

Version 0.5.0 2026-02-21 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	Specifying finite automata	5
II.1.1	Input labels	6
II.2	Command reference	7
II.3	Styling the output	19
II.4	Using <code>#cetz.canvas</code>	21
II.4.1	Element functions	22
II.4.2	Anchors	26
II.5	Layouts	28
II.5.1	Available layouts	28
II.6	Utility functions	37
III	Simulating input	45
III.1	Testing word acceptance	45
IV	FLACI support	47
IV.0.1	FLACI functions	47
V	Doing other stuff with FINITE	49
VI	Showcase	50
VII	Index	52

Part I Usage

I.1 Importing the package

Import the package in your Typst file:

```
#import "@preview/finite:0.5.0": 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-finite¹](https://github.com/jneug/typst-finite) and unpack the archive into your system dependent local repository folder² or clone it directly:

```
git clone https://github.com/jneug/typst-finite finite/0.5.0
```

In either case, make sure the files are placed in a subfolder with the correct version number: finite/0.5.0

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

```
#import "@local/finite:0.5.0": 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>

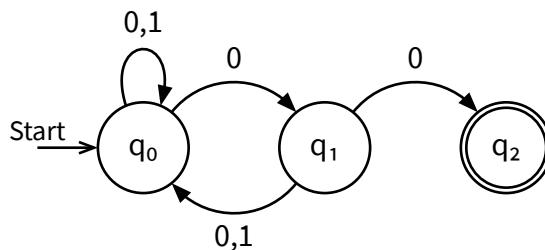
⁴<https://github.com/jneug/typst-tools4typst>

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 is defined by a dictionary of dictionaries. The keys of the top-level dictionary are the names of states to draw. The second-level dictionaries use 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.

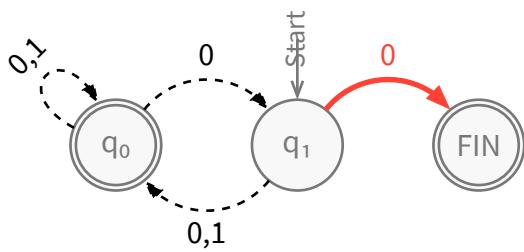
See [Section II.1](#) for more details on how to specify automata.

To modify the layout and style of the transition diagram, `#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"
```

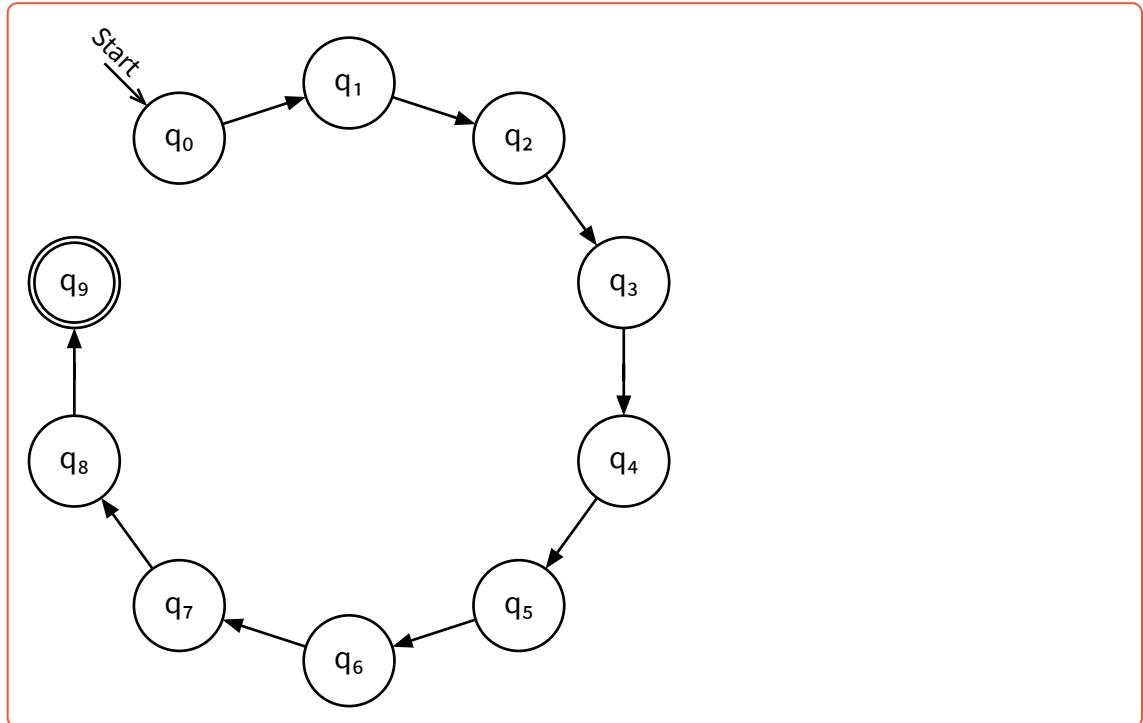
II Drawing automata

```
),
style:(
  state: (fill: luma(248), stroke:luma(120)),
  transition: (stroke: (dash: "dashed")),
  q0-q0: (anchor:top+left),
  q1: (initial:true),
  q1-q2: (stroke: 2pt + red)
)
)
```



For larger automata, 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 on layouts.

II.1 Specifying finite automata

Most of `FINITE`'s 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 transition table is a `dictionary` with state names as keys and dictionaries as values. The nested dictionaries have state names as keys and the transition inputs / labels as values.

```

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

A specification (`spec`) is composed of these keys:

```
1 (
```

```

2   transitions: (...),
3   states: (...),
4   inputs: (...),
5   initial: "...",
6   final: (...),
7   labels: (...),
8   input-labels: (...)

9 )

```

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

```

1 (
2   state1: (input1, input2, ...),
3   state2: (input1, input2, ...),
4   ...
5 )

```

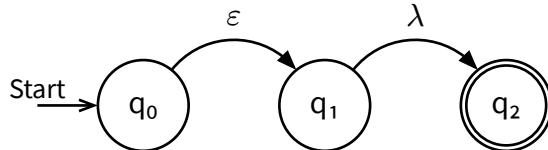
- `states` is an optional array with the names of all states. The keys of `transitions` are used by default.
- `inputs` is an optional array with all input values. The inputs found in `transitions` are used by default.
- `initial` is an optional name of the initial state. The first value in `states` is used by default.
- `final` is an optional array of final states. The last value in `states` is used by default.
- `labels` is a dictionary mapping state and transition names to display labels. An empty dictionary by default.
- `input-labels` is a dictionary mapping input strings to display content. An empty dictionary by default. Inputs that are themselves content values are automatically added here.

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

II.1.1 Input labels

By default, transition labels use the raw input strings or numbers from the transition table. The `(input-labels)` parameter of `#automaton` (and `#create-automaton`) lets you map those raw inputs to arbitrary content — useful for Greek letters, mathematical symbols, or any formatted text.

```
#automaton(
    (q0: (q1: "e"), q1: (q2: "l"), q2: none),
    input-labels: (
        e: $epsilon$,
        l: $lambda$,
    )
)
```



Input labels are stored in the spec under the `input-labels` key and are used both in transition diagrams and in `#transition-table` column headers. Inputs that are themselves content values (rather than strings or numbers) are automatically recorded as their own labels.

Input labels affect display only. The raw input strings (e.g. "e", "l") are still used internally by `#accepts` and other simulation functions.

II.2 Command reference

```
#create-automaton(
    {spec},
    {states}: auto,
    {initial}: auto,
    {final}: auto,
    {inputs}: auto,
    {labels}: (),
    {input-labels}: ()
) → automaton
```

Creates a full automaton specification (`spec`) 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: (:),
    ),
    inputs: ("0", "1"),
    input-labels: (:),
    states: ("q0", "q1", "q2"),
    initial: "q0",
    final: ("q2",),
    labels: (:),
    type: "NEA",
    finite-spec: true,
)
```

If any of the keyword arguments are set, they will overwrite the information in `(spec)`.

Argument
`(spec)` spec | transition-table
 Automaton specification.

Argument
`(states): auto` array
 The list of state names in the automaton. `auto` uses the keys in `(spec)`.

Argument
`(initial): auto` str
 The name of the initial state. `auto` uses the first key in `(spec)`.

Argument
`(final): auto` str | array | auto | none
 The list of final states. `auto` uses the last key in `(spec)`.

<p>— Argument:</p> <p><code>(inputs): auto</code></p>	<p>The list of all inputs, the automaton uses. <code>auto</code> uses the inputs provided in <code>(spec)</code>.</p>
<p>— Argument:</p> <p><code>(labels): (:)</code></p>	<p>A dictionary with custom labels for states and transitions or a <code>function</code> that generates them.</p>
<p>— Argument:</p> <p><code>(input-labels): (:)</code></p>	<p>A dictionary with custom labels for inputs. Using this, inputs can be labeled with arbitrary content instead of strings or numbers.</p>

```
#automaton(
    {spec},
    {initial}: auto,
    {final}: auto,
    {labels}: (:),
    {input-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.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
))
```

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

The `(initial)` and `(final)` will be removed in future versions in favor of automaton specs.

Argument:

`{spec}`

`spec`

Automaton specification.

Argument:

`(initial): auto`

`str | auto | none`

The name of the initial state. For `auto`, the first state in `{spec}` is used.

Argument:

`{final}: auto`

`str | array | auto | none`

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

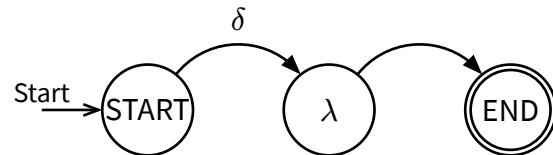
Argument:

`{labels}: (:)`

`dictionary | function`

A dictionary with custom labels for states and transitions.

```
#finite.automaton(
    (q0: (q1:none), q1: (q2:none), q2: none),
    labels: (
        q0: [START], q1: $lambda$, q2: [END],
        q0-q1: $delta$)
)
```



May be a function that gets the state or transition name and returns a computed label. If `#none` is returned, the name is not put into the label map.

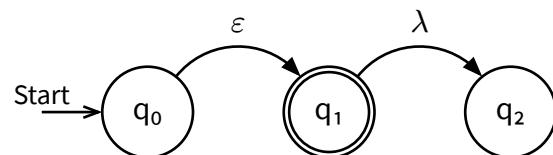
Argument

`(input-labels): (:)`

dictionary | function

A dictionary with custom labels for inputs. Using this, inputs can be labeled with arbitrary content instead of strings or numbers.

```
#finite.automaton(
    (q0: (q1:"e"), q1: (q2:"l")),
    input-labels: (
        e: $epsilon$,
        l: $lambda$)
)
```



May be a function that gets the input as a `str` and returns a computed label. If `#none` is returned, the input is not put into the label map.

Argument

`(style): (:)`

dictionary | function

A dictionary with styles for states and transitions.

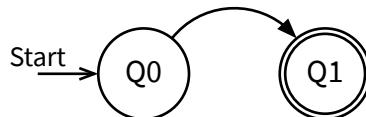
Argument:

```
(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 (`str`) \rightarrow `content` to format state labels. The function will get the state's name as a string and should return the final label as `content`.

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



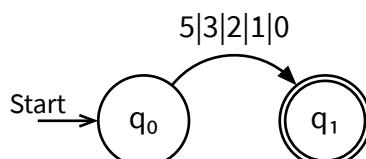
Argument:

```
(input-format): inputs => inputs.join([,])
```

function

A function (`array`) \rightarrow `content` to generate transition labels from input values. The function 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("|")
)
```



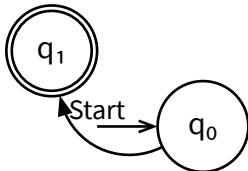
Argument:

```
{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`.

```
#transition-table(
  {spec},
  {initial}: auto,
  {final}: auto,
  {labels}: (:),
  {input-labels}: (:),
  {format}: (col, row, v) => [#v],
  {format-list}: states => states.join(", "),
  ..(table-style)
) → content
```

Displays a transition table for an automaton.

`{spec}` is a `spec` for a finite automaton.

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 `(initial)` and `(final)` arguments will be removed in future versions in favor of automaton specs.

Argument

`{spec}`

`spec`

Automaton specification.

Argument

`(initial): auto`

`str, 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.

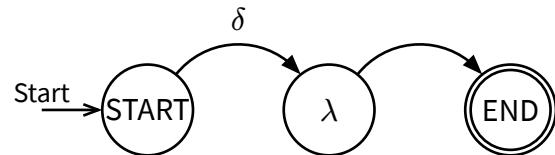
Argument

`(labels): (:)`

`dictionary | function`

A dictionary with custom labels for states and transitions.

```
#finite.automaton(
  (q0: (q1:none), q1: (q2:none), q2: none),
  labels: (
    q0: [START], q1: $lambda$, q2: [END],
    q0-q1: $delta$)
)
```



May be a function that gets the state or transition name and returns a computed label. If `#none` is returned, the name is not put into the label map.

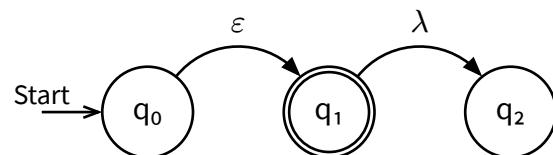
Argument

`(input-labels): (:)`

dictionary | function

A dictionary with custom labels for inputs. Using this, inputs can be labeled with arbitrary content instead of strings or numbers.

```
#finite.automaton(
  (q0: (q1:"e"), q1: (q2:"l")),
  input-labels: (
    e: $epsilon$,
    l: $lambda$)
)
```



May be a function that gets the input as a `str` and returns a computed label. If `#none` is returned, the input is not put into the label map.

Argument

`{format}: (col, row, v) => [#v]`

function

A function to format the value in a table cell. The function takes a column and row index and the cell content as a `str` and generates content: `(int, int, str) → content`.

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

δ	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 `#transition-table.format: (array) → str`

```
#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.

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

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 an automaton spec.

Argument
`{spec}` spec
 Automaton specification.

Argument
`{initial}: auto` str | auto | none
 The name of the initial state. For auto, the first state in {states} is used.

Argument
`{final}: auto` str | 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`
 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) → str.

```
#add-trap({spec}, {trap-name}: "TRAP", {trap-label}: none) → spec
```

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 usually not drawn. This function adds a trap-state to such a partial automaton and thus completes it.

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

```
#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"

str

Name for the new trap-state.

Argument:

{trap-label}: none

str

Label for the new trap-state.

```
#accepts({spec}, {word}, {format}: (spec, states) => states
    .map(((s, i)) => if i != none [
        #spec.labels.at(s, default: s) #box[#sym.arrow.r#place(top + center,
        dy: -88%)[#text(.88em, spec.input-labels.at(
            i,
            default: raw(i),
        ))]]
    ] else [#spec.labels.at(s, default: s)])
    .join()) → content
```

Tests if {word} is accepted by a given automaton.

The result is 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")

```

$q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_0 \xrightarrow{0} q_1$

false

Argument:

{spec}

spec

Automaton specification.

Argument:

{word}

str

A word to test.

Argument:

{format}: (spec, states) => states

```
.map(((s, i)) => if i != none [
    #spec.labels.at(s, default: s) #box[#sym.arrow.r#place(top +
center, dy: -88%)[#text(.88em, spec.input-labels.at(
    i,
    default: raw(i),
))]]
] else [#spec.labels.at(s, default: s)])
.join()
```

function

A function to format the result.

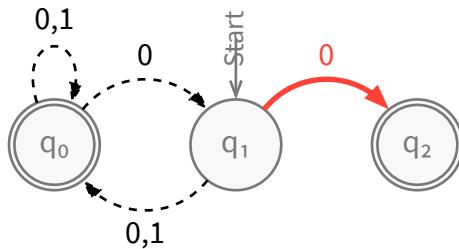
II.3 Styling the output

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 element 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 of 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 (-), for example q1-q2.

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.
- **{extrude}**: **0.88**

- ▶ label:

- **{text}**: **auto** State label.
- **{size}**: **1em** Initial text size for the labels (will be modified to fit the label into the state's circle).

- **{fill}**: **none** Color of the label text.

- **{padding}**: **auto** Padding around the label.

- ▶ initial:

- {anchor}: auto** Anchorpoint to point the initial arrow to.
- {label}: auto** Text above the arrow.
- {stroke}: auto** Stroke for the arrow.
- {scale}: auto** Scale of the arrow.
- transitions
 - {curve}: 1.0** “Curviness” of transitions. Set to **0** to get straight lines.
 - {stroke}: auto** Stroke for transition lines.
 - ▶ label:
 - {text}: ""** Transition label.
 - {size}: 1em** Size for label text.
 - {fill}: none** Color for label text. When set to **none**, the color is inherited from the transition’s stroke color.
 - {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.

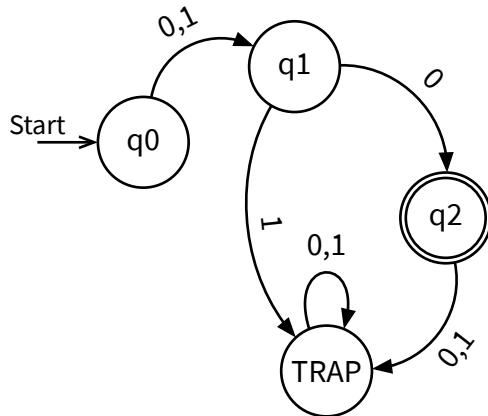
II.4 Using #cetz.canvas

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))
})
```

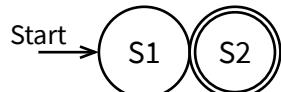


II.4.1 Element functions

```
#state(
  {position},
  {name},
  {label}: auto,
  {initial}: false,
  {final}: false,
  {anchor}: none,
  ..{style}
) → array
```

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 state's center.

Argument:

{name}

str

Name for the state.

Argument:

`{label}: auto``str | content | auto | none`

Label for the state. If set to `auto`, the `(name)` is used.

Argument:

`{initial}: false``bool | alignment | dictionary`

Whether this is an initial state. This can be either

- `true`,
- an `alignment` to specify an anchor for the initial marking,
- a `str` to specify text for the initial marking,
- an `dictionary` with the keys `anchor` and `label` to specify 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``bool`

Whether this is a final state.

Argument:

`{anchor}: none``str`

Anchor to use for drawing.

Argument:

`..{style}``any`

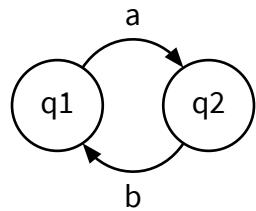
Styling options.

```
#transition(
  {from},
  {to},
  {inputs}: none,
  {label}: auto,
  {anchor}: top,
  ..{style}
) → array
```

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)``str`

Name of the starting state.

Argument:

`(to)``str`

Name of the ending state.

Argument:

`(inputs): none``str | array | none`A list of input symbols for the transition. If provided as a `str`, it is split at commas to get the list of input symbols.

Argument:

`(label): auto``str | content | auto | dictionary`A label for the transition. For `auto` the `(input)` symbols are joined with commas (,). Can be a `dictionary` with a `text` key and additional styling keys.

Argument:

`(anchor): top``alignment`

Anchor for loops. Has no effect on normal transitions.

Argument:

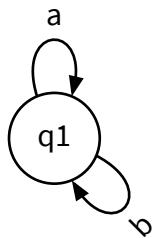
`..{style}``any`

Styling options.

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

Create a transition loop on a state.

```
#cetz.canvas({
  import finite.draw: state, loop
  state((0,0), "q1")
  loop("q1", label:"a")
  loop("q1", anchor: bottom+right, label:"b")
})
```



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

Argument:

`{state}`

`str`

Name of the state to draw the loop on.

Argument:

`{inputs}: none`

`str | array | none`

A list of input symbols for the loop. If provided as a `str`, it is split at commas to get the list of input symbols.

Argument:

`{label}: auto`

`str | content | auto | dictionary`

A label for the loop. For `auto` the `{input}` symbols are joined with commas (,). Can be a `dictionary` with a text key and additional styling keys.

Argument:

`{anchor}: top`

`alignment`

Anchor for the loop.

Argument:

..(style)

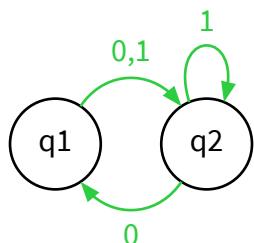
any

Styling options.

#transitions({states}, ..(style)) → content

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

```
#cetz.canvas({
    import finite.draw: state, transitions
    state((0,0), "q1")
    state((2,0), "q2")
    transitions(
        (
            q1: (q2: (0, 1)),
            q2: (q1: 0, q2: 1)
        ),
        transition: (stroke: green)
    )
})
```



Argument:

{states}

transition-table

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.

That means setting `(anchor)`: `"west"` for a state will anchor the state at the `west` anchor of the state's 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

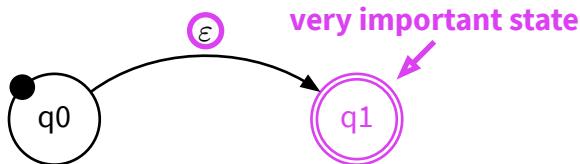
    let magenta = rgb("#dc41f1")

    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 accommodate different scenarios.

II.5.1 Available layouts

`#create-layout({positions}: (:), {anchors}: (:)) → array`

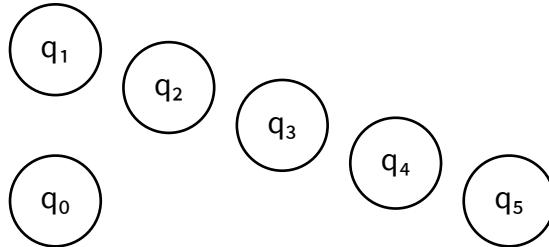
Helper function to create a layout dictionary by providing `(positions)` and/or `(anchors)`.

`#custom({spec}, {positions}: (:), {position}: (0, 0), {style}: (:)) → array`

Create a custom layout from a `dictionary` with state `coordinate`s.

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

```
#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: (0,2), rest:(rel: (1.5,-.5))
    )))
)
```



Argument

`{spec}`

`spec`

Automaton specification.

— Argument:

`{positions}: (:)`

dictionary

A dictionary with `coordinate`s for each state.

The dictionary contains each state's name as a key and the new coordinate as a value.

— Argument:

`{position}: (0, 0)`

coordinate

Position of the anchor point.

— Argument:

`{style}: (:)`

dictionary

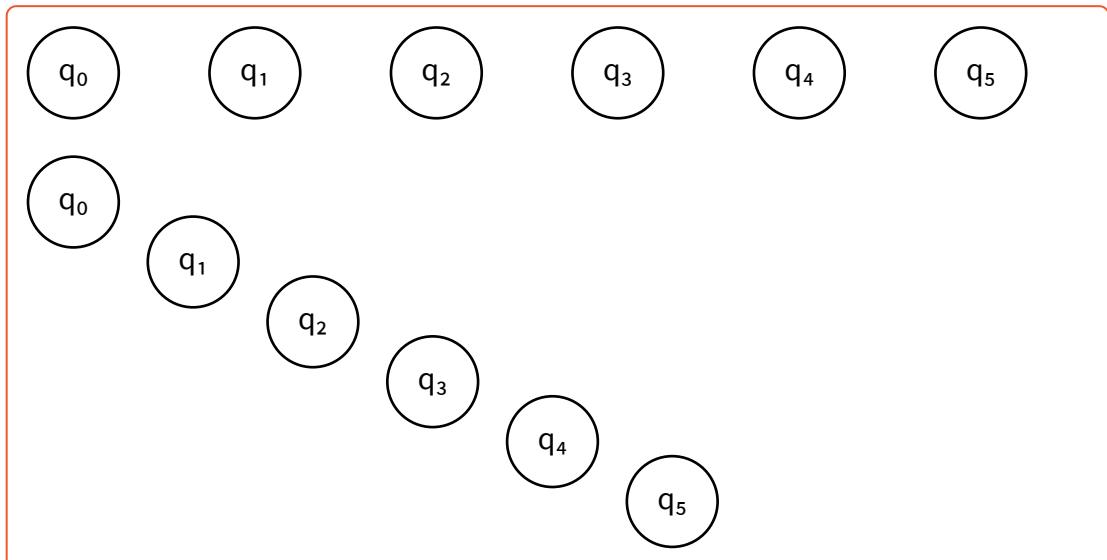
Styling options.

```
#linear(
    {spec},
    {dir}: right,
    {spacing}: default-style.state.radius * 2,
    {position}: (0, 0),
    {style}: (:)
) → array
```

Arrange states in a line.

The direction of the line can be set via `{dir}` either to an `alignment` or a direction vector with an x and y shift. Note that the length of the vector is set to `{spacing}` and only the direction is used.

```
#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(spacing: .5, dir:(2, -1))
)
```



— Argument:

{spec}

spec

Automaton specification.

— Argument:

{dir}: right

vector | alignment | 2d alignment

Direction of the line.

— Argument:

{spacing}: default-style.state.radius * 2

float

Spacing between states on the line.

— Argument:

{position}: (0, 0)

coordinate

Position of the anchor point.

— Argument:

{style}: (:)

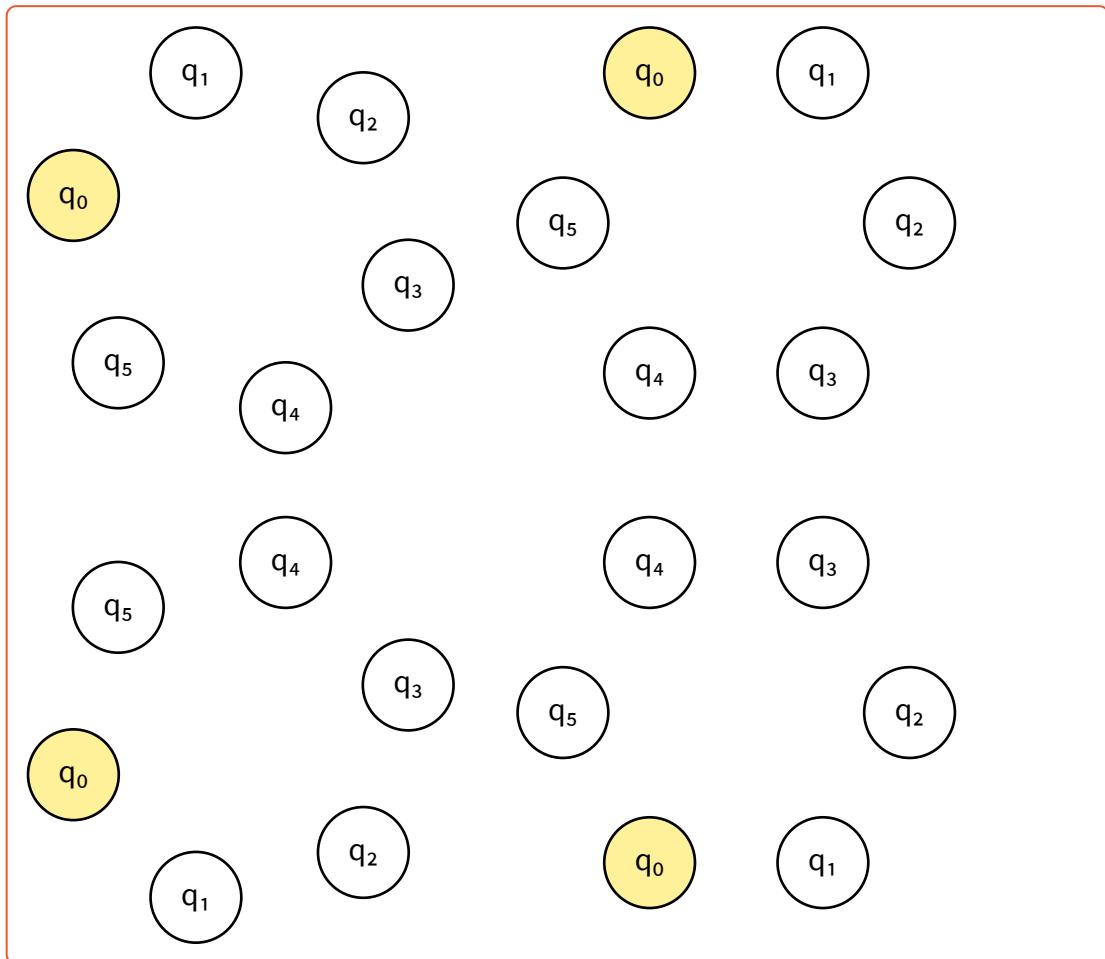
dictionary

Styling options.

```
#circular(
  {spec},
  {dir}: right,
  {spacing}: default-style.state.radius * 2,
  {radius}: auto,
  {offset}: 0deg,
  {position}: (0, 0),
  {style}: (:)
)
```

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:

{spec}

spec

Automaton specification.

Argument:

{dir}: right

alignment

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

Argument:

{spacing}: default-style.state.radius * 2

float

Spacing between states on the circle.

Argument:

{radius}: auto

float | auto

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

- Argument:
 - `{offset}: 0deg` angle
 - An offset angle to place the first state at.

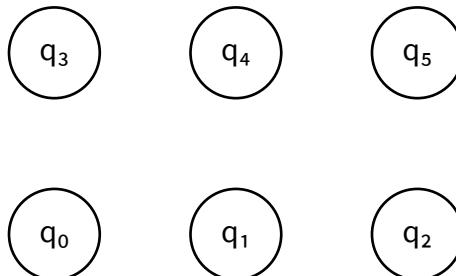
- Argument:
 - `{position}: (0, 0)` coordinate
 - Position of the anchor point.

- Argument:
 - `{style}: (:)` dictionary
 - Styling options.

```
#grid(
  {spec},
  {columns}: 4,
  {spacing}: default-style.state.radius * 2,
  {position}: (0, 0),
  {style}: (:
) → array
```

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)
)
```



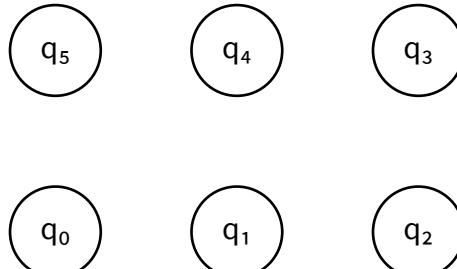
- Argument:
 - `{spec}` spec
 - Automaton specification.

Argument:	<code>(columns): 4</code>	int
Number of columns per row.		
Argument:	<code>(spacing): default-style.state.radius * 2</code>	float
Spacing between states on the grid.		
Argument:	<code>(position): (0, 0)</code>	coordinate
Position of the anchor point.		
Argument:	<code>(style): ()</code>	dictionary
Styling options.		

```
#snake(
    {spec},
    {columns}: 4,
    {spacing}: default-style.state.radius * 2,
    {position}: (0, 0),
    {style}: ())
) → array
```

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)
)
```



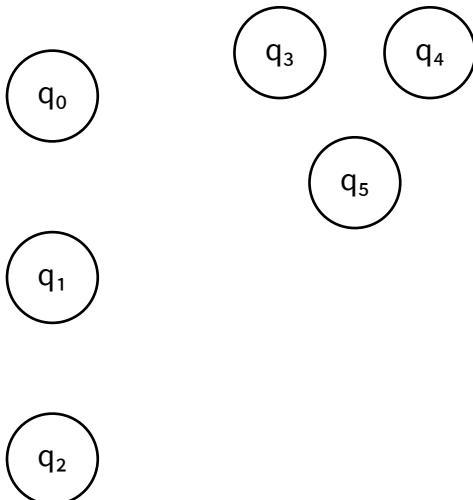
Argument:	<code>(spec)</code>	spec
Automaton specification.		
Argument:	<code>(columns): 4</code>	int
Number of columns per row.		
Argument:	<code>(spacing): default-style.state.radius * 2</code>	float
Spacing between states in the grid.		
Argument:	<code>(position): (0, 0)</code>	coordinate
Position of the anchor point.		
Argument:	<code>(style): (:)</code>	dictionary
Styling options.		

```
#group(
  {spec},
  {grouping}: auto,
  {spacing}: default-style.state.radius * 2,
  {layout}: linear.with(dir: bottom),
  {position}: (0, 0),
  {style}: (:)
) → array
```

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

```
let aut = range(6).fold((()), (d, s) => {d.insert("q"+str(s), none);
d})
#finite.automaton(
  aut,
  initial: none, final: none,
  layout: finite.layout.group.with(
    grouping: 3,
    spacing: 4,
    layout: (
      finite.layout.linear.with(dir: bottom),
```

```
    finite.layout.circular,
)
)
}
```



See [Section VI](#) for a more comprehensive example.

— Argument —

`{spec}`

`spec`

Automaton specification.

— Argument —

`{grouping}: auto`

`int | 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 each group.

— Argument —

`{spacing}: default-style.state.radius * 2`

`float`

Spacing between groups.

— 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 —

`{position}: (0, 0)`

`coordinate`

Position of the anchor point.

Argument:

`{style}: (:)`

dictionary

Styling options.

II.6 Utility functions

`#align-to-anchor`
`#align-to-vec`
`#call-or-get`
`#cubic-normal`
`#cubic-pts`
`#fit-content`

`#get-inputs`
`#get-states`
`#is-dea`
`#label-pt`
`#loop-pts`
`#mark-dir`

`#mid-point`
`#transition-pts`
`#transpose-table`
`#vector-normal`
`#vector-rotate`
`#vector-set-len`

`#align-to-anchor((align)) → str`

Return anchor name for an `alignment`.

Argument:

`(align)`

alignment

The alignment to convert.

`#align-to-vec((a)) → vector`

Returns a vector for an alignment.

Argument:

`(a)`

alignment

The alignment to convert.

`#call-or-get((value), ..(args)) → any`

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

Argument:

`(value)`

any

Value to call or return.

Argument:

`..(args)`

arguments

Arguments to pass if `(value)` is a function.

```
#cubic-normal(
  {a},
  {b},
  {c},
  {d},
  {t}
) → vector
```

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

Argument	{a}	vector
	Start point.	
Argument	{b}	vector
	End point.	
Argument	{c}	vector
	First control point.	
Argument	{d}	vector
	Second control point.	
Argument	{t}	float
	Parameter value between 0 and 1.	

```
#cubic-pts({a}, {b}, {curve}: 1) → array
```

Calculate the control points for a transition bezier curve.

Argument	{a}	vector
	Start point.	
Argument	{b}	vector
	End point.	

Argument <code>{curve}: 1</code>	<code>float</code>
Curvature factor.	
<code>#fit-content(</code> <code> {ctx},</code> <code> {width},</code> <code> {height},</code> <code> {content},</code> <code> {size}: auto,</code> <code> {min-size}: 6pt</code> <code>) → content</code>	
Fits (text) content inside the available space.	
Argument <code>{ctx}</code>	<code>dictionary</code>
The canvas context.	
Argument <code>{width}</code>	<code>float</code>
Available width.	
Argument <code>{height}</code>	<code>float</code>
Available height.	
Argument <code>{content}</code>	<code>str content</code>
The content to fit.	
Argument <code>{size}: auto</code>	<code>length auto</code>
The initial text size.	
Argument <code>{min-size}: 6pt</code>	<code>length</code>
The minimal text size to use.	

`#get-inputs({table}, {input-labels}): (:)` → array

Creates a tuple of (`array`, `dictionary`). The array is the set of unique input strings for an automaton and the dictionary is a mapping of input strings to labels.

— Argument

`(table)`

transition-table

A transition table.

`#get-states({table}) → array`

Returns a list of unique states referenced in a transition table. States are either keys in `(table)` or referenced in a transition.

— Argument

`(table)`

transition-table

A transition table.

`#is-dea({table}) → bool`

Checks if a given `spec` 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
```

) → `vector`

Calculate the location for a transition's label, based on its bezier points.

— Argument

`(a)`

vector

Start point.

Argument
(b) vector
 End point.

Argument
(c) vector
 First control point.

Argument
(d) vector
 Second control point.

Argument
(style) dictionary
 Resolved style dictionary with label and curve keys.

Argument
(loop): false bool
 Whether the transition is a loop.

#loop-pts(({start}), ({start-radius}), ({anchor}): top, ({curve}): 1) → array

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

Argument
{start} vector
 Center of the state.

Argument
{start-radius} float
 Radius of the state.

Argument
{anchor}: top alignment
 Anchor point on the state.

Argument
{curve}: 1 float
 Curvature of the transition.

```
#mark-dir(
  {a},
  {b},
  {c},
  {d},
  {scale}: 1
) → vector
```

Calculate the direction vector for a transition mark (arrowhead).

Argument:	(a)	vector
Start point.		

Argument:	(b)	vector
End point.		

Argument:	(c)	vector
First control point.		

Argument:	(d)	vector
Second control point.		

Argument:	{scale}: 1	float
Scale for the resulting direction vector.		

```
#mid-point({a}, {b}, {c}, {d}) → vector
```

Compute the midpoint of a cubic bezier curve.

Argument:	(a)	vector
Start point.		

Argument:	(b)	vector
End point.		

```

    └── Argument:
        {c}                                     vector
        First control point.

    └── Argument:
        {d}                                     vector
        Second control point.

#transition-pts(
    {start},
    {end},
    {start-radius},
    {end-radius},
    {curve}: 1,
    {anchor}: top
) → array
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}                           float
        Radius of the start state.

    └── Argument:
        {end-radius}                            float
        Radius of the end state.

    └── Argument:
        {curve}: 1                               float
        Curvature of the transition.

    └── Argument:
        {anchor}: top                           alignment

```

Anchor point for loops.

#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}) → vector

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

Argument:

`{v}`

`vector`

The vector to compute the normal for.

#vector-rotate({vec}, {angle}) → vector

Rotates a vector by `{angle}` degrees around the origin.

Argument:

`{vec}`

`vector`

The vector to rotate.

Argument:

`{angle}`

`angle`

The angle to rotate by.

#vector-set-len({v}, {len}) → vector

Set the length of a `cetz.vector`.

Argument:

`{v}`

`vector`

The vector.

Argument:

`{len}`

`float`

The new length.

Part III Simulating input

FINITE has a set of functions to simulate, test and view finite automata.

III.1 Testing word acceptance

The `#accepts` function tests whether a given word is accepted by an automaton. It takes a `spec` or `transition-table` as its first argument and a word as a `str`.

If the word is rejected, `#accepts` returns `false`. If accepted, it returns formatted content showing the path of states and transitions that led to acceptance.

```
#let aut = (
    q0: (q1: "0"),
    q1: (q1: "0", q2: "1"),
    q2: none
)

Accepted: #finite.accepts(aut, "01")

Rejected: #repr(finite.accepts(aut, "10"))

Accepted: q00 → q11 → q2
Rejected: false
```

If the automaton has no final states, `#accepts` returns `false` for every word without an error.

```
#let no-final = finite.create-automaton(
    (q0: (q0: "0", q1: "1"), q1: (q0: "1", q1: "0")),
    final: none
)
#repr(finite.accepts(no-final, "01"))

false
```

The output of `#accepts` uses the `input-labels` stored in the spec for display. To see labels in the acceptance path, first build a spec with `#create-automaton` and the `(input-labels)` parameter:

```
#let spec = finite.create-automaton(  
  (q0: (q1: "e"), q1: (q2: "l"), q2: none),  
  input-labels: (e: $epsilon$, l: $lambda$)  
)  
#finite.accepts(spec, "el")
```

$$q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{\lambda} q_2$$

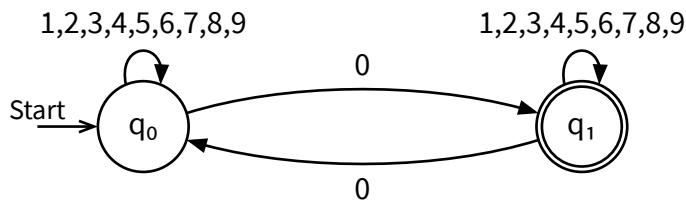
The `(word)` argument of `#accepts` is always a plain `str` using the raw input keys (e.g. `"e"`, `"l"`), not the display labels.

Part IV FLACI support

FINITE was heavily inspired by the online app [FLACI](#)⁶. FLACI lets you build automata in a visual online app and export your creations as JSON files. FINITE can import these files and render the result in your document.

FINITE currently only supports DEA and NEA automata.

```
#finite.flaci.automaton(read("flaci-export.json"))
```



Important

Read the FLACI json-file with the `#read` function, not the `#json` function. FLACI exports automata with a wrong encoding that prevents Typst from properly loading the file as JSON.

IV.0.1 FLACI functions

```
#flaci.load({data}) → spec
```

Loads `(data)` into an automaton `spec`. `(data)` needs to be a string and not a JSON dictionary.

Argument:

`(data)`

`str`

FLACI data read as a string via `#read`.

```
#flaci.automaton(  
  {data},  
  {layout}: auto,  
  {merge-layout}: true,  
  {style}: auto,  
  {merge-style}: true,  
  ..{args}  
) → content
```

⁶<https://flaci.org>

Show a FLACI file as an `#automaton`.

Read the FLACI json-file with the `#read` function, not the `#json` function. FLACI exports automata with a wrong encoding that prevents Typst from properly loading the file as JSON.

Currently only DEA and NEA automata are supported.

Argument:

`(data)`

`str`

FLACI data read as a string via `#read`.

Argument:

`(layout): auto`

`function`

Custom layout for the automaton. Will overwrite state positions from `(data)`.

Argument:

`(merge-layout): true`

`bool`

Custom state positions to merge with the ones found in `(data)`. This allows the placement of some states while the rest keeps their positions.

Argument:

`(style): auto`

`dictionary`

Custom styles to overwrite the defaults.

Argument:

`(merge-style): true`

`bool`

Custom styles to merge with the styles from `(data)`.

Argument:

`..{args}`

`any`

Further arguments for `#automaton`.

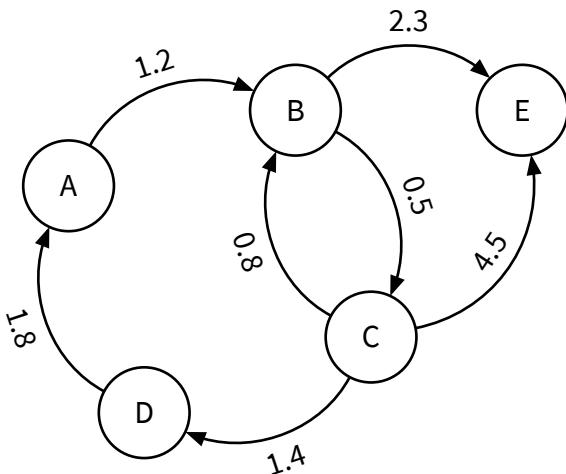
Part V Doing other stuff with FINITE

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

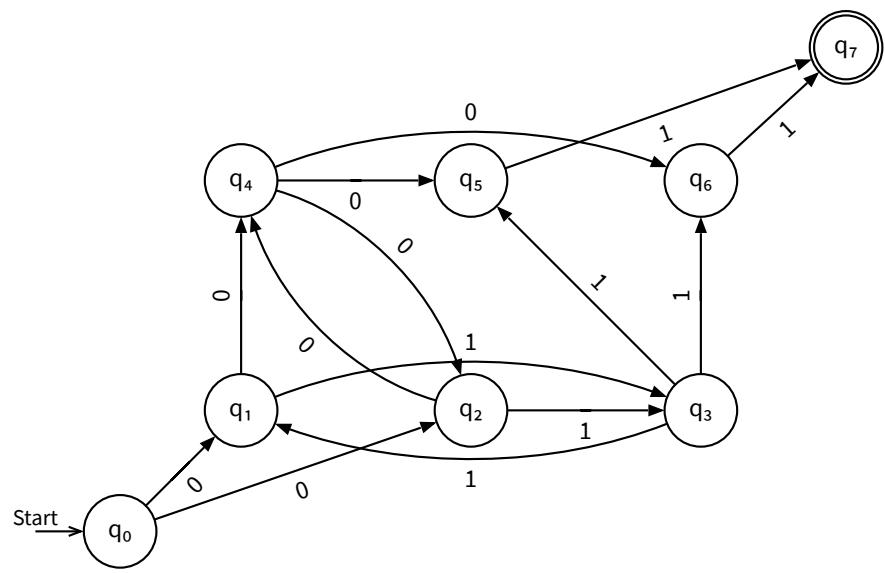
    state((0,0), "A")
    state((3,1), "B")
    state((4,-2), "C")
    state((1,-3), "D")
    state((6,1), "E")

    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 VI 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.custom.with(positions: (q0: (0, -2))),  
        finite.layout.grid.with(columns:3, spacing: 2.6, position: (2, 1)),  
        finite.layout.custom.with(positions: (q7: (8, 6)))  
    )  
,  
    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)),  
        q4-q6: (curve: 1)  
    )  
))
```



Part VII Index

A

#accepts 7, 18, 45, 46
#add-trap 17
#align-to-anchor 37
#align-to-vec 37
#automaton . 3, 6, 9, 19, 28,
47, 48

C

#call-or-get 37
#circular 31
#cetz.draw.content ... 27
 coordinate 2
#util.create-automaton.
6, 7, 45
#create-layout 28
#cubic-normal 37, 38
#cubic-pts 37, 38
#custom 28

F

#fit-content 37, 39

G

#get-inputs 37, 39
#get-states 37, 40
#grid 33
#cetz.draw.group . 26, 35

I

#is-dea 37, 40

L

#label-pt 37, 40
#cetz.draw.line 27
#linear 29
#flaci.load 47
#loop 25
#loop-pts 37, 41

M

#mark-dir 37, 42
#mid-point 37, 42

P

#powerset 17

S

#snake 34
spec 5
#state 22

T

#transition 23, 25
#transition-pts ... 37, 43
 transition-table 5
#transition-table . 7, 13
#transition-table 16
#transitions 26
#transpose-table . 37, 44

V

#vector-normal 37, 44
#vector-rotate 37, 44
#vector-set-len ... 37, 44