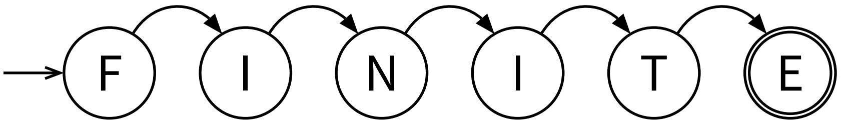


Version 0.4.1 2024-12-09 MIT

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

Part I Usage

I.1 Importing the package

Import the package in your Typst file:

```
#import "@preview/finite:0.4.1": automaton
```

I.2 Manual installation

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

Either download the current release from jneug/typst-finite[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 finite/0.4.1
```

In either case, make sure the files are placed in a subfolder with the correct version number: `finite/0.4.1`

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

```
#import "@local/finite:0.4.1": automaton
```

I.3 Dependencies

FINITE loads CeTZ[3] and the utility package T4T[4] from the `preview` package repository. The dependencies will be downloaded by Typst automatically on first compilation.

Whenever a `coordinate` type is referenced, a CeTZ coordinate can be used. Please refer to the CeTZ manual for further information on coordinate systems.

[1] https://github.com/jneug/typst-finite

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

[3] https://github.com/johannes-wolf/typst-canvas

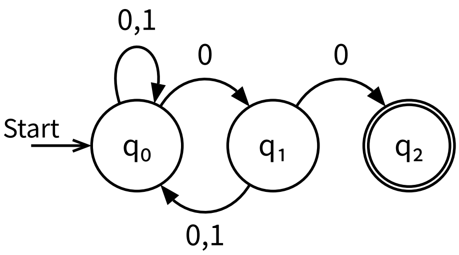
[4] https://github.com/jneug/typst-tools4typst

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,
))
```



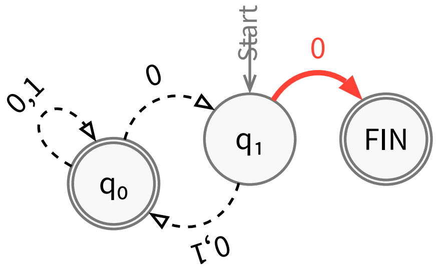
As you can see, an automaton ist defined by a dictionary of dictionaries. The keys of the top-level dictionary are the names of states to draw. The second-level dictionaries have the names of connected states as keys and transition labels as values.

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

See Section II.1 for more details on how to specify automata.

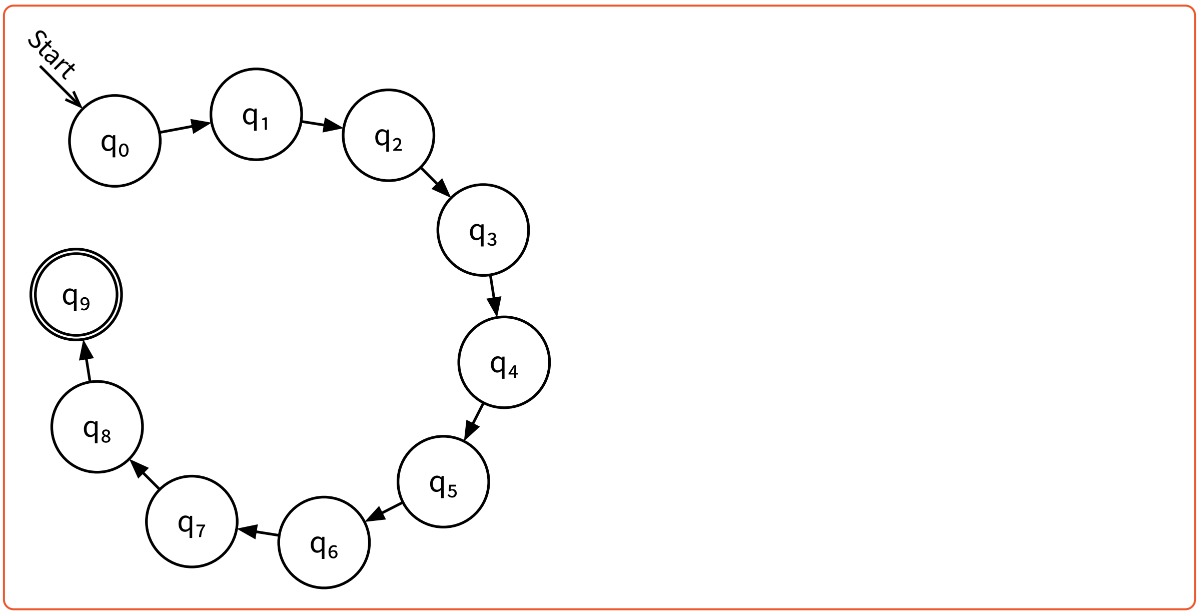
To modify how the transition diagram is displayed, #automaton accepts a set of options:

```
#automaton(
  (
    q0: (q1:0, q0:"0,1"),
    q1: (q0:(0,1), q2:"0"),
    q2: (),
  ),
  initial: "q1",
  final: ("q0", "q2"),
  labels:(
    q2: "FIN"
  ),
  style:(
    state: (fill: luma(248), stroke:luma(120)),
    transition: (stroke: (dash:"dashed")),
    q0-q0: (anchor:top+left),
    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.5 for more details about layouts.

II.1 Specifing finite automata

Most of FINITEs commands expect a finite automaton specification (“spec” in short) as the first argument. These specifications are dictionaries defining the elements of the automaton.

If an automaton has only one final state, the spec can simply be a transition table. In other cases, the specification can explicitly define the various elements.

A specification (`spec`) can have these elements:

```
(
  transitions: (...),
  states: (...),
  inputs: (...),
  initial: "...",
  final: (...)
)
```

- `transitions` is a dictionary of dictionary in the format:

  ```
  (
    state1: (input1, input2, ...),
    state2: (input1, input2, ...),
    ...
  ```

```
5 )
```

- `states` is an optional array with the names of all states. The keys of `transitions` are used by default.
- `inputs` is an optional array with all input values. The inputs found in `transitions` are used by default.
- `initial` is an optional name of the initial state. The first value in `states` is used by default.
- `final` is an optional array of final states. The last value in `states` is used by default.

The utility function `#util.to-spec` can be used to create a full spec from a partial dictionary by filling in the missing values with the defaults.

II.2 Command reference

```
#accepts(⟨spec⟩, ⟨word⟩, ⟨format⟩: states => states.map(((s, i)) => if i !=
none [
    #s #box[#sym.arrow.r#place(top + center, dy: -88%)[#text(.88em, raw(i))]]
  ] else [#s]).join())
```

Tests if a ⟨word⟩ is accepted by a given automaton.

The result if either `false` or an array of tuples with a state name and the input used to transition to the next state. The array is a possible path to an accepting final state. The last tuple always has `none` as an input.

```
#let aut = (
  q0: (q1: 0),
  q1: (q0: 1)
)
#finite.accepts(aut, "01010")

#finite.accepts(aut, "0101")
```

$q0 \xrightarrow{0} q1 \xrightarrow{1} q0 \xrightarrow{0} q1 \xrightarrow{1} q0 \xrightarrow{0} q1$

`false`

Argument

⟨spec⟩ `spec`

Automaton specification.

Argument

⟨word⟩ `string`

A word to test.

Argument

```
⟨format⟩: states => states.map(((s, i)) => if i != none [
      #s  #box[#sym.arrow.r#place(top + center, dy: -88%)[#text(.88em,
raw(i))]]
  ] else [#s]).join()
```

`function`

A function to format the result.

`#add-trap(⟨spec⟩, ⟨trap-name⟩: "TRAP")`

Adds a trap state to a partial DFA and completes it.

Deterministic automata need to specify a transition for every possible input. If those inputs don't transition to another state, a trap-state is introduced, that is not final and can't be left by any input. To simplify transition diagrams, these trap-states are oftentimes not drawn. This function adds a trap-state to such a partial automaton and thus completes it.

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

	0	1
q0	q1	TRAP
q1	TRAP	q0
TRAP	TRAP	TRAP

Argument

`⟨spec⟩` `spec`

Automaton specification.

Argument

`⟨trap-name⟩: "TRAP"` `string`

Name for the new trap-state.

```
#automaton(
  ⟨spec⟩,
  ⟨initial⟩: auto,
  ⟨final⟩: auto,
  ⟨labels⟩: (:),
  ⟨style⟩: (:),
  ⟨state-format⟩: label => {
    let m = label.match(regex(`^(\D+)(\d+)$`.text))
    if m != none {
      [#m.captures.at(0)#sub(m.captures.at(1))]
    } else {
      label
    }
  },
  ⟨input-format⟩: inputs => inputs.map(str).join(","),
  ⟨layout⟩: layout.linear,
  ..⟨canvas-styles⟩
) → content
```

Draw an automaton from a specification.

⟨spec⟩ is a dictionary with a specification for a finite automaton. See above for a description of the specification dictionaries.

The following example defines three states `q0`, `q1` and `q2`. For the input `0`, `q0` transitions to `q1` and for the inputs `0` and `1` to `q2`. `q1` transitions to `q0` for `0` and `1` and to `q2` for `0`. `q2` has no transitions.

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

⟨inital⟩ and ⟨final⟩ can be used to customize the initial and final states.

> The ⟨inital⟩ and ⟨final⟩ will be removed in future versions in favor of automaton specs.

Argument

⟨spec⟩ `spec`

Automaton specification.

Argument

⟨initial⟩: auto string | auto | none

The name of the initial state. For auto, the first state in ⟨spec⟩ is used.

Argument

⟨final⟩: auto array | auto | none

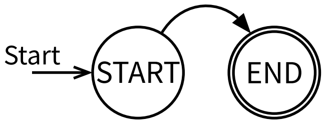
A list of final state names. For auto, the last state in ⟨spec⟩ is used.

Argument

⟨labels⟩: (:) dictionary

A dictionary with labels for states and transitions.

```
#finite.automaton(
  (q0: (q1:none), q1: none),
  labels: (q0: [START], q1: [END])
)
```



Argument

⟨style⟩: (:) dictionary

A dictionary with styles for states and transitions.

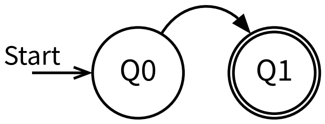
Argument

```
⟨state-format⟩: label => {
    let m = label.match(regex(`^(\D+)(\d+)$`.text))
    if m != none {
      [#m.captures.at(0)#sub(m.captures.at(1))]
    } else {
      label
    }
  }
```

function

A function (string)→content to format state labels. The function will get the states name as a string and should return the final label as content.

```
#finite.automaton(
  (q0: (q1:none), q1: none),
  state-format: (label) => upper(label)
)
```



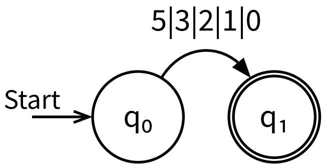
Argument

```
⟨input-format⟩: inputs => inputs.map(str).join(",")
```

function

A function (`array`)→ `content` to generate transition labels from input values. The functions will be called with the array of inputs and should return the final label for the transition. This is only necessary, if no label is specified.

```
#finite.automaton(
  (q0: (q1:(3,0,2,1,5)), q1: none),
 input-format: (inputs) => inputs.sorted().rev().map(str).join("|")
)
```



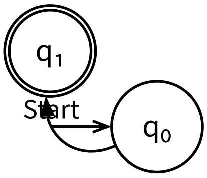
Argument

```
⟨layout⟩: layout.linear
```

`dictionary` | `function`

Either a dictionary with (`state`: `coordinate`) pairs, or a layout function. See below for more information on layouts.

```
#finite.automaton(
  (q0: (q1:none), q1: none),
  layout: (q0: (0,0), q1: (rel:(-2,1)))
)
```



Argument

`..⟨canvas-styles⟩` `any`

Arguments for `#cetz.canvas`

`#powerset(⟨spec⟩, ⟨initial⟩: auto, ⟨final⟩: auto, ⟨state-format⟩: states => "{" + states.sorted().join(",") + "}")`

Creates a deterministic finite automaton from a nondeterministic one by using powerset construction.

See the Wikipedia article on powerset construction[5] for further details on the algorithm.

`⟨spec⟩` is a dictionary with a specification for a finite automaton. See above for a description of the specification dictionaries.

Argument

`⟨spec⟩` `spec`

Automaton specification.

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

`⟨state-format⟩: states => "{" + states.sorted().join(",") + "}"` `function`

[5] https://en.wikipedia.org/wiki/Powerset_construction

A function to generate the new state names from a list of states. The function takes an array of strings and returns a string: (`array`)→ `string` .

```
#transition-table(
  ⟨spec⟩,
  ⟨initial⟩: auto,
  ⟨final⟩: auto,
  ⟨format⟩: (col, v) => raw(str(v)),
  ⟨format-list⟩: states => states.join(", "),
  ..⟨table-style⟩
) → content
```

Displays a transition table for an automaton.

⟨spec⟩ is a dictionary with a specification for a finite automaton. See above for a description of the specification dictionaries.

The table will show states in rows and inputs in columns:

```
#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

The ⟨inital⟩ and ⟨final⟩ will be removed in future versions in favor of automaton specs.

Argument

⟨spec⟩ `spec`

Automaton specification.

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

⟨format⟩: (col, v) => raw(str(v)) `function`

A function to format the value in a table column. The function takes a column index and a string and generates content: (`integer` , `string`)→ `content` .

```
#finite.transition-table((
  q0: (q1: 0, q0: (1,0)),
  q1: (q0: 1, q2: (1,0)),
  q2: (q0: 1, q2: 0),
), format: (col, value) => if col == 1 { strong(value) } else
[#value])
```

	0	1
q0	**q1**, **q0**	q0
q1	**q2**	q0, q2
q2	**q2**	q0

Argument

⟨format-list⟩: states => states.join(", ") `function`

Formats a list of states for display in a table cell. The function takes an array of state names and generates a string to be passed to ⟨format⟩: (`array`)→ `string`

```
#finite.transition-table((
  q0: (q1: 0, q0: (1,0)),
  q1: (q0: 1, q2: (1,0)),
  q2: (q0: 1, q2: 0),
), format-list: (states) => "[" + states.join(" | ") + "]")
```

	0	1
q0	[q1 | q0]	[q0]
q1	[q2]	[q0 | q2]
q2	[q2]	[q0]

Argument

..⟨table-style⟩ any

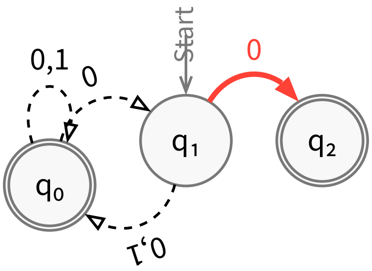
Arguments for #table.

II.3 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 (-).

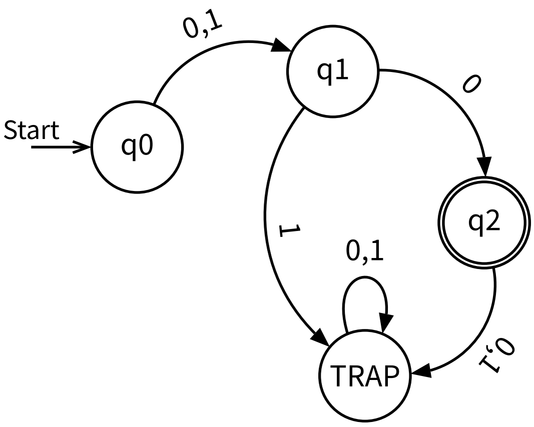
The supported styling options (and their defaults) are as follows:

- states:
  ⟨fill⟩: auto Background fill for states.
  ⟨stroke⟩: auto Stroke for state borders.
  ⟨radius⟩: 0.6 Radius of the states circle.
  - label:
    ⟨text⟩: auto State label.
    ⟨size⟩: auto Initial text size for the labels (will be modified to fit the label into the states circle).
- transitions
  ⟨curve⟩: 1.0 "Curviness" of transitions. Set to 0 to get straight lines.
  ⟨stroke⟩: auto Stroke for transitions.
  - label:
    ⟨text⟩: "" Transition label.
    ⟨size⟩: 1em 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 start, 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.4 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.south"), "trap", label:"TRAP", anchor:"north-
west")

  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.4.1 Element functions

```
#loop(
  ⟨state⟩,
  ⟨inputs⟩: none,
  ⟨label⟩: auto,
  ⟨anchor⟩: top,
  ..⟨style⟩
)
```

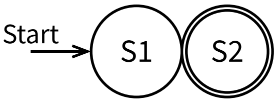
Create a transition loop on a state.

This is a shortcut for `#transition` that takes only one state name instead of two.

```
#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.east", "q2", label:"S2", final:true, anchor:"west")
})
```



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` | `dictionary`

Whether this is an initial state. This can be either

- `true`,
- an `alignment` to specify an anchor for the inital marking,
- a `string` to specify text for the initial marking,
- an `dictionary` with the keys `anchor` and `label` to specifiy both an anchor and a text label for the marking. Additionally, the keys `stroke` and `scale` can be used to style the marking.

Argument

⟨final⟩: `false` `boolean`

Whether this is a final state.

Argument

⟨anchor⟩: `"center"` `string`

Anchor to use for drawing.

Argument

`..⟨style⟩` `any`

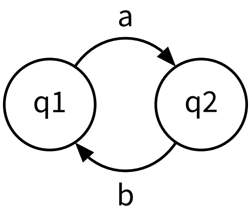
Styling options.

```
#transition(
  ⟨from⟩,
  ⟨to⟩,
  ⟨inputs⟩: none,
  ⟨label⟩: auto,
  ⟨anchor⟩: top,
  ..⟨style⟩
)
```

Draw a transition between two states.

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

```
#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 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

`⟨anchor⟩: top` — `alignment`

Anchor for loops. Has no effect on normal transitions.

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`

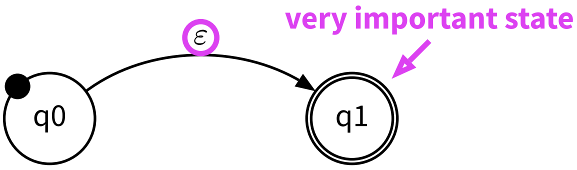
Styling options.

II.4.2 Anchors

States and transitions are created in a `#cetz.draw.group`. States are drawn with a circle named `state` that can be referenced in the group. Additionally they have a content element named `label` and optionally a line named `initial`. These elements can be referenced inside the group and used as anchors for other CETZ elements. The anchors of `state` are also copied to the state group and are directly accessible.

Transitions have an `arrow` (`#cetz.draw.line`) and `label` (`#cetz.draw.content`) element. The anchors of `arrow` are copied to the group.

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

  state((0, 0), "q0")
  state((4, 0), "q1", final: true)

  transition("q0", "q1", label: $epsilon$)

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

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



II.5 Layouts

Layouts can be used to move states to new positions within a call to #cetz.canvas. They act #cetz.draw.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.

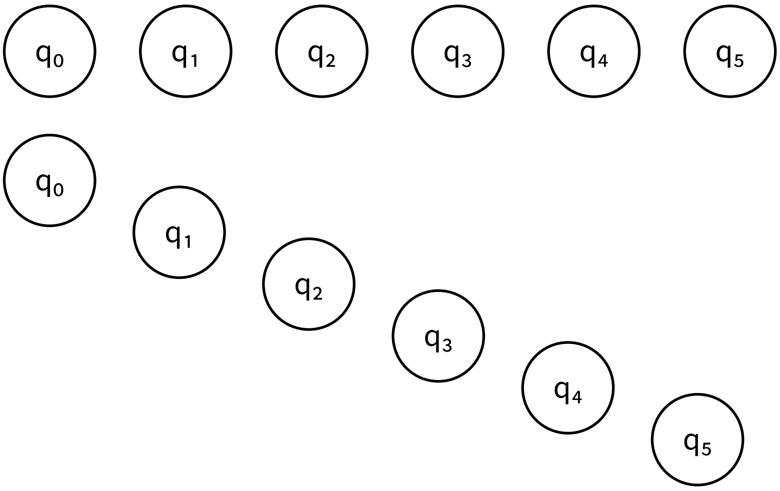
II.5.1 Available layouts

```
#linear(
  ⟨position⟩,
  ⟨name⟩: none,
  ⟨anchor⟩: "west",
  ⟨dir⟩: right,
  ⟨spacing⟩: .6,
  ⟨body⟩
)
```

Arange 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⟩: "west"` `string`

Name of the anchor to use for the layout.

Argument

`⟨dir⟩: right` `vector` | `alignment` | `2d alignment`

Direction of the line.

Argument

`⟨spacing⟩: .6` `float`

Spacing between states on the line.

Argument

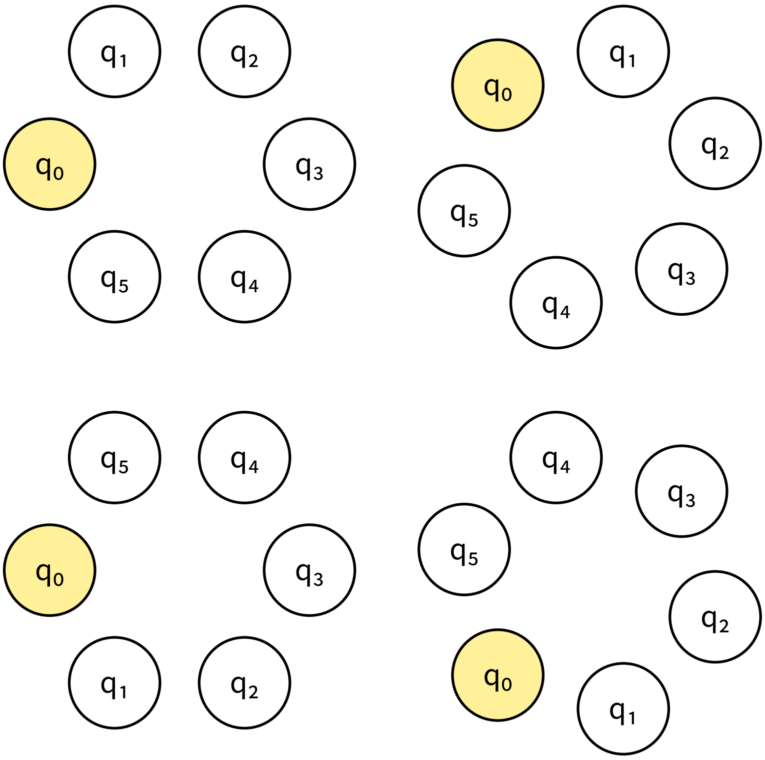
`⟨body⟩` `array`

Array of CeTZ elements to cetz.draw.

```
#circular(
  ⟨position⟩,
  ⟨name⟩: none,
  ⟨anchor⟩: "west",
  ⟨dir⟩: right,
  ⟨spacing⟩: .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%)))
  )
)
```


q_1 q_2 q_0 q_3 q_5 q_4
q_0 q_1 q_2 q_5 q_3 q_4
q_5 q_4 q_0 q_3 q_1 q_2
q_4 q_3 q_5 q_2 q_0 q_1

Argument

⟨position⟩ `coordinate`

Position of the anchor point.

Argument

`⟨name⟩: none` `string`

Name for the element to access later.

Argument

`⟨anchor⟩: "west"` `string`

Name of the anchor to use for the layout.

Argument

`⟨dir⟩: right` `alignment`

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

Argument

`⟨spacing⟩: .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

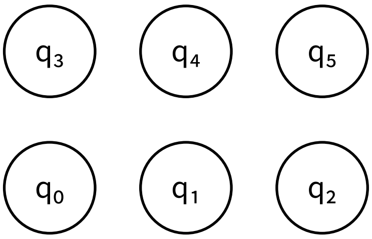
`⟨body⟩` `array`

Array of CeTZ elements to cetz.draw.

```
#grid(
  ⟨position⟩,
  ⟨name⟩: none,
  ⟨anchor⟩: "west",
  ⟨columns⟩: 4,
  ⟨spacing⟩: .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⟩: "west"` `string`

Name of the anchor to use for the layout.

Argument

`⟨columns⟩: 4` `integer`

Number of columns per row.

Argument

`⟨spacing⟩: .6` `float`

Spacing between states on the grid.

Argument

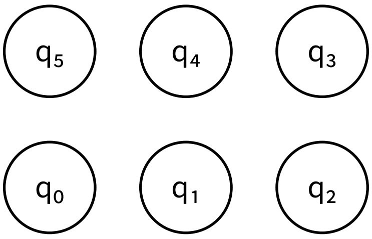
`⟨body⟩` `array`

Array of CeTZ elements to cetz.draw.

```
#snake(
  ⟨position⟩,
  ⟨name⟩: none,
  ⟨anchor⟩: "west",
  ⟨columns⟩: 4,
  ⟨spacing⟩: .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⟩: "west"` `string`

Name of the anchor to use for the layout.

Argument

`⟨columns⟩: 4` `integer`

Number of columns per row.

Argument

`⟨spacing⟩: .6` `float`

Spacing between states on the line.

Argument

`⟨body⟩` `array`

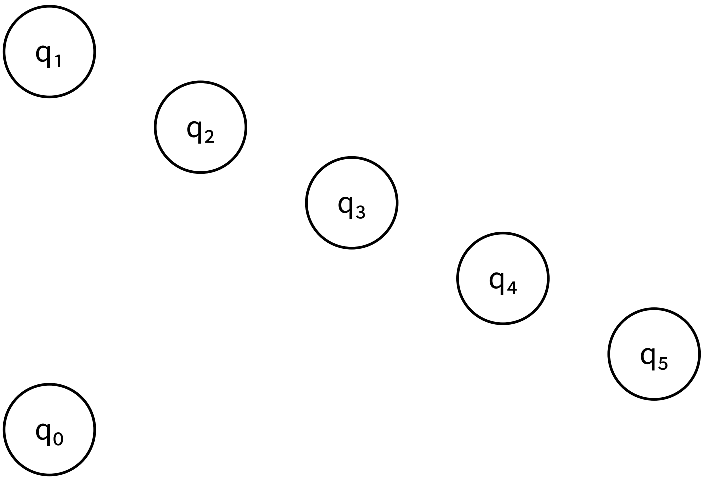
Array of CETZ elements to cetz.draw.

```
#custom(
  ⟨position⟩,
  ⟨name⟩: none,
  ⟨anchor⟩: "west",
  ⟨positions⟩: (ctx, radii, states) => (:),
  ⟨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.custom.with(positions:(..) => (
    q0: (0,0), q1: (0,5), rest:(rel: (2,-1))
  ))
)
```



Argument

`⟨position⟩` `coordinate`

Position of the anchor point.

Argument

`⟨name⟩: none` `string`

Name for the element to access later.

Argument

`⟨anchor⟩: "west"` `string`

Name of the anchor to use for the layout.

Argument

`⟨positions⟩: (ctx, radii, states) => (:)` `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.

The result may specify a `rest` key that is used as a default coordinate. This makes sense in combination with a relative coordinate like `(rel:(2,0))`.

Argument

`⟨body⟩` `array`

Array of CeTZ elements to cetz.draw.

```
#group(
  ⟨position⟩,
  ⟨name⟩: none,
  ⟨anchor⟩: "west",
  ⟨grouping⟩: 5,
  ⟨spacing⟩: .8,
  ⟨layout⟩: linear.with(dir: bottom),
  ⟨body⟩
)
```

Creates a group layout that collects states into groups that are positioned by specific sub-layouts.

See Section III for an example.

Argument

`⟨position⟩` `coordinate`

Position of the anchor point.

Argument

`⟨name⟩: none` `string`

Name for the element to access later.

Argument

`⟨anchor⟩: "west"` `string`

Name of the anchor to use for the layout.

Argument

`⟨grouping⟩: 5` `integer` | `array`

Either an integer to collect states into roughly equal sized groups or an array of arrays that specify which states (by name) are in what group.

Argument

`⟨spacing⟩: .8` `float`

A spacing between sub-group layouts.

Argument

`⟨layout⟩: linear.with(dir: bottom)` `array`

An array of layouts to use for each group. The first group of states will be passed to the first layout and so on.

Argument

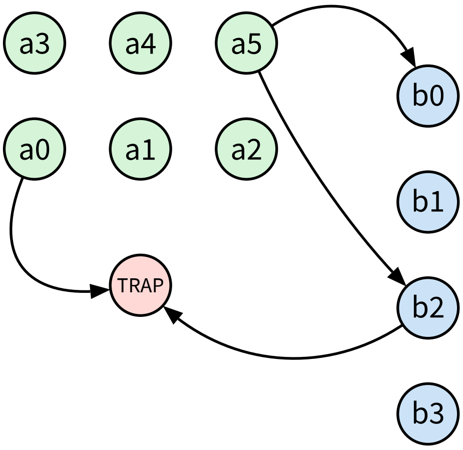
`⟨body⟩` `array`

Array of CeTZ elements to cetz.draw.

II.5.2 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(
    name: "grid",
    (0,0),
    columns:3, {
      set-style(state: (fill: green.lighten(80%)))
      for s in range(6) {
        state((), "a" + str(s))
      }
    })

  layout.linear(
    name: "line",
    (rel:(2,0), to:"grid.east"),
    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.south"), "TRAP", fill:red.lighten(80%),
label:(size:8pt))

  transition("grid.a0", "TRAP", curve:-1)
  transition("line.b2", "TRAP")
  transition("grid.a5", "line.b0")
  transition("grid.a5", "line.b2", curve:-.2)
})
```



II.6 Utility functions

#align-to-anchor
#align-to-vec
#cubic-normal
#cubic-pts
#fit-content
#get-inputs
#label-pt
#loop-pts
#mark-dir
#mid-point
#to-spec
#transition-pts
#vector-normal
#vector-rotate
#vector-set-len

```
#align-to-anchor(⟨align⟩)
```

Return anchor name for an `alignment`.

```
#align-to-vec(⟨a⟩)
```

Returns a vector for an alignment.

```
#cubic-normal(
  ⟨a⟩,
  ⟨b⟩,
  ⟨c⟩,
  ⟨d⟩,
  ⟨t⟩
)
```

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

```
#cubic-pts(⟨a⟩, ⟨b⟩, ⟨curve⟩: 1)
```

Calculate the control point for a 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.

```
#get-inputs(⟨table⟩, ⟨transpose⟩: true)
```

Gets a list of all inputs from a transition table.

```
#label-pt(
  ⟨a⟩,
  ⟨b⟩,
  ⟨c⟩,
  ⟨d⟩,
  ⟨style⟩,
  ⟨loop⟩: false
)
```

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

```
#loop-pts(⟨start⟩, ⟨start-radius⟩, ⟨anchor⟩: top, ⟨curve⟩: 1)
```

Calculate start, end and ctrl points for a transition loop.

Argument

`⟨start⟩` `vector`

Center of the state.

Argument

`⟨start-radius⟩` `length`

Radius of the state.

Argument

`⟨anchor⟩: top` `alignment`

Anchorpoint on the state

Argument

`⟨curve⟩: 1` `float`

Curvature of the transition.

```
#mark-dir(
  ⟨a⟩,
  ⟨b⟩,
  ⟨c⟩,
  ⟨d⟩,
  ⟨scale⟩: 1
)
```

Calculate the direction vector for a transition mark (arrowhead)

```
#mid-point(⟨a⟩, ⟨b⟩, ⟨c⟩, ⟨d⟩)
```

Compute the mid point of a quadratic bezier curve.

```
#to-spec(
  ⟨spec⟩,
  ⟨states⟩: auto,
  ⟨initial⟩: auto,
  ⟨final⟩: auto,
  ⟨inputs⟩: auto
)
```

Creates a full specification for a finite automaton.

```
#transition-pts(
  ⟨start⟩,
  ⟨end⟩,
  ⟨start-radius⟩,
  ⟨end-radius⟩,
  ⟨curve⟩: 1,
  ⟨anchor⟩: top
)
```

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.

#vector-normal(⟨v⟩)

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

#vector-rotate(⟨vec⟩, ⟨angle⟩)

Rotates a vector by ⟨angle⟩ degree around the origin.

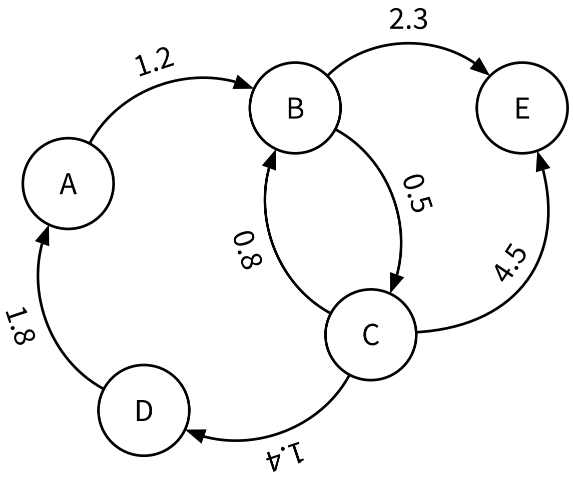
#vector-set-len(⟨v⟩, ⟨len⟩)

Set the length of a cetz.vector.

II.7 Doing other stuff with FINITE

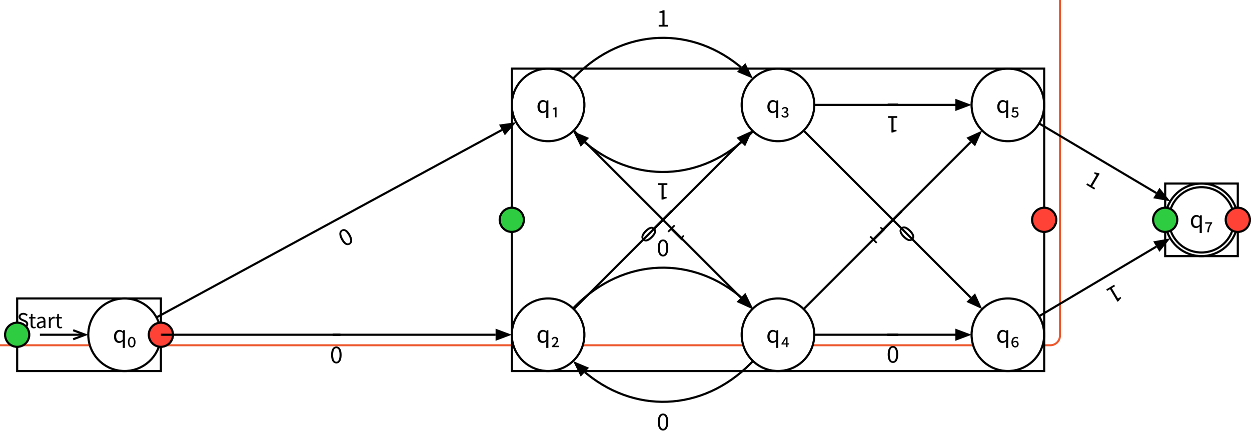
Since transition diagrams are effectively graphs, FINITE could also be used to draw graph structures:

```
#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

```
#scale(80%, automaton((
    q0: (q1: 0, q2: 0),
    q2: (q3: 1, q4: 0),
    q4: (q2: 0, q5: 0, q6: 0),
    q6: (q7: 1),
    q1: (q3: 1, q4: 0),
    q3: (q1: 1, q5: 1, q6: 1),
    q5: (q7: 1),
    q7: ()
  ),
  layout: finite.layout.group.with(grouping: (
      ("q0",),
      ("q1", "q2", "q3", "q4", "q5", "q6"),
      ("q7",)
    ),
    spacing: 2,
    layout: (
      finite.layout.linear,
      finite.layout.grid.with(columns:3, spacing:2.6),
      finite.layout.linear
    )
  ),
  style: (
    transition: (curve: 0),
    q1-q3: (curve:1),
    q3-q1: (curve:1),
    q2-q4: (curve:1),
    q4-q2: (curve:1),
    q1-q4: (label: (pos:.75)),
    q2-q3: (label: (pos:.75, dist:-.33)),
    q3-q6: (label: (pos:.75)),
    q4-q5: (label: (pos:.75, dist:-.33))
  )
))
```



Part IV Index