structure of the syllable (onset, rhyme, coda). To start with we can fill in an appropriate matrix.

```
\begin{tikzpicture}[every node/.style={draw}]
\matrix [matrix of nodes] {
& |(top)| σ & & & \\
|(O)| O & & |(R)| R & & \\
& |(N)| N & & |(C)| C & \\
& |(H)| H & |(L)| L & & \\
& |(m1)| μ & |(m2)| μ & |(m3)| μ & \\
|(r1)| rt & |(r2)| rt & |(r3)| rt & |(r4)| rt & \\
|(c1)| C & |(v1)| V & |(v2)| V & |(c2)| C & \\
|(t)| /t/ & |(a)| /a/ & |(i)| /i/ & |(ng)| /ŋ/ & \\
& & & |(nf)| [nas] & \\
};
\end{tikzpicture}
```



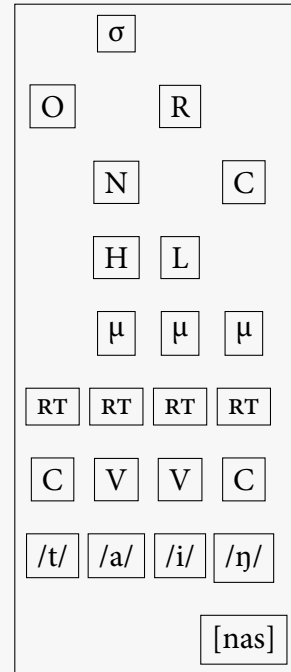
The last column is far too wide. The other columns have slightly different widths as well, making the whole thing a bit crooked looking. We can force all of the columns to have the same width by setting the `column sep` and adding that it is calculated between `origins` rather than from the left and right edges of the nodes in each column.

Since we are going to draw association lines between each row we will want to adjust the `row sep` as well. Setting this to 1 em should give enough space for lines between every row. Also all of those 'rt' on row 6 should be in small caps.

```

\begin{tikzpicture}[every node/.style={draw}]
\matrix [matrix of nodes, row sep=1em,
        column sep={2em,between origins},
        row 6/.style={font=\scshape}]
{
    & |(top)|  $\sigma$  & & & \\
|(O)| O & & |(R)| R & & \\
    & |(N)| N & & |(C)| C & \\
    & |(H)| H & |(L)| L & & \\
    & |(m1)|  $\mu$  & |(m2)|  $\mu$  & |(m3)|  $\mu$  & \\
|(r1)| rt & |(r2)| rt & |(r3)| rt & |(r4)| rt & \\
|(c1)| C & |(v1)| V & |(v2)| V & |(c2)| C & \\
|(t)| /t/ & |(a)| /a/ & |(i)| /i/ & |(ng)| /ŋ/ & \\
    & & & |(nf)| [nas] & \\
};
\end{tikzpicture}

```

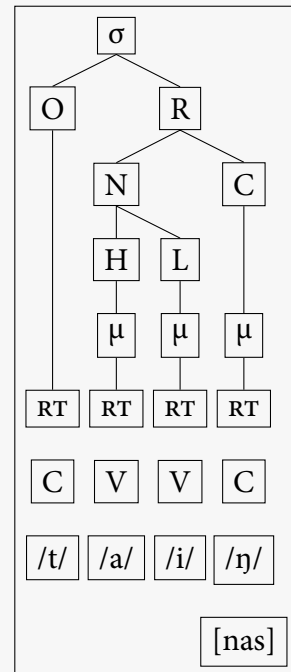


Now it's time to draw some lines. We could repeatedly say things like `\draw (top) -- (O);` over and over but this is really tiresome. TikZ helpfully supplies a special foreach operation that can apply across a list of variables. This saves us a good amount of both space and time.

```

\begin{tikzpicture}[every node/.style={draw}]
\matrix [matrix of nodes, row sep=1em,
        column sep={2em,between origins},
        row 6/.style={font=\scshape}]
{
    & |(top)|  $\sigma$  & & & \\
|(O)| O & & |(R)| R & & \\
    & |(N)| N & & |(C)| C & \\
    & |(H)| H & |(L)| L & & \\
    & |(m1)|  $\mu$  & |(m2)|  $\mu$  & |(m3)|  $\mu$  & \\
|(r1)| rt & |(r2)| rt & |(r3)| rt & |(r4)| rt & \\
|(c1)| C & |(v1)| V & |(v2)| V & |(c2)| C & \\
|(t)| /t/ & |(a)| /a/ & |(i)| /i/ & |(ng)| /ŋ/ & \\
    & & & |(nf)| [nas] & \\
};
\draw foreach \x in {O, R} {(top.south) -- (\x.north)};
\draw foreach \x in {N, C} {(R.south) -- (\x.north)};
\draw foreach \x in {H, L} {(N.south) -- (\x.north)};
\draw (H.south) -- (m1.north); \draw(L.south)--(m2.north);
\draw (O.south) -- (r1.north);
\draw (C.south) -- (m3.north);
\draw (m1.south) -- (r2.north);
\draw (m2.south) -- (r3.north);
\draw (m3.south) -- (r4.north);
\end{tikzpicture}

```

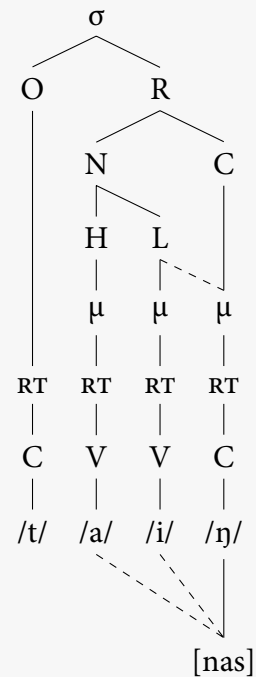


We're almost done so we can switch off the bounding boxes. To draw the linking association lines we use the [dashed] option. To avoid repeating this we enclose the affected paths inside TikZ's scope environment which applies its options to everything inside it. And that new [1.5em] after row 8 helps the nasal tier balance a bit better with all those lines.

```

\begin{tikzpicture}
\matrix [matrix of nodes, row sep=1em,
        column sep={2em,between origins},
        row 6/.style={font=\scshape}]
{
    & |(top)| σ & & & \\
|(O)| O & & & |(R)| R & & \\
    & |(N)| N & & & |(C)| C & \\
    & |(H)| H & & |(L)| L & & \\
    & |(m1)| μ & & |(m2)| μ & & |(m3)| μ & \\
|(r1)| rt & |(r2)| rt & |(r3)| rt & |(r4)| rt & & \\
|(c1)| C & |(v1)| V & |(v2)| V & |(c2)| C & & \\
|(t)| /t/ & |(a)| /a/ & |(i)| /i/ & |(ng)| /ŋ/ & |[1.5em] \\
    & & & |(nf)| [nas] & & \\
};
\draw foreach \x in {O, R} {(top.south) -- (\x.north)};
\draw foreach \x in {N, C} {(R.south) -- (\x.north)};
\draw foreach \x in {H, L} {(N.south) -- (\x.north)};
\draw (H.south) -- (m1.north);
\draw (L.south) -- (m2.north);
\draw (O.south) -- (r1.north);
\draw (C.south) -- (m3.north);
\draw (m1.south) -- (r2.north);
\draw (m2.south) -- (r3.north);
\draw (m3.south) -- (r4.north);
\draw (r1.south) -- (c1.north);
\draw (r2.south) -- (v1.north);
\draw (r3.south) -- (v2.north);
\draw (r4.south) -- (c2.north);
\draw (c1.south) -- (t.north);
\draw (v1.south) -- (a.north);
\draw (v2.south) -- (i.north);
\draw (c2.south) -- (ng.north);
\draw (ng.south) -- (nf.north);
\begin{scope}[dashed]
    \draw (L.south) -- (m3.north);
    \draw foreach \x in {a, i} {(\x.south) -- (nf.north)};
\end{scope}
\end{tikzpicture}

```



The resulting code is a bit verbose because of all those \draw commands, and there is probably some way to make it more compact. But the printed output is worthy of publication in *Phonology*, or at least it would be if it said something interesting.

## 3.2 Partial orderings and lattices

Partial orderings and lattices are easy to represent with *TikZ* matrices. The only real issue is when a diagram includes a lot of branches, in which case the required `\draw` commands get very repetitive. But as we saw in the preceding subsection, *TikZ* has a `foreach` command that allows us to map over lists of nodes. This can even be nested within another `foreach`, so that complex fanouts of lines end up being relatively compact and easy to understand.

The following example is a horizontally oriented lattice with a maximal element  $\top$  ‘top’ on the left and a minimal element  $\perp$  ‘bottom’ on the right. The columns are filled with appropriately named elements.

```
\begin{tikzpicture}
\matrix [matrix of nodes, row sep=1em,
column sep={4em,between origins},
text height=0.7em, text depth=0.3em] {
& |(a1)| a & |(a2)|  $\alpha$  & |(a3)| a & \\
& |(b1)| b & |(b2)|  $\beta$  & |(b3)| б & \\
|(T)|  $\top$  & |(c1)| c & |(c2)|  $\gamma$  & |(c3)| B & |(B)|  $\perp$  \\
& |(d1)| d & |(d2)|  $\delta$  & |(d3)| r & \\
& |(e1)| e & |(e2)|  $\varepsilon$  & |(e3)| д & \\
};
\end{tikzpicture}
```

---

	a	$\alpha$	a	
	b	$\beta$	б	
$\top$	c	$\gamma$	B	$\perp$
	d	$\delta$	r	
	e	$\varepsilon$	д	

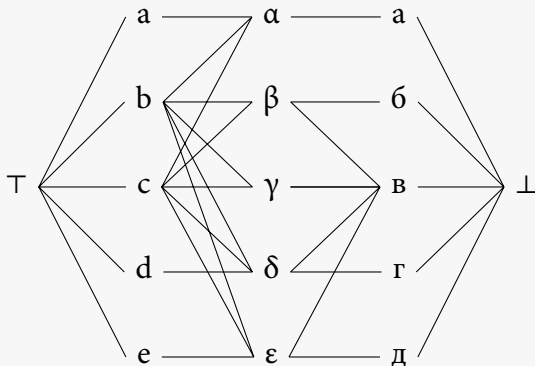
We’d like to draw some large sets of lines between the nodes in this lattice. We need the  $\top$  and  $B$  nodes to connect to every node in the columns next to them. To implement this we can use the `foreach` in much the same way as we did before. We’d also like to do a set of lines from both  $b1$  and  $c1$  to everything in the  $x2$  row. To do this we will nest one `foreach` inside of another. And we want  $c3$  to connect to everything in the 2 column except  $a2$ .



```

\begin{tikzpicture}
\matrix [matrix of nodes, row sep=1em,
        column sep={4em,between origins},
        text height=0.7em, text depth=0.3em] {
& |(a1)| a & |(a2)|  $\alpha$  & |(a3)| a & \\
& |(b1)| b & |(b2)|  $\beta$  & |(b3)| б & \\
|(T)| T & |(c1)| c & |(c2)|  $\gamma$  & |(c3)| B & |(B)|  $\perp$  \\
& |(d1)| d & |(d2)|  $\delta$  & |(d3)| r & \\
& |(e1)| e & |(e2)|  $\varepsilon$  & |(e3)| д & \\
};
\draw foreach \t in {a1, b1, c1, d1, e1} {(T.east) -- (\t.west)};
\draw foreach \b in {a3, b3, c3, d3, e3} {(\b.east) -- (B.west)};
\draw foreach \one in {b1, c1}
  {foreach \two in {a2, b2, c2, d2, e2}
   {(\one.east) -- (\two.west)}};
\draw foreach \x in {b2, c2, d2, e2} {(\x.east) -- (c3.west)};
\draw (a1.east) -- (a2.west);
\draw (a2.east) -- (a3.west);
\draw (b2.east) -- (b3.west);
\draw (c2.east) -- (c3.west);
\draw (d1.east) -- (d2.west);
\draw (d2.east) -- (d3.west);
\draw (e1.east) -- (e2.west);
\draw (e2.east) -- (e3.west);
\end{tikzpicture}

```



At the end we fill in a bunch of single lines so that the result looks like a proper lattice. For ease of reading we order these `\draw` commands first by rows and then by columns since the labels start with a row (a letter). Transforming this matrix to a vertical arrangement is left as an exercise for the reader. The labels can be kept the same with the matrix code rotated  $90^\circ$ . Then the east and west points will need to be changed to south and north respectively.

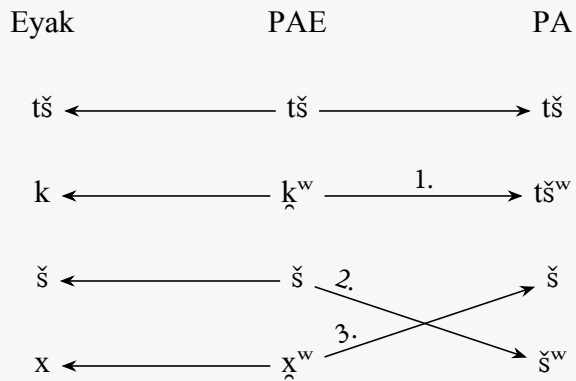
Partial ordering diagrams are essentially the same as lattices but with reduced complexity. The principles of designing them are the same, with first a choice between vertical or horizontal arrangement, then deciding on column and row separating, and finally drawing the connecting lines.

The following example illustrates a historical sound change with a simple TikZ matrix. This is taken from Michael Krauss’s “Proto-Athabaskan-Eyak fricatives and the first person singular”, an unpublished manuscript from 1977 at the Alaska Native Language Center. The original diagram was drawn by hand except for the phonemes which, like the rest of the text, were set with a typewriter. Among other things this diagram illustrates the use of nodes on paths to serve as annotations.

```

\begin{tikzpicture}[annotate/.style={sloped, font=\small, inner sep=0.5ex}]
\matrix [matrix of nodes, row sep=1em,
        column sep={8em,between origins},
        text height=0.7em, text depth=0.3em,
        nodes={font=\ipafont},
        row 1/.style={font=\normalfont}] {
Eyak      & PAE      & PA\\
|(e1)| tš & |(p1)| tš & |(a1)| tš\\
|(e2)| k  & |(p2)| k̥ʷ & |(a2)| tšʷ\\
|(e3)| š  & |(p3)| š  & |(a3)| š\\
|(e4)| x  & |(p4)| x̥ʷ & |(a4)| šʷ\\
};
\begin{scope}[exarrows]
\draw (p1) -- (e1); \draw (p1) -- (a1);
\draw (p2) -- (e2); \draw (p2) -- (a2) node [annotate, above, midway] {1.};
\draw (p3) -- (e3); \draw (p3) -- (a4) node [annotate, above, pos=0.125] {2.};
\draw (p4) -- (e4); \draw (p4) -- (a3) node [annotate, above, pos=0.125] {3.};
\end{scope}
\end{tikzpicture}

```



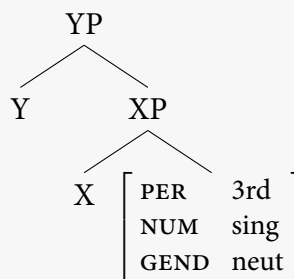
The paths with labels on them have an additional node specified at the end. The `midway` option says to place the node about midway along the path. The `pos` option gives a specific fraction of the length where the node should be positioned, here  $\frac{1}{8}$  of the length from the start (see TikZ manual III§17.8 “Placing Nodes on a Line or Curve Explicitly”). The `above` option (from the `positioning` library) places the new node above the path. Then our `annotate` style specifies that the node should be rotated to be tangent to the path with `sloped`, here tilted the same angle as the line (III§17.8 p. 237). The `inner sep` controls the size of the node beyond its content and hence the distance of the node from the line.

### 3.3 Matrices in trees

Since a TikZ matrix is a node it can appear inside of a tree. But we can't use the `\matrix` command in a tree because we are not at the beginning of a TikZ sentence where commands are allowed. Instead we can use the usual expansion of `\matrix` which is `node [matrix]`. Since children are specified as `child {node {...}}`, adding a matrix to a node is done by saying `child {node [matrix] {...}}`.

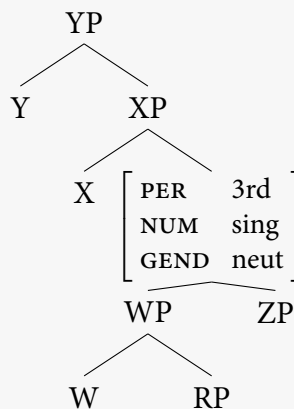
For annoying technical reasons we can't use the usual `&` symbol as a column delimiter for a matrix inside a tree. TikZ has an option `ampersand replacement` with which the `&` can be replaced by a macro (see TikZ manual III§20.5 "Considerations Concerning Active Characters"); the conventional replacement is `\&`. We define a style `tmx` that contains the appropriate `ampersand replacement` as well as the usual `matrix` and `matrix of nodes` options. Recalling our `avm` style for attribute-value matrices, we can combine this with `tmx` to get an attribute-value matrix in a tree.

```
\begin{tikzpicture}[mytree,
  tmx/.style={matrix, matrix of nodes, font=\small,
    ampersand replacement=\&}]
\node (top) {YP}
  child {node {Y}}
  child {node {XP}
    child {node {X}}
    child {node (m) [tmx, avm]
      {per \& 3rd \\
      num \& sing\\
      gend \& neut\\ }};
\end{tikzpicture}
```



It's also possible to put matrices in non-leaf nodes somewhere in the middle of a tree. Unfortunately, for reasons I have not entirely worked out, the child branches of a matrix are not positioned like we might expect.

```
\begin{tikzpicture}[mytree,
  tmx/.style={matrix, matrix of nodes, font=\small,
    ampersand replacement=\&}]
\node (top) {YP}
  child {node {Y}}
  child {node {XP}
    child {node {X}}
    child {node (m) [tmx, avm]
      {per \& 3rd \\
      num \& sing\\
      gend \& neut\\ }}
    child {node {WP}
      child {node {W}}
      child {node {RP}}
      child {node {ZP}}};
\end{tikzpicture}
```



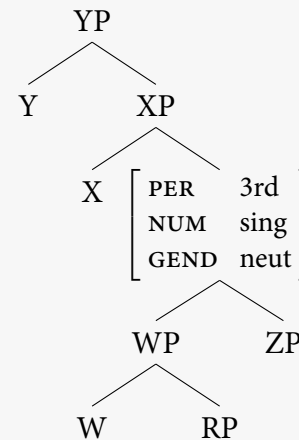
The obvious solution to this problem is to explicitly adjust the `level distance` for the children

of the node. Because of TikZ's scoping rules this change of the `level distance` would percolate downwards to further children, so it has to be set back after the aberrant level.

```

\begin{tikzpicture}[mytree,
  tmx/.style={matrix, matrix of nodes, font=\small,
    ampersand replacement=\&}]
\node (top) {YP}
  child {node {Y}}
  child {node {XP}
    child {node {X}}
    child {node (m) [tmx, avm]
      {per \& 3rd \\
        num \& sing \\
        gend \& neut \\
        }
      [level distance=3em]
      child {node {WP}
        [level distance=2em]
        child {node {W}}
        child {node {RP}}}
      child {node {ZP}}}}};
\end{tikzpicture}

```



My hunch is that the `level distance` of the children is being calculated from the `child anchor` point of the matrix which is set to north in the `mytree` style definition. I do not yet know how to tell TikZ to instead calculate this from the south point of the matrix node. I leave the problem of post-matrix level distance adjustment for future research.

### 3.4 Trees in matrices

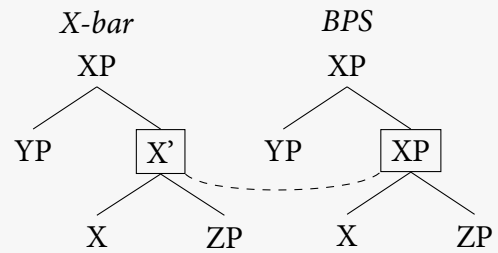
Since matrices are composed of nodes, it is entirely possible to place trees inside of a matrix. One situation where this might be useful is comparing a couple of small trees side-by-side. For larger side-by-side arrangements each tree should probably be a separate `tikzpicture` arranged together by the usual L<sup>A</sup>T<sub>E</sub>X mechanisms like the `subfigure` package.

The following example puts two trees into a matrix. Note that we do not use `matrix of nodes`. Instead we specify the nodes explicitly so that we can use the tree syntax within them. And since everything here is contained within a single `tikzpicture`, we can draw lines from a named node in one tree to a named node in the other tree. This is possible but more difficult to configure with separate `tikzpictures`.

```

\begin{tikzpicture}[mytree]
\matrix {
\node {\textit{X-bar}};
& \node {\textit{BPS}}; \\
\node {XP}
  child {node {YP}}
  child {node (x1) [draw] {X'}}
    child {node {X}}
    child {node {ZP}}};
&
\node {XP}
  child {node {YP}}
  child {node (x2) [draw] {XP}}
    child {node {X}}
    child {node {ZP}}}; \\
};
\draw [dashed] (x1.south east)
  .. controls +(south east:1em)
  and +(south west:1em)
  .. (x2.south west);
\end{tikzpicture}

```



## 4 Numbered examples

There are several packages out there to typeset numbered examples in  $\text{\LaTeX}$ , including `gb4e` and `Expex`. In principle any package that does not mangle the  $\text{\TeX}$  parser too much (e.g. changing active characters) should be compatible with `TikZ`. I have not tested `TikZ` with `gb4e` but I have combined `Expex` and `TikZ` extensively so this section illustrates only this particular combination.

### 4.1 Arrows in examples

The usual reason for wanting to draw something on a numbered example is to supply movement arrows. The technique for doing so involves using `TikZ`'s `overlay` option to draw lines on previously typeset text. But for `TikZ` to know where to put things it needs nodes to work with. Because something like a numbered example is not within a `tikzpicture` environment we have to do some extra work to inform `TikZ` of our intent.

Let's start with a simple numbered example. This has an element `_` indicating a base position as well as the word 'Which' that is moved from this position.

```
\ex {}[\textsubscript{CP} Which did John say  
      [\textsubscript{CP} PRO to move \textunderscore ]]?  
\xe
```

---

(2)  $[_{CP}$  Which did John say  $[_{CP}$  PRO to move  $_$  ])?

The first square bracket needs to be escaped otherwise the `\ex` command will think it is an option list to be parsed, so we insert some prothetic braces. Note that the sentence has been broken over two lines and the second level of bracketing is indented with respect to the first. This makes the source code easier to read and modify.

First we need to tell `TikZ` that we are going to do some stuff it should remember for later drawing (see `TikZ` manual III§17.3 "Referencing Nodes Outside the Current Picture"). We modify the every `picture` style, adding the option `remember picture` so that `TikZ` will keep track of commands in the current context inside the `\ex` command.

```
\ex%  
\tikzstyle{every picture}+=[remember picture]%  
  {}[\textsubscript{CP} Which did John say  
     [\textsubscript{CP} PRO to move \textunderscore ]]?  
\xe
```

---

(3)  $[_{CP}$  Which did John say  $[_{CP}$  PRO to move  $_$  ])?

The next thing we need to do is add some nodes for `TikZ` to draw with. Since we aren't in a `tikzpicture` environment the `\node` command won't work. We can preface this with `\tikz` which tells `TikZ` to read until it reaches a semicolon ending a `TikZ` sentence. This command is used for embedding `TikZ` diagrams in regular text.

```

\ex%
\tikzstyle{every picture}+=[remember picture]%
  {}[\textsubscript{CP} \tikz\node(wh2){Which}; did John say
    [\textsubscript{CP} PRO to move \tikz\node(wh1){\textunderscore}; ]]?
\xe

```

---

(4)  $[_{CP} \text{Which} \text{ did John say } [_{CP} \text{PRO to move } - ]]$ ?

This obviously doesn't look right. If we draw the node bounding boxes we can see why.

```

\ex%
\tikzstyle{every picture}+=[remember picture, every node/.style={draw}]%
  {}[\textsubscript{CP} \tikz\node(wh2){Which}; did John say
    [\textsubscript{CP} PRO to move \tikz\node(wh1){\textunderscore}; ]]?
\xe

```

---

(5)  $[_{CP} \boxed{\text{Which}} \text{ did John say } [_{CP} \text{PRO to move } \boxed{-} ]]$ ?

TikZ is doing its usual thing of adding some inner sep and positioning the lower edge of the nodes at the baseline. We can easily tell it not to do so.

```

\ex%
\tikzstyle{every picture}+=[remember picture, inner sep=0pt, baseline, anchor=base]%
  {}[\textsubscript{CP} \tikz\node(wh2){Which}; did John say
    [\textsubscript{CP} PRO to move \tikz\node(wh1){\textunderscore}; ]]?
\xe

```

---

(6)  $[_{CP} \text{Which} \text{ did John say } [_{CP} \text{PRO to move } \_ ]]$ ?

So now we have two nodes. We can draw a line between them by adding a tikzpicture at the end of the example but before the \xe. This tikzpicture is overlaid on the example with overlay.


```

\ex%
\tikzstyle{every picture}+=[remember picture, inner sep=0pt, baseline, anchor=base]%
  {}[\textsubscript{CP} \tikz\node(wh2){Which}; did John say
    [\textsubscript{CP} PRO to move \tikz\node(wh1){\textunderscore}; ]]?
\begin{tikzpicture}[overlay]
  \draw [exarrows] (wh1.south) -- ++(south:1.5ex) -| (wh2.south);
\end{tikzpicture}
\xe

```

---

(7)  $[_{CP} \text{Which} \text{ did John say } [_{CP} \text{PRO to move } \_ ]]$ ?



The `++(south:1.5ex)` is a polar coordinate that says to move south  $1\frac{1}{2}$  ex from the start point of the arrow, and then the `-|` creates two  $90^\circ$  angles like we saw previously for drawing square movement arrows in trees. This needs to be compiled twice: once for TikZ to register where the nodes are and a second time for it to draw the arrow correctly. A two step compilation is usually necessary for any `remember picture`.

The arrow above looks nice enough, but it's smushed right up against the baseline at its start and end points. We'll adjust the node sizes so that this problem goes away. But first, saying all that `\tikzstyle` and `\tikz\node` stuff repeatedly is annoying. We can define some commands to wrap these up conveniently. I am using a system provided by an official  $\text{\LaTeX}$  3 package called `xparse`. I encourage all  $\text{\LaTeX}$  users to switch to this from the  $\text{\LaTeX}$  2 $\epsilon$  commands like `\newcommand`.

#### Preamble

```
\DeclareDocumentCommand \tikzexsetup {} {%
  \tikzstyle{every picture}+=[remember picture, inner sep=0pt,
    baseline, anchor=base]}

% arg 1: optional strut; arg 2: node name; arg 3: node content
\DeclareDocumentCommand \ND {s m m} {%
  \tikzifinpicture{}{\tikz}\node(#2){\IfBooleanTF{#1}{\strut}{}#3};}
```

The first command `\tikzexsetup` takes no arguments and just does the `every picture` stuff. The second command called `\ND` is more interesting. It takes three arguments: the first is an optional star and the other two are mandatory arguments enclosed in the usual curly braces. The code in this first says that nothing happens if we are in a `tikzpicture` environment, but if we are not inside one then it says `\tikz`. Using the `\tikz` command inside a `tikzpicture` will cause an error, so this extra bit of code allows us to use our `\ND` both inside and outside of `tikzpictures` without worrying.

The next thing that `\ND` does is to emit a `\node` with the second argument `#2` (the first mandatory argument) in parentheses. This makes the second argument into the node name. Then the node content follows in curly braces. Inside this content the first argument `#1` – the optional star – will add a `\strut` if it is true, i.e. if the star is present, and otherwise does nothing. Finally the third argument is output and the node content is closed, followed by TikZ's obligatory semicolon.

```
\ex\tikzexsetup%
  {}[\textsubscript{CP} \ND{wh2}{Which} did John say
    [\textsubscript{CP} PRO to move \ND{wh1}{\textunderscore} ]]?
\begin{tikzpicture}[overlay]
  \draw [exarrows] (wh1.south) -- ++(south:1.5ex) -| (wh2.south);
\end{tikzpicture}
\xe
```

---

(8)  $[\text{CP} \text{Which} \text{did John say} [\text{CP} \text{PRO to move } \_ ]]?$

This is much easier to read. Now to address our arrow problem. The arrow start and end points are smushed up against the baseline because the nodes have no text below the baseline. We can fix this now by simply adding the star to our `\ND` commands which will insert a `\strut`. To anticipate our next problem we'll also add some text in a paragraph below.



```

\ex\tikzsetup%
  {}[\textsubscript{CP} \ND*{wh2}{Which} did John say
    [\textsubscript{CP} PRO to move \ND*{wh1}{\textunderscore} ]]?
\begin{tikzpicture}[overlay]
  \draw [exarrows] (wh1.south) -- ++(south:1.5ex) -| (wh2.south);
\end{tikzpicture}
\ex

```

This is `\emph{wh}`-movement from an infinitival complement clause of a quotative verb.

---

(9)  $[_{CP}$  Which did John say  $[_{CP}$  PRO to move  $_$  ]]?

This is *wh*-movement from an infinitival complement clause of a quotative verb.

The problem now is that the arrow overlaps with the text in the following paragraph. We can fix this by telling ExPex that the example needs a bit more space below it. To do so we'll define an ExPex style which modifies the `belowexskip` option. This can just be a fixed size like 1 em but we will add `plus` and `minus` components so that T<sub>E</sub>X can grow or shrink it a bit as necessary.

Preamble

```

\definelingstyle{exarrbelow}{belowexskip=1em plus 0.375em minus 0.25em}

```

Document body

```

\ex[lingstyle=exarrbelow]\tikzsetup%
  {}[\textsubscript{CP} \ND*{wh2}{Which} did John say
    [\textsubscript{CP} PRO to move \ND*{wh1}{\textunderscore} ]]?
\begin{tikzpicture}[overlay]
  \draw [exarrows] (wh1.south) -- ++(south:1.5ex) -| (wh2.south);
\end{tikzpicture}
\ex

```

This is `\emph{wh}`-movement from an infinitival complement clause of a quotative verb.

---

(10)  $[_{CP}$  Which did John say  $[_{CP}$  PRO to move  $_$  ]]?

This is *wh*-movement from an infinitival complement clause of a quotative verb.

Now what about examples with multiple parts? If we stick an arrow underneath a subpart we will see the same collision happening but this time with a following subpart. The `exarrbelow` style continues to work right since it applies to the whole `\pex ... \xe` unit, but the spacing doesn't apply to individual `\a` subparts in the example.

```

\pex[lingstyle=exarrbelow]
\atikzsetup%
  {}[\textsubscript{CP} \ND*{wh2}{Which} did John say
    [\textsubscript{CP} PRO to move \ND*{wh1}{\textunderscore} ]]?
\atikzsetup%
  {}[\textsubscript{CP} \ND*{wh4}{Which} did John say
    [\textsubscript{CP} PRO to move \ND*{wh3}{\textunderscore} ]]?
\begin{tikzpicture}[overlay]
  \draw [exarrows] (wh1.south) -- ++(south:1.5ex) -| (wh2.south);
  \draw [exarrows] (wh3.south) -- ++(south:1.5ex) -| (wh4.south);
\end{tikzpicture}
\xe

```

This is `\emph{wh}`-movement from an infinitival complement clause of a quotative verb.

- (11) a. [CP Which did John say [CP PRO to move \_ ]]?  
 b. [CP <sup>↑</sup>Which did John say [CP PRO to move \_ ]]?  
 ↑

This is *wh*-movement from an infinitival complement clause of a quotative verb.

To fix this we once again dig through ExPex's manual to find the `interpartskip` option.

```

\pex[lingstyle=exarrbelow, interpartskip=1em plus 0.125em minus 0.125em]
\atikzsetup%
  {}[\textsubscript{CP} \ND*{wh2}{Which} did John say
    [\textsubscript{CP} PRO to move \ND*{wh1}{\textunderscore} ]]?
\atikzsetup%
  {}[\textsubscript{CP} \ND*{wh4}{Which} did John say
    [\textsubscript{CP} PRO to move \ND*{wh3}{\textunderscore} ]]?
\begin{tikzpicture}[overlay]
  \draw [exarrows] (wh1.south) -- ++(south:1.5ex) -| (wh2.south);
  \draw [exarrows] (wh3.south) -- ++(south:1.5ex) -| (wh4.south);
\end{tikzpicture}
\xe

```

This is `\emph{wh}`-movement from an infinitival complement clause of a quotative verb.

- (12) a. [CP Which did John say [CP PRO to move \_ ]]?  
 b. [CP <sup>↑</sup>Which did John say [CP PRO to move \_ ]]?  
 ↑

This is *wh*-movement from an infinitival complement clause of a quotative verb.

As before, if we need to do this regularly it's a good idea to create an ExPex style in the preamble. The same problem happens with aligned glosses, in which case ExPex has options for adjusting the skip between individual lines of the gloss.




```

\ex[aboveglftskip=1.25em]\tikzsetup%
\begin{gl}
  \gla {} Avez {} {} -vous {} envoyé un livre à Paul? {} {} //
  \glb \ND*{j2}{[} have \ND*{i1}{\textunderscore} {}] -you
        \ND*{i2}{[} send.\textsc{past} a book to Paul {}]
        \ND*{j1}{\textunderscore} //
  \glft ‘Have you sent a book to Paul?’
        \trailingcitation{(Poletto \& Pollock 2004: 260)} //
\end{gl}
\begin{tikzpicture}[overlay]
  \draw[exarrows] (i1.south) -- ++(south:1.5ex) -| (i2.south);
  \draw[exarrows] (j1.south) -- ++(south:2ex) -| (j2.south);
\end{tikzpicture}
\ex

```

---

(15) Avez -vous envoyé un livre à Paul?  
 [ have \_ ] -you [ send.PAST a book to Paul ] \_  
  
 ‘Have you sent a book to Paul?’ (Poletto & Pollock 2004: 260)

Now this example is all ready for printing in the next volume of the OUP Cartography series. Certainly Rizzi and Cinque will appreciate how stylish it looks in their book, and with the arrows in place it's much easier to read than the original style below that uses only index letters.

```

\ex%
\begin{gl}
  \gla {} Avez {} {} -vous {} envoyé un livre à Paul? {} {} //
  \glb {}[ have \textunderscore\textsubscript{i} {}]\textsubscript{j} -you
        {}[ send.\textsc{past} a book to Paul {}]\textsubscript{i}
        \textunderscore\textsubscript{j} //
  \glft ‘Have you sent a book to Paul?’
        \trailingcitation{(Poletto \& Pollock 2004: 260)} //
\end{gl}
\ex

```

---

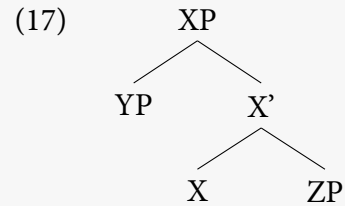
(16) Avez -vous envoyé un livre à Paul?  
 [ have <sub>i</sub> ]<sub>j</sub> -you [ send.PAST a book to Paul ]<sub>i</sub> <sub>j</sub>  
 ‘Have you sent a book to Paul?’ (Poletto & Pollock 2004: 260)

## 4.2 Trees in examples

Syntactic trees are often too large to reasonably fit within a numbered example and so should instead be placed in figure floats. But small trees can easily fit in a numbered example, usually as long as

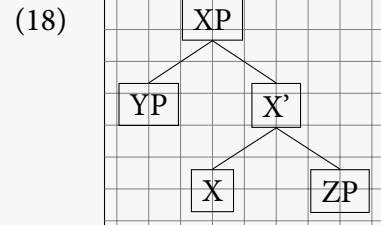
the tree has no more than two or maybe three levels. Putting a tree in a simple numbered example is straightforward, just place a tikzpicture inside of ExPex's `\ex ... \xe` and you're done.

```
\ex
\begin{tikzpicture}[mytree]
\node (top) {XP}
  child {node {YP}}
  child {node {X'}}
    child {node {X}}
    child {node {ZP}}};
\end{tikzpicture}
\xe
```



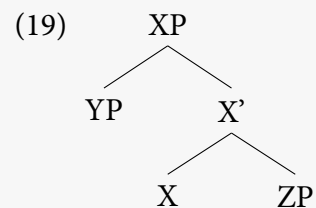
Well, it's almost that easy. Although the example above is certainly acceptable, there are some things that would be nice to adjust. For starters, there's a lot of whitespace between the example number and the tree. We'll turn on the framed option and the background grid (both from the backgrounds library, see TikZ manual V§43 "Background Library") to see the bounding boxes and geometry of the tikzpicture.

```
\ex
\begin{tikzpicture}[mytree, framed,
  background grid/.style={draw, help lines, step=1em},
  show background grid, every node/.style={draw}]
\node (top) {XP}
  child {node {YP}}
  child {node {X'}}
    child {node {X}}
    child {node {ZP}}};
\end{tikzpicture}
\xe
```



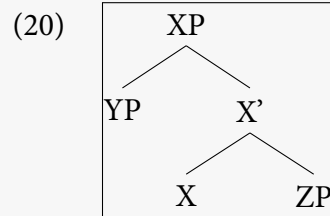
The whitespace between number and tree has two sources: one is from the ExPex settings and the other is from the TikZ settings. In this document the ExPex option `textoffset` is set to `0.75em`; this is the primary option for controlling the indentation distance between the example number and the content. TikZ also provides an option `trim left` (see TikZ manual III§15.8 "Establishing a Bounding Box") which can adjust the relative position of the left side of the bounding box. This is similar to using `\hspace{-...}` but it uses the tikzpicture's internal geometry from the root of the tree.

```
\ex
\begin{tikzpicture}[mytree, trim left=-2em]
\node (top) {XP}
  child {node {YP}}
  child {node {X'}}
    child {node {X}}
    child {node {ZP}}};
\end{tikzpicture}
\xe
```



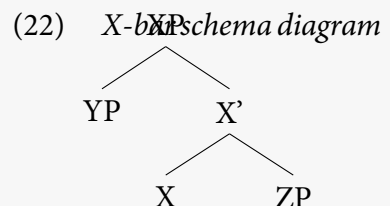
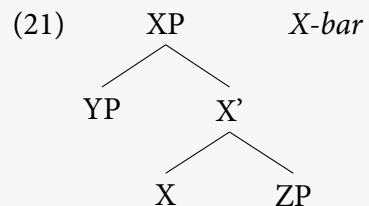
TikZ also provides an explicit way to specify the bounding box independent of any actual content in the tikzpicture. The tikzpicture is not clipped: if the content is too large for the bounding box then bits of it will stick out. Usually this should only be used as a last resort when fitting a diagram into its context is particularly difficult (e.g. posters, grant applications). The bounding box is an option `use as bounding box` applied to a path, or with the shortcut `\useasboundingbox` which is essentially equivalent to `\path [use as bounding box]` (see TikZ manual III§15.8 “Establishing a Bounding Box”). The root node is at  $(0,0)$  and trees grow negatively (down).

```
\ex
\begin{tikzpicture}[mytree]
\draw[use as bounding box]
(-2.625em,0) rectangle (4.625em,-6.75em);
\node (top) {XP}
  child {node {YP}}
  child {node {X'}}
    child {node {X}}
    child {node {ZP}}};
\end{tikzpicture}
\xe
```



Bounding boxes aside, we may want to add a label on the side of the example, helping readers to distinguish the purpose of this example from others nearby. ExPeX provides a convenient way of adding annotations that hang on the right margin with the `\rightcomment{...}` command. This typesets whatever it is given, right aligned flush with the right margin of the page. This command is perfectly compatible with TikZ, but it completely ignores the tikzpicture (and indeed everything else) and will happily print on top of its contents.

```
\ex\rightcomment{\textit{X-bar}}
\begin{tikzpicture}[mytree, trim left=-2em]
\node (top) {XP}
  child {node {YP}}
  child {node {X'}}
    child {node {X}}
    child {node {ZP}}};
\end{tikzpicture}
\xe
\ex\rightcomment{\textit{X-bar schema diagram}}
\begin{tikzpicture}[mytree, trim left=-2em]
\node (top) {XP}
  child {node {YP}}
  child {node {X'}}
    child {node {X}}
    child {node {ZP}}};
\end{tikzpicture}
\xe
```



Getting the right comment to fit with its example content is mostly an exercise in saying as little as possible while still being informative rather than fiddling with spacing and tweaking sizes.

A short, simple comment like “X-bar” should be enough for readers to connect the example with some discussion in surrounding paragraphs. I have sometimes used notional feature sets like “[+neg, -pl]” or “[−raise, +agree]” to capture multiple contrasts between examples. If you find that a short comment is not enough to identify an example without overprinting on your data then you probably need more examples, more discussion, or both.

### 4.3 Tables with drawings in examples

The ExPex documentation details the construction of tables inside of numbered examples. The documentation uses plain T<sub>E</sub>X which many L<sup>A</sup>T<sub>E</sub>X users are unfamiliar with. Trying to substitute L<sup>A</sup>T<sub>E</sub>X tables is often underwhelming because L<sup>A</sup>T<sub>E</sub>X expects these to be placed in floats and so is rather inconsiderate about spacing and alignment issues. It turns out that T<sub>E</sub>X tables are actually quite simple, even simpler than those provided by L<sup>A</sup>T<sub>E</sub>X, but their syntax is more abbreviated. In this section I introduce some basics of small T<sub>E</sub>X table design for numbered examples and then show how these can be combined with TikZ to produce very sophisticated presentations of data.

A T<sub>E</sub>X table is specified with the `\halign` command. It takes a single argument which is basically a set of logical lines ending with `\cr` (carriage return). I say ‘logical’ because the actual line endings in the source code are largely irrelevant, and instead the logical line is a sequence of characters ending in `\cr` that forms one horizontal row of an `\halign` table.

```
\halign{%
#\quad\hfil%
&#\quad\hfil%
&#\hfil\cr
first column & second column & third column\cr
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}
```

---

first column	second column	third column
row two col 1	row two col 2	row two col 3
row 3 col 1	row 3 col 2	row 3 col 3

The first logical line of an `\halign` is the table formatting specification, or ‘header’. This spells out how each column of the table will be formatted. The header can be all on a single line of code, but I prefer to break it up into one code line for each column. Columns are separated by `&` both in the header and the body of the table. In the header the contents of the column are indicated by the placeholder `#` (compare the numbered placeholders like `#1` and `#2` in commands). Thus the statement `#\quad\hfil` says to print the contents of the column and follow it with a `\quad` (short for `\hspace{1em}`) and an `\hfil`. This latter command is a rubber space, one which will grow horizontally (h) to fill (fil) the rest of the column.

The example above specifies three columns with right alignment by filling the right end of each column with the rubber `\hfil`. Note that the final column does not have a `\quad` since there is no following columns to space out from it. You could add a `\quad` and it will be there, but you won’t see it, so it’s better to leave it out and avoid surprising yourself with otherwise invisible space later.

Right alignment of the columns is as simple as moving the `\hfil` to the other side of the `#`. The example below illustrates this, swapping the positions of the two in the header lines.

```
\halign{%
\hfil#\quad%
&\hfil#\quad%
&\hfil#\cr
first column & second column & third column\cr
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}
```

---

first column	second column	third column
row two col 1	row two col 2	row two col 3
row 3 col 1	row 3 col 2	row 3 col 3

Centre alignment is achieved with an `\hfil` on both sides of the `#`.

```
\halign{%
\hfil#\hfil\quad%
&\hfil#\hfil\quad%
&\hfil#\hfil\cr
first column & second column & third column\cr
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}
```

---

first column	second column	third column
row two col 1	row two col 2	row two col 3
row 3 col 1	row 3 col 2	row 3 col 3

Arbitrary text can be inserted in the header just as well as formatting commands.

```
\halign{%
#\hfil\quad+\quad%
&#\hfil\quad\rightarrow\quad%
&#\hfil\cr
first column & second column & third column\cr
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}
```

---

first column	+ second column	→ third column
row two col 1	+ row two col 2	→ row two col 3
row 3 col 1	+ row 3 col 2	→ row 3 col 3

To draw a horizontal rule across the whole table the rule needs to be contained inside a `\noalign` command. This tells  $\TeX$  that the table alignment is temporarily suspended.



```

\halign{%
#\hfil\quad+\quad%
&#\hfil\quad\rightarrow\quad%
&#\hfil\cr
first column & second column & third column\cr
\noalign{\hrule}
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}

```

---

first column	+	second column	→	third column
row two col 1	+	row two col 2	→	row two col 3
row 3 col 1	+	row 3 col 2	→	row 3 col 3

To improve that we can add a `\smallskip` before and after the rule which will insert a small amount of vertical space between the row above and the row below.

```

\halign{%
#\hfil\quad+\quad%
&#\hfil\quad\rightarrow\quad%
&#\hfil\cr
first column & second column & third column\cr
\noalign{\smallskip\hrule\smallskip}
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}

```

---

first column	+	second column	→	third column
row two col 1	+	row two col 2	→	row two col 3
row 3 col 1	+	row 3 col 2	→	row 3 col 3

The spacing looks odd here because  $\text{T}_{\text{E}}\text{X}$  is inserting its usual `\baselineskip` between each line, and the `\hrule` is appearing just before it is inserted. We can suppress the `\baselineskip` using a special command `\offinterlineskip`. Then we can control the spacing explicitly by inserting a strut at the beginning of the first column.

```

\offinterlineskip\halign{%
\strut#\hfil\quad+\quad%
&#\hfil\quad→\quad%
&#\hfil\cr
first column & second column & third column\cr
\noalign{\smallskip\hrule\smallskip}
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}

```

---

first column	+	second column	→	third column
row two col 1	+	row two col 2	→	row two col 3
row 3 col 1	+	row 3 col 2	→	row 3 col 3

ExPex is perfectly willing to accept this as the contents of an `\ex ... \xe`.

```

\ex\offinterlineskip\halign{%
\strut#\hfil\quad+\quad%
&#\hfil\quad→\quad%
&#\hfil\cr
first column & second column & third column\cr
\noalign{\smallskip\hrule\smallskip}
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}
\xe

```

---

(23)

first column	+	second column	→	third column
row two col 1	+	row two col 2	→	row two col 3
row 3 col 1	+	row 3 col 2	→	row 3 col 3

That's not exactly what we were expecting, however. To repair this we can place the whole table inside a `\vtop` which will create a box aligned to its surroundings at the top of the box.

```

\ex\vtop{\offinterlineskip\halign{%
\strut#\hfil\quad+\quad%
&#\hfil\quad→\quad%
&#\hfil\cr
first column & second column & third column\cr
\noalign{\smallskip\hrule\smallskip}
row two col 1 & row two col 2 & row two col 3\cr
row 3 col 1 & row 3 col 2 & row 3 col 3\cr
}}
\ex

```

---

(24)	<table style="border-collapse: collapse; margin: auto;"> <tr> <td style="padding: 0 10px;">first column</td> <td style="padding: 0 10px;">+</td> <td style="padding: 0 10px;">second column</td> <td style="padding: 0 10px;">→</td> <td style="padding: 0 10px;">third column</td> </tr> <tr> <td style="padding: 0 10px;">row two col 1</td> <td style="padding: 0 10px;">+</td> <td style="padding: 0 10px;">row two col 2</td> <td style="padding: 0 10px;">→</td> <td style="padding: 0 10px;">row two col 3</td> </tr> <tr> <td style="padding: 0 10px;">row 3 col 1</td> <td style="padding: 0 10px;">+</td> <td style="padding: 0 10px;">row 3 col 2</td> <td style="padding: 0 10px;">→</td> <td style="padding: 0 10px;">row 3 col 3</td> </tr> </table>	first column	+	second column	→	third column	row two col 1	+	row two col 2	→	row two col 3	row 3 col 1	+	row 3 col 2	→	row 3 col 3
first column	+	second column	→	third column												
row two col 1	+	row two col 2	→	row two col 3												
row 3 col 1	+	row 3 col 2	→	row 3 col 3												

Now we're totally ready for some TikZ magic. Using our previous experience with TikZ in other numbered examples, we know that we will need to tell TikZ to remember picture, specify some nodes with our `\ND` command, and then finally draw with those nodes in a `tikzpicture` environment at the end of the `\ex ... \xe`. Making this mess look nice is left as an exercise for the reader.

```

\ex\tikzsetup%
\vtop{\offinterlineskip\halign{%
\strut#\hfil\quad+\quad%
&#\hfil\quad→\quad%
&#\hfil\cr
first \ND{c1}{column} & second column & \ND{c3}{third} column\cr
\noalign{\smallskip\hrule\smallskip}
row two col 1 & row two col \ND{r2c2}{2} & row two col \ND{r2c3}{3}\cr
row 3 col 1 & row 3 col \ND{r3c2}{2} & row 3 col \ND{r3c3}{3}\cr
}}%
\begin{tikzpicture}[overlay]
\draw [dotted] (c1.north east)
.. controls +(north east:1em) and +(north west:1em)
.. (c3.north west);
\draw (r2c2.south) |- (r3c2.east);
\draw (r2c3) to[out=0, in=0] (r3c3);
\end{tikzpicture}
\ex

```

---

(25)	<table style="border-collapse: collapse; margin: auto;"> <tr> <td style="padding: 0 10px;">first column</td> <td style="padding: 0 10px;">+</td> <td style="padding: 0 10px;">second column</td> <td style="padding: 0 10px;">→</td> <td style="padding: 0 10px;">third column</td> </tr> <tr> <td style="padding: 0 10px;">row two col 1</td> <td style="padding: 0 10px;">+</td> <td style="padding: 0 10px;">row two col 2</td> <td style="padding: 0 10px;">→</td> <td style="padding: 0 10px;">row two col 3</td> </tr> <tr> <td style="padding: 0 10px;">row 3 col 1</td> <td style="padding: 0 10px;">+</td> <td style="padding: 0 10px;">row 3 col 2</td> <td style="padding: 0 10px;">→</td> <td style="padding: 0 10px;">row 3 col 3</td> </tr> </table>	first column	+	second column	→	third column	row two col 1	+	row two col 2	→	row two col 3	row 3 col 1	+	row 3 col 2	→	row 3 col 3
first column	+	second column	→	third column												
row two col 1	+	row two col 2	→	row two col 3												
row 3 col 1	+	row 3 col 2	→	row 3 col 3												