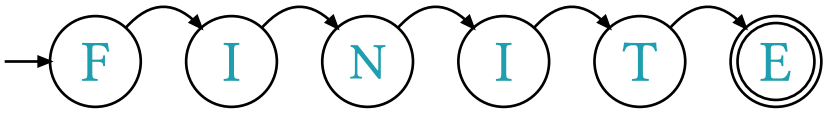


v0.1.0 2023-09-13 MIT

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 4

II.2. Styling the output 7

II.3. Using `#cetz.canvas()` 8

II.3.1. Element functions 9

II.3.2. Anchors 12

II.4. Layouts 12

II.4.1. Available layouts 12

II.4.2. Using layouts 19

II.4.3. Creating custom layouts 20

II.4.3.1. Using `#layout.custom()` 20

II.4.3.2. Creating a layout element 21

II.5. Utility functions 22

II.6. Doing other stuff with finite 24

III. Showcase

IV. 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 GitHub[1] and unpack the archive into your system dependent local repository folder[2] 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.

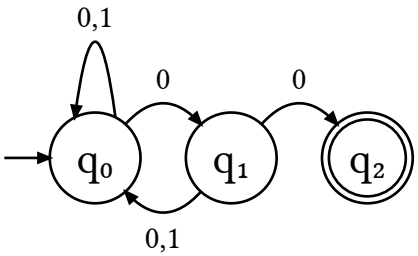
[1] https://github.com/jneug/typst-finite
[2] https://github.com/typst/packages#local-packages

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:

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

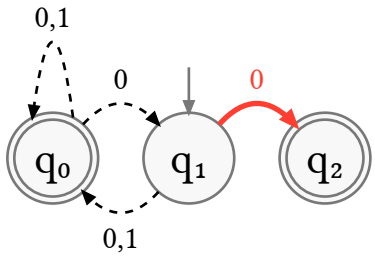


As you can see, an automaton ist defined by a dictionary of dictionaries. The keys of the top-level dictionary are the names of states to draw. The second-level dictionaries have the names of connected states as keys and transition labels as values.

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 initiat state and the last (`q2`) to be a final state.

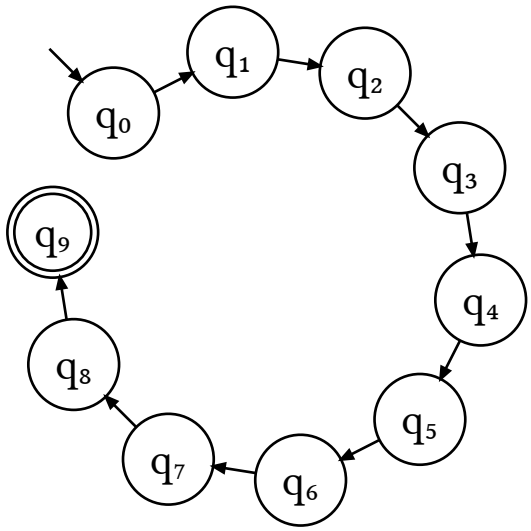
To modify the defaults, `#automaton()` accepts a set of options:

```
#automaton(
  (
    q0: (q1:0, q0:"0,1"),
    q1: (q0:(0,1), q2:"0"),
    q2: (),
  ),
  initial: "q1",
  final: ("q0", "q2"),
  style:(
    state: (fill: luma(248), stroke:luma(120)),
    transition: (stroke: (dash:"dashed")),
    q1: (initial:top),
    q1-q2: (stroke: 2pt + red)
  )
)
```



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

```
#let aut = (:)
#for i in range(10) {
  let name = "q"+str(i)
  aut.insert(name, (:))
  if i < 9 {
    aut.at(name).insert("q" + str(i + 1), none)
  }
}
#automaton(
  aut,
  layout: finite.layout.circular.with(offset: 45deg),
  style: (
    transition: (curve: 0),
    q0: (initial: top+left)
  )
)
```



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 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 `q0`, `q1` and `q2`. For the input `0` `q0` transitions to `q1` and to `q2` for the inputs `0` and `1`. `q1` transitions to `q0` for `0` and `1` and `q2` for `0`. `q2` 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 (`q0` and `q2` 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.

`inital` 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`.

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()`.

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

	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`

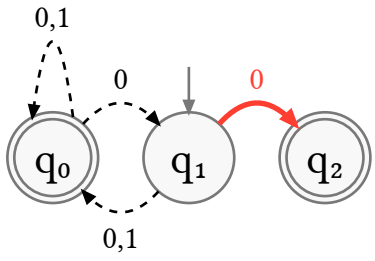
Arguments for `table`.

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.

```
#automaton(
  (
    q0: (q1:0, q0:"0,1"),
    q1: (q0:(0,1), q2:"0"),
    q2: (),
  ),
  initial: "q1",
  final: ("q0", "q2"),
  style:(
    state: (fill: luma(248), stroke:luma(120)),
    transition: (stroke: (dash:"dashed")),
    q1: (initial:top),
    q1-q2: (stroke: 2pt + red)
  )
)
```



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

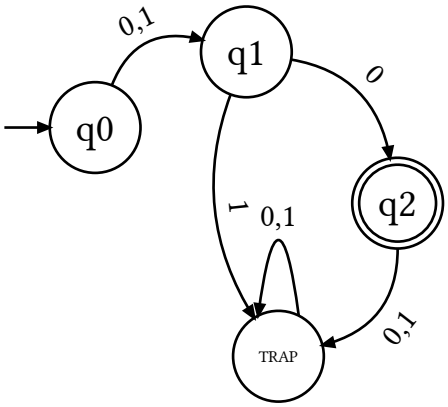
The supported styling options (and their defaults) are as follows:

- 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()`.

```
#cetz.canvas({
  import cetz.draw: set-style
  import finite.draw: state, transition

  state((0,0), "q0", initial:true)
  state((2,1), "q1")
  state((4,-1), "q2", final:true)
  state((rel:(0, -3), to:"q1.bottom"), "trap", label:"TRAP", anchor:"top-
left")

  transition("q0", "q1", inputs:(0,1))
  transition("q1", "q2", inputs:(0))
  transition("q1", "trap", inputs:(1), curve:-1)
  transition("q2", "trap", inputs:(0,1))
  transition("trap", "trap", inputs:(0,1))
})
```



II.3.1. Element functions

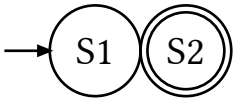
```
#content-box(
  a,
  b,
  cnt,
  angle: 0deg,
  clip: "false",
  anchor: none,
  name: none,
  ..style-args
)
```

Draws content but tries to fit the text into the box defined by the given coordinates.

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

Draw a state at the given `position`.

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



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.

Argument

`..style` `any`

Styling options.

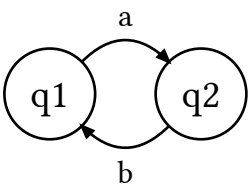
```
#transition(
  from,
  to,
  inputs: none,
  label: auto,
  ..style
)
```

Draw a transition between two states.

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

2.3.1 Using `#cetz.canvas()`

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



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`

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

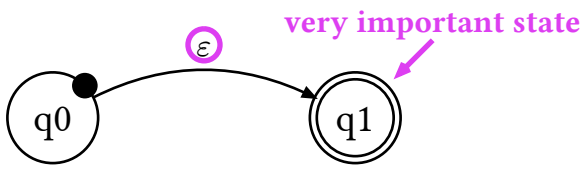
`..style` `any`

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:

```
#cetz.canvas({
  import cetz.draw: circle, line, place-marks, content
  import finite.draw: state, transition

  state((0,0), "q0")
  state((4,0), "q1", final:true)
  transition("q0", "q1", label:$epsilon$)

  circle("q0.top-right", radius:.4em, stroke:none, fill:black)

  let magenta-stroke = 2pt+rgb("#dc41f1")
  circle("q0-q1.label", radius:.5em, stroke:magenta-stroke)
  place-marks(
    line(
      name: "q0-arrow",
      (rel:(.6,.6), to:"q1.top-right"),
      (rel:(.15,.15), to:"q1.top-right"),
      stroke:magenta-stroke
    ),
    (mark:">", pos:1, stroke:magenta-stroke)
  )
  content(
    (rel:(0,.25), to:"q0-arrow.start"),
    text(fill:rgb("#dc41f1"), [*very important state*])
  )
})
```



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

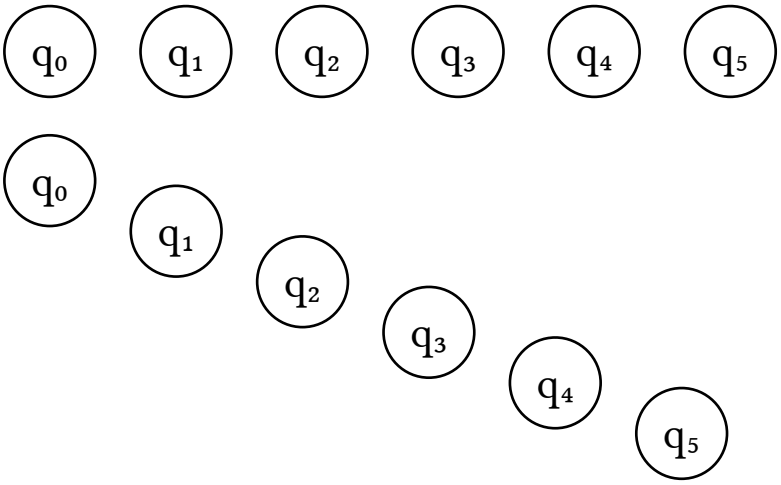
```
#circular()                #custom()                #grid()
```

```
#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.

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



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

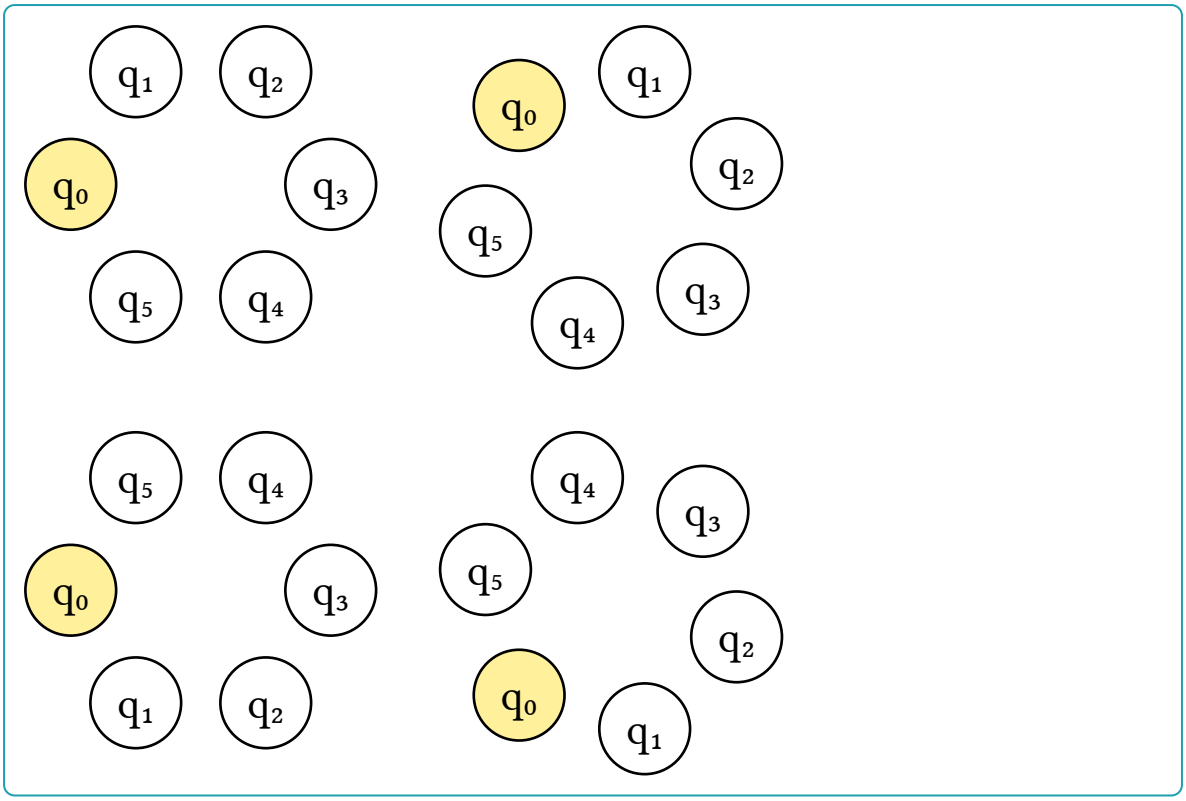
`body` `array`

Array of CeTZ elements to draw.

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

Arrange states in a circle.

```
#let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
#grid(columns: 2, gutter: 2em,
  finite.automaton(
    aut,
    initial: none, final: none,
    layout:finite.layout.circular,
    style: (q0: (fill: yellow.lighten(60%)))
  ),
  finite.automaton(
    aut,
    initial: none, final: none,
    layout:finite.layout.circular.with(offset:45deg),
    style: (q0: (fill: yellow.lighten(60%)))
  ),
  finite.automaton(
    aut,
    initial: none, final: none,
    layout:finite.layout.circular.with(dir:left),
    style: (q0: (fill: yellow.lighten(60%)))
  ),
  finite.automaton(
    aut,
    initial: none, final: none,
    layout:finite.layout.circular.with(dir:left, offset:45deg),
    style: (q0: (fill: yellow.lighten(60%)))
  )
)
```



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` `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

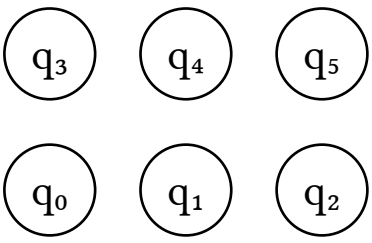
`body` `array`

Array of CeTZ elements to draw.

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

Arrange states in rows and columns.

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



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

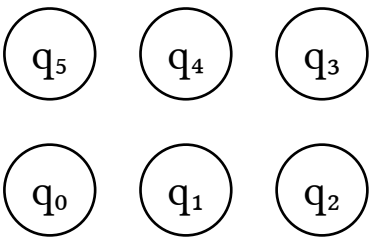
`body` `array`

Array of CeTZ elements to draw.

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

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

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



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`

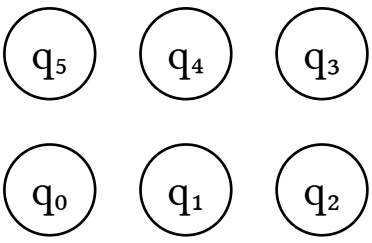
Array of CeTZ elements to draw.

```
#custom(
  position,
  name: none,
  anchor: ""left"",
  positions: (...) => ...,
  body
)
```

Create a custom layout from a positioning function.

See “Creating custom layouts” for more information.

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



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

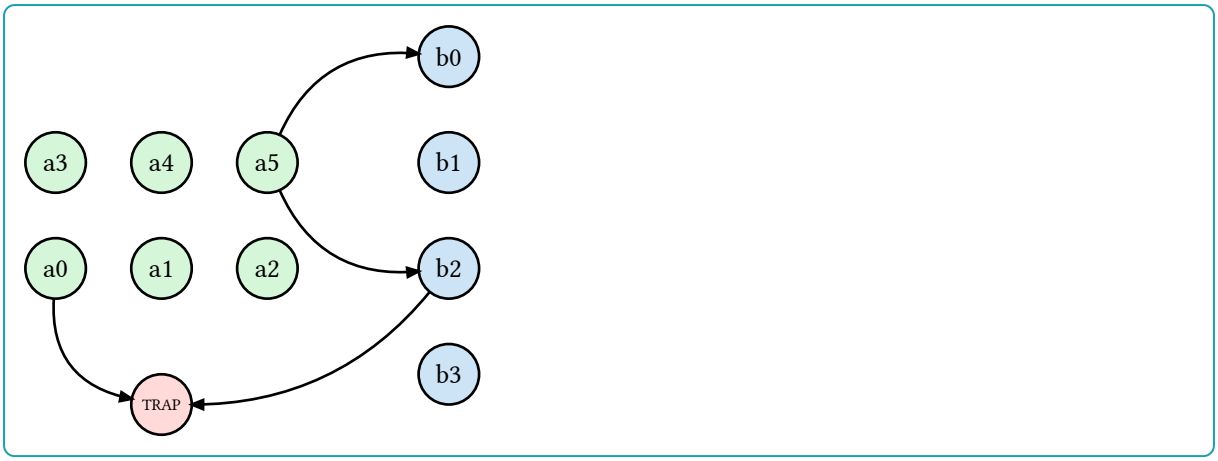
`body` `array`

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.

```
#cetz.canvas({
  import cetz.draw: set-style
  import finite.draw: *

  set-style(state: (radius: .4))

  layout.grid(
    (0,0),
    name:"grid", columns:3, {
      set-style(state: (fill: green.lighten(80%)))
      for s in range(6) {
        state((), "a" + str(s))
      }
    })

  layout.linear(
    (rel:(2,0), to:"grid.right"),
    dir: bottom, anchor: "center", {
      set-style(state: (fill: blue.lighten(80%)))
      for s in range(4) {
        state((), "b" + str(s))
      }
    })

  state((rel: (0, -1.4), to:"grid.bottom"), "TRAP", fill:red.lighten(80%))

  transition("a0", "TRAP", curve:-1)
  transition("b2", "TRAP")
  transition("a5", "b0")
  transition("a5", "b2", curve:-1)
})
```



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()`

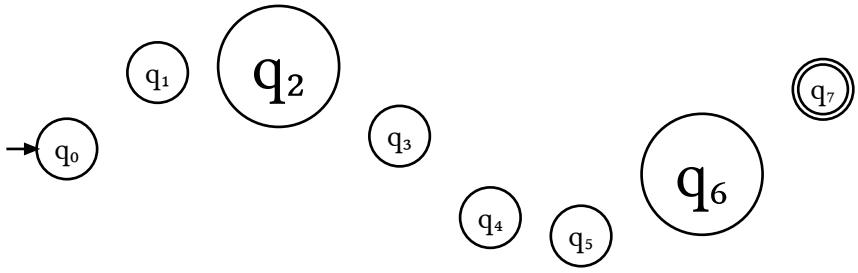
The `custom` layout passes information about states to a `positions` function, that computes a dictionary with new coordinates for all states. The `custom` layout will then place the states at the given locations and handle any other elements other than states.

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:

```
#let wave-layout = finite.layout.custom.with(
  positions: (ctx, radii, states) => {
    let (i, at) = (0, 0)
    let pos = (:)
    for (name, r) in radii {
      at += r
      pos.insert(name, (at, 1.2 * calc.sin(i)))
      i += 1
      at += r + .4
    }
    return pos
  }
)

#let aut = (:)
#for i in range(8) {
  aut.insert("q"+str(i), none)
}

#automaton(
  aut,
  layout: wave-layout,
  style: (
```

```
    state: (radius: .4),
    q2: (radius: .8),
    q6: (radius: .8)
  )
)
```



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:

```
#let my-layout(
  position, name: none, anchor: "left", body
) = {
  // Layouts always need to have a name.
  // If none was provided, we create one.
  if is.n(name) {
    name = "layout" + body.map((e) => e.at("name", default:"")).join("-")
  }

  // Get the base layout element
  let layout = base(position, name, anchor, body)

  // We need to supply a function to compute new locations for the elements.
  layout.children = (ctx) => {
    let elements = ()
    for element in elements {
      // states have a custom "radius" key
      if "radius" in element {
        // Change the position of the state
        element.coordinates = ((rel:(.6,0)),)
      }
      elements.push(element)
    }
    elements
  }

  return (layout,)
}
```

II.5. Utility functions

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

#vector-set-len(v, len)

Set the length of a vactor.

#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.

```
#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.

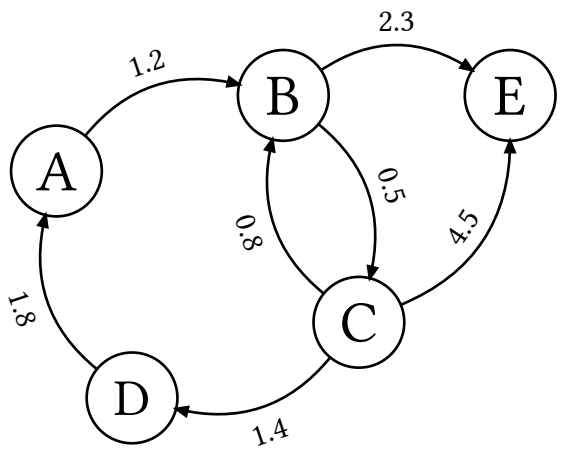
```
#prepare-ctx(ctx, force: "false")
```

Prepares the CeTZ context for use with finite

II.6. Doing other stuff with finite

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

```
#cetz.canvas({
  import cetz.draw: set-style
  import finite.draw: state, transitions

  state((0,0), "A")
  state((3,1), "B")
  state((4,-2), "C")
  state((1,-3), "D")
  state((6,1), "E")

  transitions((
      A: (B: 1.2),
      B: (C: .5, E: 2.3),
      C: (B: .8, D: 1.4, E: 4.5),
      D: (A: 1.8),
      E: (:)
    ),
    C-E: (curve: -1.2))
})
```



Part III.
Showcase

Part IV.
Index

A
#align-to-vec 22
#automaton 3, 4

C
#circular 14
#content-box 9
#ctrl-pt 22
#custom 1, 18, 20

F
#fit-content 23

G
#grid 16

L
#label-pt 22
#linear 13

M
#mark-dir 22
#mid-point 22

P
#prepare-ctx 23

Q
#quadratic-normal 22

S
#snake 17
#state 9

T
#transition 10
#transition-pts 22
#transition-table 6
#transitions 11

V
#vector-normal 22
#vector-set-len 22