

CeTZ

ein Typst
Zeichenpaket

Johannes Wolf
fenjalien

Version 0.2.0

1 Introduction	3	4.10 Function	32
2 Usage	3	5 Utility	32
2.1 Argument Types	3	5.1.1 for-each-anchor	32
2.2 Anchors	3	6 Libraries	33
3 Draw Function Reference	4	6.1 Tree	33
3.1 Canvas	4	6.1.1 tree	33
3.2 Styling	4	6.1.2 Node	35
3.3 Elements	5	6.2 Plot	35
3.3.1 line	5	6.2.1 Types	35
3.3.2 rect	6	6.2.2 add-anchor	35
3.3.3 arc	7	6.2.3 plot	36
3.3.4 circle	9	6.2.4 add	39
3.3.5 circle-through	10	6.2.5 add-fill-between	42
3.3.6 bezier	11	6.2.6 add-hline	43
3.3.7 bezier-through	12	6.2.7 add-vline	44
3.3.8 content	13	6.2.8 add-contour	45
3.3.9 catmull	14	6.2.9 add-boxwhisker	47
3.3.10 hobby	15	6.2.10 sample-fn	48
3.3.11 grid	16	6.2.11 sample-fn2	49
3.3.12 mark	17	6.2.12 Examples	50
3.4 Path Transformations	18	6.2.13 Styling	50
3.4.1 merge-path	18	6.3 Chart	51
3.4.2 group	19	6.3.1 Examples – Bar Chart	52
3.4.3 anchor	20	6.3.2 Examples – Column Chart	53
3.4.4 copy-anchors	20	6.3.3 boxwhisker	54
3.4.5 place-anchors	21	6.3.4 Styling	55
3.4.6 place-marks	22	6.4 Palette	55
3.4.7 intersections	22	6.4.1 new	56
3.5 Layers	23	6.4.2 List of predefined palettes	56
3.5.1 on-layer	23	6.5 Angle	56
3.6 Transformations	24	6.6 Decorations	57
3.6.1 translate	24	6.6.1 brace	57
3.6.2 set-origin	24	6.6.2 flat-brace	59
3.6.3 set-viewport	25	7 Advanced Functions	61
3.6.4 rotate	25	7.1 Coordinate	61
3.6.5 scale	26	7.1.1 resolve	61
3.7 Context Modification	26	7.2 Styles	62
3.7.1 set-ctx	26	7.2.1 resolve	62
3.7.2 get-ctx	27	7.2.2 Default Style	63
4 Coordinate Systems	27	8 Creating Custom Elements	64
4.1 XYZ	27	9 Internals	64
4.2 Previous	28	9.1 Context	64
4.3 Relative	28	9.2 Elements	64
4.4 Polar	28		
4.5 Barycentric	29		
4.6 Anchor	29		
4.7 Tangent	30		
4.8 Perpendicular	30		
4.9 Interpolation	31		

1 Introduction

This package provides a way to draw stuff using a similar API to [Processing](#) but with relative coordinates and anchors from [TikZ](#). You also won't have to worry about accidentally drawing over other content as the canvas will automatically resize. And remember: up is positive!

The name CeTZ is a recursive acronym for “CeTZ, ein Typst Zeichenpaket” (german for “CeTZ, a Typst drawing package”) and is pronounced like the word “Cats”.

2 Usage

This is the minimal starting point:

```
#import "@preview/cetz:0.2.0"
#cetZ.canvas({
  import cetZ.draw: *
  ...
})
```

Note that draw functions are imported inside the scope of the canvas block. This is recommended as draw functions override Typst's functions such as `line`.

2.1 Argument Types

Argument types in this document are formatted in monospace and encased in angle brackets `<>`. Types such as `<integer>` and `<content>` are the same as Typst but additional are required:

<coordinate> Any coordinate system. See Section 4.

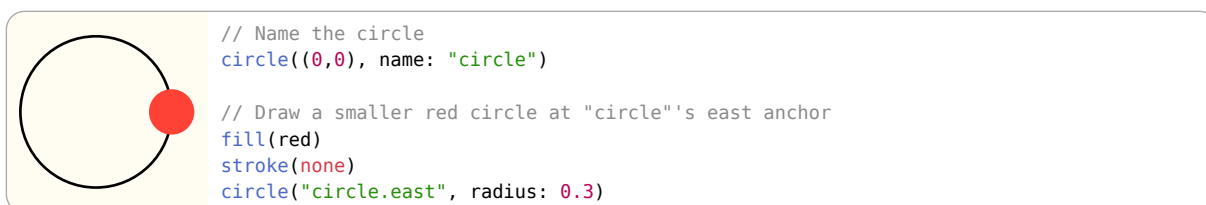
<number> `<integer>` or `<float>`

<style> Named arguments (or a dictionary if used for a single argument) of style key-values

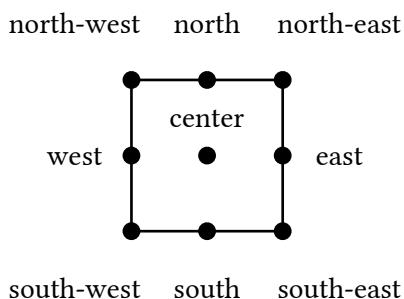
2.2 Anchors

Anchors are named positions relative to named elements.

To use an anchor of an element, you must give the element a name using the `name` argument.

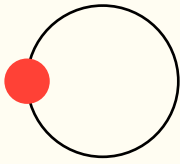


Group elements will have default anchors based on their axis aligned bounding box, they are:



Other elements will have their own anchors.

Elements can be placed relative to their own anchors if they have an argument called `anchor`:



```
// An element does not have to be named
// in order to use its own anchors.
circle((0,0), anchor: "west")

// Draw a smaller red circle at the origin
fill(red)
stroke(none)
circle((0,0), radius: 0.3)
```

3 Draw Function Reference

3.1 Canvas

`canvas`(background: `none`, length: `1cm`, debug: `false`, body)

background <color> (default: none)

A color to be used for the background of the canvas.

length <length> (default: 1cm)

Used to specify what 1 coordinate unit is.

debug <bool> (default: false)

Shows the bounding boxes of each element when `true`.

body

A code block in which functions from `draw.typ` have been called.

3.2 Styling

You can style draw elements by passing the relevant named arguments to their draw functions. All elements have stroke and fill styling unless said otherwise.

fill <color> or <none> (default: none)

How to fill the draw element.

stroke <none> or <auto> or <length> (default: black + 1pt)

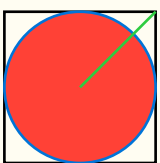
or <color> or <dictionary> or <stroke>

How to stroke the border or the path of the draw element. See Typst's line documentation for more details: <https://typst.app/docs/reference/visualize/line/#parameters-stroke>



```
// Draws a red circle with a blue border
circle((0, 0), fill: red, stroke: blue)
// Draws a green line
line((0, 0), (1, 1), stroke: green)
```

Instead of having to specify the same styling for each time you want to draw an element, you can use the `set-style` function to change the style for all elements after it. You can still pass styling to a draw function to override what has been set with `set-style`. You can also use the `fill()` and `stroke()` functions as a shorthand to set the fill and stroke respectively.

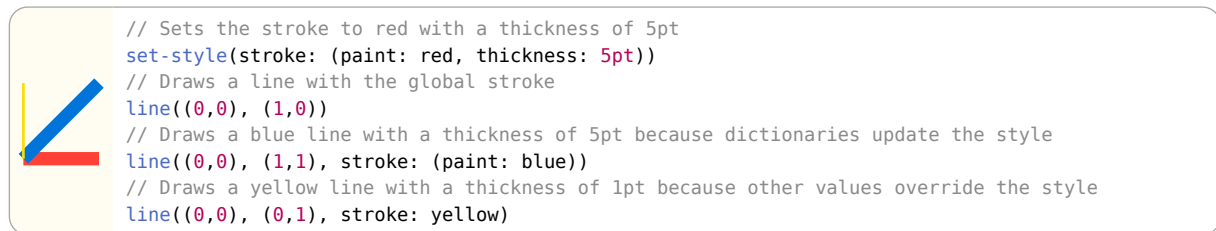


```
// Draws an empty square with a black border
rect((-1, -1), (1, 1))

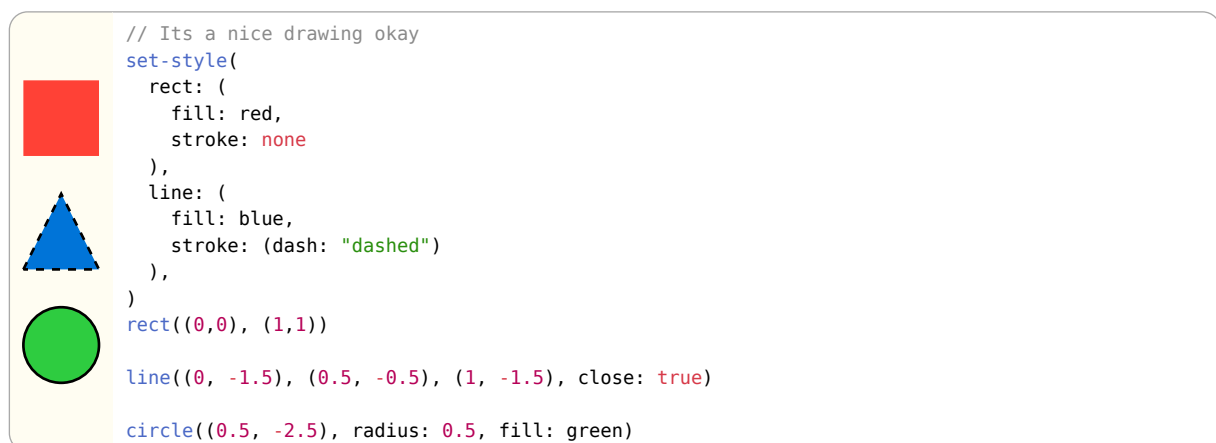
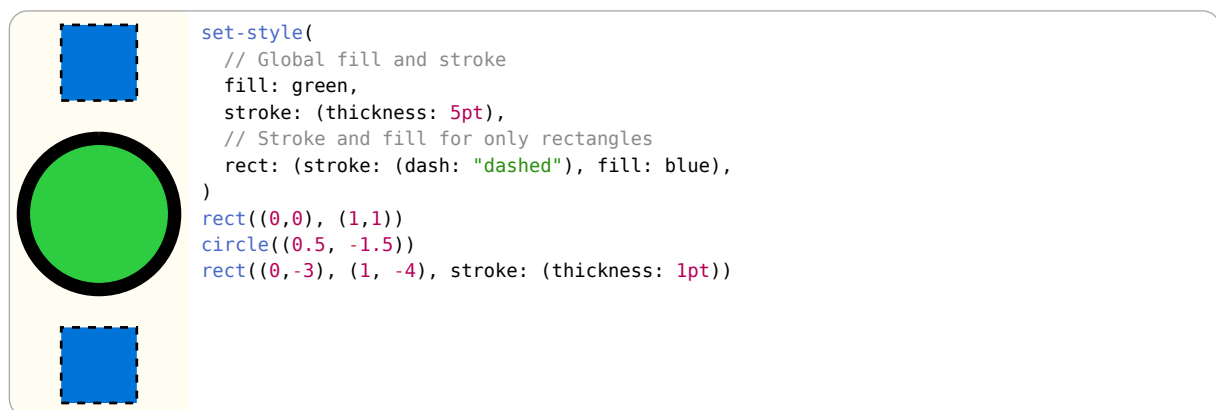
// Sets the global style to have a fill of red and a stroke of blue
set-style(stroke: blue, fill: red)
circle((0,0))

// Draws a green line despite the global stroke is blue
line((0, 0), (1,1), stroke: green)
```

When using a dictionary for a style, it is important to note that they update each other instead of overriding the entire option like a non-dictionary value would do. For example, if the stroke is set to (paint: red, thickness: 5pt) and you pass (paint: blue), the stroke would become (paint: blue, thickness: 5pt).



You can also specify styling for each type of element. Note that dictionary values will still update with its global value, the full hierarchy is function > element type > global. When the value of a style is auto, it will become exactly its parent style.



For the default styles of all elements see Section 7.2.2

3.3 Elements

3.3.1 line

Draw a line or a line-strip

Style Root line

Anchors

"start" The lines start position

"end" The lines start position

Style Root line**Parameters**

```
line(
  ..pts-style: coordinate style,
  close: bool,
  name: none string
)
```

..pts-style coordinate or style

Positional two or more coordinates to draw lines between. Accepts style key-value pairs.

close bool

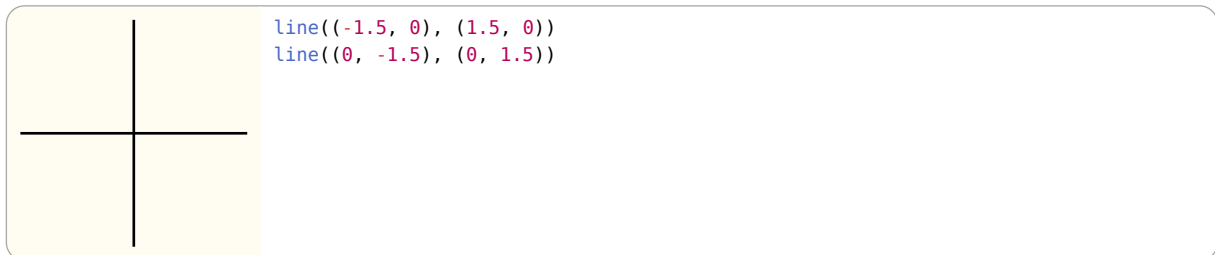
If true, the line-strip gets closed to form a polygon

Default: **false**

name none or string

Element name

Default: **none**

**Styling**

mark <dictionary> or <auto>

(default: **auto**)

The styling to apply to marks on the line, see mark

3.3.2 rect

Draw a rect between two coordinates

Style Root rect

Tip: To draw a rect with a specified size instead of two coordinates, use relative coordinates for the second: (rel: (<width>, <height>)).

Parameters

```
rect(
  a: coordinate,
  b: coordinate,
  name: none string,
  anchor,
  ..style: style
)
```

a `coordinate`

First coordinate

b `coordinate`

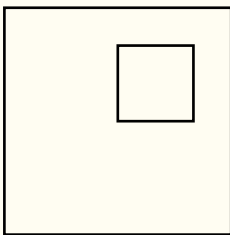
Second coordinate

name `none` or `string`

Element name

Default: `none`**anchor**Default: `none`**..style** `style`

Style key-value pairs



```
rect((0,0), (1,1))
rect((-1.5, 1.5), (1.5, -1.5))
```

3.3.3 arc

Draw a circular segment

Style Root arc

anchors

"center" The center of the arc**"arc-center"** Mid-point on the arc border**"chord-center"** Center of the chord**"origin"** Arc origin**"arc-start"** Arc start coordinate**"arc-end"** Arc end coordinate

Parameters

```
arc(
  position: coordinate,
  start: none angle,
  stop: none angle,
  delta: auto angle,
  name: none string,
  anchor,
  ..style: style
)
```

position coordinate

Position to place the arc at. If anchor is unset, this is the arcs start position.

start none or angle

Start angle

Default: auto

stop none or angle

Stop angle

Default: auto

delta auto or angle

Angle delta from either start or stop. Exactly two of the three angle arguments must be set.

Default: auto

name none or string

Element name

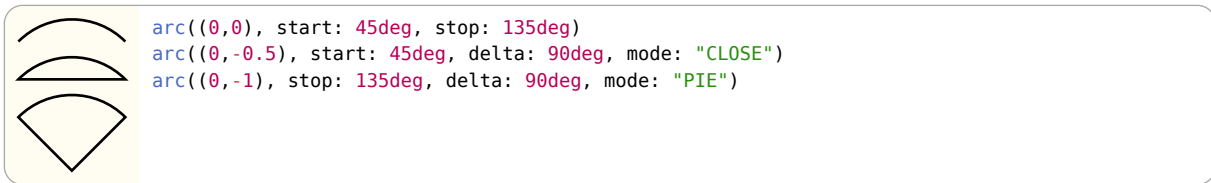
Default: none

anchor

Default: none

..style style

Style key-values



Styling

radius <number> or <array> (default: 1)

The radius of the arc. This is also a global style shared with circle!

mode <string> (default: "OPEN")

The options are "OPEN" (the default, just the arc), "CLOSE" (a circular segment) and "PIE" (a circular sector).

3.3.4 circle

Draw an ellipse

The radii of the ellipse can be set via the style key `radius`, which takes a number or a tuple of numbers for the x- and y-radius.

Style Root circle

Anchors

"center" The center of the ellipse

Parameters

```
circle(
  position: coordinate,
  name: none string,
  anchor: none string,
  ..style: style
)
```

position coordinate

Anchor position, by default this is the ellipses center

name none or string

Element name

Default: none

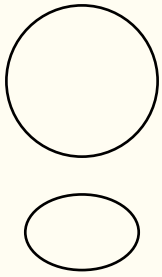
anchor none or string

Anchor to position the element relative to

Default: none

..style style

Style key-values



```
circle((0,0))
// Draws an ellipse
circle((0,-2), radius: (0.75, 0.5))
```

3.3.5 circle-through

Draw a circle through three coordinates

Style Root circle

anchors

"center" The center of the ellipse

Parameters

```
circle-through(
  a: coordinate,
  b: coordinate,
  c: coordinate,
  name: none string,
  anchor: none string,
  ..style: style
)
```

a coordinate

Coordinate a

b coordinate

Coordinate b

c coordinate

Coordinate c

name none or string

Element name

Default: none

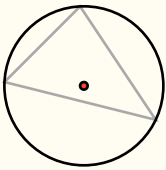
anchor none or string

Anchor to position the element relative to

Default: none

..style style

Style key-values



```
let (a, b, c) = ((0,0), (2,-.5), (1,1))
line(a, b, c, close: true, stroke: gray)
circle-through(a, b, c, name: "c")
circle("c.center", radius: .05, fill: red)
```

Styling**radius** <number> or <length> or <array of <number> or <length>> (default: 1)

The circle's radius. If an array is given an ellipse will be drawn where the first item is the x radius and the second item is the y radius. This is also a global style shared with arc!

3.3.6 bezier

Draw a quadratic or cubic bezier curve

anchors**ctrl-<n>** Nth control point (n is an integer starting at 0)**Style Root** bezier**Parameters**

```
bezier(
  start: coordinate,
  end: coordinate,
  ..ctrl-style: coordinate style,
  name: none string
)
```

start coordinate

Start position

end coordinate

End position (last coordinate)

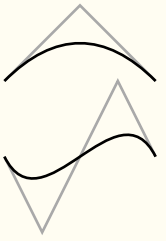
..ctrl-style coordinate or style

One or two control point coordinates. Accepts style key-value pairs.

name none or string

Element name

Default: none



```
let (a, b, c) = ((0, 0), (2, 0), (1, 1))
line(a, c, b, stroke: gray)
bezier(a, b, c)

let (a, b, c, d) = ((0, -1), (2, -1), (.5, -2), (1.5, 0))
line(a, c, d, b, stroke: gray)
bezier(a, b, c, d)
```

3.3.7 bezier-through

Draw a cubic bezier curve through a set of three points

See bezier for style and anchor details.

Parameters

```
bezier-through(
  start: coordinate,
  pass-through: coordinate,
  end: coordinate,
  name: none string,
  ..style: style
)
```

start coordinate

Start position

pass-through coordinate

Curve mid-point

end coordinate

End coordinate

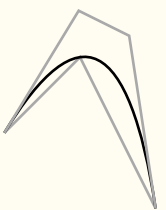
name none or string

Element name

Default: none

..style style

Style key-value pairs



```
let (a, b, c) = ((0, 0), (1, 1), (2, -1))
line(a, b, c, stroke: gray)
bezier-through(a, b, c, name: "b")

// Show calculated control points
line(a, "b.ctrl-0", "b.ctrl-1", c, stroke: gray)
```

3.3.8 content

Position typst content in the canvas

You can call the function with one or two coordinates:

- One coordinate `content((..), [..])`: The content gets placed at the coordinate
- Two coordinates `content((..), (..), [..])`: The content gets placed inside the rect between the two coordinates

Style Root content

Parameters

```
content(
  ..args-style: coordinate content,
  angle: angle coordinate,
  clip: bool,
  anchor: none string,
  name: none string
)
```

angle **angle** or **coordinate**

Rotation of the content. If a coordinate instead of an angle is used the angle between it and the contents first coordinate is used for rotation

Default: **0deg**

clip **bool**

If true, the content is placed inside a box that gets clipped

Default: **false**

anchor **none** or **string**

Anchor to position the content relative to. Defaults to the contents center.

Default: **none**

name **none** or **string**

Element name

Default: **none**


Hello World! `content((0,0), [Hello World!])`

To put text on a line you can let content calculate the angle between its position and a second coordinate by passing it to angle:

Text on a line

```
line((0, 0), (3, 1), name: "line")
content(("line.start", .5, "line.end"),
  angle: "line.end", padding: .1,
  [Text on a line], anchor: "south")
```

This example uses linear interpolated coordinates (*a*, *t*, *b*) to place the content at the center of the line, see Section 4.9.



This is
a long
text.

```
// Place content in a rect between two coordinates
content((0,0), (2,2), box(par(justify: false)[This is a long text.],
  stroke: 1pt, width: 100%, height: 100%, inset: 1em))
```

Styling

This draw element is not affected by fill or stroke styling.

padding <length> (default: 0pt)

3.3.9 catmull

Draw a Catmull-Rom curve through a set of points

The curves tension can be adjusted using the style key *tension*.

Anchors

"start" First point

"end" Last point

"pt-<n>" Nth point (*n* is an integer starting at 0)

Style Root catmull

Parameters

```
catmull(
  ..pts-style: coordinate style,
  close: bool,
  name: none string
)
```

..pts-style coordinate or style

List of points to run the curve through. Accepts style key-value pairs.

close bool

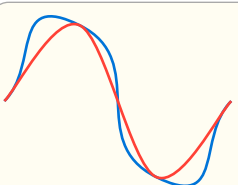
Auto-close the curve

Default: false

name none or string

Element name

Default: none



```
catmull((0,0), (1,1), (2,-1), (3,0), tension: .4, stroke: blue)
catmull((0,0), (1,1), (2,-1), (3,0), tension: .5, stroke: red)
```

3.3.10 hobby

Draw a Hobby curve through a set of points

The curves curlyness can be adjusted using the style key `omega`. The rho function can be set using the style key `rho`.

anchors

"start" First point

"end" Last point

"pt-<n>" Nth point (n is an integer starting at 0)

Style Root hobby

Parameters

```
hobby(
  ..pts-style: coordinate style,
  ta: auto array,
  tb: auto array,
  close: bool,
  name: none string
)
```

..pts-style coordinate or style

List of points to run the curve through. Accepts style key-value pairs.

ta auto or array

Outgoing tension at point.at(n) from point.at(n) to point.at(n+1). Length must be the length of points minus one

Default: auto

tb auto or array

Incoming tension at point.at(n+1) from point.at(n) to point.at(n+1). Length must be the length of points minus one:

Default: auto

close bool

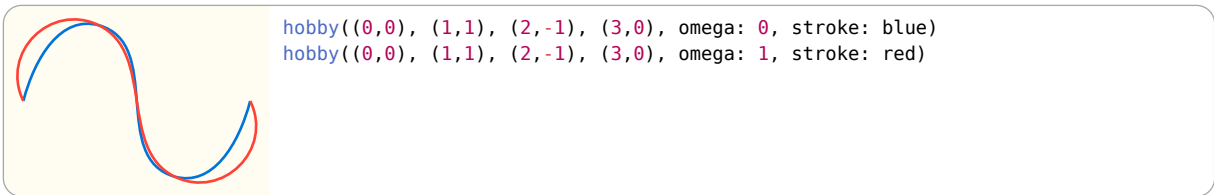
Auto-close the curve

Default: false

name none or string

Element name

Default: none



3.3.11 grid

Draw a grid between two coordinates

Style Root grid

Parameters

```
grid(
  from: coordinate,
  to: coordinate,
  step: number,
  name: none string,
  help-lines,
  ..style: style
)
```

from coordinate

Start coordinate

to coordinate

End coordinate

step number

Grid spacing in canvas units

Default: 1

name none or string

Element name

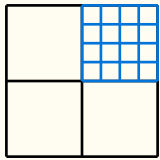
Default: none

help-lines

Default: false

..style style

Style key-value pairs



```
// Draw a grid
grid((0,0), (2,2))

// Draw a smaller blue grid
grid((1,1), (2,2), stroke: blue, step: .25)
```

3.3.12 mark

Draw a single mark pointing at a target coordinate

Style Root mark

Note: The size of the mark depends on its style values, not the distance between `from` and `to`, which only determine its orientation.

Parameters

```
mark(
  from: coordinate,
  to: coordinate,
  ..style: style
)
```

from coordinate

Starting position used for orientation calculation

to coordinate

The marks target position at which it points

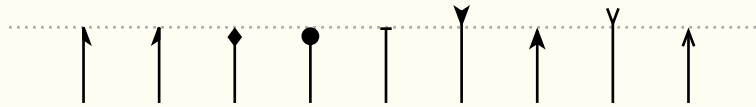
..style style

Style key-value pairs



```
mark((0,0), (1,0), symbol: ">", fill: black)
mark((0,0), (1,1), symbol: ">", scale: 3, fill: black)
```

Or as part of a line, using lines mark style key:



```
rotate(-90deg)
set-style(mark: (fill: black))
line((1, -1), (1, 9), stroke: (paint: gray, dash: "dotted"))
line((0, 8), (rel: (1, 0)), mark: (end: "left-harpoon"))
line((0, 7), (rel: (1, 0)), mark: (end: "right-harpoon"))
line((0, 6), (rel: (1, 0)), mark: (end: "<>"))
line((0, 5), (rel: (1, 0)), mark: (end: "o"))
line((0, 4), (rel: (1, 0)), mark: (end: "|"))
line((0, 3), (rel: (1, 0)), mark: (end: "<"))
line((0, 2), (rel: (1, 0)), mark: (end: ">"))

set-style(mark: (fill: none))
line((0, 1), (rel: (1, 0)), mark: (end: "<"))
line((0, 0), (rel: (1, 0)), mark: (end: ">"))
```

Styling

- symbol** <string> (default: >)
The type of mark to draw when using the mark function.
- start** <string>
The type of mark to draw at the start of a path.
- end** <string>
The type of mark to draw at the end of a path.
- size** <number> (default: 0.15)
The size of the marks.
- angle** <angle> (default: 45deg)
Angle for triangle style marks (“<” and “>”)

3.4 Path Transformations

3.4.1 merge-path

Merge two or more paths by concatenating their elements

Note that the draw direction of the joined elements is important to be continuous, as jumps get connected by straight lines!

Parameters

```
merge-path(
  body: drawables,
  close: bool,
  name: none string,
  ..style: style
)
```

body drawables

Drawables to be merged into one

close `bool`

Auto-close the path (by a straight line)

Default: `false`**name** `none` or `string`

Element name

Default: `none`**..style** `style`

Style key-value pairs



```
// Merge two different paths into one
merge-path({
  line((0, 0), (1, 0))
  bezier((0, 0), (0, 0), (1,1), (0,1))
}, fill: white)
```

3.4.2 group

Group one or more elements.

All state changes but anchors remain group local. That means, that changes to the transformation matrix (i.e. `rotate(...)`) are applied to the groups child elements only.

A group creates compass anchors of its axis aligned bounding box, that are accessible using the following names: north, north-east, east, south-east, south, south-west, west and north-west. All anchors created using `anchor(<name>, <position>)` are also accessible, both in the group (by name "anchor") and outside the group (by the groups name + the anchor name, i.e. "group.anchor")

Parameters

```
group(
  name: none string,
  anchor: none string,
  ..body-style: drawables style
)
```

name `none` or `string`

Group element name

Default: `none`

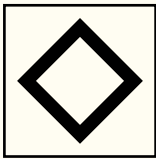
anchor `none` or `string`

Anchor of the group to position itself relative to. This is done by calculating the distance to the groups "default" anchor and translating all grouped elements by that distance. The groups "default" anchor gets set to the groups "center" anchor, but the user can supply it's own by calling `anchor("default", ...)`.

Default: `none`

..body-style `drawables` or `style`

Requires one positional parameter, the list of the groups child elements. Accepts style key-value pairs.



```
// Create group
group({
  stroke(5pt)
  scale(.5); rotate(45deg)
  rect((-1,-1),(1,1))
})
rect((-1,-1),(1,1))
```

3.4.3 anchor

Create a new named anchor.

An anchor is a named position insides a group, that can be referred to by other positions. Anchors can not be defined outsides groups, trying so will emit an error.

Setting an anchor which name already exists overwrites the previously defined one.

Parameters

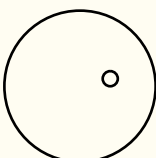
```
anchor(
  name: string,
  position: position
)
```

name `string`

Anchor name

position `position`

The anchors position



```
group(name: "g", {
  circle((0,0))
  anchor("x", (.4,.1))
})
circle("g.x", radius: .1)
```

3.4.4 copy-anchors

Copy multiple anchors from one element into the current group.

Parameters

```
copy-anchors(
  element: string,
  filter: auto array
)
```

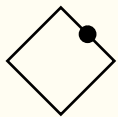
element string

Element name

filter auto or array

If set to an array, the function copies only anchors that are both in the source element and the filter list

Default: auto



```
group(name: "g", {
  rotate(45deg)
  rect((0,0), (1,1), name: "r")
  copy-anchors("r")
})
circle("g.north", radius: .1, fill: black)
```

3.4.5 place-anchors

Place multiple anchors along a path

Parameters

```
place-anchors(
  path: drawable,
  ..anchors: array,
  name
)
```

path drawable

Single drawable

..anchors array

List of anchor dictionaries of the form (pos: <float>, name: <string>), where pos is a relative position on the path from 0 to 1.

- name: (auto,string): If auto, take the name of the passed drawable. Otherwise sets the elements name

name

Default: auto

```
place-anchors(name: "demo",
  bezier((0,0), (3,0), (1,-1), (2,1)),
```

```

        (name: "a", pos: .15),
        (name: "mid", pos: .5))
circle("demo.a", radius: .1, fill: black)
circle("demo.mid", radius: .1, fill: black)

```

3.4.6 place-marks

Place one or more marks along a path

Mark items must get passed as positional arguments. A mark-item is an dictionary of the format: (mark: "<symbol>", pos: <float>), where the position pos is a relative position from 0 to 1 along the path.

Parameters

```

place-marks(
  path: drawable,
  ..marks-style: mark-item style,
  name: none string
)

```

path drawable

A single drawable

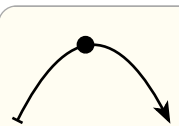
..marks-style mark-item or style

Positional mark-items and style key-value pairs

name none or string

Element name

Default: none



```

place-marks(bezier-through((0,0), (1,1), (2,0)),
  (mark: "|", size: .1, pos: 0),
  (mark: "o", size: .2, pos: .5),
  (mark: ">", size: .3, pos: 1),
  fill: black)

```

3.4.7 intersections

Calculate intersection between multiple paths and create one anchor per intersection point.

All resulting anchors will be named numerically, starting by 0. I.e., a call `intersections("a", ...)` will generate the anchors "a.0", "a.1", "a.2" to "a.n", depending of the number of intersections.

Parameters

```

intersections(
  name: string,
  body: drawables,
  samples: int
)

```

name string

Name of the node containing the anchors

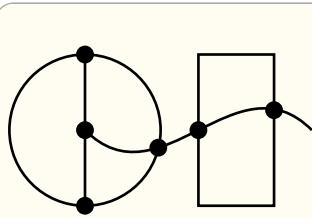
body drawables

Drawables to calculate intersections for

samples int

Number of samples to use for non-linear path segments. A higher sample count can give more precise results but worse performance.

Default: 10



```
intersections("demo", {
  circle((0, 0))
  bezier((0,0), (3,0), (1,-1), (2,1))
  line((0,-1), (0,1))
  rect((1.5,-1),(2.5,1))
})
for-each-anchor("demo", (name) => {
  circle("demo." + name, radius: .1, fill: black)
})
```

3.5 Layers

You can use layers to draw elements below or on top of other elements by using layers with a higher or lower index. When rendering, all draw commands are sorted by their layer (0 being the default).

3.5.1 on-layer

Place elements on a specific layer

A layer determines the position of an element in the draw queue. A lower layer is drawn before a higher layer.

Layers can be used to draw behind or in front of other elements, even if the other elements got created before/after. An example would be drawing a background behind a text, but using the text's calculated bounding box for positioning the background.

Parameters

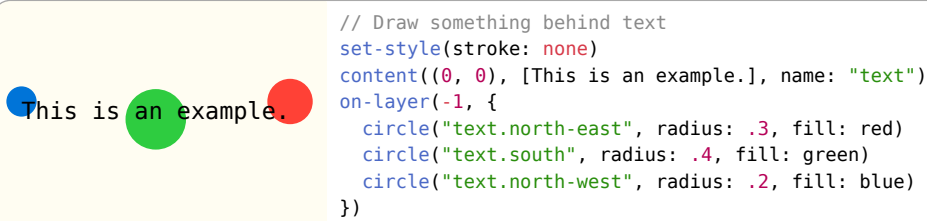
```
on-layer(
  layer: number,
  body: drawables
)
```

layer number

Layer number. The default layer of elements is layer 0.

body drawables

Elements to draw on the layer specified.



3.6 Transformations

All transformation functions push a transformation matrix onto the current transform stack. To apply transformations scoped use a `group(...)` object.

Transformation matrices get multiplied in the following order:

$$M_{\text{world}} = M_{\text{world}} \cdot M_{\text{local}}$$

3.6.1 translate

Push translation matrix

Parameters

```
translate(
  vec: vector dictionary,
  pre: bool
)
```

vec vector or dictionary

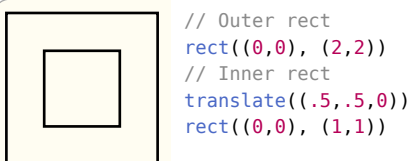
Translation vector

pre bool

Specify matrix multiplication order

- false: $\text{World} = \text{World} * \text{Translate}$
- true: $\text{World} = \text{Translate} * \text{World}$

Default: **true**



3.6.2 set-origin

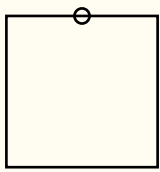
Sets the given position as the origin

Parameters

```
set-origin(origin: coordinate)
```

origin coordinate

Coordinate to set as new origin (0,0,0)



```
// Outer rect
rect((0,0), (2,2), name: "r")
// Move origin to top edge
set-origin("r.north")
circle((0, 0), radius: .1)
```

3.6.3 set-viewport

Span viewport between two coordinates and set-up scaling and translation

Parameters

```
set-viewport(
  from: coordinate,
  to: coordinate,
  bounds: vector
)
```

from coordinate

Bottom-Left corner coordinate

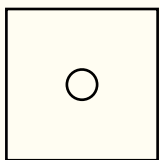
to coordinate

Top right corner coordinate

bounds vector

Viewport bounds vector that describes the inner width, height and depth of the viewport

Default: (1, 1, 1)



```
rect((0,0), (2,2))
set-viewport((0,0), (2,2), bounds: (10, 10))
circle((5,5))
```

3.6.4 rotate

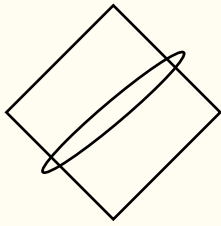
Rotate on z-axis (default) or specified axes if angle is of type dictionary.

Parameters

```
rotate(angle: typst-angle dictionary)
```

angle typst-angle or dictionary

Angle (z-axis) or dictionary of the form (x: <typst-angle>, y: <angle>, z: <angle>) specifying per axis rotation typst-angle.



```
// Rotate on z-axis
rotate((z: 45deg))
rect((-1,-1), (1,1))
// Rotate on y-axis
rotate((y: 80deg))
circle((0,0))
```

3.6.5 scale

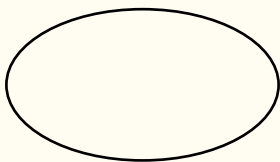
Push scale matrix

Parameters

`scale`(factor: `float` | `dictionary`)

factor `float` or `dictionary`

Scaling factor for all axes or per axis scaling factor dictionary.



```
// Scale x-axis
scale((x: 1.8))
circle((0,0))
```

3.7 Context Modification

The context of a canvas holds the canvas' internal state like style and transformation. Note that the fields of the context of a canvas are considered private and therefore unstable. You can add custom values to the context, but in order to prevent naming conflicts with future CeTZ versions, try to assign unique names.

3.7.1 set-ctx

Modify the current canvas context

A context object holds the canvas' state, such as the element dictionary, the current transformation matrix, group and canvas unit length. The following fields are considered stable:

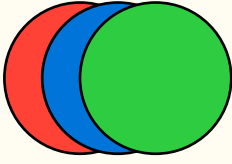
- `length` (`length`): Length of one canvas unit as typst length
- `transform` (`cetz.matrix`): Current 4x4 transformation matrix
- `debug` (`bool`): True if the canvas' debug flag is set

Parameters

`set-ctx`(callback: `function`)

callback `function`

Function accepting a context object and returning the new one: `(ctx) => <ctx>`



```
// Setting a custom transformation matrix
set-ctx(ctx => {
  let mat = ((1, 0, .5, 0),
             (0, 1, 0, 0),
             (0, 0, 1, 0),
             (0, 0, 0, 1))
  ctx.transform = mat
  return ctx
})
circle((z: 0), fill: red)
circle((z: 1), fill: blue)
circle((z: 2), fill: green)
```

3.7.2 get-ctx

Get the current context

Some functions such as `coordinate.resolve(...)` or `styles.resolve(...)` require a context object as argument. See `set-ctx` for a description of the context object.

Parameters

`get-ctx`(callback: `function`)

callback `function`

Function accepting the current context object and returning drawables or none

```
(
  (1, 0, 0.5, 0),
  (0, -1, -0.5, 0),
  (0, 0, 1, 0),
  (0, 0, 0, 1),
)

// Print the transformation matrix
get-ctx(ctx => {
  content(), [#repr(ctx.transform)]
})
```

4 Coordinate Systems

A *coordinate* is a position on the canvas on which the picture is drawn. They take the form of dictionaries and the following sub-sections define the key value pairs for each system. Some systems have a more implicit form as an array of values and CeTZ attempts to infer the system based on the element types.

4.1 XYZ

Defines a point *x* units right, *y* units upward, and *z* units away.

x <number> or <length> (default: 0)

The number of units in the x direction.

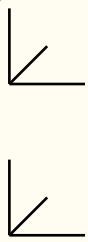
y <number> or <length> (default: 0)

The number of units in the y direction.

z <number> or <length> (default: 0)

The number of units in the z direction.

The implicit form can be given as an array of two or three <number> or <length>, as in `(x,y)` and `(x,y,z)`.

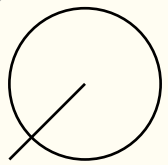


```
line((0,0), (x: 1))
line((0,0), (y: 1))
line((0,0), (z: 1))

// Implicit form
line((0, -2), (1, -2))
line((0, -2), (0, -1, 0))
line((0, -2), (0, -2, 1))
```

4.2 Previous

Use this to reference the position of the previous coordinate passed to a draw function. This will never reference the position of a coordinate used in to define another coordinate. It takes the form of an empty array (). The previous position initially will be (0, 0, 0).



```
line((0,0), (1, 1))

// Draws a circle at (1,1)
circle()
```

4.3 Relative

Places the given coordinate relative to the previous coordinate. Or in other words, for the given coordinate, the previous coordinate will be used as the origin. Another coordinate can be given to act as the previous coordinate instead.

rel <coordinate>

The coordinate to be place relative to the previous coordinate.

update <bool>

(default: **true**)

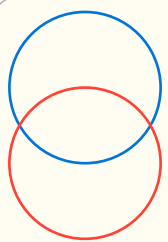
When false the previous position will not be updated.

to <coordinate>

(default: ())

The coordinate to treat as the previous coordinate.

In the example below, the red circle is placed one unit below the blue circle. If the blue circle was to be moved to a different position, the red circle will move with the blue circle to stay one unit below.



```
circle((0, 0), stroke: blue)
circle((rel: (0, -1)), stroke: red)
```

4.4 Polar

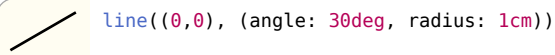
Defines a point a radius distance away from the origin at the given angle.

angle <angle>

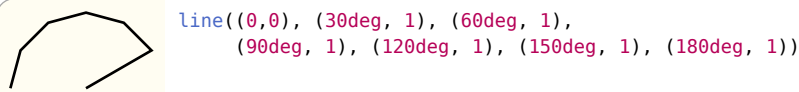
The angle of the coordinate. An angle of 0deg is to the right, a degree of 90deg is upward. See <https://typst.app/docs/reference/layout/angle/> for details.

radius <number> or <length> or <array of length or number>

The distance from the origin. An array can be given, in the form (x, y) to define the x and y radii of an ellipse instead of a circle.



The implicit form is an array of the angle then the radius (angle, radius) or (angle, (x, y)).



4.5 Barycentric

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2, \dots, v_n and numbers $\alpha_1, \alpha_2, \dots, \alpha_n$. Then the barycentric coordinate specified by these vectors and numbers is

$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \dots + \alpha_n}$$

bary <dictionary>

A dictionary where the key is a named element and the value is a <float>. The center anchor of the named element is used as v and the value is used as a .



4.6 Anchor

Defines a point relative to a named element using anchors, see Section 2.2.

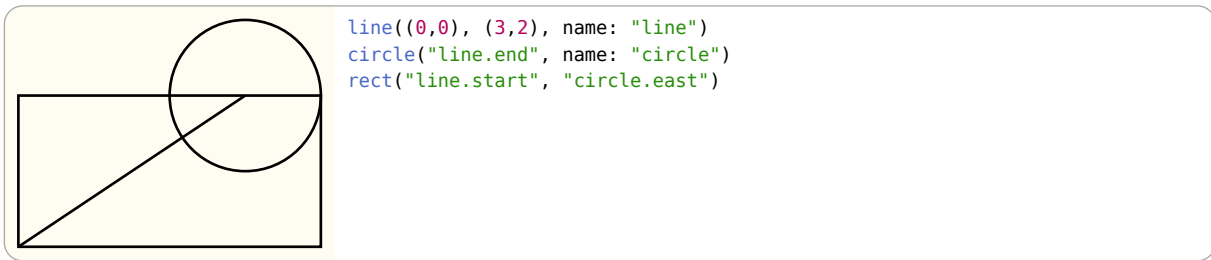
name <string>

The name of the element that you wish to use to specify a coordinate.

anchor <string>

An anchor of the element. If one is not given a default anchor will be used. On most elements this is center but it can be different.

You can also use implicit syntax of a dot separated string in the form "name.anchor".



4.7 Tangent

This system allows you to compute the point that lies tangent to a shape. In detail, consider an element and a point. Now draw a straight line from the point so that it “touches” the element (more formally, so that it is *tangent* to this element). The point where the line touches the shape is the point referred to by this coordinate system.

element <string>

The name of the element on whose border the tangent should lie.

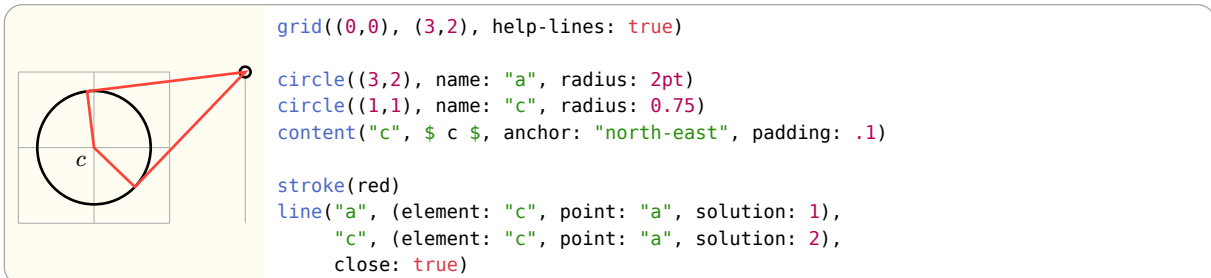
point <coordinate>

The point through which the tangent should go.

solution <integer>

Which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently it does this by assuming the distance between the center and top anchor (See Section 2.2) is the radius of a circle.



4.8 Perpendicular

Can be used to find the intersection of a vertical line going through a point p and a horizontal line going through some other point q .

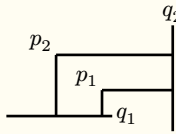
horizontal <coordinate>

The coordinate through which the horizontal line passes.

vertical <coordinate>

The coordinate through which the vertical line passes.

You can use the implicit syntax of (horizontal, "-|", vertical) or (vertical, "|-", horizontal)



```

set-style(content: (padding: .05))
content((30deg, 1), $ p_1 $, name: "p1")
content((75deg, 1), $ p_2 $, name: "p2")

line((-0.2, 0), (1.2, 0), name: "xline")
content("xline.end", $ q_1 $, anchor: "west")
line((2, -0.2), (2, 1.2), name: "yline")
content("yline.end", $ q_2 $, anchor: "south")

line("p1.south-east", (horizontal: ()), vertical: "xline.end")
line("p2.south-east", ((), "|-", "xline.end")) // Short form
line("p1.south-east", (vertical: ()), horizontal: "yline.end")
line("p2.south-east", ((), "-|", "yline.end")) // Short form

```

4.9 Interpolation

Use this to linearly interpolate between two coordinates *a* and *b* with a given factor number. If number is a <length> the position will be at the given distance away from *a* towards *b*. An angle can also be given for the general meaning: “First consider the line from *a* to *b*. Then rotate this line by angle around point *a*. Then the two endpoints of this line will be *a* and some point *c*. Use this point *c* for the subsequent computation.”

a <coordinate>

The coordinate to interpolate from.

b <coordinate>

The coordinate to interpolate to.

number <number> or <length>

The factor to interpolate by or the distance away from *a* towards *b*.

angle <angle>

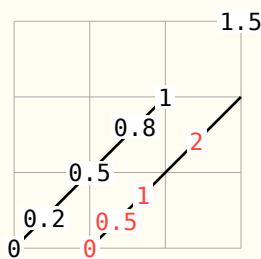
(default: 0deg)

abs <bool>

(default: false)

Interpret number as absolute distance, instead of a factor.

Can be used implicitly as an array in the form (*a*, number, *b*) or (*a*, number, angle, *b*).



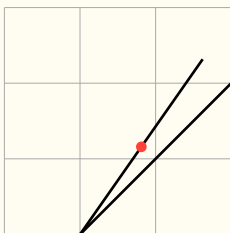
```

grid((0,0), (3,3), help-lines: true)

line((0,0), (2,2))
for i in (0, 0.2, 0.5, 0.8, 1, 1.5) { /* Relative distance */
  content(((0,0), i, (2,2)),
    box(fill: white, inset: 1pt, [#i]))
}

line((1,0), (3,2))
for i in (0, 0.5, 1, 2) { /* Absolute distance */
  content((a: (1,0), number: i, abs: true, b: (3,2)),
    box(fill: white, inset: 1pt, text(red, [#i])))
}

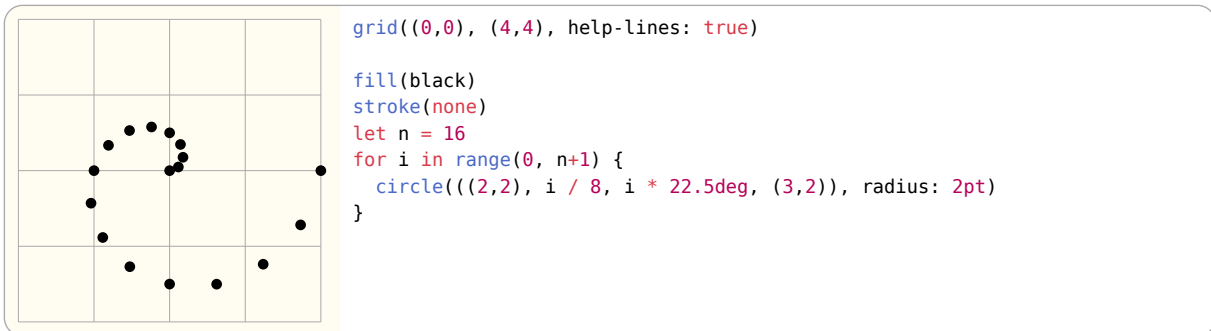
```



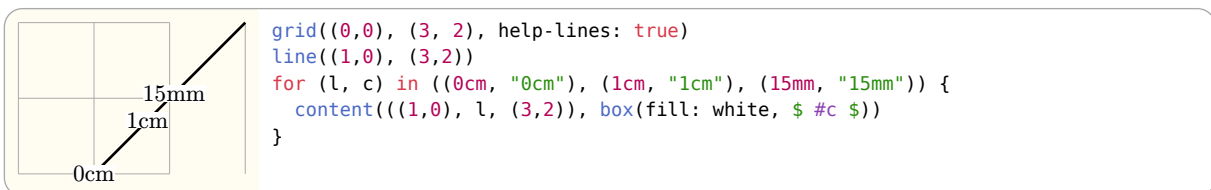
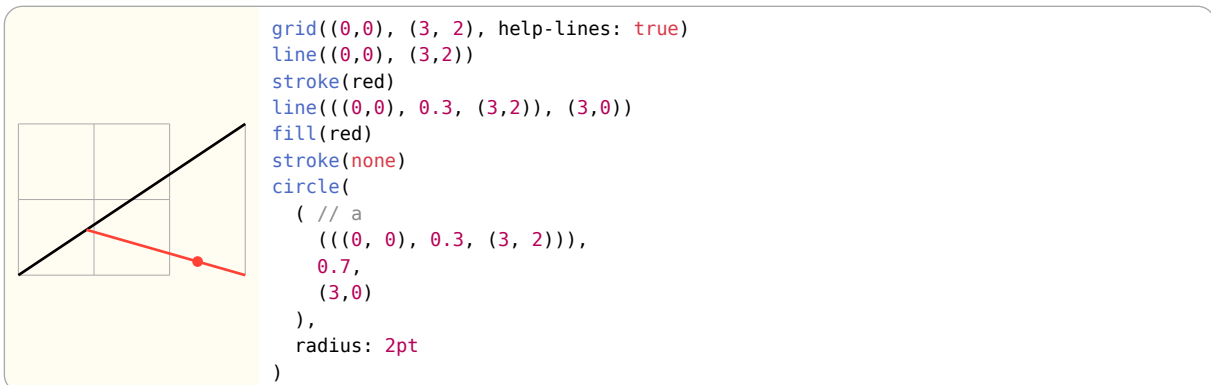
```

grid((0,0), (3,3), help-lines: true)
line((1,0), (3,2))
line((1,0), ((1, 0), 1, 10deg, (3,2)))
fill(red)
stroke(none)
circle(((1, 0), 0.5, 10deg, (3, 2)), radius: 2pt)

```



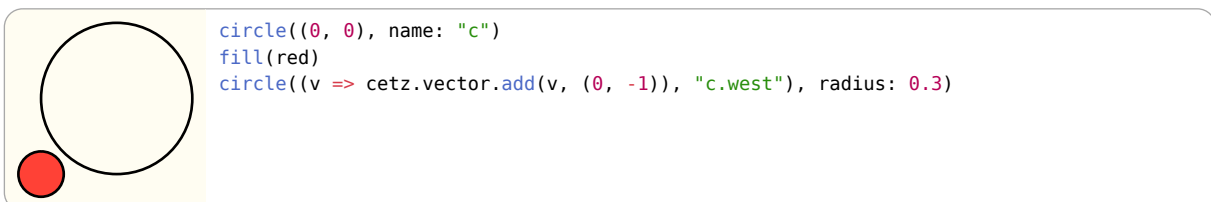
You can even chain them together!



4.10 Function

An array where the first element is a function and the rest are coordinates will cause the function to be called with the resolved coordinates. The resolved coordinates have the same format as the implicit form of the 3-D XYZ coordinate system, Section 4.1.

The example below shows how to use this system to create an offset from an anchor, however this could easily be replaced with a relative coordinate with the `to` argument set, Section 4.3.



5 Utility

5.1.1 for-each-anchor

Iterates through all anchors of an element, calling a callback per anchor

Parameters

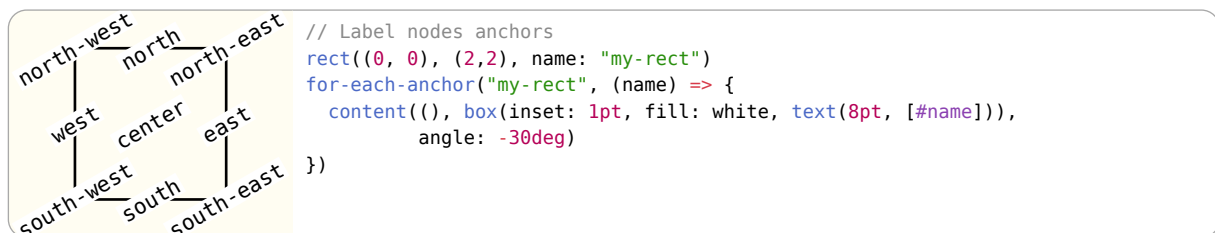
```
for-each-anchor(
  name: string,
  callback: function
)
```

name string

The target elements name to iterate over

callback function

Callback function accepting an anchor name string: (name) => {} that must return drawables or none.



6 Libraries

6.1 Tree

With the tree library, CeTZ provides a simple tree layout algorithm.

- [tree\(\)](#)

6.1.1 tree

Layout and render tree nodes

Parameters

```
tree(
  root: array,
  draw-node: function,
  draw-edge: function,
  direction: string,
  parent-position: string,
  grow: float,
  spread: float,
  name,
  ..style
)
```

root array

Tree structure represented by nested lists Example: ([root], [child 1], ([child 2], [grandchild 1]))

draw-node `function`

Callback for rendering a node. Signature: (node) => elements. The nodes position is accessible through the anchor "center" or the last position ().

Default: `auto`

draw-edge `function`

Callback for rendering edges between nodes Signature: (source-name, target-name, target-node) => elements

Default: `auto`

direction `string`

Tree grow direction (up, down, left, right)

Default: `"down"`

parent-position `string`

Positioning of parent nodes (begin, center, end)

Default: `"center"`

grow `float`

Depth grow factor (default 1)

Default: `1`

spread `float`

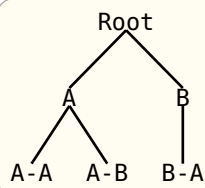
Sibling spread factor (default 1)

Default: `1`

name

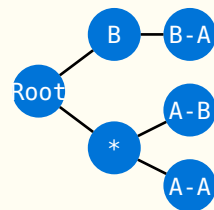
Default: `none`

..style



```
import cetz.tree

let data = ([Root], ([A], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), line: (stroke: blue))
```



```
import cetz.tree

let data = ([Root], ([\*], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), direction: "right",
  mark: (end: ">", fill: none),
  draw-node: (node, ..) => {
    circle(( ), radius: .35, fill: blue, stroke: none)
    content(( ), text(white, [#node.content]))
  },
  draw-edge: (from, to, ..) => {
    let (a, b) = (from + ".center",
      to + ".center")

    line((a: a, b: b, abs: true, number: .35),
      (a: b, b: a, abs: true, number: .35))
  })
```

6.1.2 Node

A tree node is an array of nodes. The first array item represents the current node, all following items are direct children of that node. The node itself can be of type content or dictionary with a key content.

6.2 Plot

The library plot of CeTZ allows plotting 2D data.

6.2.1 Types

Types commonly used by function of the plot library:

domain Tuple representing a functions domain as closed interval. Example domains are: (0, 1) for [0, 1] or (-calc.pi, calc.pi) for $[-\pi, \pi]$.

- [add-anchor\(\)](#)
- [plot\(\)](#)

6.2.2 add-anchor

Add an anchor to a plot environment

Parameters

```
add-anchor(
  name: string,
  position: array,
  axes: array
)
```

name string

Anchor name

position array

Tuple of x and y values. Both values can have the special values “min” and “max”, which resolve to the axis min/max value. Position is in axis space!

axes array

Name of the axes to use (“x”, “y”), note that both axes must exist!

Default: (“x”, “y”)

6.2.3 plot

Create a plot environment

Note: Data for plotting must be passed via `plot.add(...)`

Note that different axis-styles can show different axes. The “school-book” and “left” style shows only axis “x” and “y”, while the “scientific” style can show “x2” and “y2”, if set (if unset, “x2” mirrors “x” and “y2” mirrors “y”). Other axes (e.G. “my-axis”) work, but no ticks or labels will be shown.

Options

The following options are supported per axis and must be prefixed by `<axis-name>-`, e.G. `x-min: 0` or `y-label: [y]`.

- `label (content)`: Axis label
- `min (int)`: Axis minimum value
- `max (int)`: Axis maximum value
- `tick-step (none, float)`: Distance between major ticks (or no ticks if none)
- `minor-tick-step (none, float)`: Distance between minor ticks (or no ticks if none)
- `ticks (array)`: List of ticks values or value/label tuples. Example `(1,2,3)` or `((1, [A]), (2, [B]),)`
- `format (string)`: Tick label format, “float”, “sci” (scientific) or a custom function that receives a value and returns a content (`value => content`).
- `grid (bool,string)`: Enable grid-lines at tick values:
 - “major”: Enable major tick grid
 - “minor”: Enable minor tick grid
 - “both”: Enable major & minor tick grid
 - `false`: Disable grid
- `unit (none, content)`: Tick label suffix
- `decimals (int)`: Number of decimals digits to display for tick labels

Parameters

```
plot(
  body: body,
  size: array,
  axis-style: none string,
  name: string,
  plot-style: style function,
  mark-style: style function,
  fill-below: bool,
  legend: none auto coordinate,
  legend-anchor: auto string,
  legend-style: style,
  ..options: any
)
```

body `body`

Calls of `plot.add` or `plot.add-*` commands

size `array`

Plot canvas size tuple of width and height in canvas units

Default: `(1, 1)`

axis-style `none` or `string`

Axis style “scientific”, “left”, “school-book”

- “scientific”: Frame plot area and draw axes y , x , y_2 , and x_2 around it
- “school-book”: Draw axes x and y as arrows with both crossing at $(0, 0)$
- “left”: Draw axes x and y as arrows, the y axis stays on the left (at $x.\min$) and the x axis at the bottom (at $y.\min$)
- `none`: Draw no axes (and no ticks).

Default: “scientific”

name `string`

Element name

Default: `none`

plot-style `style` or `function`

Style used for drawing plot graphs This style gets inherited by all plots.

Default: `default-plot-style`

mark-style style or function

Style used for drawing plot marks. This style gets inherited by all plots.

Default: default-mark-style

fill-below bool

Fill functions below the axes (draw axes above fills)

Default: true

legend none or auto or coordinate

Position to place the legend at. The following anchors are considered optimal for legend placement:

- legend.north, legend.south, legend.east, legend.west
- legend.north-east, legend.north-west, legend.south-east, legend.south-west
- legend.inner-north, legend.inner-south, legend.inner-east, legend.inner-west
- legend.inner-north-east, legend.inner-north-west, legend.inner-south-east, legend.inner-south-west

If set to auto, the placement is read from the legend style (root legend).

Default: auto

legend-anchor auto or string

Anchor of the legend group to use as origin of the legend group.

Default: auto

legend-style style

Legend style overwrites.

Default: (:)

..options any

The following options are supported per axis and must be prefixed by <axis-name>-, e.G. x-min: 0.

- min (int): Axis minimum
- max (int): Axis maximum
- horizontal (bool): Axis orientation; note that each plot must use one vertical and one horizontal axis! The default value for this parameter is guessed: Axes starting with “x” are considered horizontal by default. This does not affect the side the ticks of the axis are drawn, but only the drawing direction.
- tick-step (float): Major tick step
- minor-tick-step (float): Major tick step
- ticks (array): List of ticks values or value/label tuples
- unit (content): Tick label suffix
- decimals (int): Number of decimals digits to display

- [add\(\)](#)
- [add-fill-between\(\)](#)
- [add-hline\(\)](#)
- [add-vline\(\)](#)

6.2.4 add

Add data to a plot environment.

Note: You can use this for scatter plots by setting the stroke style to none: `add(..., style: (stroke: none))`.

Must be called from the body of a `plot(...)` command.

Parameters

```
add(
  domain: domain,
  hypograph: bool,
  epigraph: bool,
  fill: bool,
  fill-type: string,
  style: style,
  mark: string,
  mark-size: float,
  mark-style,
  samples: int,
  sample-at: array,
  line: string dictionary,
  axes: array,
  label,
  data: array function
)
```

domain domain

Domain of data, if data is a function. Has no effect if data is not a function.

Default: **auto**

hypograph `bool`

Fill hypograph; uses the hypograph style key for drawing

Default: `false`

epigraph `bool`

Fill epigraph; uses the epigraph style key for drawing

Default: `false`

fill `bool`

Fill to y zero

Default: `false`

fill-type `string`

Fill type:

"**axis**" Fill to $y = 0$

"**shape**" Fill the functions shape

Default: `"axis"`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

mark `string`

Mark symbol to place at each distinct value of the graph. Uses the mark style key of style for drawing.

The following marks are supported:

- "*" or "x" – X
- "+" – Cross
- "|" – Bar
- "-" – Dash
- "o" – Circle
- "triangle" – Triangle
- "square" – Square

Default: `none`

mark-size `float`

Mark size in canvas units

Default: `.2`

mark-style

Default: `(:)`

samples `int`

Number of times the data function gets called for sampling y-values. Only used if data is of type function.

Default: `50`

sample-at `array`

Array of x-values the function gets sampled at in addition to the default sampling.

Default: `()`

line `string` or `dictionary`

Line type to use. The following types are supported:

- `"linear"` Linear line segments
- `"spline"` A smoothed line
- `"vh"` Move vertical and then horizontal
- `"hv"` Move horizontal and then vertical
- `"vhv"` Add a vertical step in the middle
- `"raw"` Like linear, but without linearization.

`"linear"` *should* never look different than `"raw"`.

If the value is a dictionary, the type must be supplied via the `type` key. The following extra attributes are supported:

- `"samples" <int>` Samples of splines
- `"tension" <float>` Tension of splines
- `"mid" <float>` Mid-Point of vhv lines (0 to 1)
- `"epsilon" <float>` Linearization slope epsilon for use with `"linear"`, defaults to 0.

Default: `"linear"`

axes `array`

Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

label

Default: **none**

data array or function

Array of 2D data points (numeric) or a function of the form $x \Rightarrow y$, where x is a value inside domain and y must be numeric or a 2D vector (for parametric functions).

Examples

- $((0,0), (1,1), (2,-1))$
- $x \Rightarrow \text{calc.pow}(x, 2)$

6.2.5 add-fill-between

Fill the area between two graphs. This behaves same as `plot` but takes a pair of data instead of a single data array/function. The area between both function plots gets filled.

This can be used to display an error-band of a function.

Parameters

```
add-fill-between(
  data-a: array function,
  data-b: array function,
  domain: domain,
  samples: int,
  sample-at: array,
  line: string dictionary,
  axes: array,
  label,
  style: style
)
```

data-a array or function

Data of the first plot, see `add`

data-b array or function

Data of the second plot, see `add`

domain domain

Domain of both `data-a` and `data-b`. The domain is used for sampling functions only and has no effect on data arrays.

Default: **auto**

samples `int`

Number of times the `data-a` and `data-b` function gets called for sampling y-values. Only used if `data-a` or `data-b` is of type function.

Default: `50`

sample-at `array`

Array of x-values the function(s) get sampled at in addition to the default sampling.

Default: `()`

line `string` or `dictionary`

Line type to use, see `add`

Default: `"linear"`

axes `array`

Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

label

Default: `none`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

6.2.6 add-hline

Add horizontal lines at values y

Parameters

```
add-hline(
  ..y: number,
  axes: array,
  style: style,
  label
)
```

..y number

Y axis value(s) to add a line at

axes array

Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!

Default: ("x", "y")

style style

Style to use, can be used with a palette function

Default: (:)

label

Default: none

6.2.7 add-vline

Add vertical lines at values x.

Parameters

```
add-vline(
  ..x: number,
  axes: array,
  style: style,
  label
)
```

..x number

X axis values to add a line at

axes array

Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!

Default: ("x", "y")

style style

Style to use, can be used with a palette function

Default: (:)

labelDefault: `none`

- `add-contour()`

6.2.8 add-contour

Add a contour plot of a sampled function or a matrix.

Parameters

```
add-contour(
  data: array function,
  label,
  z: float array,
  x-domain: domain,
  y-domain: domain,
  x-samples: int,
  y-samples: int,
  interpolate: bool,
  op: auto string function,
  axes: array,
  style: style,
  fill: bool,
  limit: int
)
```

data array or function

A function of the signature $(x, y) \Rightarrow z$ or an array of floats where the first index is the row and the second index is the column.

Examples:

- $(x, y) \Rightarrow x > 0$
- $(x, y) \Rightarrow 30 - (\text{calc.pow}(1 - x, 2) + \text{calc.pow}(1 - y, 2))$

labelDefault: `none`**z** float or array

Z values to plot. Contours containing values above z ($z \geq 0$) or below z ($z < 0$) get plotted. If you specify multiple z values, they get plotted in order.

Default: `(1,)`**x-domain** domain

X axis domain used if data is a function.

Default: `(0, 1)`

y-domain `domain`

Y axis domain used if data is a function.

Default: `(0, 1)`

x-samples `int`

X axis domain samples ($2 < n$)

Default: `25`

y-samples `int`

Y axis domain samples ($2 < n$)

Default: `25`

interpolate `bool`

Use linear interpolation between sample values

Default: `true`

op `auto` or `string` or `function`

Z value comparison operator:

`">`", `">="`, `"<`", `"<="`, `"!="`, `"=="` Use the operator for comparison.

auto Use `">="` for positive z values, `"<="` for negative z values.

<function> Call comparison function of the format `(plot-z, data-z) => boolean`, where `plot-z` is the z-value from the plots `z` argument and `data-z` is the z-value of the data getting plotted.

Default: `auto`

axes `array`

Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

fill bool

Fill each contour

Default: **false****limit** int

Limit of contours to create per z value before the function panics

Default: **50**

- [add-boxwhisker\(\)](#)

6.2.9 add-boxwhisker

Add one or more box or whisker plots

Parameters

```
add-boxwhisker(
    data: array<dictionary>,
    label,
    axes: array,
    style: style,
    box-width: float,
    whisker-width: float,
    mark: string,
    mark-size: float
)
```

data array or dictionary

dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

The following fields are supported:

- x (number) X-axis value
- min (number) Minimum value
- max (number) Maximum value
- q1, q2, q3 (number) Quartiles from lower to upper
- outliers (array of numbers) Optional outliers

Examples:

```
• (x: 1 // Location on x-axis
  outliers: (7, 65, 69), // Optional outlier values
  min: 15, max: 60 // Minimum and maximum
  q1: 25, // Quartiles: Lower
  q2: 35, // Median
  q3: 50) // Upper
```

labelDefault: **none**

axes `array`

Name of the axes to use ("x", "y"), note that not all plot styles are able to display a custom axis!

Default: `("x", "y")`**style** `style`

Style to use, can be used with a palette function

Default: `(:)`**box-width** `float`

Width from edge-to-edge of the box of the box and whisker in plot units. Defaults to 0.75

Default: `0.75`**whisker-width** `float`

Width from edge-to-edge of the whisker of the box and whisker in plot units. Defaults to 0.5

Default: `0.5`**mark** `string`

Mark to use for plotting outliers. Set none to disable. Defaults to "x"

Default: `"*"`**mark-size** `float`

Size of marks for plotting outliers. Defaults to 0.15

Default: `0.15`

- [sample-fn\(\)](#)
- [sample-fn2\(\)](#)

6.2.10 sample-fn

Sample the given one parameter function with `samples` values evenly spaced within the range given by domain and return each sampled y value in an array as (x, y) tuple.

If the functions first return value is a tuple (x, y), then all return values must be a tuple.

Parameters

```
sample-fn(
  fn: function,
  domain: domain,
  samples: int,
  sample-at: array
) -> array: Array of (x y) tuples
```


fn `function`

Function to sample of the form $(x) \Rightarrow y$ or $(x) \Rightarrow (x, y)$.

domain `domain`

Domain of `fn` used as bounding interval for the sampling points.

samples `int`

Number of samples in domain.

sample-at `array`

List of x values the function gets sampled at in addition to the `samples` number of samples. Values outside the specified domain are legal.

Default: `()`

6.2.11 sample-fn2

Samples the given two parameter function with `x-samples` and `y-samples` values evenly spaced within the range given by `x-domain` and `y-domain` and returns each sampled output in an array.

Parameters

```
sample-fn2(
  fn: function,
  x-domain: domain,
  y-domain: domain,
  x-samples: int,
  y-samples: int
) -> array: Array of z scalars
```

fn `function`

Function of the form $(x, y) \Rightarrow z$ with all values being numbers.

x-domain `domain`

Domain used as bounding interval for sampling point's x values.

y-domain `domain`

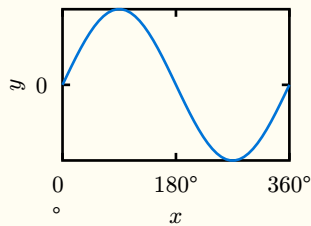
Domain used as bounding interval for sampling point's y values.

x-samples `int`

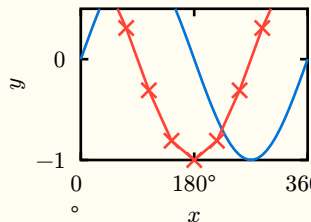
Number of samples in the x-domain.

y-samples `int`

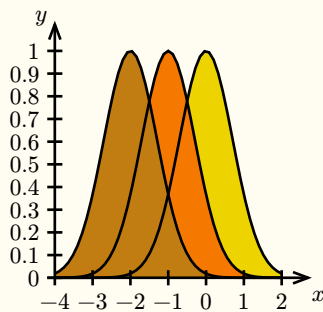
Number of samples in the y-domain.

6.2.12 Examples

```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 180, y-tick-step: 1,
          x-unit: $degree$, {
    plot.add(domain: (0, 360), x => calc.sin(x * 1deg))
  })
```



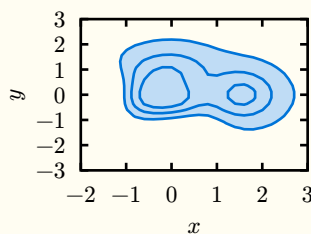
```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 180, y-tick-step: 1,
          x-unit: $degree$, y-max: .5, {
    plot.add(domain: (0, 360), x => calc.sin(x * 1deg))
    plot.add(domain: (0, 360), x => calc.cos(x * 1deg),
              samples: 10, mark: "x", style: (mark: (stroke: blue)))
  })
```



```
import cetz.plot
import cetz.palette

// Axes can be styled!
// Set the tick length to .1:
set-style(axes: (tick: (length: .1)))

// Plot something
plot.plot(size: (3,3), x-tick-step: 1, axis-style: "left", {
  for i in range(0, 3) {
    plot.add(domain: (-4, 2),
              x => calc.exp(-(calc.pow(x + i, 2))),
              fill: true, style: palette.tango)
  }
})
```



```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 1, y-tick-step: 1, {
  let z(x, y) = {
    (1 - x/2 + calc.pow(x,5) + calc.pow(y,3)) * calc.exp(-(x*x) - (y*y))
  }
  plot.add-contour(x-domain: (-2, 3), y-domain: (-3, 3),
                  z, z: (.1, .4, .7), fill: true)
})
```

6.2.13 Styling

The following style keys can be used (in addition to the standard keys) to style plot axes. Individual axes can be styled differently by using their axis name as key below the axes root.

```
set-style(axes: ( /* Style for all axes */ ))
set-style(axes: (bottom: ( /* Style axis "bottom" */)))
```

Axis names to be used for styling:

- School-Book and Left style:

- x: X-Axis
- y: Y-Axis

- Scientific style:

- left: Y-Axis
- right: Y2-Axis
- bottom: X-Axis
- top: X2-Axis

Default scientific Style

```
(
  fill: none,
  stroke: luma(0%),
  label: (offset: 0.2, anchor: auto),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.2, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: luma(66.67%), dash: "dotted"),
    fill: none,
  ),
)
```

Default school-book Style

```
(
  fill: none,
  stroke: luma(0%),
  label: (offset: 0.2, anchor: auto),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.1, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: luma(66.67%), dash: "dotted"),
    fill: none,
  ),
  mark: (end: ">"),
  padding: 0.4,
)
```

6.3 Chart

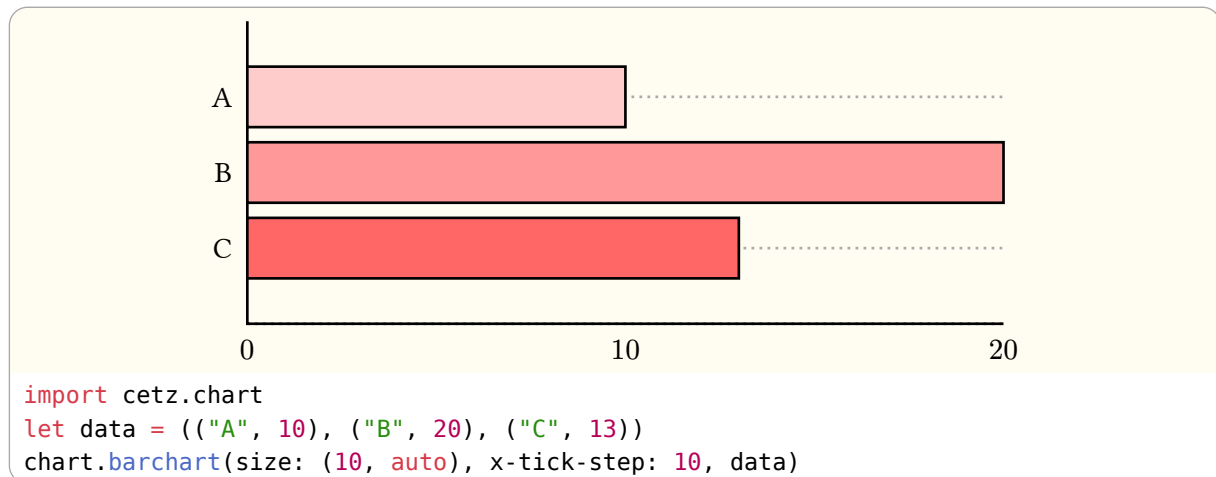
With the chart library it is easy to draw charts.

Supported charts are:

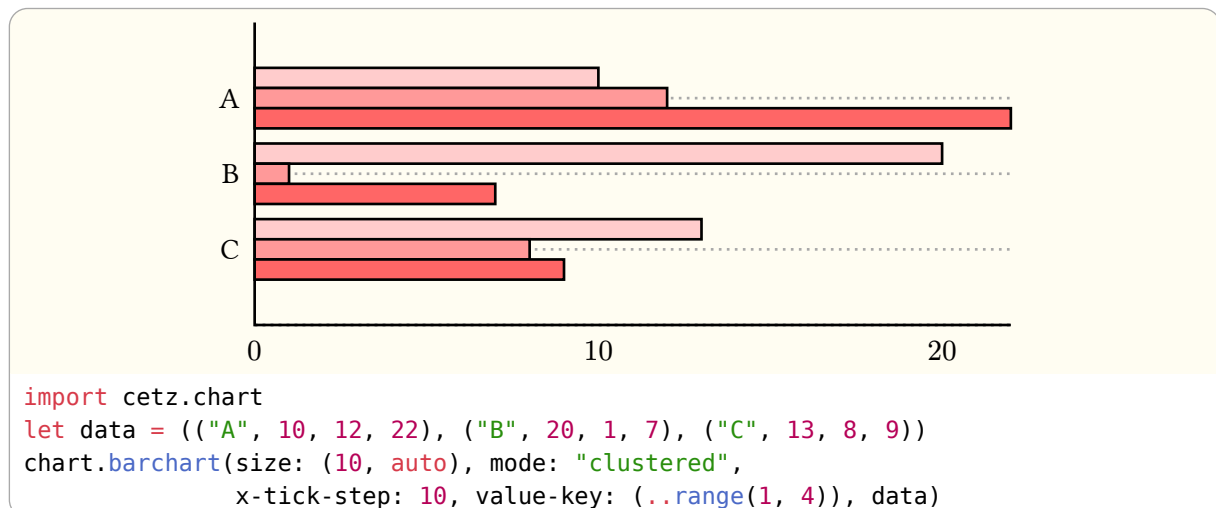
- `barchart(..)` and `columnchart(..)`: A chart with horizontal/vertical growing bars
 - `mode: "basic"`: (default): One bar per data row
 - `mode: "clustered"`: Multiple grouped bars per data row
 - `mode: "stacked"`: Multiple stacked bars per data row
 - `mode: "stacked100"`: Multiple stacked bars relative to the sum of a data row
- `boxwhisker(..)`: A box-plot chart

6.3.1 Examples – Bar Chart

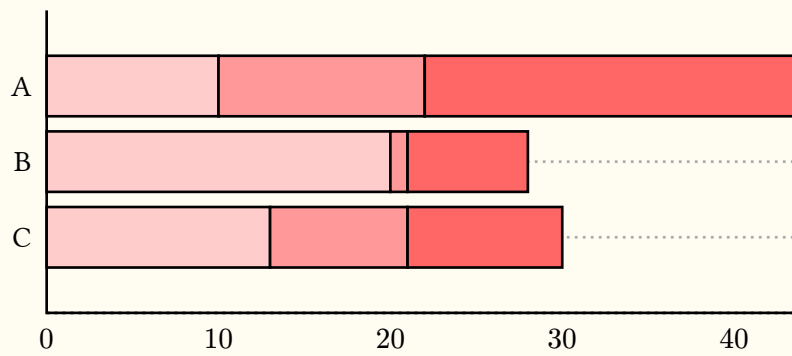
Basic



Clustered



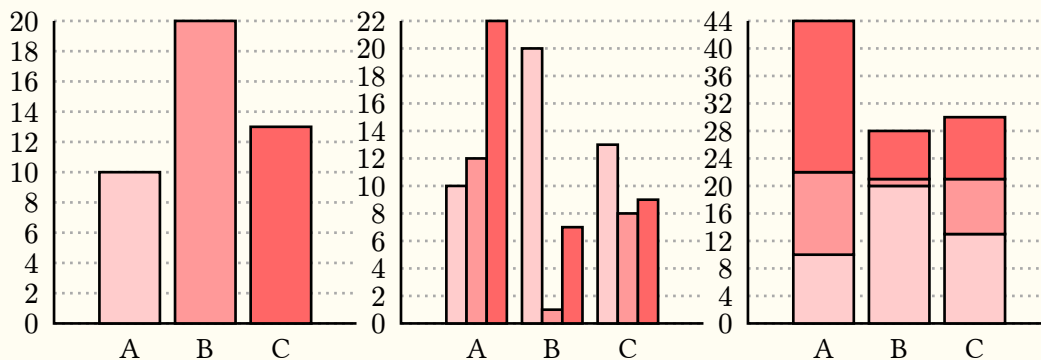
Stacked



```
import cetz.chart
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
chart.barchart(size: (10, auto), mode: "stacked",
  x-tick-step: 10, value-key: (..range(1, 4)), data)
```

6.3.2 Examples – Column Chart

Basic, Clustered and Stacked



```
import cetz.chart
// Left
let data = (("A", 10), ("B", 20), ("C", 13))
group(name: "a", {
  anchor("default", (0,0))
  chart.columnchart(size: (auto, 4), data)
})
// Center
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
set-origin("a.south-east")
group(name: "b", anchor: "south-west", {
  anchor("center", (0,0))
  chart.columnchart(size: (auto, 4),
    mode: "clustered", value-key: (1,2,3), data)
})
// Right
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
set-origin("b.south-east")
group(name: "c", anchor: "south-west", {
  anchor("center", (0,0))
  chart.columnchart(size: (auto, 4),
    mode: "stacked", value-key: (1,2,3), data)
})
```

- `boxwhisker()`

6.3.3 boxwhisker

Add one or more box or whisker plots.

Parameters

```
boxwhisker(
  data: array | dictionary,
  size,
  y-min,
  y-max,
  label-key: integer | string,
  box-width: float,
  whisker-width: float,
  mark: string,
  mark-size: float,
  ..arguments: any
)
```

data array or dictionary

Dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

See `plot.add-boxwhisker` for more details.

Examples:

- (x: 1 // Location on x-axis
 outliers: (7, 65, 69), // Optional outliers
 min: 15, max: 60 // Minimum and maximum
 q1: 25, // Quartiles: Lower
 q2: 35, // Median
 q3: 50) // Upper
- size (array) : Size of chart. If the second entry is auto, it automatically scales to accommodate the number of entries plotted
- y-min (float) : Lower end of y-axis range. If auto, defaults to lowest outlier or lowest min.
- y-max (float) : Upper end of y-axis range. If auto, defaults to greatest outlier or greatest max.

size

Default: (1, auto)

y-min

Default: auto

y-max

Default: auto

label-key integer or string

Index in the array where labels of each entry is stored

Default: 0

box-width float

Width from edge-to-edge of the box of the box and whisker in plot units. Defaults to 0.75

Default: 0.75

whisker-width float

Width from edge-to-edge of the whisker of the box and whisker in plot units. Defaults to 0.5

Default: 0.5

mark string

Mark to use for plotting outliers. Set none to disable. Defaults to "x"

Default: "*"

mark-size float

Size of marks for plotting outliers. Defaults to 0.15

Default: 0.15

..arguments any

Additional arguments are passed to plot.plot

6.3.4 Styling

Charts share their axis system with plots and therefore can be styled the same way, see Section 6.2.13.

Default barchart Style

```
(axes: (tick: (length: 0)))
```

Default columnchart Style

```
(axes: (tick: (length: 0)))
```

Default boxwhisker Style

```
(axes: (tick: (length: -0.1)), grid: none)
```

6.4 Palette

A palette is a function that returns a style for an index. The palette library provides some predefined palettes.

- [new\(\)](#)

6.4.1 new

Define a new palette

A palette is a function in the form `index -> style` that takes an index (int) and returns a canvas style dictionary. If passed the string "len" it must return the length of its styles.

Parameters

```
new(
  stroke: stroke,
  fills: array
) -> function
```

stroke stroke

Single stroke style.

fills array

List of fill styles.

6.4.2 List of predefined palettes

- gray



- red



- blue



- rainbow



- tango-light



- tango

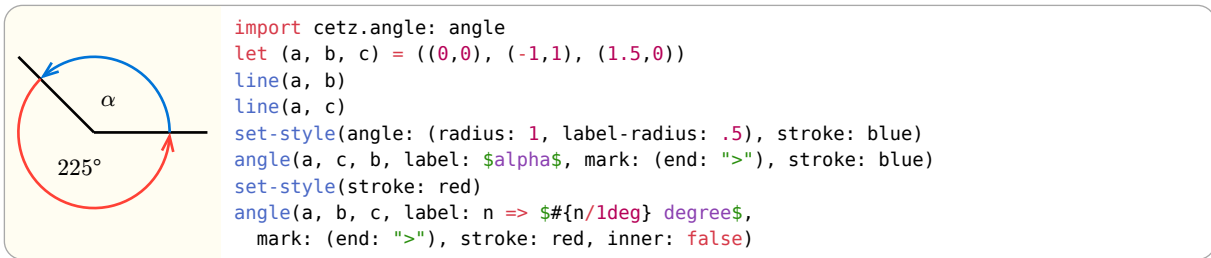


- tango-dark



6.5 Angle

The angle function of the angle module allows drawing angles with an optional label.



Default angle Style

```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 0.25,
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: auto,
    fill: none,
  ),
)
```

6.6 Decorations

Various pre-made shapes and lines.

6.6.1 brace

Draw a curly brace between two points.

Style root: brace.

Anchors:

start Where the brace starts, same as the start parameter.

end Where the brace end, same as the end parameter.

spike Point of the spike, halfway between start and end and shifted by amplitude towards the pointing direction.

content Point to place content/text at, in front of the spike.

center Center of the enclosing rectangle.

(a-k) Debug points a through k.

Parameters

```
brace(
  start: coordinate,
  end: coordinate,
  flip: bool,
  debug: bool,
  name: string none,
  ..style: style
)
```

start coordinate

Start point

end coordinate

End point

flip bool

Flip the brace around

Default: `false`**debug** bool

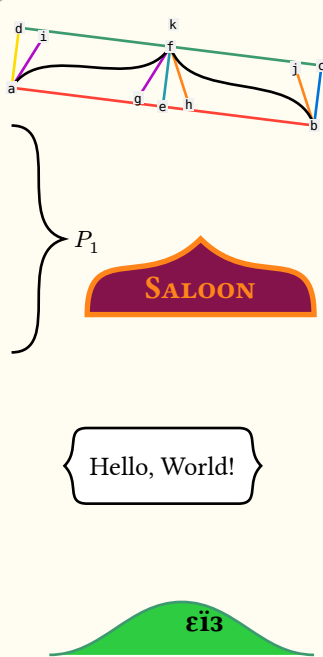
Show debug lines and points

Default: `false`**name** string or none

Element name

Default: `none`**..style** style

Style attributes



```
import cetz.decorations: brace
let text = text.with(size: 12pt, font: "Linux Libertine")

brace((0, 0), (4, -.5), pointiness: 25deg, outer-pointiness: auto,
amplitude: .8, debug: true)
brace((0, -.5), (0, -3.5), name: "brace")
content("brace.content", [P_1$])

// styling can be passed to the underlying `merge-path` call
brace((1, -3), (4, -3), amplitude: 1, pointiness: .5, stroke: orange + 2pt,
fill: maroon, close: true, name: "saloon")
content((rel: (0, -.15), to: "saloon.center"), text(fill: orange,
smallcaps[*SaLoon*]))

// as part of another path
set-origin((2, -5))
merge-path({
  brace(+1, .5), (+1, -.5), amplitude: .3, pointiness: .5)
  brace(-1, -.5), (-1, .5), amplitude: .3, pointiness: .5)
}, fill: white, close: true)
content((0, 0), text(size: 10pt)[Hello, World!])

brace((-1.5, -2.5), (2, -2.5), pointiness: 1, outer-pointiness: 1, stroke:
olive, fill: green, name: "hill")
content((rel: (.3, .1), to: "hill.center"), text[*εἰ3*])
```

Styling

- amplitude** <number> (default: 0.7)
Determines how much the brace rises above the base line.
- pointiness** <number> or <angle> (default: 15deg)
How pointy the spike should be. 0deg or 0 for maximum pointiness, 90deg or 1 for minimum.
- outer-pointiness** <number> or <angle> or <auto> (default: 0)
How pointy the outer edges should be. 0deg or 0 for maximum pointiness (allowing for a smooth transition to a straight line), 90deg or 1 for minimum. Setting this to **auto** will use the value set for pointiness.
- content-offset** <number> (default: 0.3)
Offset of the content anchor from the spike.
- debug-text-size** <length> (default: 6pt)
Font size of displayed debug points when debug is **true**.

Default brace Style

```
(
  amplitude: 0.7,
  pointiness: 15deg,
  outer-pointiness: 0,
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

6.6.2 flat-brace

Draw a flat curly brace between two points.

This mimics the braces from TikZ's `decorations.pathreplacing` library¹. In contrast to `brace()`, these braces use straight line segments, resulting in better looks for long braces with a small amplitude.

Style root: flat-brace.

anchors:

- start** Where the brace starts, same as the `start` parameter.
- end** Where the brace end, same as the `end` parameter.
- spike** Point of the spike's top.
- content** Point to place content/text at, in front of the spike.
- center** Center of the enclosing rectangle.
- (a-h)** Debug points a through h.

Parameters

```
flat-brace(
  start: coordinate,
  end: coordinate,
  flip: bool,
  debug: bool,
  name: string none,
  ..style: style
)
```

¹<https://github.com/pgf-tikz/pgf/blob/6e5fd71581ab04351a89553a259b57988bc28140/tex/generic/pgf/libraries/decorations/pgflibrarydecorations.pathreplacing.code.tex#L136-L185>

start coordinate

Start point

end coordinate

End point

flip bool

Flip the brace around

Default: `false`**debug** bool

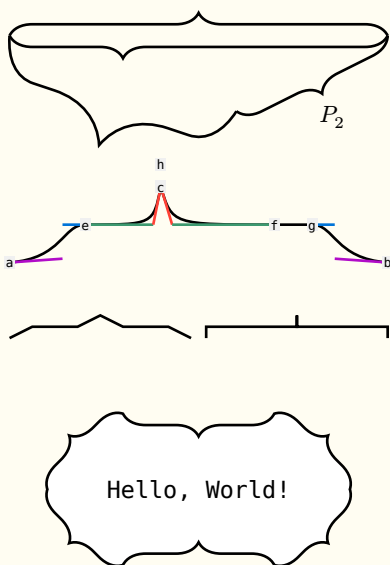
Show debug lines and points

Default: `false`**name** string or none

Element name

Default: `none`**..style** style

Style attributes



```
import cetz.decorations: flat-brace
```

```
flat-brace((x: 5))
flat-brace((0, 0), (5, 0), flip: true, aspect: .3)
flat-brace((rel: (-2, -1)), name: "a")
flat-brace((0, 0), amplitude: 1, curves: 1.5, outer-curves: .5)
content("a.content", [$P_2$])
```

```
flat-brace((0, -3), (5, -3), debug: true, amplitude: 1, aspect: .4,
curves: (1.5, .9, 1, .1), outer-curves: (1, .3, .1, .7))
```

```
// triangle and square braces
```

```
flat-brace((0, -4), (2.4, -4), curves: (auto, 0, 0, 0))
flat-brace((2.6, -4), (5, -4), curves: 0)
```

```
merge-path(close: true, fill: white, {
  move-to((.5, -6))
  flat-brace((rel: (1, 1)))
  flat-brace((rel: (2, 0)), flip: true, name: "top")
  flat-brace((rel: (1, -1)))
  flat-brace((rel: (-1, -1)))
  flat-brace((rel: (-2, 0)), flip: true, name: "bottom")
  flat-brace((rel: (-1, 1)))
})
content(("top.spike", .5, "bottom.spike"), [Hello, World!])
```

Styling

- amplitude** <number> (default: 0.3)
Determines how much the brace rises above the base line.
- aspect** <number> (default: 0.5)
Determines the fraction of the total length where the spike will be placed.
- curves** <array> or <number> (default: (1, 0.5, 0.6, 0.15))
Customizes the control points of the curved parts. Setting a single number is the same as setting (num, auto, auto, auto). Setting any item to auto will use its default value. The first item specifies the curve widths as a fraction of the amplitude. The second item specifies the length of the green and blue debug lines as a fraction of the curve's width. The third item specifies the vertical offset of the red and purple debug lines as a fraction of the curve's height. The fourth item specifies the horizontal offset of the red and purple debug lines as a fraction of the curve's width.
- outer-curves** <array> or <number> or <auto> (default: auto)
Customizes the control points of just the outer two curves (just the blue and purple debug lines). Overrides settings from curves. Setting the entire value or individual items to auto uses the values from curves as fallbacks.
- content-offset** <number> (default: 0.3)
Offset of the content anchor from the spike.
- debug-text-size** <length> (default: 6pt)
Font size of displayed debug points when debug is true.

Default flat-brace Style

```
(
  amplitude: 0.3,
  aspect: 0.5,
  curves: (1, 0.5, 0.6, 0.15),
  outer-curves: auto,
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

7 Advanced Functions

7.1 Coordinate

7.1.1 resolve

Resolve a list of coordinates to a absolute vectors

Parameters

```
resolve(
  ctx: context,
  ..coordinates: coordinate,
  update: bool
) -> (ctx vector..) Returns a list of the new context object plus the
```

ctx context

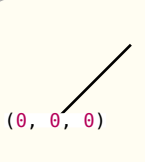
Canvas context object

..coordinates `coordinate`

List of coordinates

update `bool`

Update the context's last position resolved coordinate vectors

Default: `true`


```

line((0,0), (1,1), name: "l")
get-ctx(ctx => {
  // Get the vector of coordinate "l.start"
  content("l.start", [#cetz.coordinate.resolve(ctx, "l.start").at(1)], frame: "rect",
    stroke: none, fill: white)
})

```

7.2 Styles

7.2.1 resolve

Resolve the current style root

Parameters

```

resolve(
  current: style,
  new: style,
  root: none | str,
  base: none | style
)

```

current `style`

Current context style (ctx.style).

new `style`Style values overwriting the current style (or an empty dict). I.e. inline styles passed with an element: `line(..., stroke: red)`.**root** `none` or `str`

Style root element name.

Default: `none`**base** `none` or `style`Base style. For use with custom elements, see `lib/angle.typ` as an example.Default: `none`

```
(
  fill: none,
  stroke: 1pt + luma(0%),
  radius: 1,
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: 1pt + luma(0%),
    fill: none,
  ),
)

get-ctx(ctx => {
  // Get the current line style
  content((0,0), [#cetz.styles.resolve(ctx.style, (:), root: "line")])
})
```

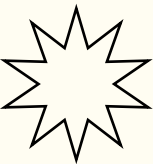
7.2.2 Default Style

This is a dump of the style dictionary every canvas gets initialized with. It contains all supported keys for all elements.

```
(
  root: (fill: none, stroke: 1pt + luma(0%), radius: 1),
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: auto,
    fill: none,
  ),
  group: (padding: none),
  line: (
    mark: (
      scale: 1,
      length: 0.2,
      width: 0.15,
      inset: 0.05,
      sep: 0.1,
      z-up: (0, 1, 0),
      start: none,
      end: none,
      stroke: auto,
      fill: none,
    ),
  ),
  bezier: (
    mark: (
      scale: 1,
      length: 0.2,
      width: 0.15,
      inset: 0.05,
      sep: 0.1,
      z-up: (0, 1, 0),
      start: none,
      end: none,
      stroke: auto,
      fill: none,
      flex: true,
      position-samples: 30,
    ),
    shorten: "LINEAR",
  ),
  catmull: (
    tension: 0.5,
    mark: (
      scale: 1,
      length: 0.2,
      width: 0.15,
      inset: 0.05,
      sep: 0.1,
      z-up: (0, 1, 0),
      start: none,
      end: none,
      stroke: auto,
      fill: none,
      flex: true,
      position-samples: 30,
    ),
    shorten: "LINEAR",
  ),
  hobby: (
    omega: (1, 1),
    rho: auto,
    mark: (
      scale: 1,
      length: 0.2,
      width: 0.15,
      inset: 0.05,
      sep: 0.1,
      z-up: (0, 1, 0),
      start: none,
      end: none,
      stroke: auto,
      fill: none,
      flex: true,
      position-samples: 30,
    ),
    shorten: "LINEAR",
  ),
  arc: (
    mode: "OPEN",
    mark: (
      scale: 1,
      length: 0.2,
      width: 0.15,
      inset: 0.05,
      sep: 0.1,
      z-up: (0, 1, 0),
      start: none,
      end: none,
      stroke: auto,
      fill: none,
    ),
  ),
  content: (padding: 0, frame: none, fill: auto, stroke: auto),
)
```

8 Creating Custom Elements

The simplest way to create custom, reusable elements is to return them as a group. In this example we will implement a function `my-star(center)` that draws a star with `n` corners and a style specified inner and outer radius.



```
let my-star(center, name: none, ..style) = {
  group(name: name, ctx => {
    // Define a default style
    let def-style = (n: 5, inner-radius: .5, radius: 1)

    // Resolve the current style ("star")
    let style = cetz.styles.resolve(ctx.style, style.named(),
      base: def-style, root: "star")

    // Compute the corner coordinates
    let corners = range(0, style.n * 2).map(i => {
      let a = 90deg + i * 360deg / (style.n * 2)
      let r = if calc.rem(i, 2) == 0 { style.radius } else { style.inner-radius }

      // Output a center relative coordinate
      (rel: (calc.cos(a) * r, calc.sin(a) * r, 0), to: center)
    })

    line(..corners, ..style, close: true)
  })
}

// Call the element
my-star((0,0))
my-star((0,3), n: 10)

set-style(star: (fill: yellow)) // set-style works, too!
my-star((0,6), inner-radius: .3)
```

9 Internals

9.1 Context

The state of the canvas is encoded in its context object. Elements or other draw calls may return a modified context element to the canvas to change its state, e.g. modifying the transforming matrix, adding a group or setting a style.

```
(
  typst-style: ...,
  length: 28.35pt,
  debug: false,
  prev: (pt: (0, 0, 0)),
  em-size: (width: 8.8pt, height: 8.8pt),
  style: (:),
  transform: (
    (1, 0, 0.5, 0),
    (0, -1, -0.5, 0),
    (0, 0, 1, 0),
    (0, 0, 0, 1),
  ),
  nodes: (:),
  groups: (),
)

// Show the current context
get-ctx(ctx => {
  content((), raw(repr(ctx), lang: "typc"))
})
```

9.2 Elements

Each CeTZ element (`line`, `bezier`, `circle`, ...) returns an array of functions for drawing to the canvas. Such function takes the canvas' context object and must return an dictionary of the following keys:

- `ctx` (required): The (modified) canvas context object

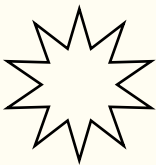
- **drawables:** List of drawables to render to the canvas
- **anchors:** A function of the form (`<anchor-identifier>`) => `<vector>`
- **name:** The elements name

An element that does only modify the context could be implemented like the following:

```
let my-element() = {
  (ctx => {
    // Do something with ctx ...
    (ctx: ctx)
  },)
}

// Call the element
my-element()
```

For drawing, elements must not use Typst native drawing functions, but output CeTZ paths. The `drawable` module provides functions for path creation (`path(...)`), the `path-util` module provides utilities for path segment creation. For demonstration, we will recreate the custom element `my-star` from Section 8:



```
import cetz.drawable: path
import cetz.vector

let my-star(center, ..style) = {
  (ctx => {
    // Define a default style
    let def-style = (n: 5, inner-radius: .5, radius: 1)

    // Resolve center to a vector
    let (ctx, center) = cetz.coordinate.resolve(ctx, center)

    // Resolve the current style ("star")
    let style = cetz.styles.resolve(ctx.style, style.named(),
      base: def-style, root: "star")

    // Compute the corner coordinates
    let corners = range(0, style.n * 2).map(i => {
      let a = 90deg + i * 360deg / (style.n * 2)
      let r = if calc.rem(i, 2) == 0 { style.radius } else { style.inner-radius }
      vector.add(center, (calc.cos(a) * r, calc.sin(a) * r, 0))
    })

    // Build a path through all three coordinates
    let path = cetz.drawable.path((cetz.path-util.line-segment(corners)),
      stroke: style.stroke, fill: style.fill, close: true)

    (ctx: ctx,
     drawables: cetz.drawable.apply-transform(ctx.transform, path),
    ),)
  },)
}

// Call the element
my-star((0,0))
my-star((0,3), n: 10)
my-star((0,6), inner-radius: .3, fill: yellow)
```

Using custom elements instead of groups (as in Section 8) makes sense when doing advanced computations or even applying modifications to passed in elements.