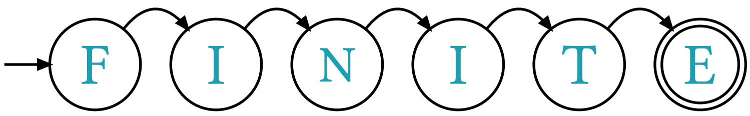


v0.1.0 2023-08-24 MIT

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 (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. 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 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.

I.3. Quick start

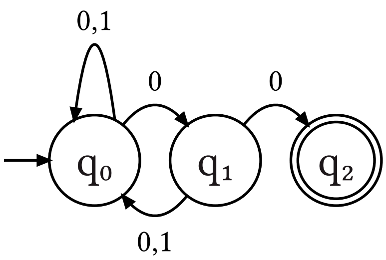
FINITE helps you draw transition diagrams for finite automata in your Typst douments, using the power of CETZ.

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

```
#automaton((
  q0: (q1:0, q0:"0,1"),
  q1: (q0:(0,1), q2:"0"),
  q2: (),
))
```

The output looks like this:

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

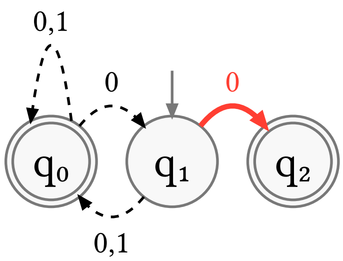


As you can see, an automaton ist defined by a dictionary of dictionaries. The keys of the top-level dictionary are the names of the states to draw. The second-level dictionaries have the names of connected states as keys and transition labels as values.

In the example above, the states `q0`, `q1` and `q2` are defined. `q0` is connected to `q1` nad 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.

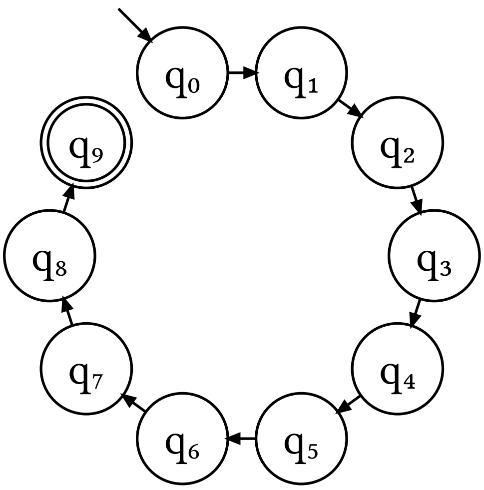
To modify the defaults, `#automaton()` accepts a set of options:

```
#automaton(
  (
    q0: (q1:0, q0:"0,1"),
    q1: (q0:(0,1), q2:"0"),
    q2: (),
  ),
  start: "q1",
  stop: ("q0", "q2"),
  style:(
    state: (fill: luma(248), stroke:luma(120)),
    transition: (stroke: (dash:"dashed")),
    q1: (start: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: 2),
  style: (
    transition: (curve: 0),
    q0: (start: top+left)
  )
)
```



See Section I.6 for more details about layouts.

I.4. Command reference

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

Draw an automaton from a transition table.

Argument

`states` — `dictionary`

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

Argument

`start: auto` — string | auto | none

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

Argument
`stop: 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.
- layout (function: A layout function.

Argument
`..canvas-styles` — any

Arguments for `#cetz.canvas()`

`#transition-table(states, start: auto, stop: auto, format: (...) => ...)`

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),
  q1: (q0: 1, q2: 0),
  q2: (q0: 1, q2: 0),
))
```

	0	1
q0	q1	q0
q1	q2	q0
q2	q2	q0

Argument
`states` — dictionary

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

Argument
`start: auto` — string | auto | none

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

Argument

`stop: auto` — `array` | `auto` | `none`

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

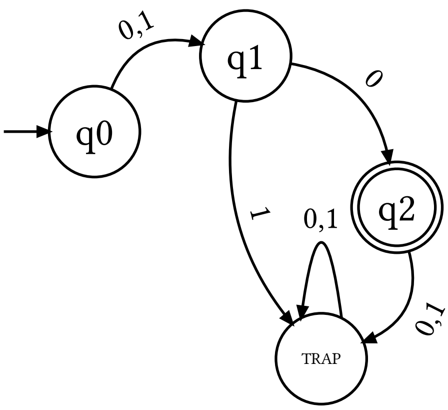
I.5. Drawing automata

The above commands use custom CETZ elements, to draw states and transitions. For complex automata, the functions in the `draw` module can be used directly.

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

  state((0,0), "q0", start:true)
  state((2,1), "q1")
  state((4,-1), "q2", stop:true)
  state((3,-3), "trap", label:"TRAP")

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




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

Draw a state at the given `position`.

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

Draw a transition between two states.

The two states `from` and `to` have to be drawn first.

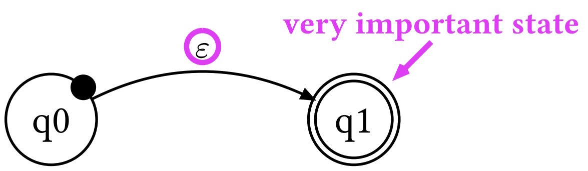
States have the common anchors (like `top`, `top-left` …), transitions have a `start`, `end`, `center` and `label` anchors. These can be used to draw additional elements:

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

  state((0,0), "q0")
  state((4,0), "q1", stop: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*])
  )
})
```



I.6. Using layouts

Layouts calculate coordinates for every state in a transition table and return a dictionary with all computed locations.

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

> ⚠ Layouts currently are very simple. They will most likely see improvements in the future. At the moment, layouts don't know about the context and can't resolve coordinates other than absolute coordinate pairs.

I.6.1. Available layouts

`#circular()` `#grid()` `#linear()`
`#fixed()` `#group()` `#snake()`

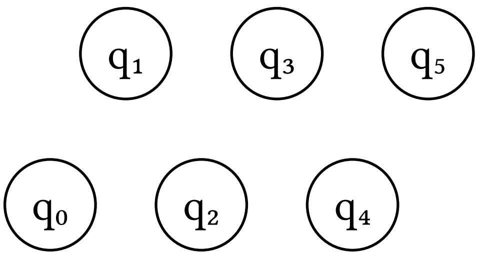
`#fixed(states, start, stop, pos: "(:)")`

A fixed list of coordinates.

Can be used to manually sepcify individual coordinates for each state. Note, that coordinates may be any specified in any coordinate system CETZ knows. If you want to use fixed coordinates as a sub-layout (e.g. in `group()`, all coordinates must be in the (x,y)-system.

Notes not in `pos` will be placed at `(0, 0)`.

```
#let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
#finite.automaton(
  aut,
  start: none, stop: none,
  layout:finite.layout.fixed.with(pos:(
    q0: (0,-1), q1: (1,1), q2: (2,-1),
    q3: (3,1), q4: (4,-1), q5: (5,1)
  ))
)
```



Argument

`states` — `dictionary`

Transition table.

Argument

`start` — `string` | `none`

Initial state.

Argument

`stop` — `array` | `none`

List of final states.

Argument

`pos: (:)` — `dictionary`

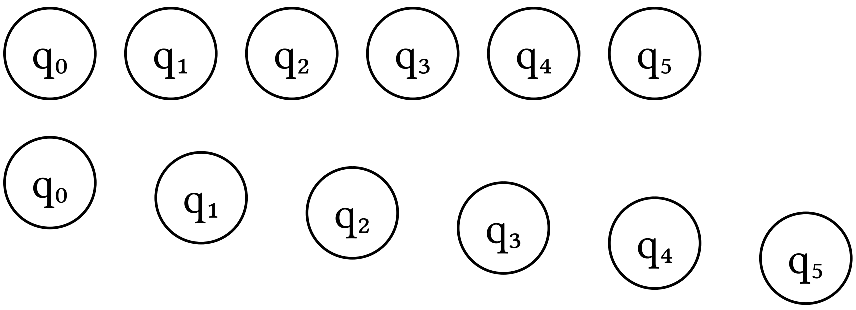
Dictionary with coordinates for each state.

```
#linear(
  states,
  start,
  stop,
  x: 1.6,
  y: 0
)
```

Arrange states in a line.

The direction of the line can be set by `x` and `y`.

```
#let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
#finite.automaton(
  aut,
  start: none, stop: none,
  layout:finite.layout.linear
)
#finite.automaton(
  aut,
  start: none, stop: none,
  layout:finite.layout.linear.with(x:2, y:-.2)
)
```



Argument

`states` — `dictionary`

Transition table.

Argument

`start` — `string` | `none`

Initial state.

Argument

`stop` — `array` | `none`

List of final states.

Argument

`x: 1.6` — `float`

Step size along the x-axis.

Argument

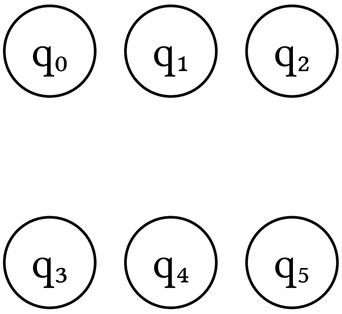
`y: 0` — `float`

Step size along the y-axis.

```
#grid(
  states,
  start,
  stop,
  columns: 6,
  x: 1.6,
  y: 2.8
)
```

Arrange states in rows and columns.

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



Argument

`states` — `dictionary`

Transition table.

Argument

`start` — `string` | `none`

Initial state.

Argument

`stop` — `array` | `none`

List of final states.

Argument

`columns: 6` — `integer`

Number of columns per row.

Argument

`x: 1.6` `float`

Distance between the center of each column.

Argument

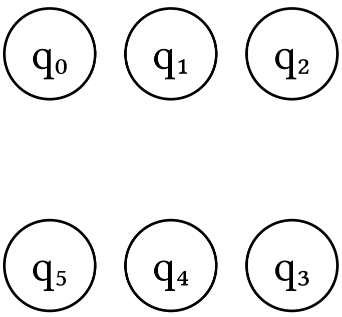
`y: 2.8` `float`

Distance between the center of each row.

```
#snake(
  states,
  start,
  stop,
  columns: 6,
  x: 1.6,
  y: 2.8
)
```

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,
  start: none, stop: none,
  layout:finite.layout.snake.with(columns: 3)
)
```



Argument

`states` `dictionary`

Transition table.

Argument

`start` `string` | `none`

Initial state.

Argument

`stop` `array` | `none`

List of final states.

Argument

`columns: 6` `integer`

Number of columns per row.

Argument

`x: 1.6` `float`

Distance between the center of each column.

Argument

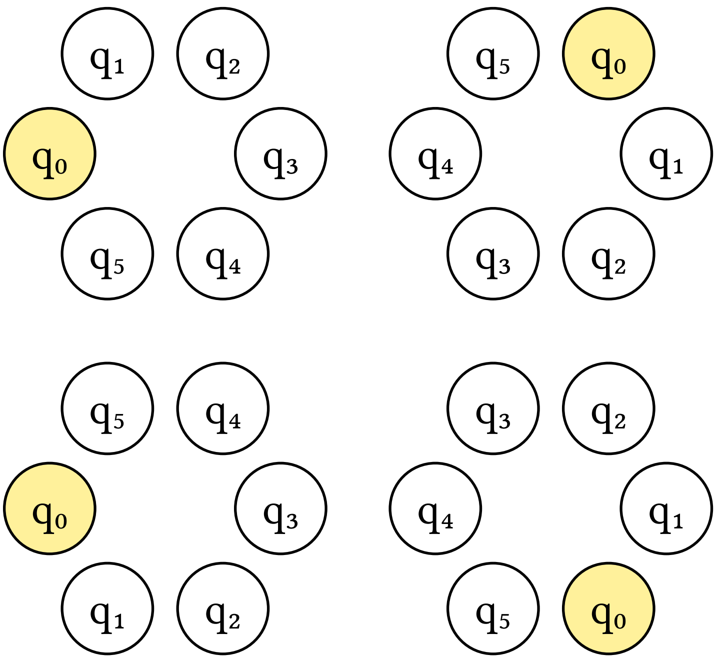
`y: 2.8` `float`

Distance between the center of each row.

```
#circular(
  states,
  start,
  stop,
  radius: auto,
  spacing: 1.6,
  dir: right,
  offset: 0
)
```

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,
    start: none, stop: none,
    layout:finite.layout.circular,
    style: (q0: (fill: yellow.lighten(60%)))
  ),
  finite.automaton(
    aut,
    start: none, stop: none,
    layout:finite.layout.circular.with(offset:2),
    style: (q0: (fill: yellow.lighten(60%)))
  ),
  finite.automaton(
    aut,
    start: none, stop: none,
    layout:finite.layout.circular.with(dir:left),
    style: (q0: (fill: yellow.lighten(60%)))
  ),
  finite.automaton(
    aut,
    start: none, stop: none,
    layout:finite.layout.circular.with(dir:left, offset:2),
    style: (q0: (fill: yellow.lighten(60%)))
  )
)
```



Argument

`states` `dictionary`

Transition table.

Argument

`start` `string` | `none`

Initial state.

Argument

`stop` `array` | `none`

List of final states.

Argument

`radius: auto` `float` | `auto`

Radius of the circle, the states are placed on. With `auto`, the radius is calculated based on `spacing`.

Argument

`spacing: 1.6` `float`

Distance between the center of each state on the circle.

Argument

`dir: right` `alignment`

`left` or `right` to indicate the direction, the states are arranged on the circle.

Argument

`offset: 0` `integer`

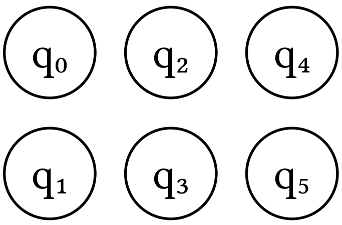
An offset, where the inital state should be placed on the circle. For `offset: 0`, the initial state is placed to the left or right (depending on `dir`) of the circle. For `offset: 1`, the initial state is placed one step in the direction indicated by `dir`.

```
#group(
  states,
  start,
  stop,
  x: 1.6,
  y: 0,
  grouping: 5,
  layout: "linear.with(x:0, y:-1.6)"
)
```

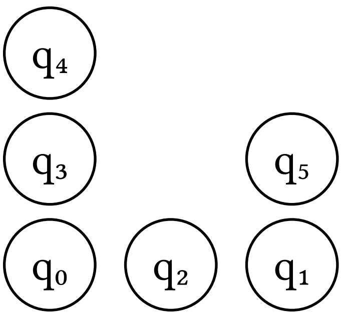
Group states and layout each group with its own layout.

```
#let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
Group every two states together:
#finite.automaton(
  aut,
  start: none, stop: none,
  layout:finite.layout.group.with(
    grouping: 2
  )
)
Group specific states and arrange from bottom to top:
#finite.automaton(
  aut,
  start: none, stop: none,
  layout:finite.layout.group.with(
    grouping: (
      ("q0", "q3", "q4"),
      ("q2",),
      ("q1", "q5"),
    ),
    layout: finite.layout.linear.with(x:0, y:1.4)
  )
)
```

Group every two states together:



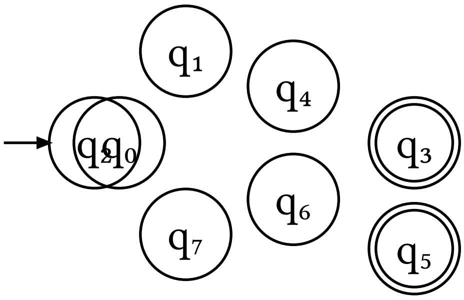
Group specific states and arrange from bottom to top:



`#start-stop(states, start, stop, layout: "snake")`

Arrange initial state on the left, final states on the right and the rest in the center with a custom layout.

```
#let aut = range(8).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
#finite.automaton(
  aut,
  start: "q2", stop: ("q3", "q5"),
  layout:finite.layout.start-stop.with(
    layout: finite.layout.circular
  ),
)
```



I.6.2. Creating custom layouts

A layout is a function, that, provided with information about the automaton, returns a dictionary with the state names as keys and valid coordinates as values.

`#linear-layout()` is a simple example:

```
let linear-layout(states, start, stop, x:2.2, y:0) = {
  let positions = (:)
  for (i, name) in states.keys().enumerate() {
    positions.insert(name, (i * x, i * y))
  }
  return positions
}
```

A layout function always gets passed the `states` dictionary (the same dictionary that is passed to `#automaton()`), the `start` state and the list of end states `stop`.

Additional named arguments may be used to configure the layout via `with`:

```
linear-layout.with(x:-1, y:2.2)
```

This example arranges the states in a wave:

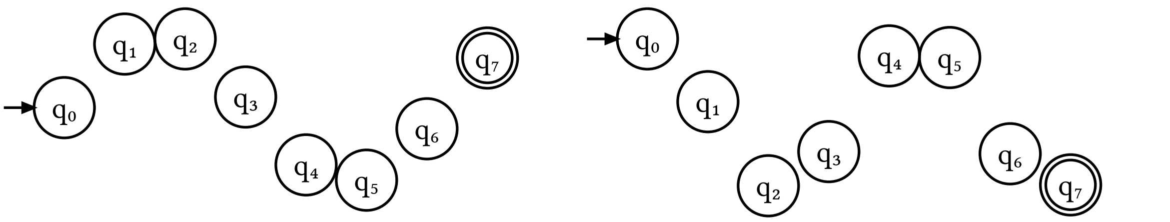
```
#let wave-layout(
  states, start, stop,
  x: 1.6, amp: 1,
  generator: calc.sin
) = {
  let positions = (:)

  for (i, state) in states.keys().enumerate() {
    positions.insert(state, (
      i * x, generator(i * amp)
    ))
  }

  return positions
}

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

#grid(
  columns:(1fr,1fr),
  automaton(
    aut,
    layout: wave-layout.with(generator: calc.sin, x:.8),
    style: (state: (radius: .4))
  ),
  automaton(
    aut,
    layout: wave-layout.with(generator: calc.cos, x:.8, amp:1.4),
    style: (state: (radius: .4))
  )
)
```



I.7. Utility functions

`#ctrl-pt()`	`#label-pt()`	`#prepare-ctx()`
`#fit-content()`	`#mark-dir()`	`#transition-pts()`

`#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)
```

Prepares the CeTZ context for use with finite

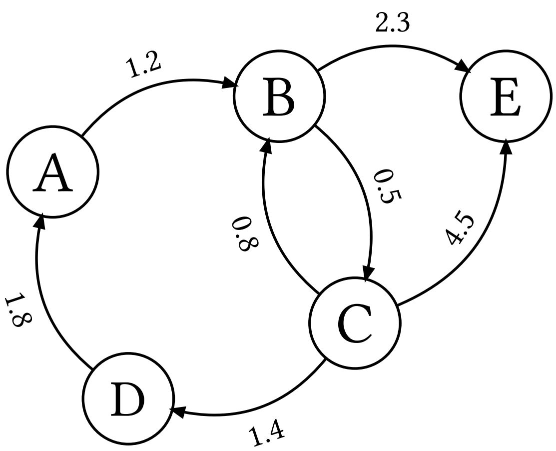
I.8. Doing other stuff with finite

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 II.
Index