

v0.1.0

2023-08-29

MIT

Typst-setting finite automata with CeTZ

Jonas NEUGEBAUER

<https://github.com/jneug/typst-finite>

FINITE is a Typst package to draw transition diagrams for finite automata (finite state machines) with the power of **CeTZ**.

The package provides new elements for manually drawing states and transitions on any **CeTZ** canvas, but also comes with commands to quickly create automata from a transition table.

Table of contents

I. Usage

I.1. Load from package repository (Typst 0.6.0 and later)	2
I.2. Dependencies	2

II. Drawing automata

II.1. Command reference	5
II.2. Styling the output	8
II.3. Using <code>#cetz.canvas()</code>	9
II.3.1. Element functions	10
II.3.2. Anchors	12
II.4. Layouts	13
II.4.1. Available layouts	13
II.4.2. Using layouts	20
II.4.3. Creating custom layouts	21
II.4.3.1. Using <code>#layout.custom()</code>	
21	
II.4.3.2. Creating a layout element	22
II.5. Utility functions	22
II.6. Doing other stuff with FINITE	25

III. Index

Part I.

Usage

I.1. Load from package repository (Typst 0.6.0 and later)

For Typst 0.6.0 and later, the package can be imported from the *preview* repository:

```
#import "@preview/finite:0.1.0": automaton
```

Alternatively, the package can be downloaded and saved into the system dependent local package repository.

Either download the current release from GitHub¹ and unpack the archive into your system dependent local repository folder² or clone it directly:

```
git clone https://github.com/jneug/typst-finite.git finite/0.1.0
```

In either case, make sure the files are placed in a subfolder with the correct version number: `finite/0.1.0`

After installing the package, just import it inside your `typ` file:

```
#import "@local/finite:0.1.0": automaton
```

I.2. Dependencies

`FINITE` loads `CeTZ` and the utility package `T4T` from the preview package repository. The dependencies will be downloaded by Typst automatically on first compilation.

¹<https://github.com/jneug/typst-finite>

²<https://github.com/typst/packages#local-packages>

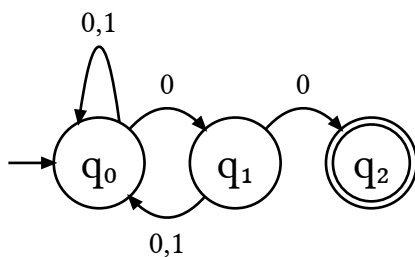
Part II.

Drawing automata

FINITE helps you draw transition diagrams for finite automata in your Typst documents, using the power of **CeTZ**.

To draw an automaton, simply import `#automaton()` from **FINITE** and use it like this:

```
1 #automaton((
2   q0: (q1:0, q0:"0,1"),
3   q1: (q0:(0,1), q2:"0"),
4   q2: none,
5 ))
```



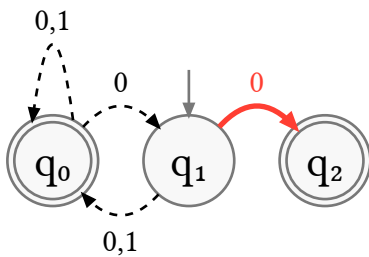
As you can see, an automaton is defined by a dictionary of dictionaries. The keys of the top-level dictionary are the names of states to draw. The second-level dictionaries have the names of connected states as keys and transition labels as values.

In the example above, the states `q0`, `q1` and `q2` are defined. `q0` is connected to `q1` and has a loop to itself. `q1` transitions to `q2` and back to `q0`. `#automaton()` selected the first state in the dictionary (in this case `q0`) to be the initial state and the last (`q2`) to be a final state.

To modify the defaults, `#automaton()` accepts a set of options:

2 Dependencies

```
1 #automaton(  
2   (  
3     q0: (q1:0, q0:"0,1"),  
4     q1: (q0:(0,1), q2:"0"),  
5     q2: (),  
6   ),  
7   initial: "q1",  
8   final: ("q0", "q2"),  
9   style:(  
10    state: (fill: luma(248), stroke:luma(120)),  
11    transition: (stroke: (dash:"dashed")),  
12    q1: (initial:top),  
13    q1-q2: (stroke: 2pt + red)  
14  )  
15 )
```

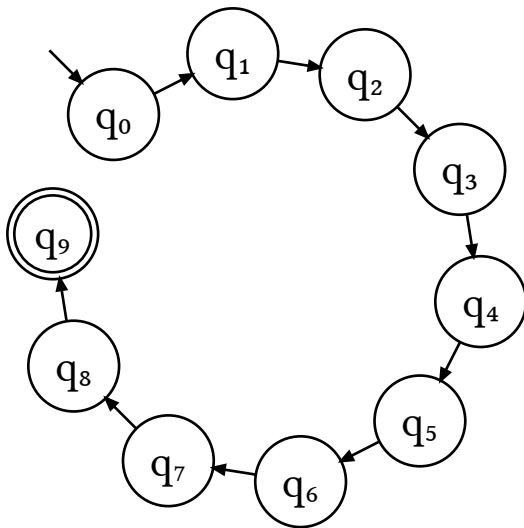


For larger automata, the states can be arranged in different ways:

```

1 #let aut = (:)
2 #for i in range(10) {
3   let name = "q"+str(i)
4   aut.insert(name, (:))
5   if i < 9 {
6     aut.at(name).insert("q" + str(i + 1), none)
7   }
8 }
9 #automaton(
10  aut,
11  layout: finite.layout.circular.with(offset: 45deg),
12  style: (
13    transition: (curve: 0),
14    q0: (initial: top+left)
15  )
16 )

```



See Section II.4 for more details about layouts.

II.1. Command reference

```

#automaton(
  states,
  initial: auto,
  final: auto,
  style: "(:)",
  label-format: (...) => ...,
  layout: "layout.linear",
  ..canvas-styles
)

```

Draw an automaton from a transition table.

The transition table `states` has to be a dictionary of dictionaries, having the names of all states as keys in the first level dictionary and names of states, the state has a transition to

2.1 Command reference

as keys in the second level dictionaries. The values in the second level dictionary are labels (inputs) for the transitions.

The following example, defines three states q_0 , q_1 and q_2 . For the input 0 q_0 transitions to q_1 and to q_2 for the inputs 0 and 1. q_1 transitions to q_0 for 0 and 1 and q_2 for 0. q_2 has no transitions.

```
#automaton((
  q0: (q1:0, q0:"0,1"),
  q1: (q0:(0,1), q2:"0"),
  q2: none
))
```

If no initial and final states are defined, `#automaton()` selects the first and last state in the dictionary respectively (q_0 and q_2 in this example).

As you can see, the transition labels can be provided as a single value or an array. Arrays are joined with a comma (,) to generate the final label.

For now, there is no difference in providing inputs as arrays or strings. Internally, string are split on commas, to get atomic symbols, and later joined again. Future versions might use these symbols, though, to actually simulate the automaton and decide if a word is accepted or not.

`initial` and `final` can be used to customize the initial and final states.

Argument

`states`

dictionary

A dictionary of dictionaries, defining the transition table of an automaton.

Argument

`initial: auto`

string | auto | none

The name of the initial state. For `auto`, the first state in `states` is used.

Argument

`final: auto`

array | auto | none

A list of final state names. For `auto`, the last state in `states` is used.

Argument

`style: "(:)"`

dictionary

A dictionary with styles for states and transitions.

Argument

`label-format: (...) => ...`

function

A function (`string`, `boolean`) => `string` to format labels. The function will get the label as a string and a boolean `is-state`, if the label is generated for a state (`true`) or a transition (`false`). It should return the final label as `content`.

2.1 Command reference

Argument

layout: "layout.linear"

function

A layout function. See below for more information on layouts.

Argument

..canvas-styles

any

Arguments for #cetz.canvas()

```
#transition-table(  
  states,  
  initial: auto,  
  final: auto,  
  format: (...) => ...,  
  format-list: (...) => ...,  
  ..table-style  
)
```

Displays a transition table for an automaton.

The format for states is the same as for automaton().

```
1 #finite.transition-table(  
2   q0: (q1: 0, q0: (1,0)),  
3   q1: (q0: 1, q2: (1,0)),  
4   q2: (q0: 1, q2: 0),  
5 ))
```

	0	1
q0	{q1,q0}	q0
q1	q2	{q0,q2}
q2	q2	q0

Argument

states

dictionary

A dictionary of dictionaries, defining the transition table of an automaton.

Argument

initial: auto

string | auto | none

The name of the initial state. For auto, the first state in states is used.

Argument

final: auto

array | auto | none

A list of final state names. For auto, the last state in states is used.

Argument

`..table-style`

any

Arguments for table.

II.2. Styling the output

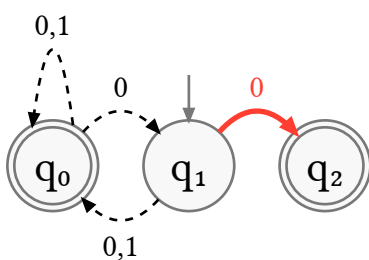
As common in `CeTZ`, you can pass general styles for states and transitions to the `#cetz.set-style()` function within a call to `#cetz.canvas()`. The elements functions `#state()` and `#transition()` (see below) can take their respective styling options as arguments, to style individual elements.

`automaton` takes a `style` argument that passes the given style to the above functions. The example below sets a background and stroke color for all states and draws transitions with a dashed style. Additionally, the state `q1` has the arrow indicating an initial state drawn from above instead from the left. The transition from `q1` to `q2` is highlighted in red.

```

1  #automaton(
2    (
3      q0: (q1:0, q0:"0,1"),
4      q1: (q0:(0,1), q2:"0"),
5      q2: (),
6    ),
7    initial: "q1",
8    final: ("q0", "q2"),
9    style:(
10     state: (fill: luma(248), stroke:luma(120)),
11     transition: (stroke: (dash:"dashed")),
12     q1: (initial:top),
13     q1-q2: (stroke: 2pt + red)
14   )
15 )

```



Every state can be accessed by its name and every transition is named with its initial and end state joined with a dash (-).

The supported styling options (and their defaults) are as follows:

- states:
 - fill:** `auto` Background fill for states.
 - stroke:** `auto` Stroke for state borders.
 - radius:** `0.6` Radius of states.
- label:

- text:** **auto** Default state label.
- size:** **auto** Default text size for state labels.
- transitions
 - curve:** **1.0** Curviness of transitions. Set to **0** to get straight lines.
 - stroke:** **auto** Stroke for transitions.
- label:
 - text:** **""** Default transition label.
 - size:** **1em** Default size for label text.
 - color:** **auto** Color for label text.
 - pos:** **0.5** Position on the transition, between **0** and **1**. **0** sets the text at the initial, **1** at the end of the transition.
 - dist:** **0.33** Distance of the label from the transition.
 - angle:** **auto** Angle of the label text. **auto** will set the angle based on the transitions direction.

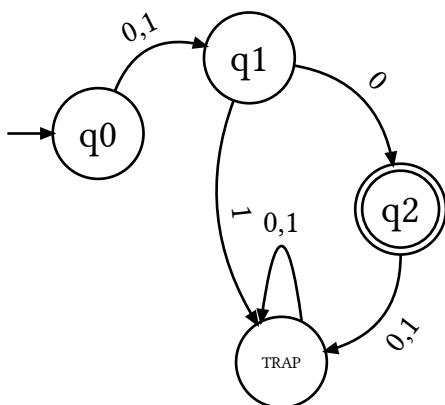
II.3. Using `#cetz.canvas()`

The above commands use custom `CeTZ` elements to draw states and transitions. For complex automata, the functions in the `draw` module can be used inside a call to `#cetz.canvas()`.

```

1  #cetz.canvas({
2      import cetz.draw: set-style
3      import finite.draw: state, transition
4
5      state((0,0), "q0", initial:true)
6      state((2,1), "q1")
7      state((4,-1), "q2", final:true)
8      state((rel:(0, -3), to:"q1.bottom"), "trap", label:"TRAP", anchor:"top-
9  left")
10
11     transition("q0", "q1", inputs:(0,1))
12     transition("q1", "q2", inputs:(0))
13     transition("q1", "trap", inputs:(1), curve:-1)
14     transition("q2", "trap", inputs:(0,1))
15     transition("trap", "trap", inputs:(0,1))
16 })

```

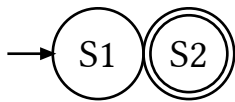


II.3.1. Element functions

```
#state(
  position,
  name,
  label: auto,
  initial: "false",
  final: "false",
  anchor: "center",
  ..style
)
```

Draw a state at the given position.

```
1 #cetz.canvas({
2   import finite.draw: state
3   state((0,0), "q1", label:"S1", initial:true)
4   state("q1.right", "q2", label:"S2", final:true, anchor:"left")
5 })
```



—Argument—

position

coordinate

Position of the states center.

—Argument—

name

string

Name for the state.

—Argument—

label: auto

string | content | auto | none

Label for the state. If set to **auto**, the name is used.

—Argument—

initial: "false"

boolean | alignment

Whether this is an initial state.

—Argument—

final: "false"

boolean

Whether this is a final state.

—Argument—

anchor: "center"

string

Anchor to use for drawing.

2.3.1 Using `#cetz.canvas()`

Argument

`..style`

any

Styling options.

`#transition(`

`from,`

`to,`

`inputs: none,`

`label: auto,`

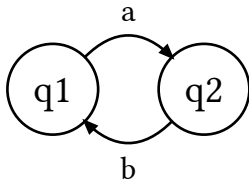
`..style`

`)`

Draw a transition between two states.

The two states `from` and `to` have to be existing names of states.

```
1 #cetz.canvas({
2     import finite.draw: state, transition
3     state((0,0), "q1")
4     state((2,0), "q2")
5     transition("q1", "q2", label:"a")
6     transition("q2", "q1", label:"b")
7 })
```



Argument

`from`

string

Name of the starting state.

Argument

`to`

string

Name of the ending state.

Argument

`inputs: none`

string | array | none

A list of atomic input symbols for the transition. If provided as a `string`, it is split on commas to get the list of input symbols.

Argument

`label: auto`

string | content | auto | dictionary

2.3.1 Using `#cetz.canvas()`

A label for the transition. For `auto` the input symbols are joined with commas. Can be a `dictionary` with a text and additional styling keys.

—Argument—

`..style`

any

Styling options.

`#transitions(states, ..style)`

Draws all transitions from a transition table with a common style.

—Argument—

`states`

dictionary

A transition table given as a dictionary of dictionaries.

—Argument—

`..style`

any

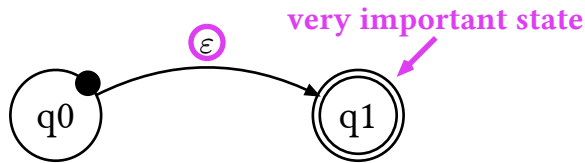
Styling options.

II.3.2. Anchors

States have the common anchors (like `top`, `top-left` ...), transitions have a `initial`, `end`, `center` and `label` anchors. These can be used to add elements to an automaton:

```
1 #cetz.canvas({
2   import cetz.draw: circle, line, place-marks, content
3   import finite.draw: state, transition
4
5   state((0,0), "q0")
6   state((4,0), "q1", final:true)
7   transition("q0", "q1", label:$epsilon$)
8
9   circle("q0.top-right", radius:.4em, stroke:none, fill:black)
10
11   let magenta-stroke = 2pt+rgb("#dc41f1")
12   circle("q0-q1.label", radius:.5em, stroke:magenta-stroke)
13   place-marks(
14     line(
15       name: "q0-arrow",
16       (rel:(.6,.6), to:"q1.top-right"),
17       (rel:(.15,.15), to:"q1.top-right"),
18       stroke:magenta-stroke
19     ),
20     (mark:">", pos:1, stroke:magenta-stroke)
21   )
22   content(
23     (rel:(0,.25), to:"q0-arrow.start"),
24     text(fill:rgb("#dc41f1"), [*very important state*])
25   )
26 }
```

```
26    )
    })
```



II.4. Layouts

Layouts can be used to move states to new positions within a call to `#cetz.canvas()`. They act similar to `CETZ` groups and have their own transform. Any other elements than states will keep their original coordinates, but be translated by the layout, if necessary.

`FINITE` ships with a bunch of layouts, to accomodate different scenarios.

II.4.1. Available layouts

`#circular()`

`#custom()`

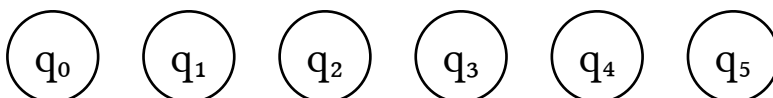
`#grid()`

```
#linear(
  position,
  name: none,
  anchor: "left",
  dir: right,
  spacing: 0.6,
  body
)
```

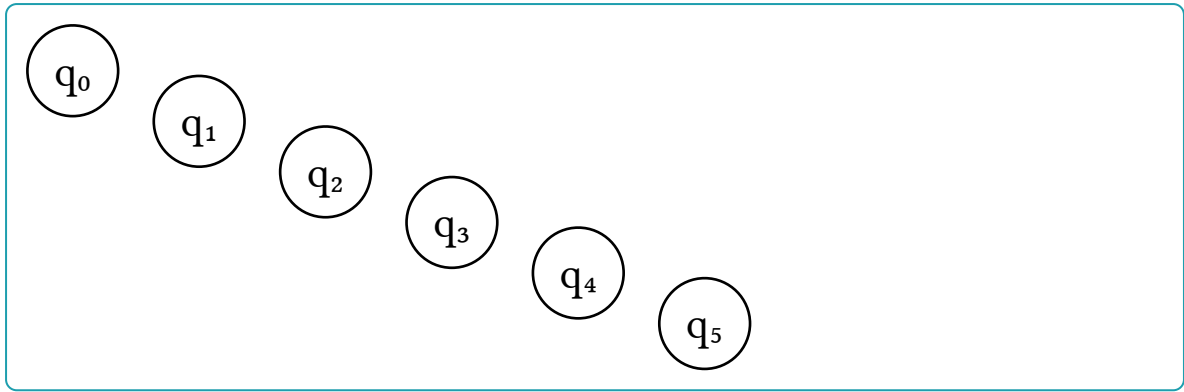
Arrange states in a line.

The direction of the line can be set via `dir` either to an `alignment` or a vector with a x and y shift.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #finite.automaton(
3   aut,
4   initial: none, final: none,
5   layout: finite.layout.linear.with(dir: right)
6 )
7 #finite.automaton(
8   aut,
9   initial: none, final: none,
10  layout: finite.layout.linear.with(dir: (.5, -.2))
11 )
```



2.4.1 Layouts



Argument
position coordinate
Position of the anchor point.

Argument
name: `none` string
Name for the element to access later.

Argument
anchor: `"left"` string
Name of the anchor to use for the layout.

Argument
dir: `right` vector | alignment | 2d alignment
Direction of the line.

Argument
spacing: `0.6` float
Spacing between states on the line.

Argument
body array
Array of `CETZ` elements to draw.

```
#circular(  
  position,  
  name: none,  
  anchor: "left",  
  dir: right,  
  spacing: 0.6,  
  radius: auto,  
  offset: 0deg,  
  body  
)
```

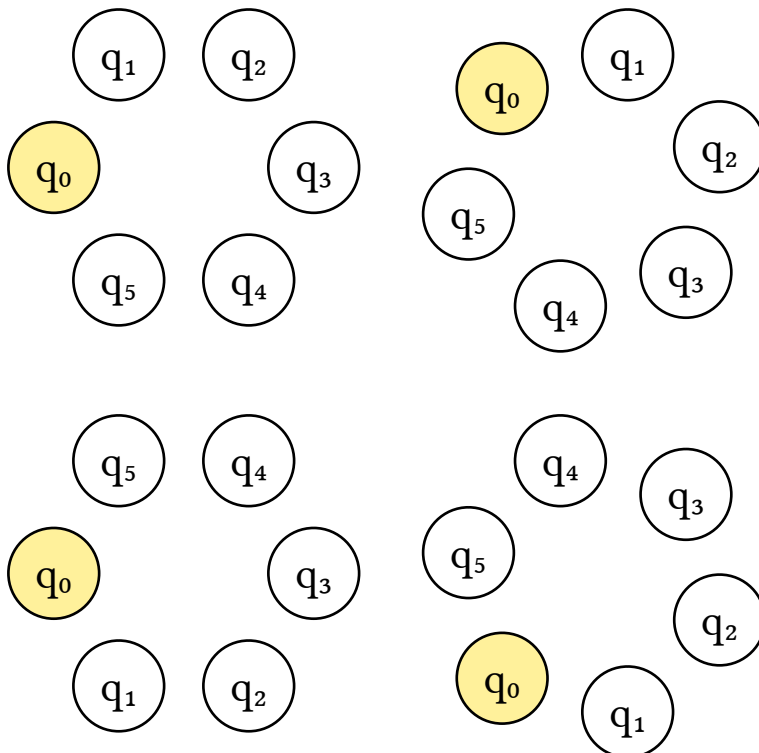
2.4.1 Layouts

Arrange states in a circle.

```

1  #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2  #grid(columns: 2, gutter: 2em,
3    finite.automaton(
4      aut,
5      initial: none, final: none,
6      layout:finite.layout.circular,
7      style: (q0: (fill: yellow.lighten(60%)))
8    ),
9    finite.automaton(
10     aut,
11     initial: none, final: none,
12     layout:finite.layout.circular.with(offset:45deg),
13     style: (q0: (fill: yellow.lighten(60%)))
14   ),
15   finite.automaton(
16     aut,
17     initial: none, final: none,
18     layout:finite.layout.circular.with(dir:left),
19     style: (q0: (fill: yellow.lighten(60%)))
20   ),
21   finite.automaton(
22     aut,
23     initial: none, final: none,
24     layout:finite.layout.circular.with(dir:left, offset:45deg),
25     style: (q0: (fill: yellow.lighten(60%)))
26   )
27 )

```



Argument
position

coordinate

2.4.1 Layouts

Position of the anchor point.

Argument

name: `none`

string

Name for the element to access later.

Argument

anchor: `"left"`

string

Name of the anchor to use for the layout.

Argument

dir: `right`

alignment

Direction of the circle. Either `left` or `right`.

Argument

spacing: `0.6`

float

Spacing between states on the line.

Argument

radius: `auto`

float | auto

Either a fixed radius or `auto` to calculate a suitable the radius.

Argument

offset: `0deg`

angle

An offset angle to place the first state at.

Argument

body

array

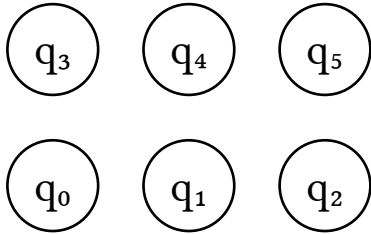
Array of `CETZ` elements to draw.

```
#grid(  
  position,  
  name: none,  
  anchor: "left",  
  columns: 4,  
  spacing: 0.6,  
  body  
)
```

Arrange states in rows and columns.

2.4.1 Layouts

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #finite.automaton(
3   aut,
4   initial: none, final: none,
5   layout: finite.layout.grid.with(columns: 3)
6 )
```



Argument

position

coordinate

Position of the anchor point.

Argument

name: none

string

Name for the element to access later.

Argument

anchor: "left"

string

Name of the anchor to use for the layout.

Argument

columns: 4

integer

Number of columns per row.

Argument

spacing: 0.6

float

Spacing between states on the line.

Argument

body

array

Array of `CETZ` elements to draw.

```
#snake(
  position,
  name: none,
  anchor: "left",
  columns: 4,
  spacing: 0.6,
```

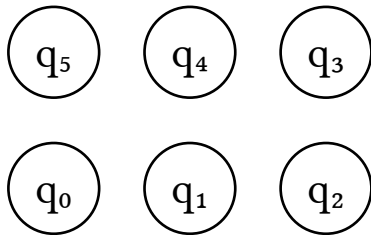
2.4.1 Layouts

body

)

Arrange states in a grid, but alternate the direction in every even and odd row.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #finite.automaton(
3   aut,
4   initial: none, final: none,
5   layout: finite.layout.snake.with(columns: 3)
6 )
```



Argument

position

coordinate

Position of the anchor point.

Argument

name: none

string

Name for the element to access later.

Argument

anchor: "left"

string

Name of the anchor to use for the layout.

Argument

columns: 4

integer

Number of columns per row.

Argument

spacing: 0.6

float

Spacing between states on the line.

Argument

body

array

Array of [CeTZ](#) elements to draw.

```
#custom(  
  position,  
  name: none,
```

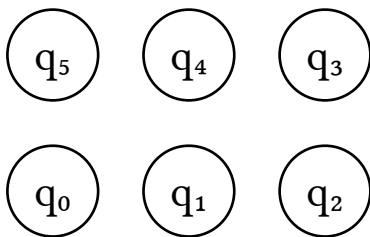
2.4.1 Layouts

```
anchor: "left",
positions: (...) => ...,
body
)
```

Create a custom layout from a positioning function.

See “Creating custom layouts” for more information.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #finite.automaton(
3   aut,
4   initial: none, final: none,
5   layout: finite.layout.snake.with(columns:3)
6 )
```



Argument

position

coordinate

Position of the anchor point.

Argument

name: none

string

Name for the element to access later.

Argument

anchor: "left"

string

Name of the anchor to use for the layout.

Argument

positions: (...) => ...

function

A function (dictionary, dictionary, array) => dictionary to compute coordinates for each state.

The function gets the current **CeTZ** context, a dictionary of computed radii for each state and a list with all state elements to position. The returned dictionary contains each states name as a key and the new coordinate as a value.

Argument

body

array

Array of **CeTZ** elements to draw.

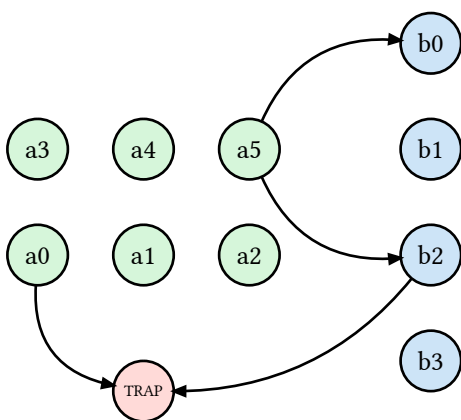
II.4.2. Using layouts

Layouts are elements themselves. This means, they have a coordinate to be moved on the canvas and they can have anchors. Using layouts allows you to quickly create complex automata, without the need to pick each states coordinate by hand.

```

1  #cetz.canvas({
2      import cetz.draw: set-style
3      import finite.draw: *
4
5      set-style(state: (radius: .4))
6
7      layout.grid(
8          (0,0),
9          name:"grid", columns:3, {
10             set-style(state: (fill: green.lighten(80%)))
11             for s in range(6) {
12                 state(( ), "a" + str(s))
13             }
14         })
15
16      layout.linear(
17          (rel:(2,0), to:"grid.right"),
18          dir: bottom, anchor: "center", {
19             set-style(state: (fill: blue.lighten(80%)))
20             for s in range(4) {
21                 state(( ), "b" + str(s))
22             }
23         })
24
25      state((rel: (0, -1.4), to:"grid.bottom"), "TRAP", fill:red.lighten(80%))
26
27      transition("a0", "TRAP", curve:-1)
28      transition("b2", "TRAP")
29      transition("a5", "b0")
30      transition("a5", "b2", curve:-1)
31  })

```



II.4.3. Creating custom layouts

There are two ways to create custom layouts. Using the `#layout.custom()` layout or building your own from the ground up.

II.4.3.1. Using `#layout.custom()`

The custom layout passes information about states to a `positions` function, that computes a dictionary with new coordinates for all states. The custom layout will then place the states at the given locations and handle any other elements other than states.

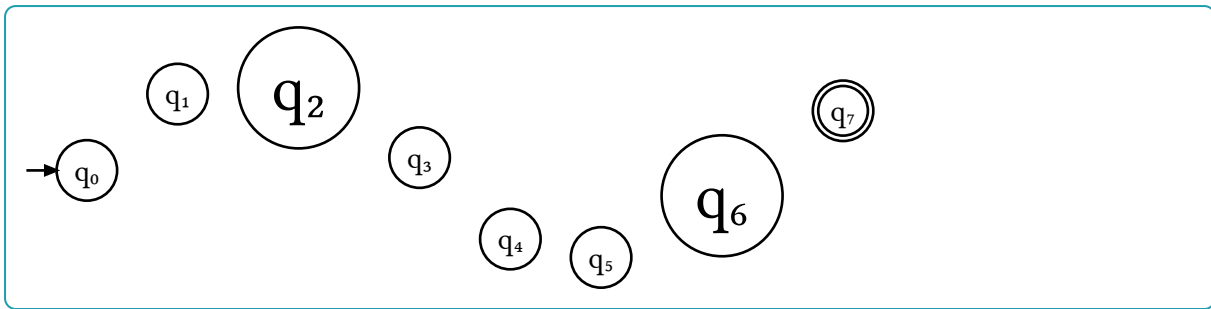
The position function gets passed the `CETZ` context, a dictionary of state names and matching radii (for drawing the states circle) and the list of `#state()` elements.

This example arranges the states in a wave:

```

1  #let wave-layout = finite.layout.custom.with(
2    positions: (ctx, radii, states) => {
3      let (i, at) = (0, 0)
4      let pos = (:)
5      for (name, r) in radii {
6        at += r
7        pos.insert(name, (at, 1.2 * calc.sin(i)))
8        i += 1
9        at += r + .4
10     }
11     return pos
12   }
13 )
14
15 #let aut = (:)
16 #for i in range(8) {
17   aut.insert("q"+str(i), none)
18 }
19
20 #automaton(
21   aut,
22   layout: wave-layout,
23   style: (
24     state: (radius: .4),
25     q2: (radius: .8),
26     q6: (radius: .8)
27   )
28 )

```



II.4.3.2. Creating a layout element

Layout are elements and are similar to [CETZ](#)'s groups. A layout takes an array of elements and computes a new positions for each state in the list.

To create a layout, [FINITE](#) provides a base element that can be extended. A basic layout can look like this:

```

1 #let my-layout(
2   position, name: none, anchor: "left", body
3 ) = {
4   // Layouts always need to have a name.
5   // If none was provided, we create one.
6   if is.n(name) {
7     name = "layout" + body.map((e) => e.at("name", default:"")).join("-")
8   }
9
10  // Get the base layout element
11  let layout = base(position, name, anchor, body)
12
13  // We need to supply a function to compute new locations for the elements.
14  layout.children = (ctx) => {
15    let elements = ()
16    for element in elements {
17      // states have a custom "radius" key
18      if "radius" in element {
19        // Change the position of the state
20        element.coordinates = ((rel:(.6,0)),)
21      }
22      elements.push(element)
23    }
24    elements
25  }
26
27  return (layout,)
28 }
```

II.5. Utility functions

#align-to-vec()	#label-pt()	#prepare-ctx()
#ctrl-pt()	#mark-dir()	#quadratic-normal()
#fit-content()	#mid-point()	#transition-pts()

2.5 Utility functions

#vector-set-len(v, len)

Set the length of a vector.

#vector-normal(v)

Compute a normal for a 2d vector. The normal will be pointing to the right of the original vector.

#align-to-vec(a)

Returns a vector for an alignment.

#quadratic-normal(a, b, c, t)

Compute a normal vector for a point on a quadratic bezier curve.

#mid-point(a, b, c)

Compute the mid point of a quadratic bezier curve.

#ctrl-pt(a, b, curve: 1)

Calculate the controlpoint for a transition.

#mark-dir(a, b, c, scale: 1)

Calculate the direction vector for a transition mark (arrowhead)

#label-pt(

a,

b,

c,

style,

loop: "false"

)

Calculate the location for a transitions label, based on its bezier points.

2.5 Utility functions

```
#transition-pts(  
  start,  
  end,  
  start-radius,  
  end-radius,  
  curve: 1  
)
```

Calculate start, end and ctrl points for a transition.

—Argument—

start

vector

Center of the start state.

—Argument—

end

vector

Center of the end state.

—Argument—

start-radius

length

Radius of the start state.

—Argument—

end-radius

length

Radius of the end state.

—Argument—

curve: 1

float

Curvature of the transition.

```
#fit-content(  
  ctx,  
  width,  
  height,  
  content,  
  size: auto,  
  min-size: "6pt"  
)
```

Fits (text) content inside the available space.

—Argument—

ctx

dictionary

The canvas context.

—Argument—

content

string | content

The content to fit.

Argument

size: **auto**

length | **auto**

The initial text size.

Argument

min-size: **"6pt"**

length

The minimal text size to set.

#prepare-ctx(ctx)

Prepares the CeTZ context for use with finite

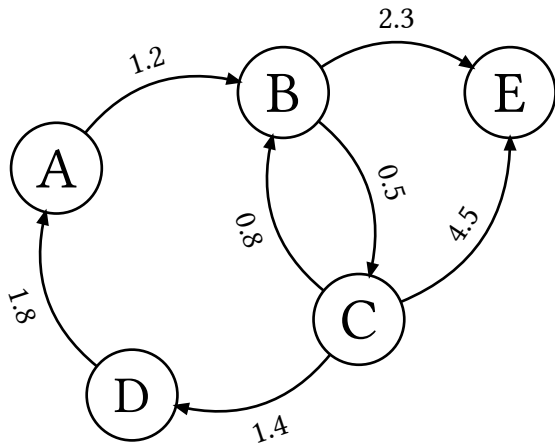
II.6. Doing other stuff with **finite**

Since transition diagrams are effectively graphs, **FINITE** could also be used to draw graph structures:

```

1  #cetz.canvas({
2    import cetz.draw: set-style
3    import finite.draw: state, transitions
4
5    state((0,0), "A")
6    state((3,1), "B")
7    state((4,-2), "C")
8    state((1,-3), "D")
9    state((6,1), "E")
10
11    transitions((
12      A: (B: 1.2),
13      B: (C: .5, E: 2.3),
14      C: (B: .8, D: 1.4, E: 4.5),
15      D: (A: 1.8),
16      E: (:)
17    ),
18    C-E: (curve: -1.2))
19 })

```



Part III.

Index

A

#align-to-vec 23

#automaton 3, 5

C

#circular 14

#ctrl-pt 23

#custom 1, 18, 21

F

#fit-content 24

G

#grid 16

L

#label-pt 23

#linear 13

M

#mark-dir 23

#mid-point 23

P

#prepare-ctx 25

Q

#quadratic-normal 23

S

#snake 17

#state 10

T

#transition 11

#transition-pts 24

#transition-table 7

#transitions 12

V

#vector-normal 23

#vector-set-len 23