Arborly

A library for producing beautiful syntax tree graphs.

Version 0.3.1 2025-05-15 MIT

Table of Contents

I Usage 2

I.1 Importing the Package 2

I.2 Building a Syntax Tree 2

II Arguments 3

III Styling and Attributes 5

III.1 Hierarchy 5

III.2 Defaults 6

III.3 Effects 6

III.3.1 Align 6

III.3.2 Align-Content 7

III.3.3 Triangle 7

III.3.4 Lines 8

III.3.5 Anchor 9

III.3.6 Text 10

III.3.7 Padding 11

III.3.8 Name 11

III.3.9 Fit 12

IV Examples 13

IV.1 The Quick Brown Fox 14

IV.2 Sphinx of Black Quartz 15

IV.3 Math Nodes 16

IV.4 Content Nodes 17

IV.5 Tight 18

IV.6 Long Sentence 19

IV.7 Deep Sentence 20

Part I Usage

I.1 Importing the Package

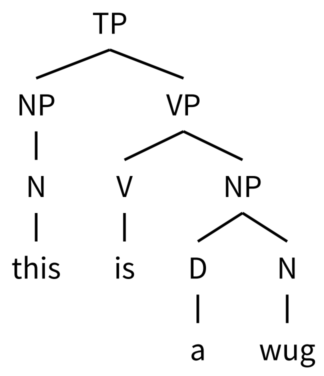
```
#import "@preview/arborly:0.3.1": *
```

I.2 Building a Syntax Tree

Technically each node is some content optionally followed by a number of bracket-enclosed nodes. For example: `content [node] [node]`. This is so that you can place a node into the body position of tree, and all nodes appear bracket-enclosed without needing an additional layer. Which is to say, you get to type `#tree[A[B]]` instead of `#tree[[A[B]]]`.

To give a demonstration:

```
#tree[TP
  [NP
    [N [this]]
  ]
  [VP
    [V [is]]
    [NP
      [D [a]]
      [N [wug]]
    ]
  ]
]
```



Part II Arguments

```
#a(..⟨args⟩)
```

Used for adding attributes to nodes. See the Styling and Attributes section for more details.

```
#tree(
  ⟨style⟩: (:),
  ⟨vertical-snapping-threshold⟩: 2pt,
  ⟨vertical-gap⟩: 8mm,
  ⟨horizontal-gap⟩: 7mm,
  ⟨code⟩: none
)[body] → content
```

Generate a syntax tree

Argument

`⟨style⟩: (:)` — `dictionary`

A root style dictionary that is inherited by all nodes. Any more specific configuration within the tree takes precedence.

Argument

`⟨vertical-snapping-threshold⟩: 2pt` — `length`

If the height of a node's content differs from regular text by less than this amount, it will be vertically spaced as though it had the same height.

This is not useful if most nodes are significantly larger than simple text (eg. being surrounded by `rect`).

Argument

`⟨vertical-gap⟩: 8mm` — `length`

The vertical gap between nodes

Argument

`⟨horizontal-gap⟩: 7mm` — `length`

The horizontal gap between nodes

Argument

`⟨code⟩: none`

A code block to be inserted into the cetz canvas after the syntax tree. It can be used for drawing arrows between nodes. Remember to name nodes using `#a` in order to reference them. See Styling and Arguments for more.

Argument

⟨`body`⟩ `content`

The tree’s structure denoted using bracket-enclosed values, as described in Building a Syntax Tree

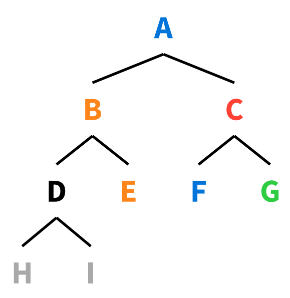
Part III Styling and Attributes

The attribute function `#a` is exposed for two purposes: styling the syntax tree, and naming nodes for usage with injected cetz code. A fallback set of attributes may be provided to the `tree` function.

III.1 Hierarchy

Attributes directly in a node take the highest precedence, followed by those in the `inherit` key of its parent's attributes, and so on until the root node. If none of those places specify the key, it goes to the values in the `style` argument, and then the default value. Here's an example with edge cases:

```
#show text: strong
#tree(
  style: (text: (fill: blue))
)[
  A
  [B #a(inherit: (text: (fill: orange)))
    [D #a(text: (fill: black), inherit: (text: (fill: gray)))
      [H] [I]
    ]
    [E]
  ]
  [C #a(text: (fill: red))
    [F]
    [G #a(text: (fill: green))]
  ]
]
```



Note that F is blue, since the red styling of C was not put under `inherit`. Whereas E is orange just like B, since it was set to be inherited.

It is also possible to specify both node-specific and inherited styles at the same time, in which case the node-specific styles will take precedence for that node. In the above example that's used for setting D to black while its children are gray.

III.2 Defaults

This is the set of default attributes. They are used when a style is not specified for a node or inherited from any of its ancestors.

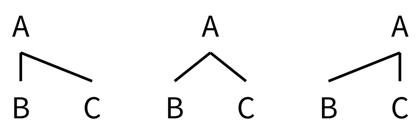
```
(
  align: center,
  align-content: center,
  triangle: false,
  parent-line: (:),
  child-lines: (:),
  parent-anchor: "north",
  child-anchor: "south",
  text: (:),
  padding: 0.5em,
  name: none,
  fit: "tight",
)
```

III.3 Effects

III.3.1 Align

Accepts an alignment of `left`, `center`, or `right`. In future might support an auto, acting like center but snapping in some situations. Currently does not.

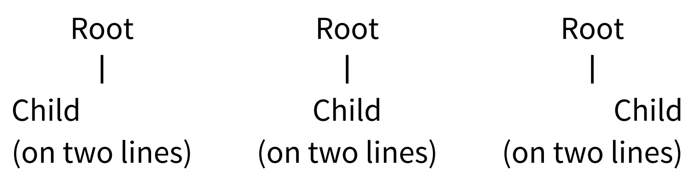
```
#grid(
  gutter: 1em,
  columns: 3,
  ..(left, center, right).map(alignment => {
    tree(style: (align: alignment))[A[B][C]]
  })
)
```



III.3.2 Align-Content

Shorthand for wrapping nodes in `#align(alignment)[...]`

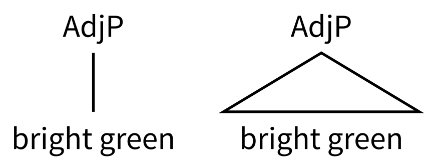
```
#grid(
  gutter: 1em,
  columns: 3,
  ..(left, center, right).map(alignment => {
   tree(style: (align-content: alignment))[Root[Child \ (on two lines)]]
  })
)
```



III.3.3 Triangle

This is commonly used to indicate that the content could be further broken down.

```
#grid(
  gutter: 1em,
  columns: 2,
  ..(false, true).map(boolean => {
    tree(vertical-gap: 1.2cm)[
      AdjP
      [bright green #a(triangle: boolean)]
    ]
  })
)
```

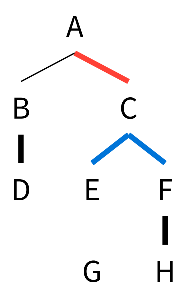


III.3.4 Lines

Used to style either the line to a node’s parent, or the lines to its children. If both are set then the options are merged, with `parent-line`’s configuration taking precedence as it is more specific (a node only has one line to its parent).

These options are equivalent to the options for styling `cetz.draw.line`

```
#tree[
  A #a(inherit: (child-lines: (stroke: (thickness: 2pt))))
  [B #a(parent-line: (stroke: (thickness: 0.5pt)))
    [D]
  ]
   [C  #a(parent-line:  (stroke:  (paint:  red)),  child-lines:  (stroke:
(paint: blue)))
    [E #a(child-lines: (stroke: none))
      [G]
    ]
    [F [H]]
  ]
]
```

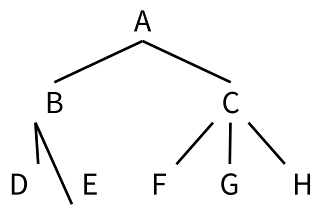


Note that the style merging can only merge dictionaries, so to merge you will need to use more explicit syntax. For example, C’s parent-line would have overridden the thickness if A had simply used `(stroke: 2pt)`.

III.3.5 Anchor

The cetz anchor for a connecting line's ends to attach to. A value of none indicates the center.

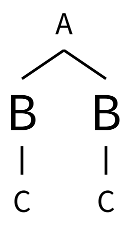
```
#tree(
  style: (padding: 0.3em)
)[
  A
  [B #a(child-anchor: "south-west")
    [D #a(parent-anchor: "north-east")]
    [E #a(parent-anchor: "south-west")]
  ]
   [C #a(parent-anchor: "north", inherit: (parent-anchor: none, child-
anchor: none))
    [F] [G] [H]
  ]
]
```



III.3.6 Text

These are equivalent to the named arguments of the `text` function, which you would usually customize with `#set text(...)`. Unfortunately set rules cannot be used within the tree's body, unless they are tightly scoped to only the content of one node like so.

```
#tree[A
  [
    #[
      #set text(size: 20pt)
      B
    ]
    [C]
  ]
  [B #a(text: (size: 20pt))
  [C]
  ]
]
```



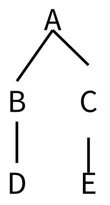
This is very verbose, especially when used for several nodes. So the provided parameter (used on for the right branch of this tree) is exposed (and as usual can be inherited if specified).

However a similar option is not exposed for all elements that support `set`, so this is a useful trick to know.

III.3.7 Padding

This is simply passed on to the padding argument of cetz.draw.content

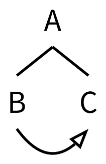
```
#tree(
  style: (padding: none)
)[A
  [B #a(inherit: (padding: 0.3em))
    [D]
  ]
  [C #a(padding: 0.8em)
    [E]
  ]
]
```



III.3.8 Name

This is used as the name for cetz.draw.content if it is set to a non-none value. It's crucial to use this attribute to label nodes that you plan to reference in cetz code, such as when drawing arrows between nodes. Here is an example.

```
#let arrow = {
  import "@preview/cetz:0.3.4"
  cetz.draw.set-style(mark: (end: ">"))
  cetz.draw.bezier(
    "bee.south", "see.south",
    (rel: (-90deg, 1cm), to: ("bee", 50%, "see"))
  )
}
#tree(code: arrow)[A
  [B #a(name: "bee")]
  [C #a(name: "see")]
]
```



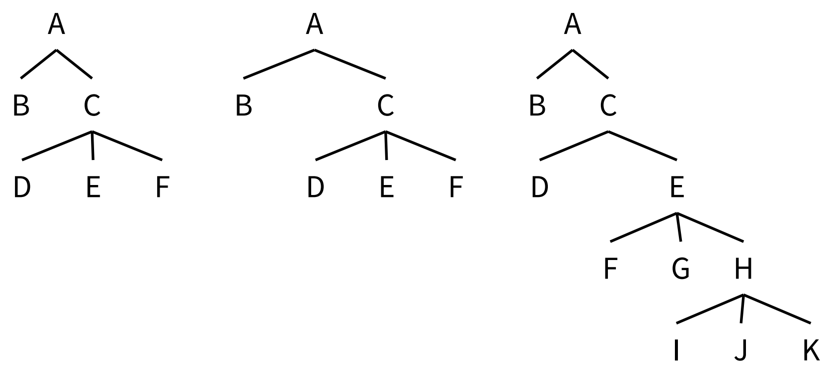
III.3.9 Fit

This determines which horizontal positional algorithm is used. The default is `"tight"`, which allows nodes to hang over other nodes for more compact spacing. This is valuable for larger syntax trees which otherwise get very spread out.

Currently the only other style is `"band"`, in which each node has nothing under it (other than its children of course).

It's even possible to use multiple fits in one tree, as in the third case. But be careful as there may be malfunctioning edge cases I haven't found.

```
#grid(gutter: 1em, columns: 3,
  ..("tight", "band").map(fit => {
    tree(style: (fit: fit))[A
      [B]
      [C [D] [E] [F]]
    ]
  }),
  tree[A
    [B]
    [C #a(inherit: (fit: "band"))
      [D]
      [E
        [F] [G] [H #a(inherit: (fit: "tight"))
          [I]
          [J]
          [K]
        ]
      ]
    ]
  ]
)
```

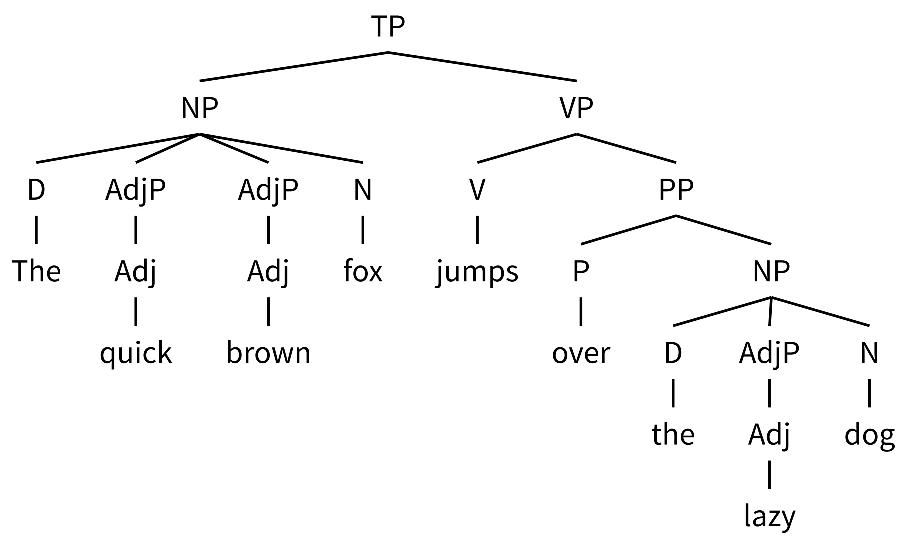


Part IV Examples

Note that I am not a linguist, so these analyses may be wrong. There are intended to show the appearance of arborly syntax trees. Please create a github issue if you see a mistake, or would like to submit another example.

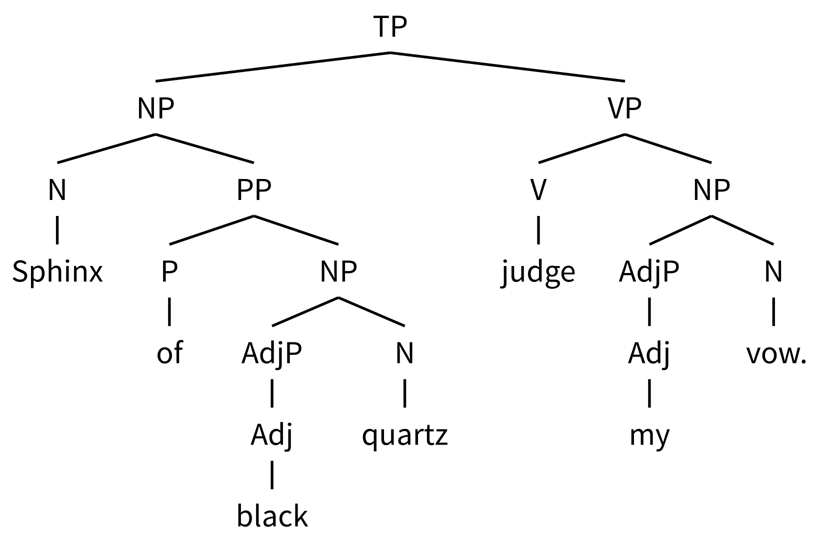
IV.1 The Quick Brown Fox

```
#tree[TP
  [NP
    [D [The]]
    [AdjP
      [Adj [quick]]
    ]
    [AdjP
      [Adj [brown]]
    ]
    [N [fox]]
  ]
  [VP
    [V [jumps]]
    [PP
      [P [over]]
      [NP
        [D [the]]
        [AdjP
          [Adj [lazy]]
        ]
        [N [dog]]
      ]
    ]
  ]
]
```



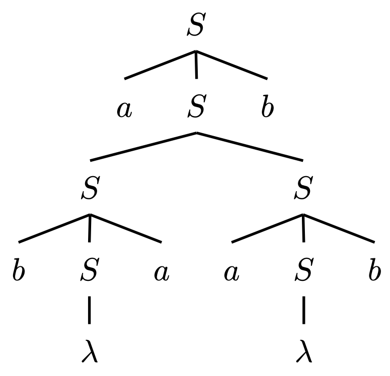
IV.2 Sphinx of Black Quartz

```
#tree(
  style: (fit: "band")
)[TP
  [NP
    [N [Sphinx]]
    [PP
      [P [of]]
      [NP
        [AdjP
          [Adj [black]]
        ]
        [N [quartz]]
      ]
    ]
  ]
  [VP
    [V [judge]]
    [NP
      [AdjP
        [Adj [my]]
      ]
      [N [vow.]]
    ]
  ]
]
```



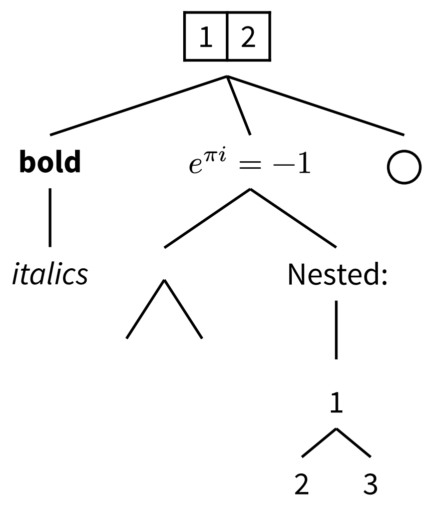
IV.3 Math Nodes

```
#tree[$S$
  [$a$]
  [$S$
    [$S$
      [$b$]
      [$S$ [$lambda$]]
      [$a$]
    ]
    [$S$
      [$a$]
      [$S$ [$lambda$]]
      [$b$]
    ]
  ]
  [$b$]
]
```



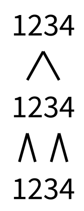
IV.4 Content Nodes

```
#tree(
  vertical-gap: 1.2cm,
  horizontal-gap: 1cm,
)[#table(columns: 2, [1],[2])
  [*bold* [_italics_]]
  [$e^(pi i) = -1$
    [[] []]
    [Nested:
    [#tree[1 [2] [3]]]
    ]
  ]
  [#circle(radius: 0.5em)]
]
```



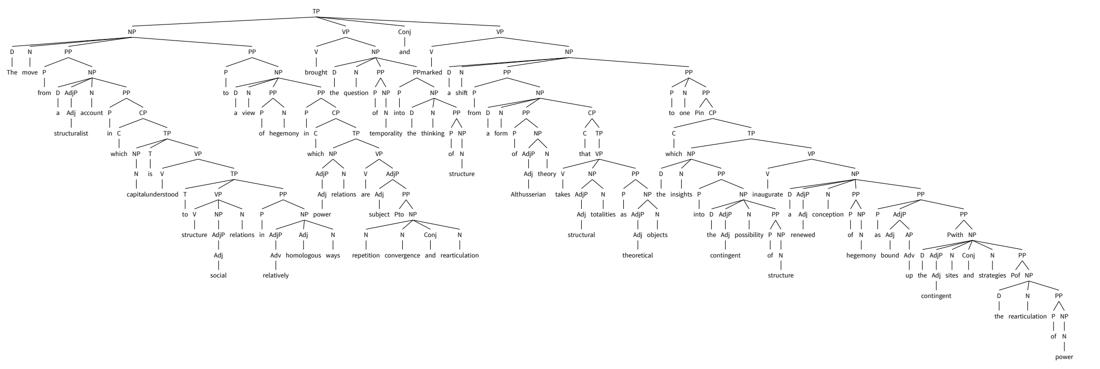
IV.5 Tight

```
#tree(
  horizontal-gap: 0pt,
)[1234
  [12 [1] [2]]
  [34 [3] [4]]
]
```



IV.6 Long Sentence

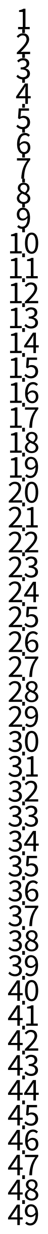
```
#let sentence = [TP[NP[D[The]][N[move]][PP[P[from]][NP[D[a]]
[AdjP[Adj[structuralist]]][N[account]][PP[P[in]][CP[C[which]]
[TP[NP[N[capital]]][T[is]][VP[V[understood]][TP[T[to]]
[VP[V[structure]][NP[AdjP[Adj[social]]]][N[relations]]][PP[P[in]]
[NP[AdjP[Adv[relatively]]][Adj[homologous]][N[ways]]]]]]]]]]]
[PP[P[to]][NP[D[a]][N[view]][PP[P[of]][N[hegemony]]][PP[P[in]]
[CP[C[which]][TP[NP[AdjP[Adj[power]]][N[relations]]][VP[V[are]]
[AdjP[Adj[subject]][PP[Pto][NP[N[repetition]][N[convergence]]
[Conj[and]][N[rearticulation]]]]]]]]]]]]][VP[V[brought]][NP[D[the]]
[N[question]][PP[P[of]][NP[N[temporality]]]][PP[P[into]][NP[D[the]]
[N[thinking]][PP[P[of]][NP[N[structure]]]]]]]][Conj[and]]
[VP[V[marked]][NP[D[a]][N[shift]][PP[P[from]][NP[D[a]][N[form]]
[PP[P[of]][NP[AdjP[Adj[Althusserian]]][N[theory]]]][CP[C[that]]
[TP[VP[V[takes]][NP[AdjP[Adj[structural]]][N[totalities]]][PP[P[as]]
[NP[AdjP[Adj[theoretical]]][N[objects]]]]]]]]][PP[P[to]][N[one]]
[PP[Pin][CP[C[which]][TP[NP[D[the]][N[insights]][PP[P[into]]
[NP[D[the]][AdjP[Adj[contingent]]][N[possibility]][PP[P[of]]
[NP[N[structure]]]]]]]][VP[V[inaugurate]][NP[D[a]][AdjP[Adj[renewed]]]
[N[conception]][PP[P[of]][NP[N[hegemony]]]][PP[P[as]][AdjP[Adj[bound]]
[AP[Adv[up]]]][PP[Pwith][NP[D[the]][AdjP[Adj[contingent]]][N[sites]]
[Conj[and]][N[strategies]][PP[Pof][NP[D[the]][N[rearticulation]]
[PP[P[of]][NP[N[power]]]]]]]]]]]]]]]]]]]

#scale(
  21%,
  reflow: true,
  tree(
    horizontal-gap: 3mm,
    vertical-gap: 1cm,
    sentence
  )
)
```



IV.7 Deep Sentence

As of writing the node depth limit is 61, but this is reduced the deeper the call stack where you use the tree function.

```
#tree(style: (padding: 0.0), vertical-gap: 0.05cm)[1[2[3[4[5[6[7[8[9
[10[11[12[13[14[15[16[17[18[19[20[21[22[23[24[25[26[27[28[29[30[31[32
[33[34[35[36[37[38[39[40[41[42[43[44[45[46[47[48[49]]]]]]]]]]]]]]]]]
]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
```


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49