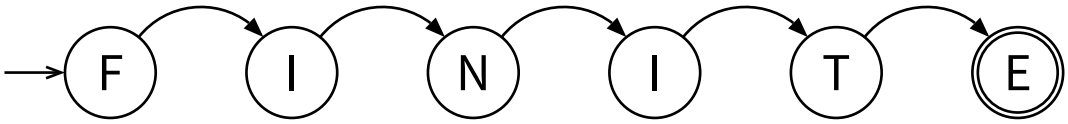



Version 0.4.1 2025-01-14 MIT

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

The package provides commands to quickly draw automata from a transition table but also lets you manually create and customize transition diagrams on any CETZ canvas.

## Table of Contents

I Usage ..... 2
I.1 Importing the package ..... 2
I.2 Manual installation ..... 2
I.3 Dependencies ..... 2
II Drawing automata ..... 3
II.1 Specifing finite automata ..... 5
II.2 Command reference ..... 6
II.3 Styling the output ..... 12
II.4 Using `#cetz.canvas` ..... 14
II.4.1 Element functions ..... 15
II.4.2 Anchors ..... 17
II.5 Layouts ..... 18
II.5.1 Available layouts ..... 19
II.5.2 Using layouts ..... 19
II.6 Utility functions ..... 19
III Simulating input ..... 23
IV Working with grammars ..... 24
V Index ..... 25

# 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.

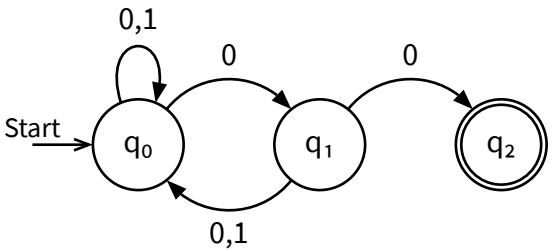
[1] https://github.com/jneug/typst-finite
[2] https://github.com/typst/packages#local-packages
[3] https://github.com/johannes-wolf/typst-canvas
[4] https://github.com/jneug/typst-tools4typst

# 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.

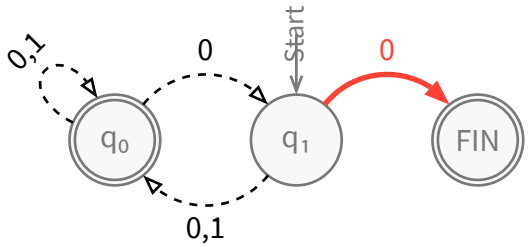
In the example above, the states `q0`, `q1` and `q2` are defined. `q0` is connected to `q1` and has a loop to itself. `q1` transitions to `q2` and back to `q0`. `#automaton` selected the first state in the dictionary (in this case `q0`) to be the Initial state and the last (`q2`) to be a final state.

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:

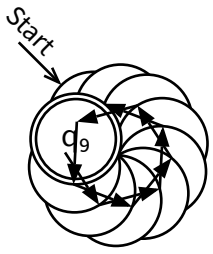
```
#automaton(
  (
    q0: (q1:0, q0:"0,1"),
    q1: (q0:(0,1), q2:"0"),
    q2: (),
  ),
  initial: "q1",
  final: ("q0", "q2"),
  labels:(
    q2: "FIN"
  ),
```

```
  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 trnasition table is a `dictionary` with state names as keys `dictionary` as values. The nested dictionaries have state names as keys and the transition inputs as values.

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

```
(
 ⟨finite⟩ boolean
 ⟨type⟩ string
)
```

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, ...),
  ...
)
```

- `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): (spec, 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`

- spec (spec): Automaton specification.
- word (string): A word to test.
- format (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 |

- spec (spec): Automaton specification.
- trap-name (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.

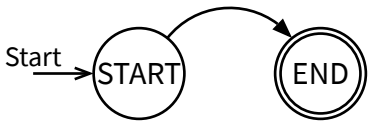
```
#automaton((
  q0: (q1:0, q0:(0, 1)),
  q1: (q0:(0, 1), q2:0),
  q2: none
))
```

⟨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.

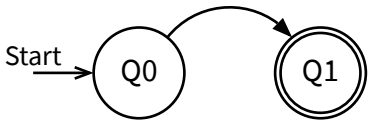
- spec (spec): Automaton specification.
- initial (string, auto, none): The name of the initial state. For `auto`, the first state in ⟨spec⟩ is used.
- final (array, auto, none): A list of final state names. For `auto`, the last state in ⟨spec⟩ is used.
- labels (dictionary): A dictionary with labels for states and transitions.

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



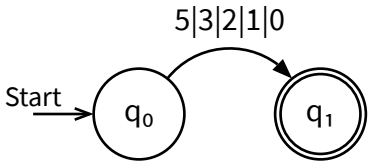

- style (dictionary): A dictionary with styles for states and transitions.
- state-format (function): A function `(string)→content` to format state labels. The function will get the states name as a string and should return the final label as `content`.

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



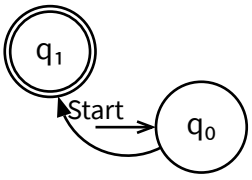

- input-format (function): A function `(array)→content` to generate transition labels from input values. The functions will be called with the array of inputs and should return the final label for the transition. This is only necessary, if no label is specified.

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




- layout (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)))
)
```




- ..canvas-styles (any): Arguments for `#cetz.canvas`

```
#create-automaton(
  ⟨spec⟩,
  ⟨states⟩: auto,
  ⟨initial⟩: auto,
  ⟨final⟩: auto,
  ⟨inputs⟩: auto
) → automaton
```

Creates a full `automaton` specification for a finite automaton. The function accepts either a partial specification and adds the missing keys by parsing the available information or takes a `transition-table` and parses it into a full specification.

```
#finite.create-automaton((
  q0: (q1: 0, q0: (0,1)),
  q1: (q0: (0,1), q2: "0"),
  q2: none,
))
```

---

```
(
  transitions: (
    q0: (q1: ("0",), q0: ("0", "1")),
    q1: (q0: ("0", "1"), q2: ("0",)),
    q2: (:),
  ),
  states: ("q0", "q1", "q2"),
  initial: "q0",
  final: ("q2",),
  inputs: ("0", "1"),
  type: "NEA",
  finite-spec: true,
)
```

If any of the keyword arguments are set, they will overwrite the information in ⟨spec⟩.

```
#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.

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

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

- spec (spec): Automaton specification.
- initial (string, auto, none): The name of the initial state. For `auto`, the first state in ⟨states⟩ is used.
- final (array, auto, none): A list of final state names. For `auto`, the last state in ⟨states⟩ is used.
- state-format (function): 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.

- spec (spec): Automaton specification.

- initial (string, auto, none): The name of the initial state. For `auto`, the first state in ⟨`states`⟩ is used.
- final (array, auto, none): A list of final state names. For `auto`, the last state in ⟨`states`⟩ is used.
- format (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 |

- format-list (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] |

- ..table-style (any): Arguments for `#table`.

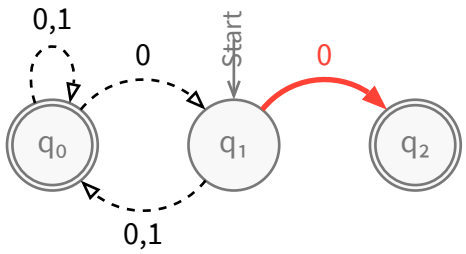
## II.3 Styling the output

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.

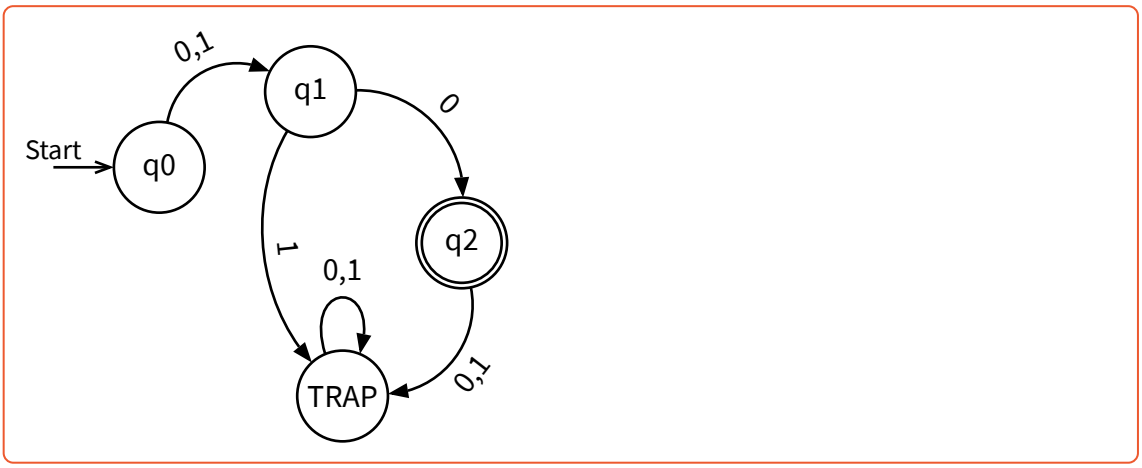
  **(dist): 0.33** Distance of the label from the transition.

  **(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⟩
)
```

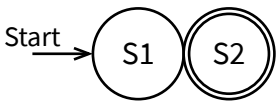
Create a transition loop on a state.

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⟩: none,
  ..⟨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")
})
```



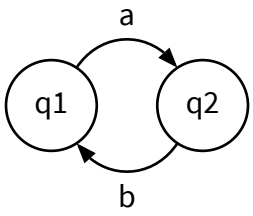

- position (coordinate): Position of the states center.
- name (string): Name for the state.
- label (string,content,auto,none): Label for the state. If set to `auto`, the ⟨`name`⟩ is used.
- initial (boolean,alignment,dictionary): Whether this is an initial state. This can be either
  - `true`,
  - 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.
- final (boolean): Whether this is a final state.
- anchor (string): Anchor to use for drawing.
- ..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")
})
```




- from (string): Name of the starting state.
- to (string): Name of the ending state.
- inputs (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.
- label (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.
- anchor (alignment): Anchor for loops. Has no effect on normal transitions.
- ..style (any): Styling options.

**`#transitions(⟨states⟩, ..⟨style⟩)`**

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

- states (dictionary): A transition table given as a dictionary of dictionaries.
- ..style (any): Styling options.

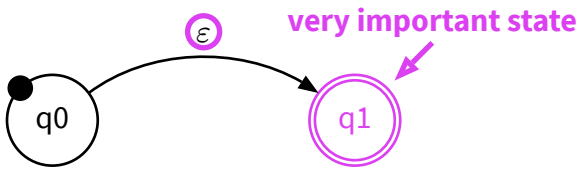
### II.4.2 Anchors

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.

> That means setting `⟨anchor⟩: "west"` for a state will anchor the state at the `west` anchor of the states circle, not of the bounding box of the group.

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

  let magenta = rgb("#dc41f1")

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

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

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

  let magenta-stroke = 2pt + magenta
  circle("q0-q1.label", 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: magenta, [*very important state*]),
  )
})
```




## II.5 Layouts

> Layouts changed in FINITE version 0.5 and are no longer compatible with FINITE 0.4 and before.

Layouts can be passed to `#automaton` to position states on the canvas without the need to give specific coordinates for each state. FINITE ships with a bunch of layouts, to accomodate different scenarios.

### II.5.1 Available layouts

### 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.

## II.6 Utility functions

| | | |
|---|---|---|
| `#align-to-anchor` | `#get-inputs` | `#transition-pts` |
| `#align-to-vec` | `#is-dea` | `#transpose-table` |
| `#call-or-get` | `#label-pt` | `#vector-normal` |
| `#cubic-normal` | `#loop-pts` | `#vector-rotate` |
| `#cubic-pts` | `#mark-dir` | `#vector-set-len` |
| `#fit-content` | `#mid-point` | |

**`#align-to-anchor(⟨align⟩)`**

Return anchor name for an `alignment`.

**`#align-to-vec(⟨a⟩)`**

Returns a vector for an alignment.

**`#call-or-get(⟨value⟩, ..⟨args⟩)`**

Calls `⟨value⟩` with `..⟨args⟩`, if it is a `function` and returns the result or `⟨value⟩` otherwise.

```
#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.

- ctx (dictionary): The canvas context.
- content (string, content): The content to fit.
- size (length,auto): The initial text size.
- min-size (length): The minimal text size to set.

**#get-inputs((table), (transpose): true)**

Gets a list of all inputs from a transition table.

Argument

(table) `transition-table`

A transition table.

Argument

(transpose): true `boolean`

If (table) needs to be transposed first. Set this to `false` if the table already is in the format (input: states).

**#is-dea((table)) → boolean**

Checks if a given `automaton` specification represents a deterministic automaton.

```
#util.is-dea((
  q0: (q1: 1, q2: 1),
))
#util.is-dea((
  q0: (q1: 1, q2: 0),
))
```

---

```
false  true
```

Argument

(table) `transition-table`

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.

- start (vector): Center of the state.
- start-radius (length): Radius of the state.
- curve (float): Curvature of the transition.
- anchor (alignment): Anchorpoint on the state

```
#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.

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

Calculate start, end and ctrl points for a transition.

- start (vector): Center of the start state.
- end (vector): Center of the end state.
- start-radius (length): Radius of the start state.
- end-radius (length): Radius of the end state.
- curve (float): Curvature of the transition.

**#transpose-table(⟨table⟩) → dictionary**

Changes a `transition-table` from the format (`state`: `inputs`) to (`input`: `states`) or vice versa.

Argument

⟨table⟩ `transition-table`

A transition table in any format.

**#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.

# Part III Simulating input

FINITE has a set of

# Part IV Working with grammars

# Part V Index


## A

#accepts ..... 6
#add-trap ..... 6
#align-to-anchor ..... 19
#align-to-vec ..... 19
#automaton . 3, 5, 7, 13, 19

## C

#call-or-get ..... 19
#cetz.draw.content ... 18
`coordinate` ..... 2
#create-automaton ..... 10
#cubic-normal ..... 19
#cubic-pts ..... 19

## F

#fit-content ..... 19, 20

## G

#get-inputs ..... 19, 20
#cetz.draw.group ..... 17

## I

#is-dea ..... 19, 20

## L

#label-pt ..... 19, 21
#cetz.draw.line ..... 18
#loop ..... 15
#loop-pts ..... 19, 21

## M

#mark-dir ..... 19, 21
#mid-point ..... 19, 21

## P

#powerset ..... 10

## S

`spec` ..... 5
#state ..... 15

## T

#util.to-spec ..... 6
#transition ..... 15, 16
#transition-pts ... 19, 21
`transition-table` .... 5, 11
#transitions ..... 17
#transpose-table . 19, 22

## V

#vector-normal ..... 19, 22
#vector-rotate ..... 19, 22
#vector-set-len ... 19, 22