Regular expressions from scratch

Kian Kasad

Outline

Get these slides!

https://kasad.com/regex.pdf

Background

What are regular expressions?

What are regular expressions?

Also called “regex”

What are regular expressions?

Also called “regex”

They’re ways of matching (finding) patterns in text

What are regular expressions?

Also called “regex”

They’re ways of matching (finding) patterns in text

We define a language called “regular expressions” which match certain patterns so we don’t have to write the logic ourselves

Motivation: finding phone numbers

Let's say a phone number is `XXX-XXX-XXXX`, where `X` represents a numeric digit

Motivation: finding phone numbers

Let's say a phone number is `XXX-XXX-XXXX`, where `X` represents a numeric digit

Can't do a plain substring search; there are 10^{10} possibilities

Motivation: finding phone numbers

Let's say a phone number is `XXX-XXX-XXXX`, where `X` represents a numeric digit

Can't do a plain substring search; there are 10^{10} possibilities

We can instead write code to search for phone numbers

Motivation: finding phone numbers

```
for each position i in text, do
    if
        text[i] is a digit
        and text[i+1] is a digit
        and text[i+2] is a digit
        and text[i+3] is “-”
        ...
    then
        text[i...(i+9)] is a phone number
```

Regex language

Atoms

Regex language

Atoms

Any character other than the special characters that form the language matches itself

Regex language

Atoms

Any character other than the special characters that form the language matches itself

- `.` matches any single character

Regex language

Atoms

Any character other than the special characters that form the language matches itself

`.` matches any single character

`[xyz]` and `[x-z]` both match either “x” or “y” or “z”

Regex language

Atoms

Any character other than the special characters that form the language matches itself

`.` matches any single character

`[xyz]` and `[x-z]` both match either “x” or “y” or “z”

`(<pattern>)` treats the inner `<pattern>`, which may consist of multiple atoms, as a single atom

Regex language

Quantifiers

Regex language

Quantifiers

? matches zero or one of the preceding atom

Regex language

Quantifiers

? matches zero or one of the preceding atom

* matches zero or more of the preceding atom

Regex language

Quantifiers

`?` matches zero or one of the preceding atom

`*` matches zero or more of the preceding atom

`+` matches one or more of the preceding atom

Regex language

Quantifiers

`?` matches zero or one of the preceding atom

`*` matches zero or more of the preceding atom

`+` matches one or more of the preceding atom

`<pattern>|<pattern>` matches either the left or the right `<pattern>`

Example revisited: finding phone numbers

[1] `{N}` is a quantifier that repeats the preceding atom exactly N times

Example revisited: finding phone numbers

Without dashes:[1] `[0-9]{10}`

[1] `{N}` is a quantifier that repeats the preceding atom exactly N times

Example revisited: finding phone numbers

Without dashes:[1] `[0-9]{10}`

With dashes: `[0-9]{3}-[0-9]{3}-[0-9]{4}`

[1] `{N}` is a quantifier that repeats the preceding atom exactly N times

Example revisited: finding phone numbers

Without dashes:[1] `[0-9]{10}`

With dashes: `[0-9]{3}-[0-9]{3}-[0-9]{4}`

Optional dashes: `([0-9]{3}-?){2}[0-9]{4}`

[1] `{N}` is a quantifier that repeats the preceding atom exactly N times

Example revisited: finding phone numbers

Without dashes:[1] `[0-9]{10}`

With dashes: `[0-9]{3}-[0-9]{3}-[0-9]{4}`

Optional dashes: `([0-9]{3}-?){2}[0-9]{4}`

In JavaScript:

```
text.match(/([0-9]{3}-?){2}[0-9]{4}/)
```

[1] `{N}` is a quantifier that repeats the preceding atom exactly N times

How do we match regular expressions?

Finite state automata

Also called “finite state machines” or “FSA” or “FSM”

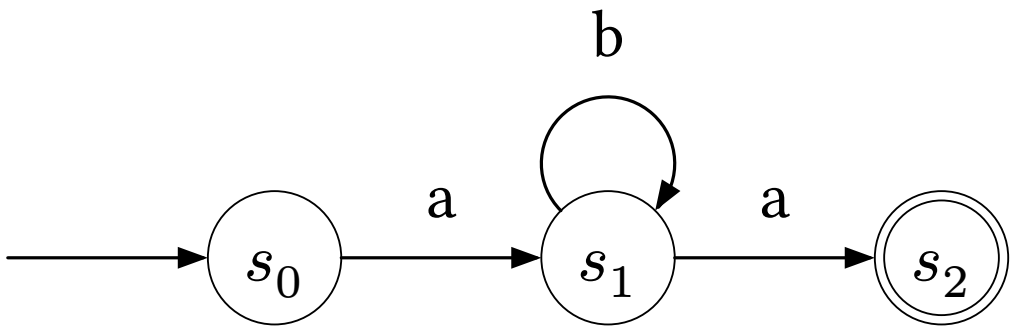

Figure 1: A simple finite state automaton representing a regular expression.

Finite state automata

Also called “finite state machines” or “FSA” or “FSM”

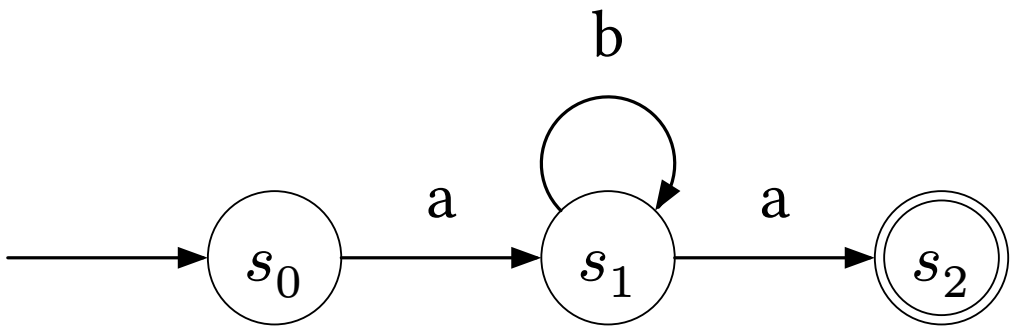

Figure 1: A simple finite state automaton representing a regular expression.

Represents the regular expression `ab*a`

Determinism

In order to efficiently execute a FSA, it must be deterministic

Determinism

In order to efficiently execute a FSA, it must be deterministic

An FSA is deterministic if for each state, no matter the next character in the text, there is only one possible next state

Determinism

In order to efficiently execute a FSA, it must be deterministic

An FSA is deterministic if for each state, no matter the next character in the text, there is only one possible next state

Two things make FSAs non-deterministic:

Determinism

In order to efficiently execute a FSA, it must be deterministic

An FSA is deterministic if for each state, no matter the next character in the text, there is only one possible next state

Two things make FSAs non-deterministic:

- Ambiguous transitions

Determinism

In order to efficiently execute a FSA, it must be deterministic

An FSA is deterministic if for each state, no matter the next character in the text, there is only one possible next state

Two things make FSAs non-deterministic:

- Ambiguous transitions
- Epsilon transitions

Determinism

Ambiguous transitions

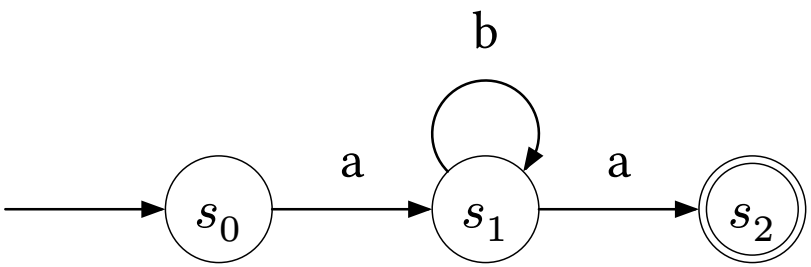

Figure 2: Deterministic finite automaton (DFA). Represents `ab*a`

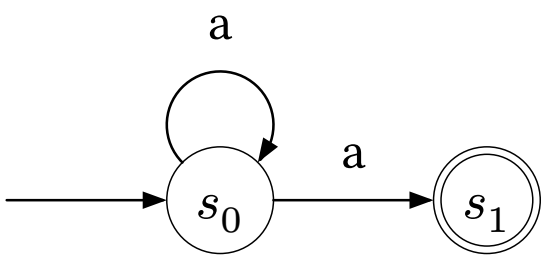

Figure 3: Non-deterministic finite automaton (NFA). Represents `a*a`

Determinism

Epsilon transitions

An epsilon transition (ε-transition) is a transition we can take at any time without reading another character of input

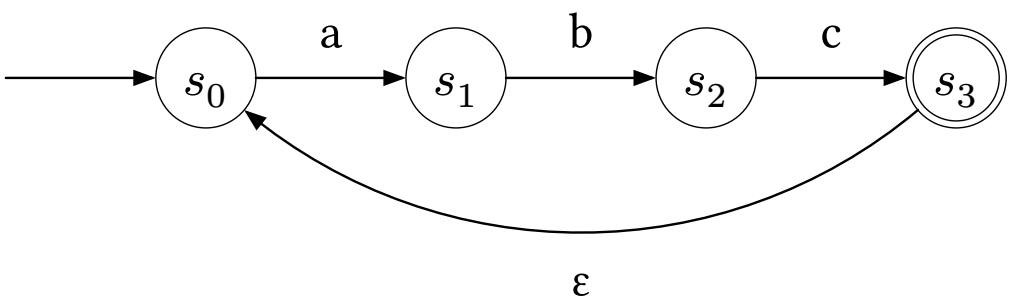

Figure 4: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

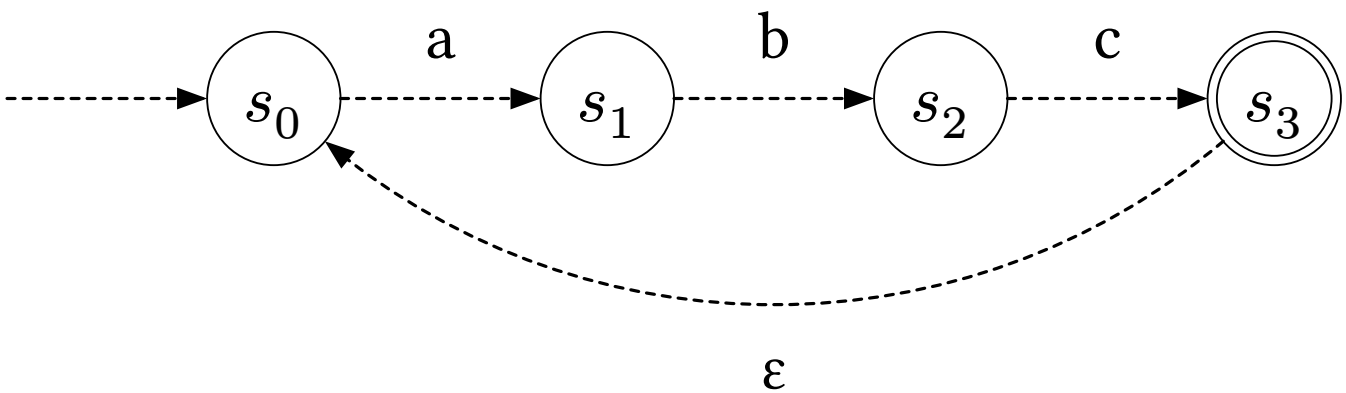

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

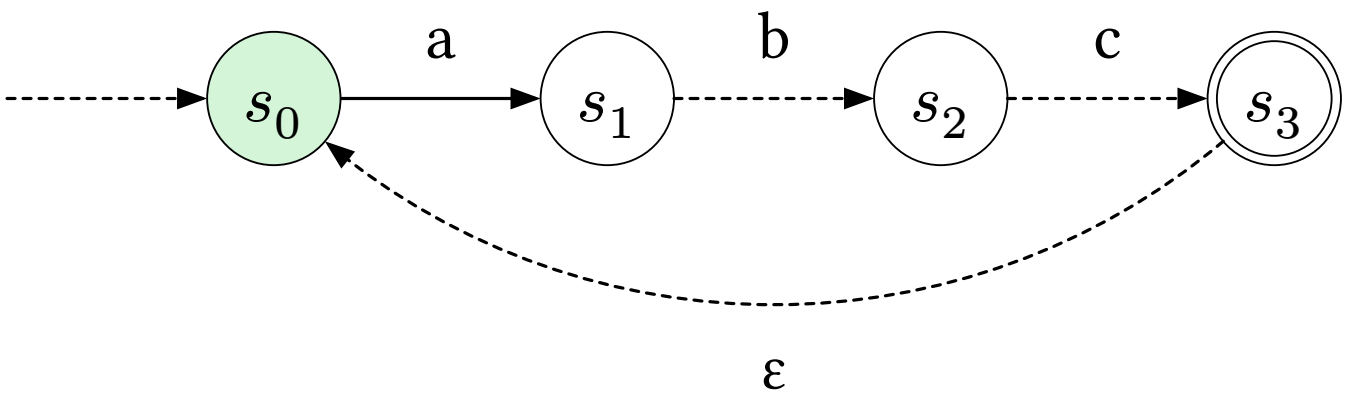

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: abcabc

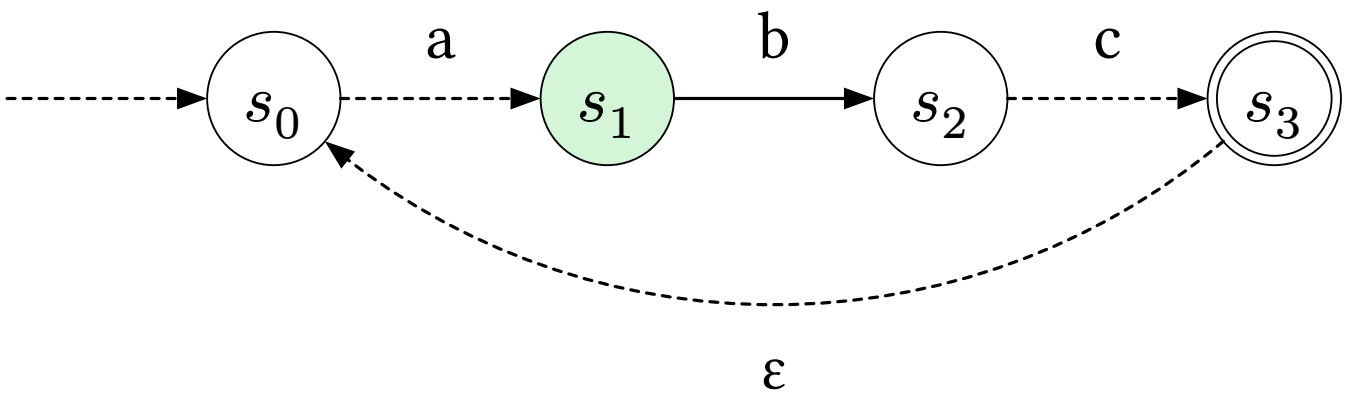

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

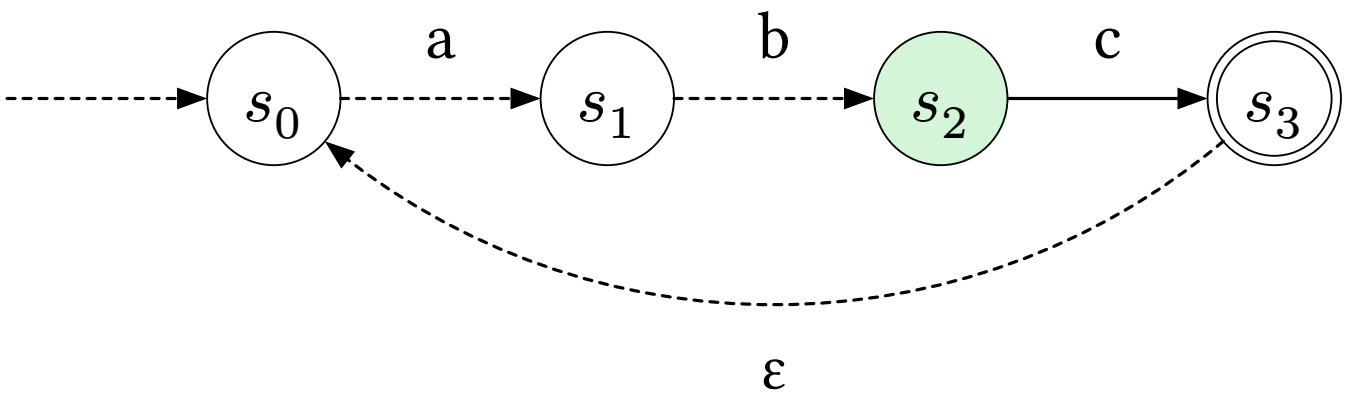

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

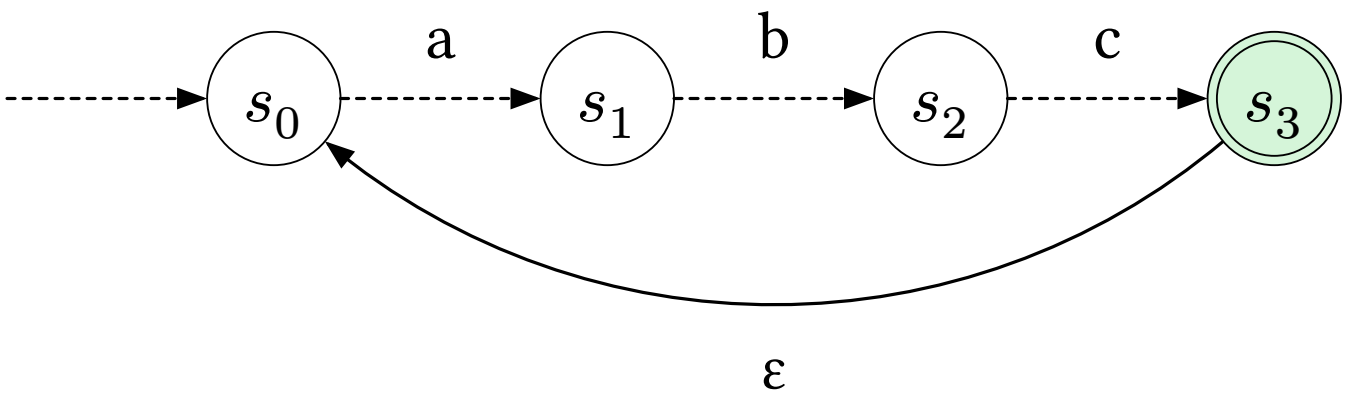

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

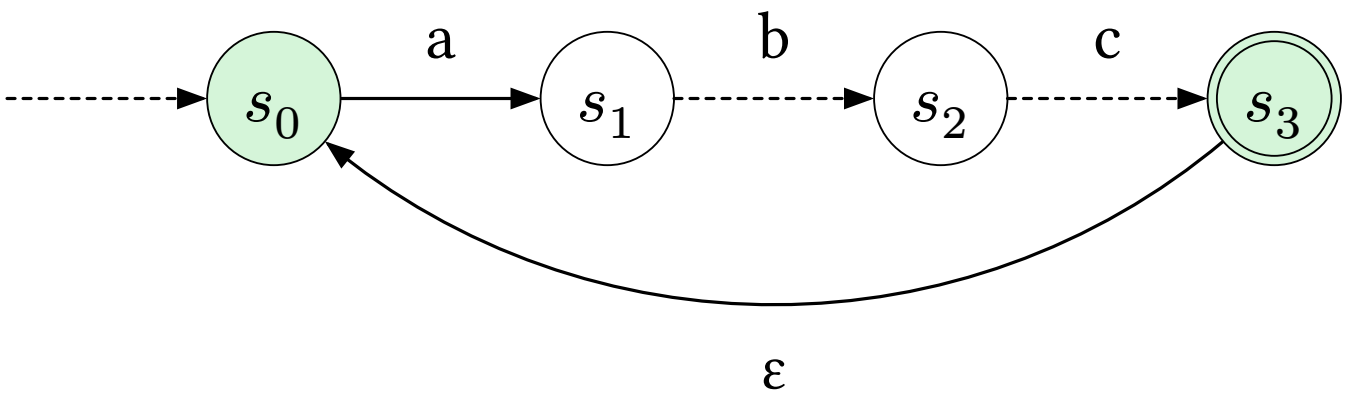

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

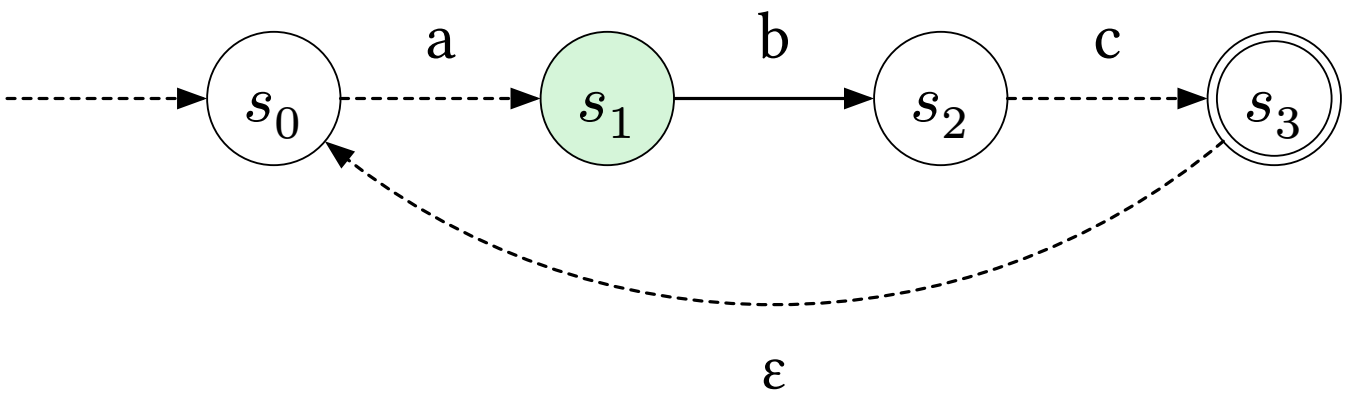

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

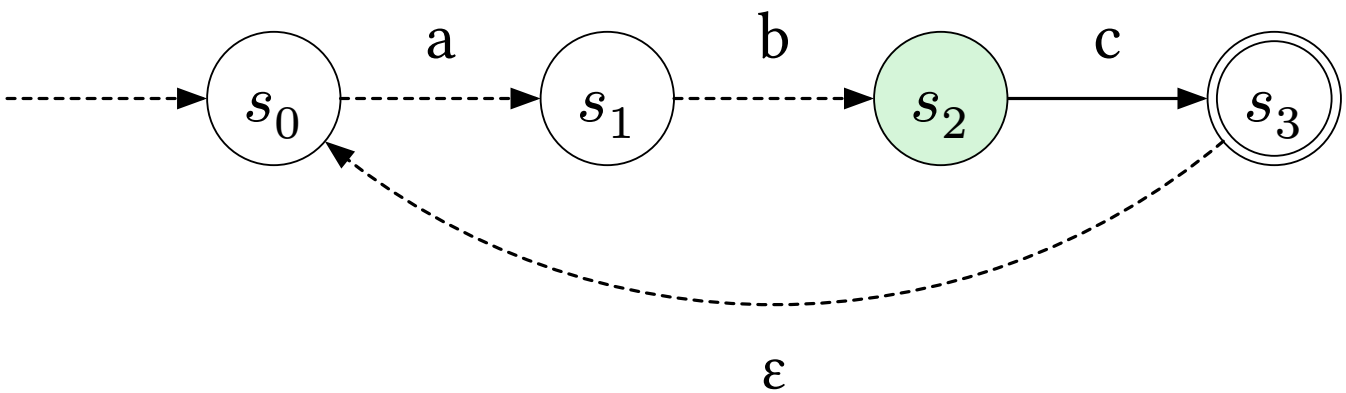

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

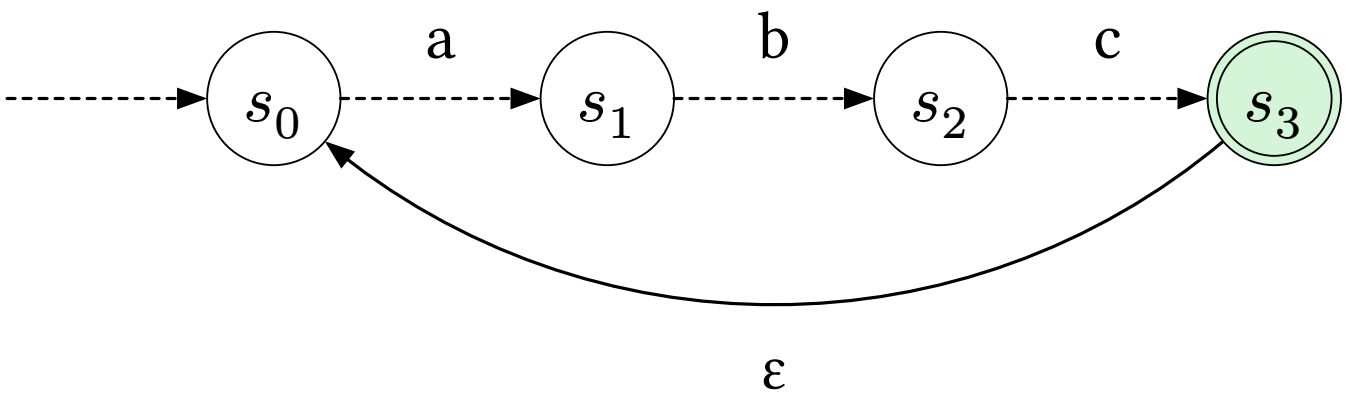

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ε-transitions

Input: `abcabc`

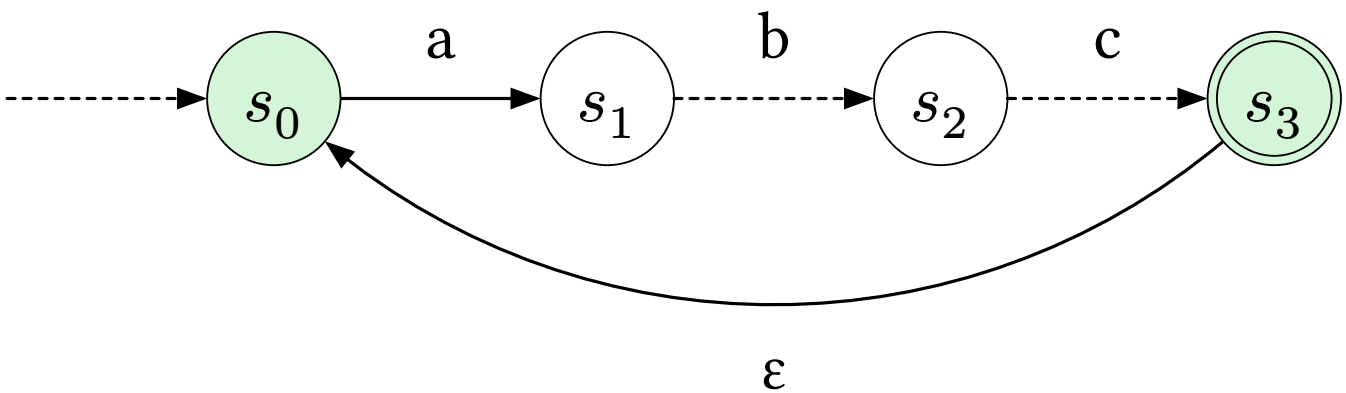

Figure 5: An NFA with ε-transitions. Represents `(abc)+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

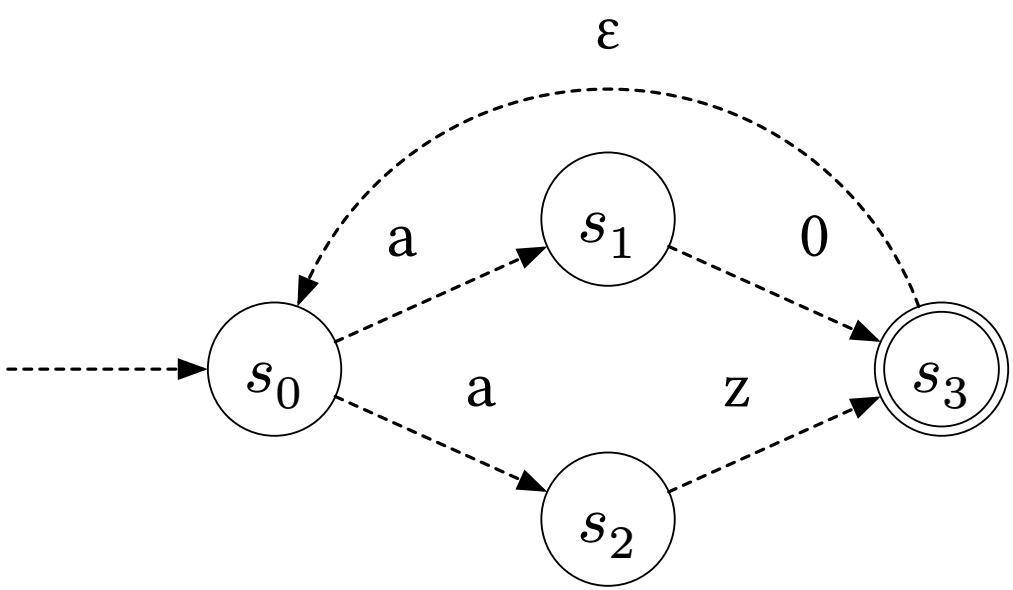

Figure 6: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

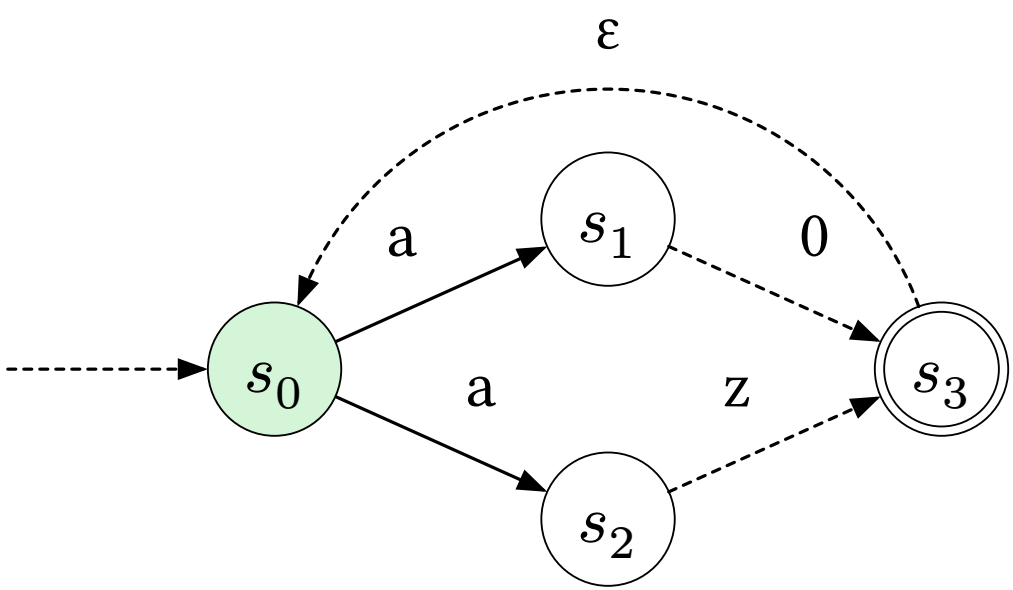

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

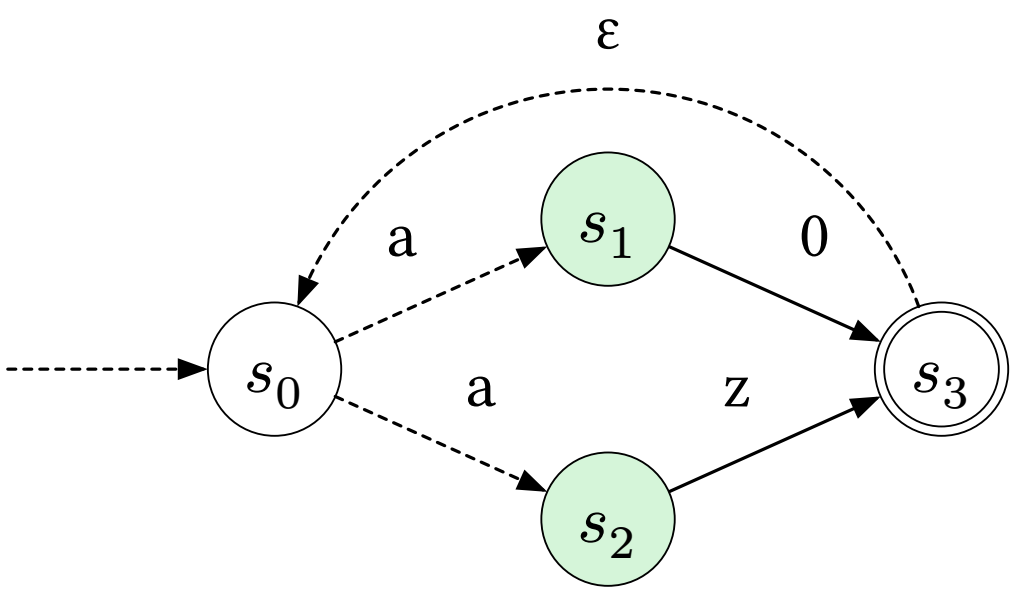

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `azа0aza0aza0`

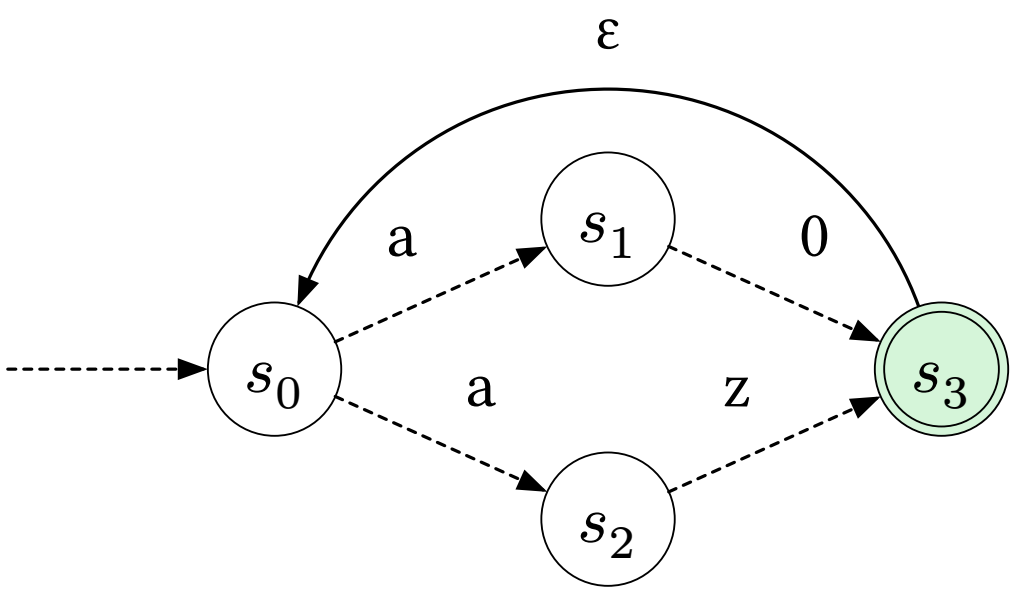

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `azа0aza0aza0`

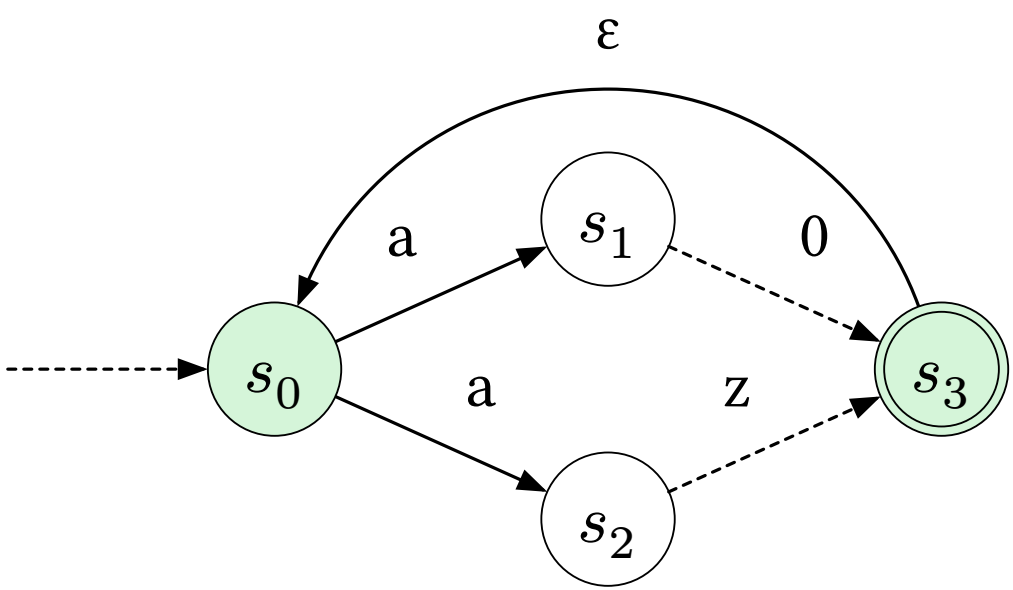

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

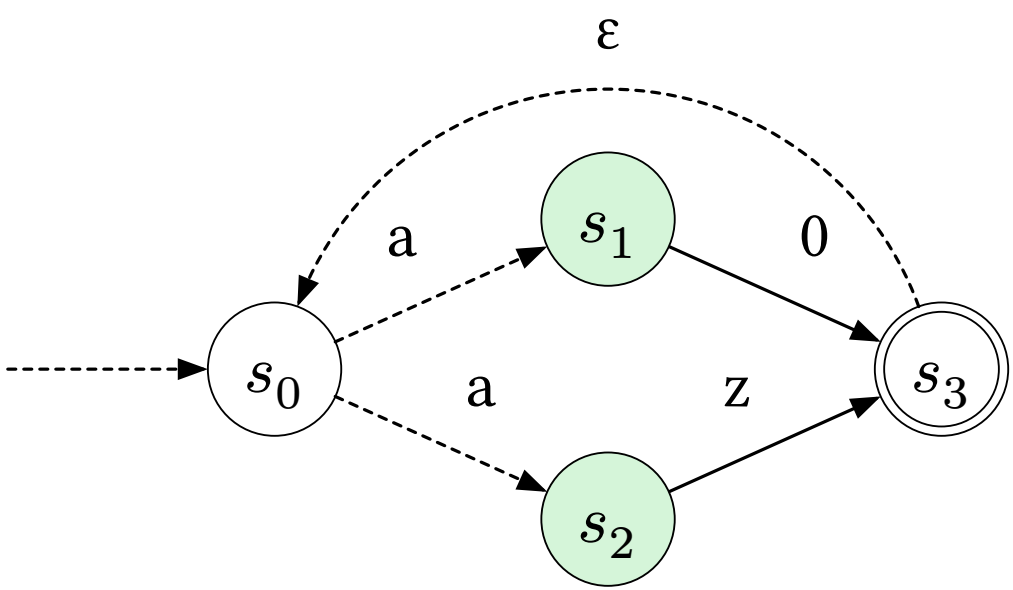

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

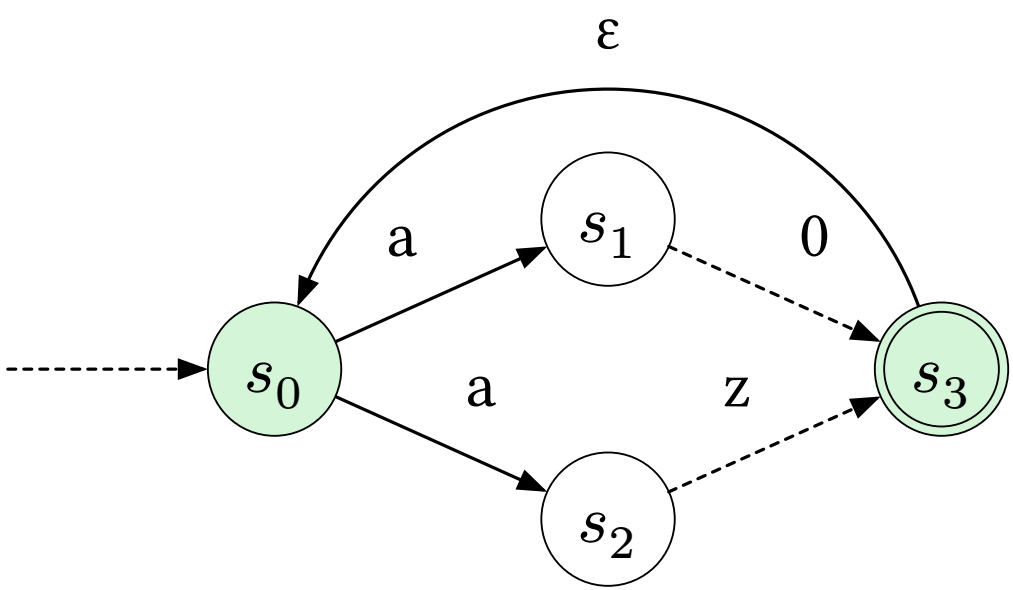

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

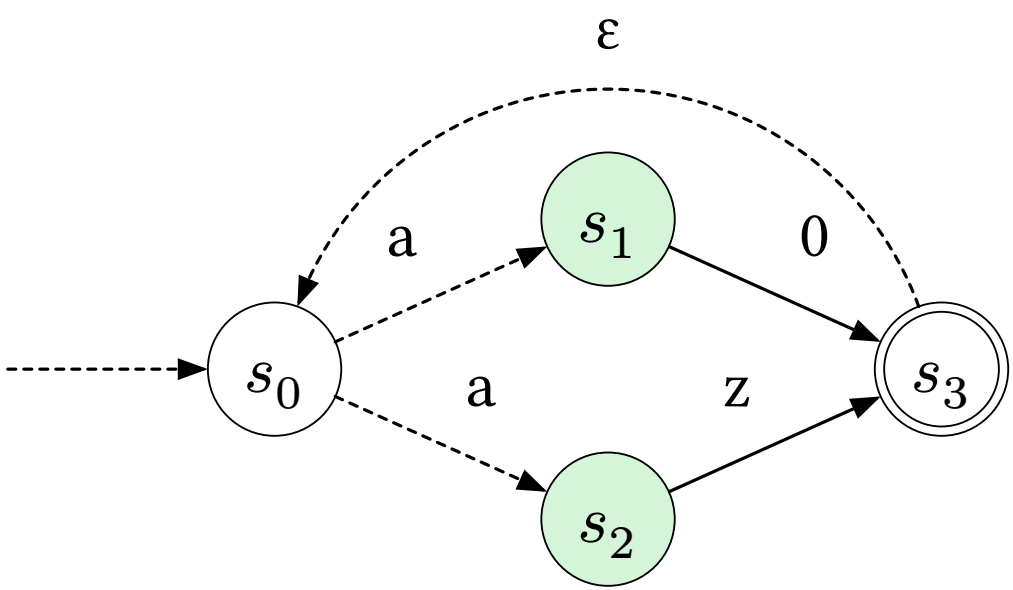

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

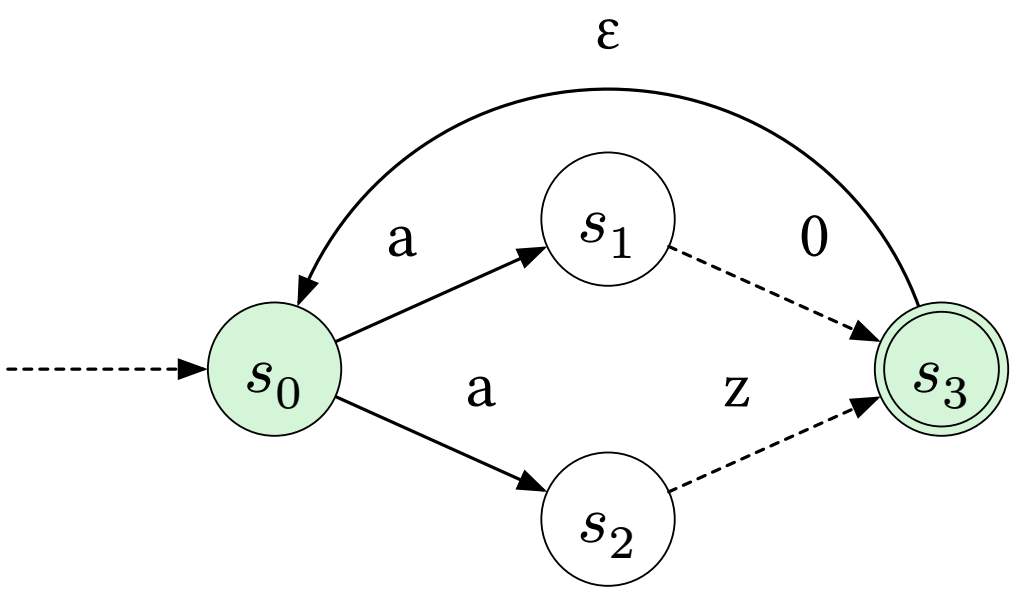

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

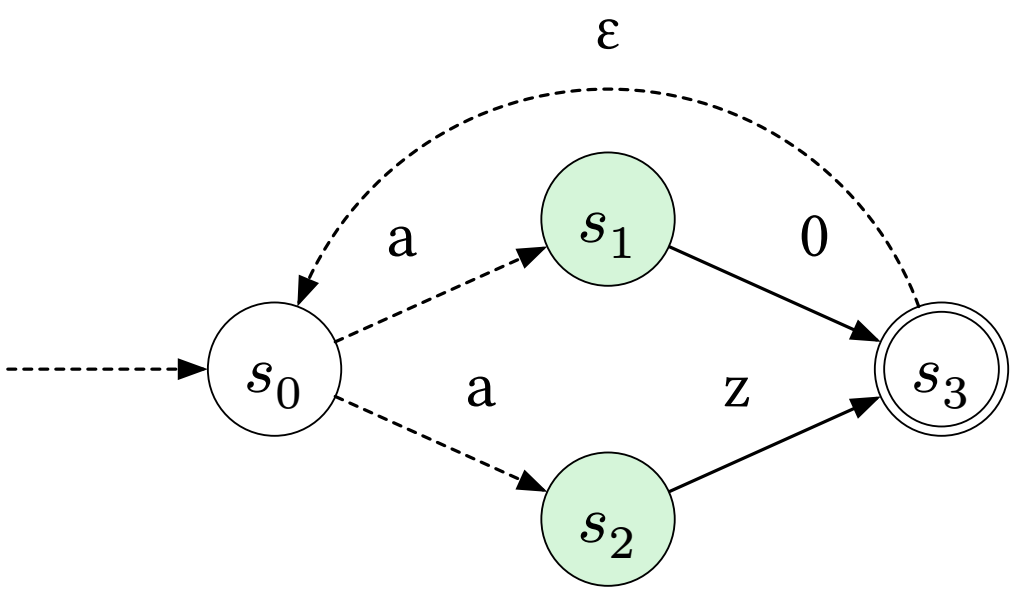

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

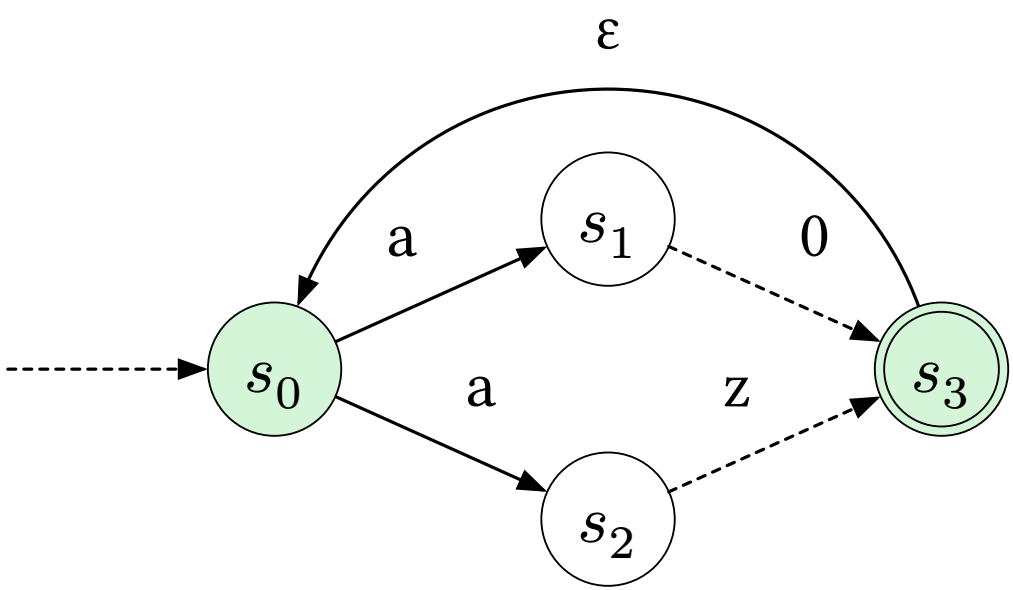

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

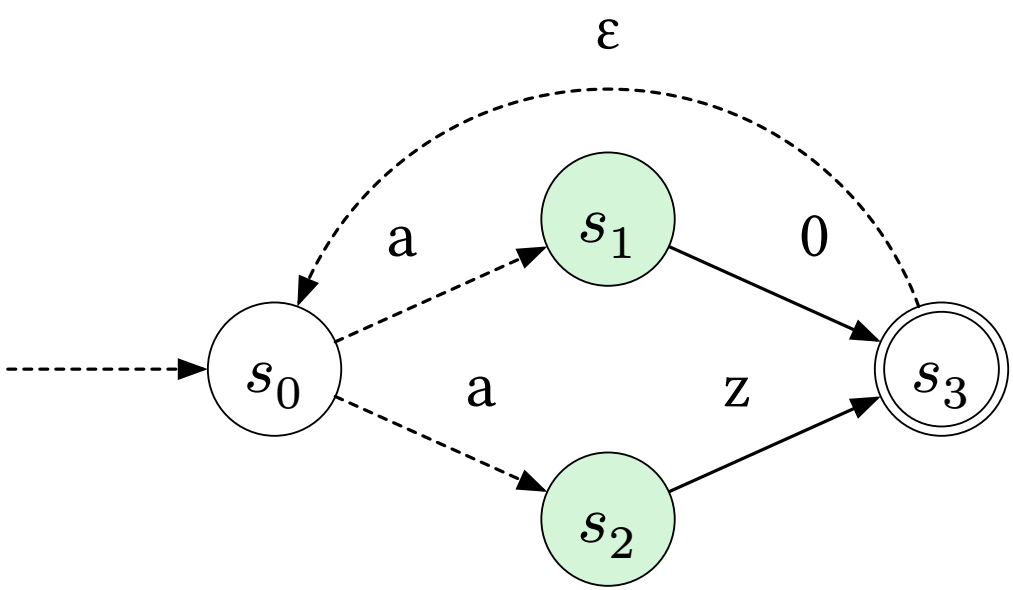

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

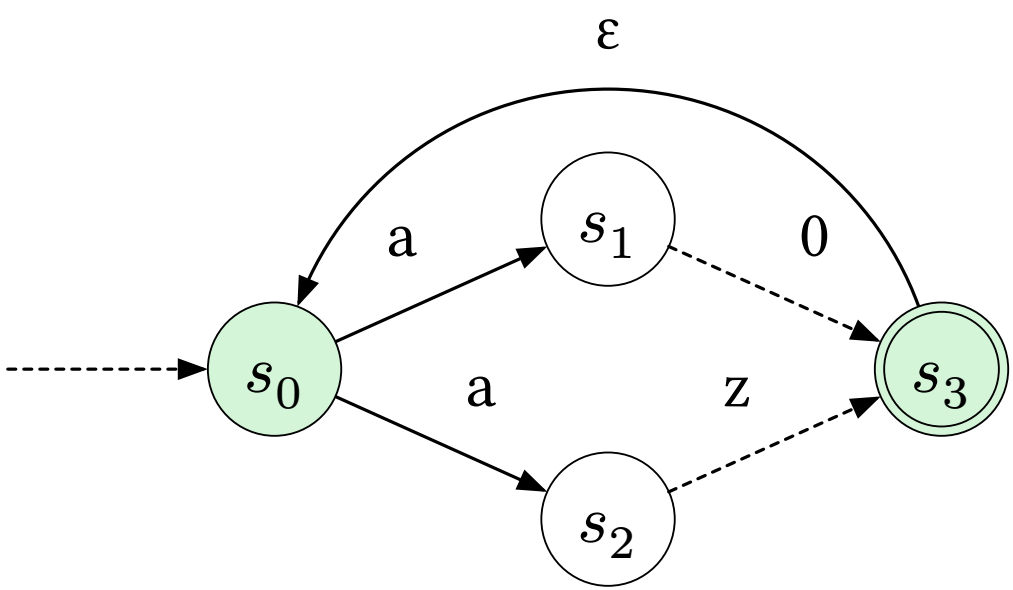

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

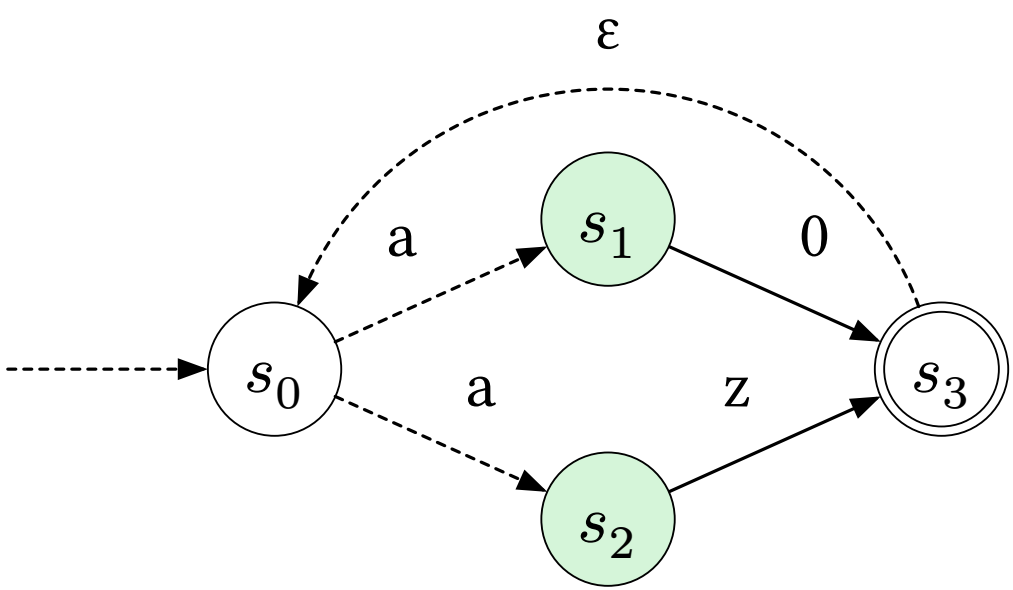

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Simulating NFAs with ambiguous transitions

Input: `aza0aza0aza0`

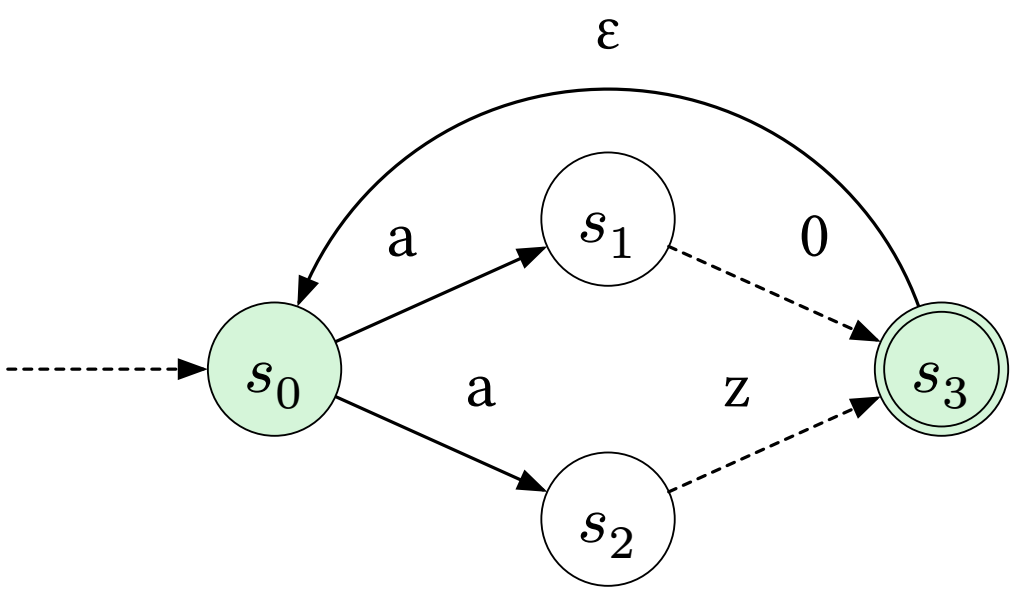

Figure 7: An NFA with ambiguous transitions. Represents `(a[z0])+`

Removing non-determinism

We can enumerate every possible set of active states and map each one to one state in a new FSA

This forms a DFA!

Removing non-determinism

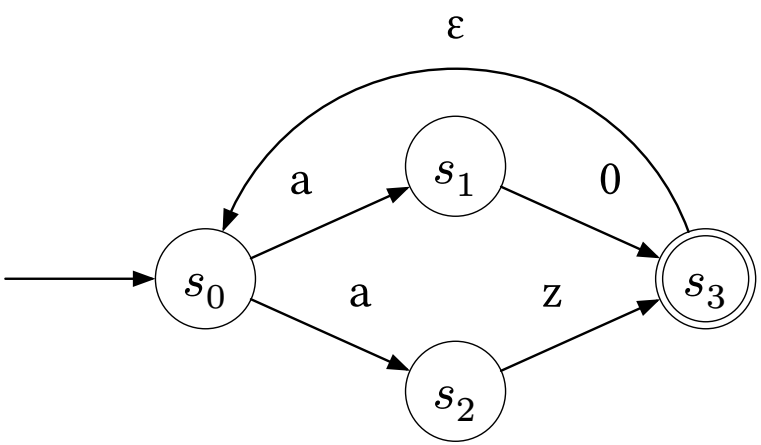

Figure 8: Conversion of an NFA into a DFA.

Removing non-determinism

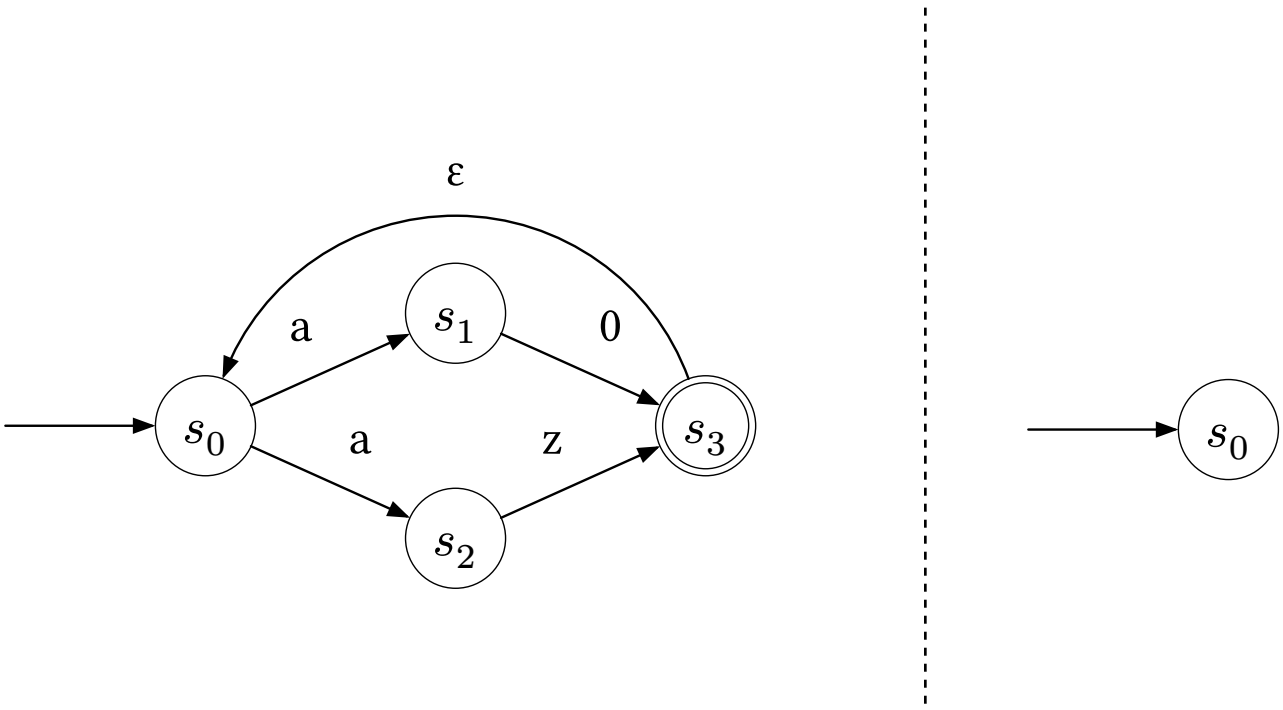

Figure 8: Conversion of an NFA into a DFA.

Removing non-determinism

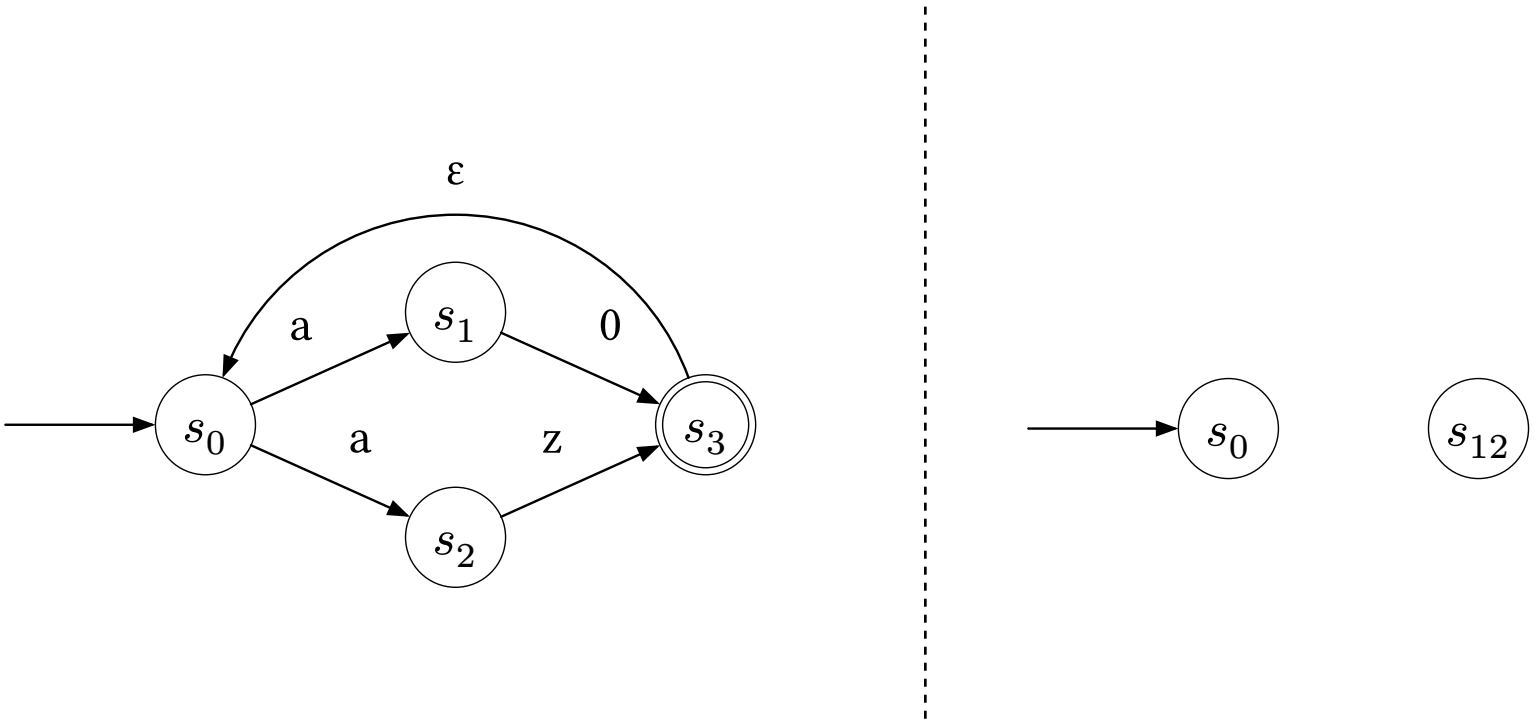

Figure 8: Conversion of an NFA into a DFA.

Removing non-determinism

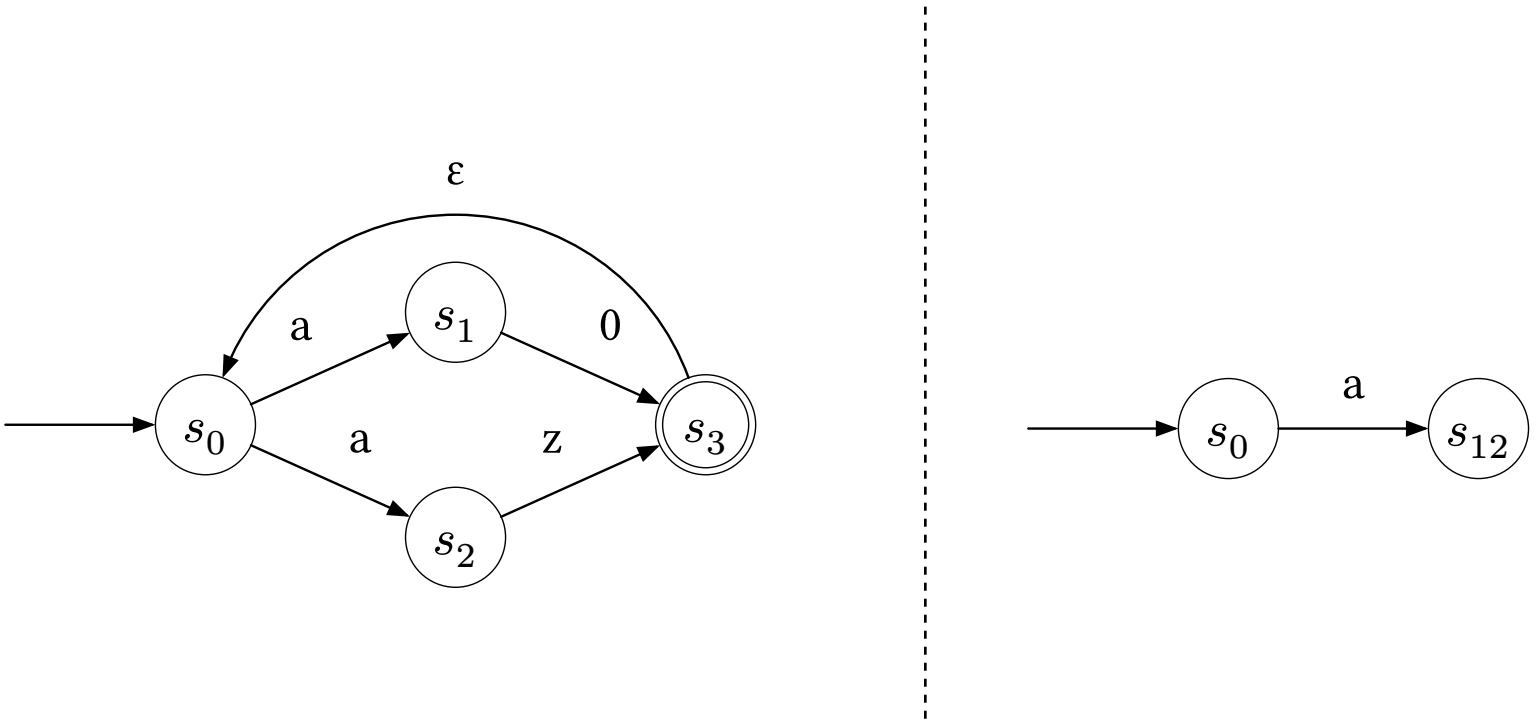

Figure 8: Conversion of an NFA into a DFA.

Removing non-determinism

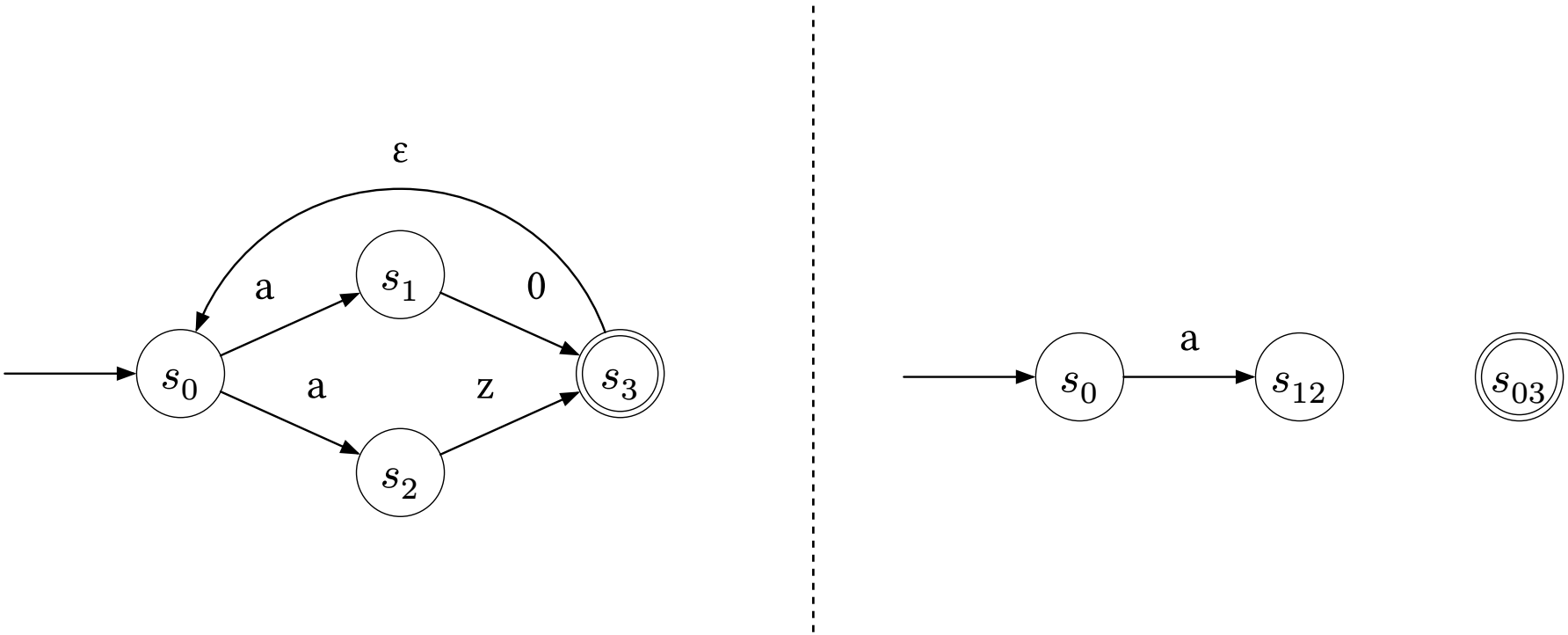

Figure 8: Conversion of an NFA into a DFA.

Removing non-determinism

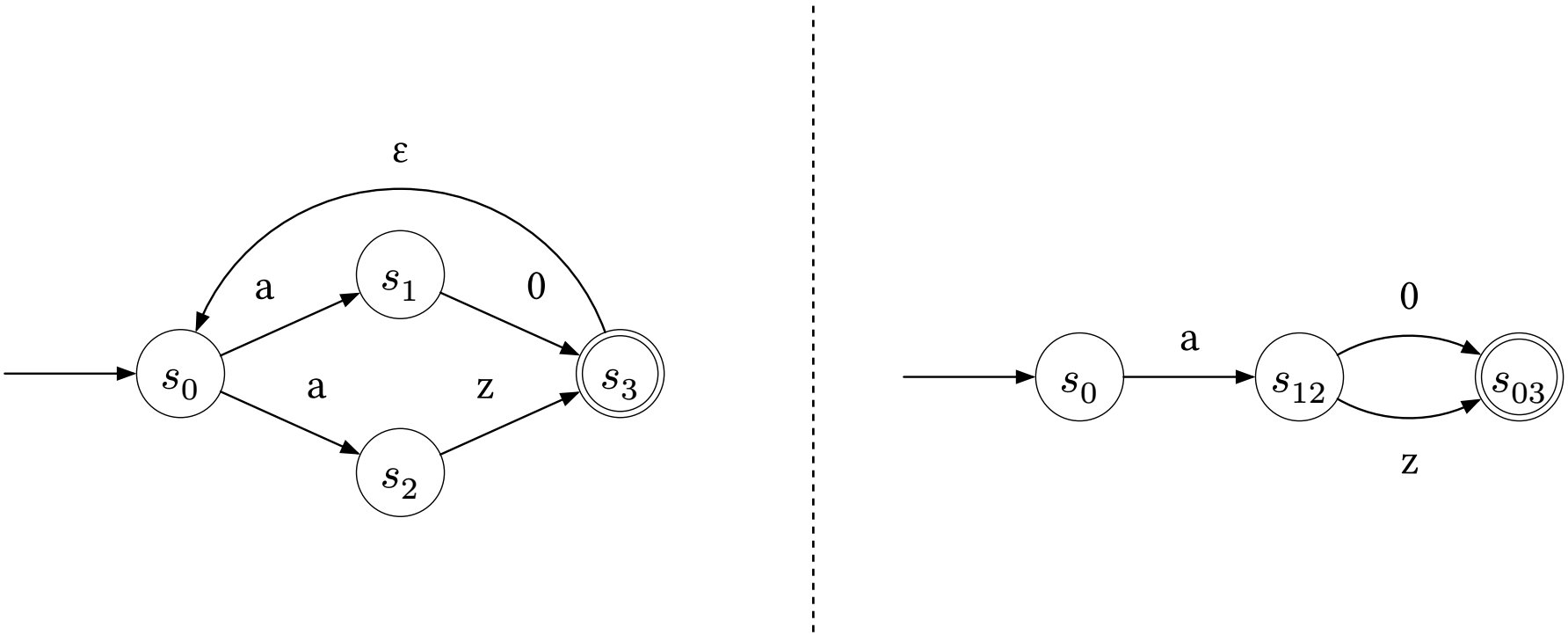

Figure 8: Conversion of an NFA into a DFA.

Removing non-determinism

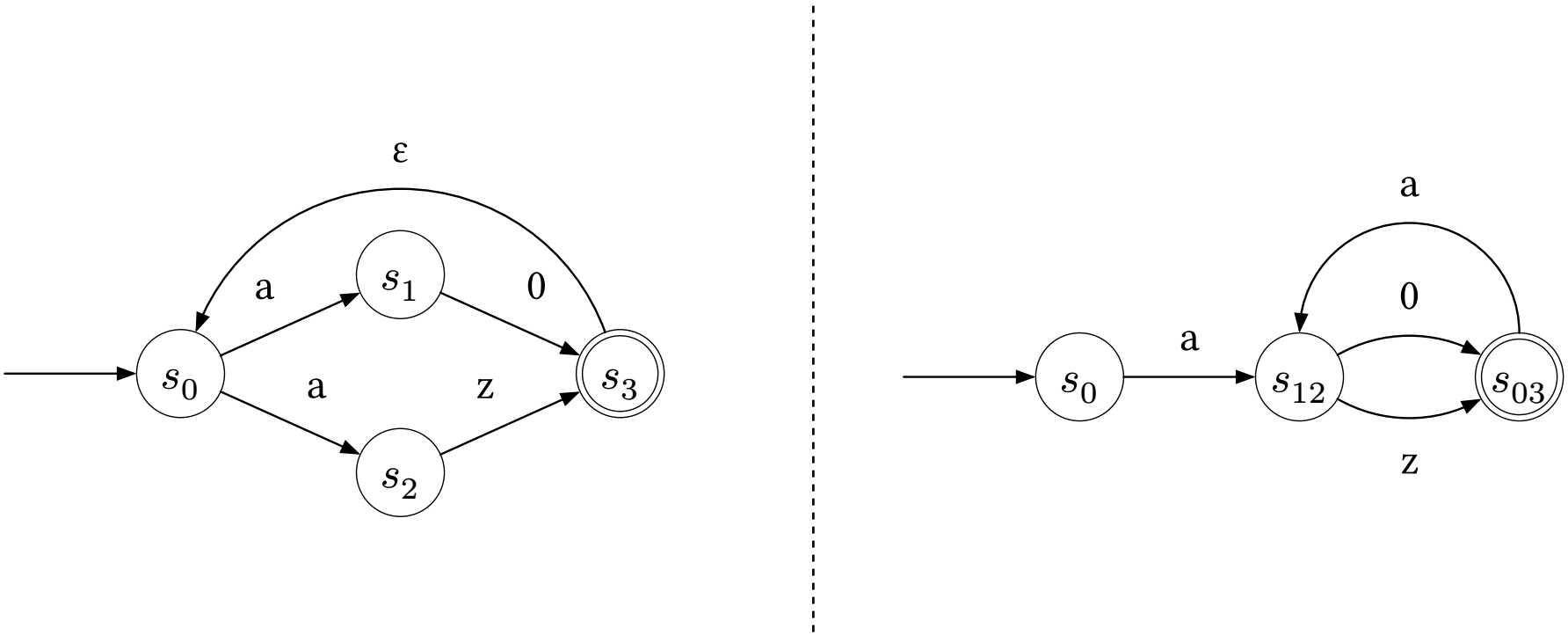

Figure 8: Conversion of an NFA into a DFA.

Simulating DFAs

There is only one possible transition we can take from each state

This means there is only one state we can be in at a time

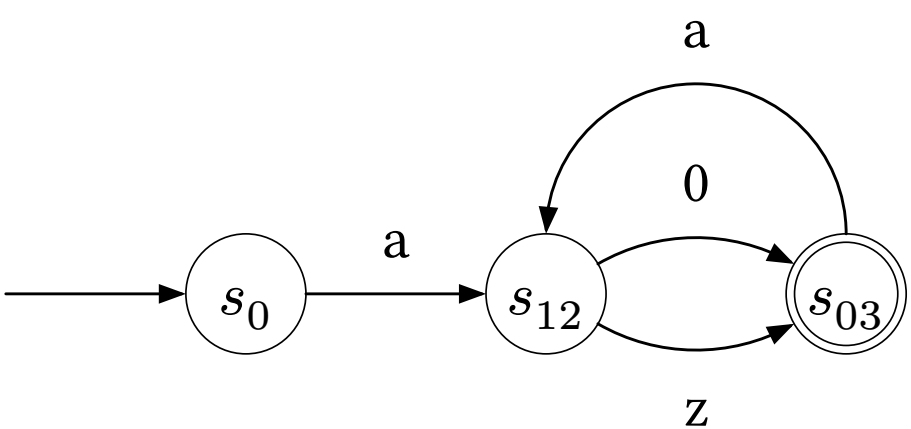

Figure 9: The DFA converted from our previous NFA. Represents `(a[z0])+`

Simulating DFAs

We now have $\Theta(1)$ space complexity[1]

And since there is one transition per input character, we have $\Theta(\|s\|)$ time complexity, where s is the input string

[1]For simulating/executing the DFA. Representing the DFA still takes $O(n + m)$ space, where n is the number of states and m is the number of transitions.

How do we convert regular expressions to FSAs?

Atoms

Single character

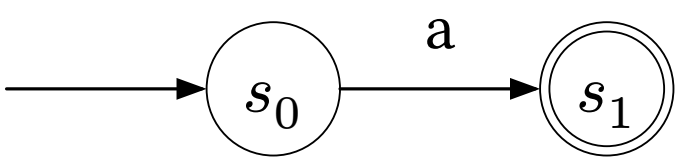

Figure 10: FSA representing a single character atom a.

Any character

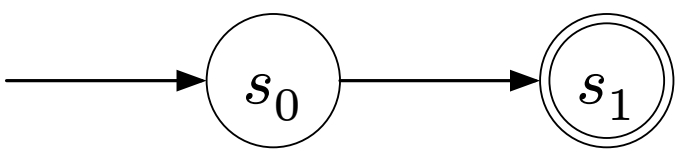

Figure 11: FSA representing the wildcard atom .. Unlabeled edge means it accepts any character.

Character ranges

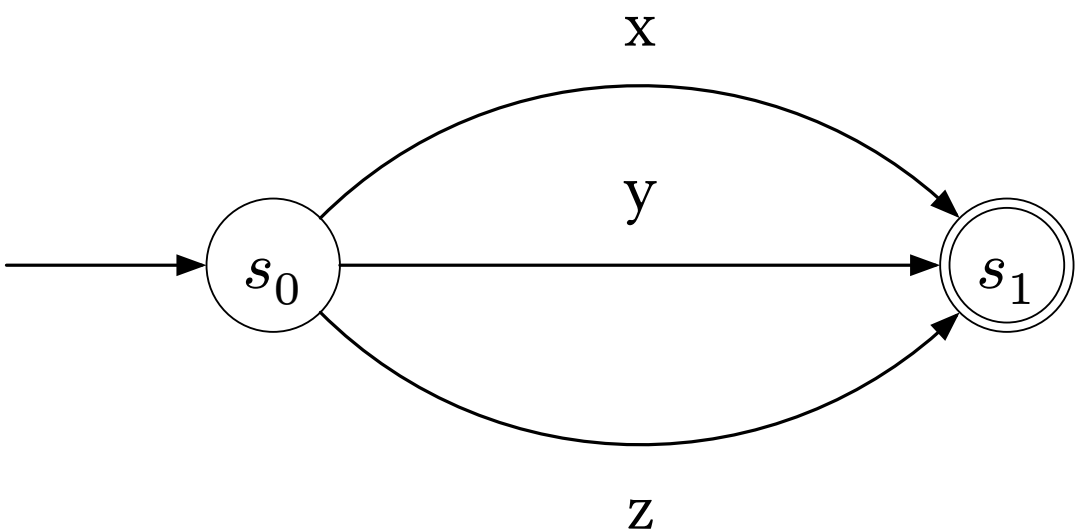

Figure 12: FSA representing the character range `[xyz]`. Unlabeled edge means it accepts any character.

Disclaimer

Typst blows up starting here so the figure numbers start jumping around

I have no idea why…

Concatenation

We simply connect the accepting state(s) of one atom to the start state of the next atom

Figure 13: Concatenation of two sub-machines.

Concatenation

We simply connect the accepting state(s) of one atom to the start state of the next atom

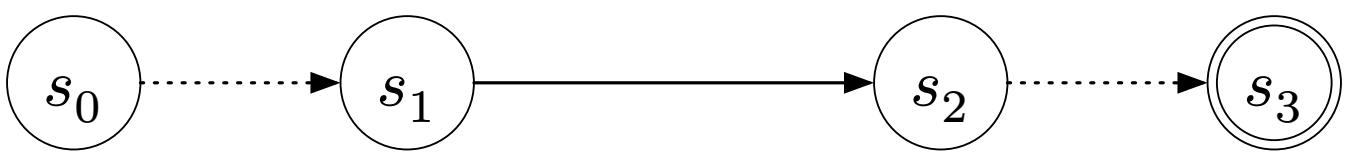

Figure 22: Concatenation of two sub-machines.

Concatenation

We simply connect the accepting state(s) of one atom to the start state of the next atom

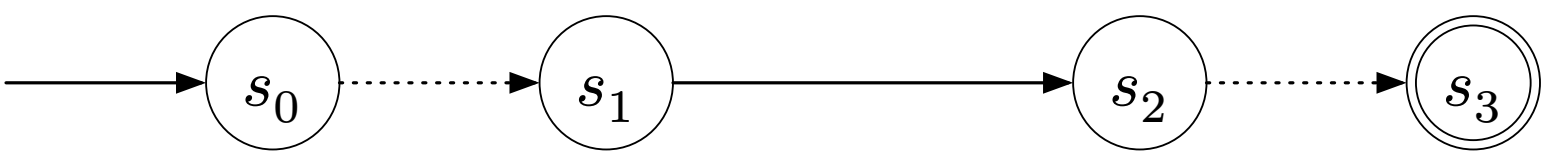

Figure 22: Concatenation of two sub-machines.

Quantifiers

? — Zero or one (method 1)

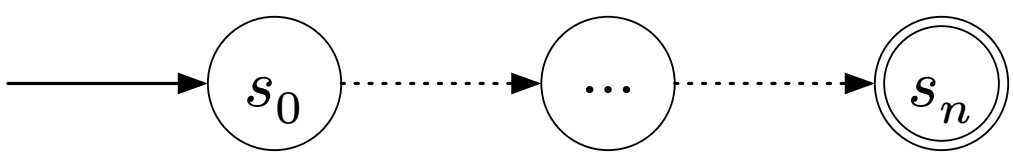

Figure 23: Zero or one operator expressed as an FSA.

Quantifiers

? — Zero or one (method 1)

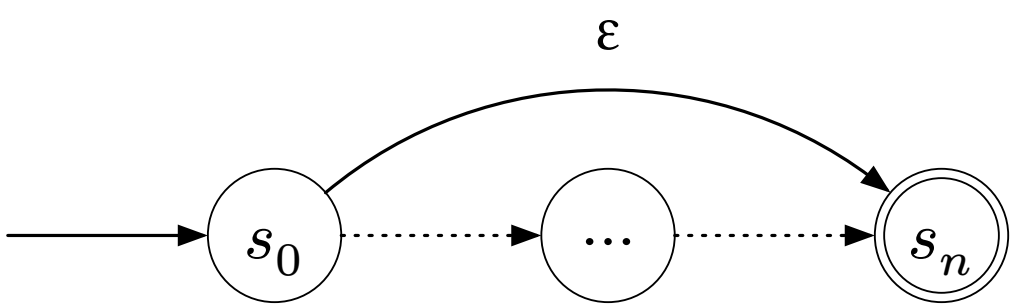

Figure 37: Zero or one operator expressed as an FSA.

Quantifiers

? — Zero or one (method 2)

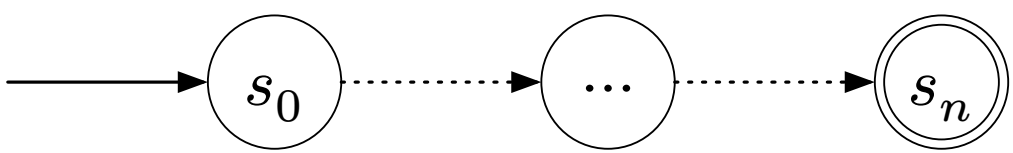

Figure 38: Zero or one operator expressed as an FSA.

Quantifiers

? — Zero or one (method 2)

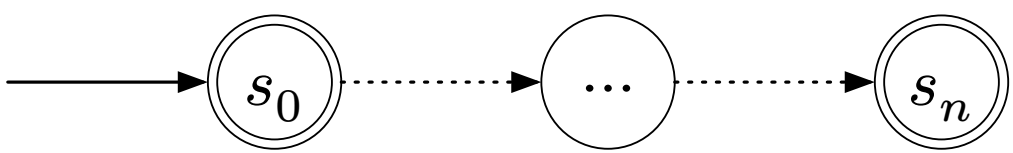

Figure 44: Zero or one operator expressed as an FSA.

Quantifiers

+ — One or more

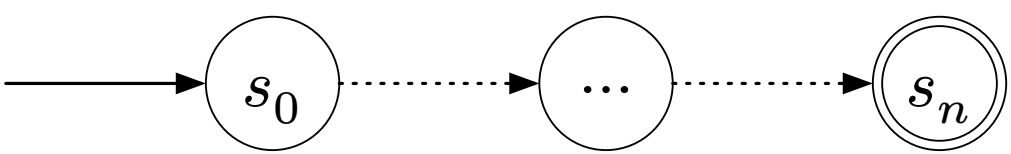

Figure 45: One or more operator expressed as an FSA.

Quantifiers

+ — One or more

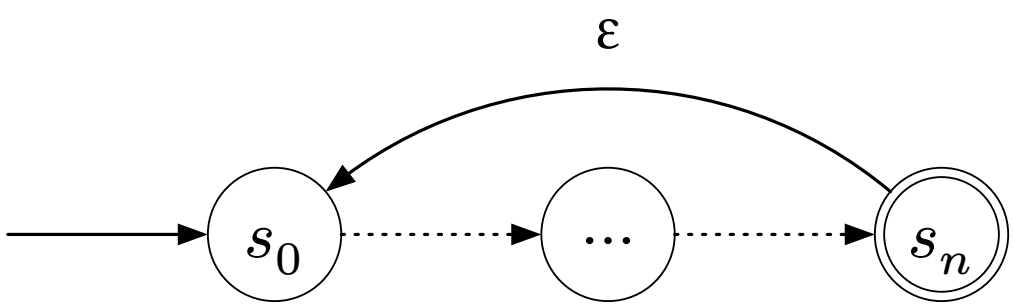

Figure 47: One or more operator expressed as an FSA.

Quantifiers

* — Zero or more

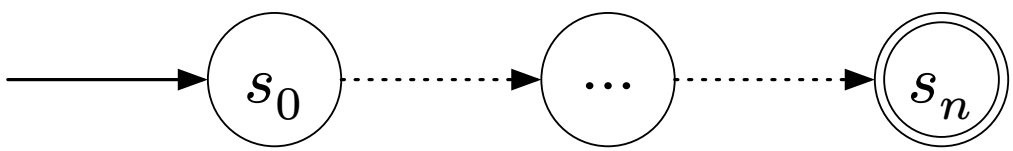

Figure 48: One or more operator expressed as an FSA.

Quantifiers

* — Zero or more

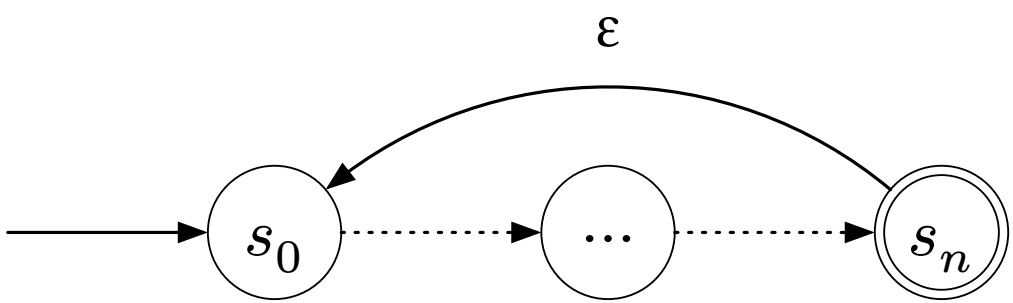

Figure 49: One or more operator expressed as an FSA.

Quantifiers

* — Zero or more

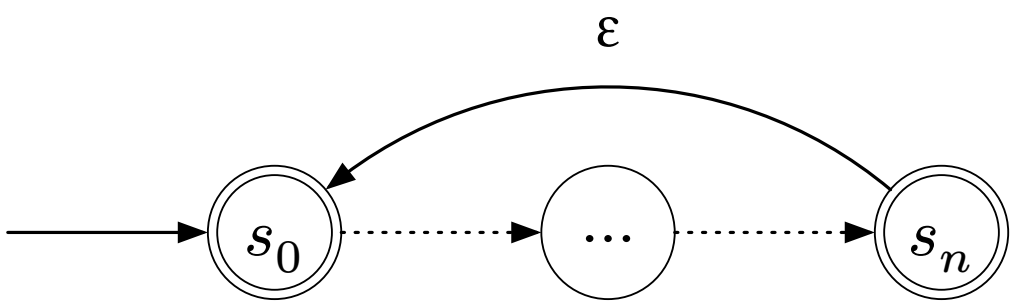

Figure 49: One or more operator expressed as an FSA.

Alternation

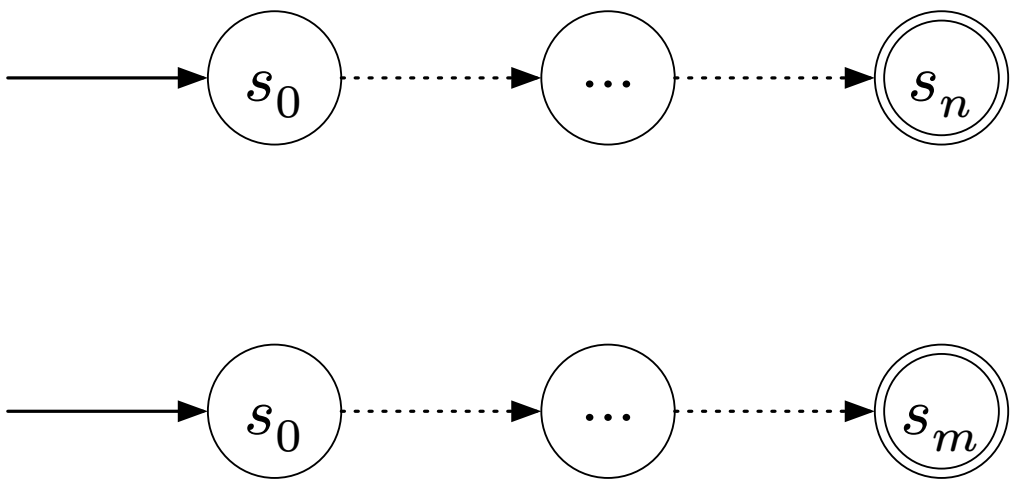

Figure 50: Alternation expressed as an FSA.

Alternation

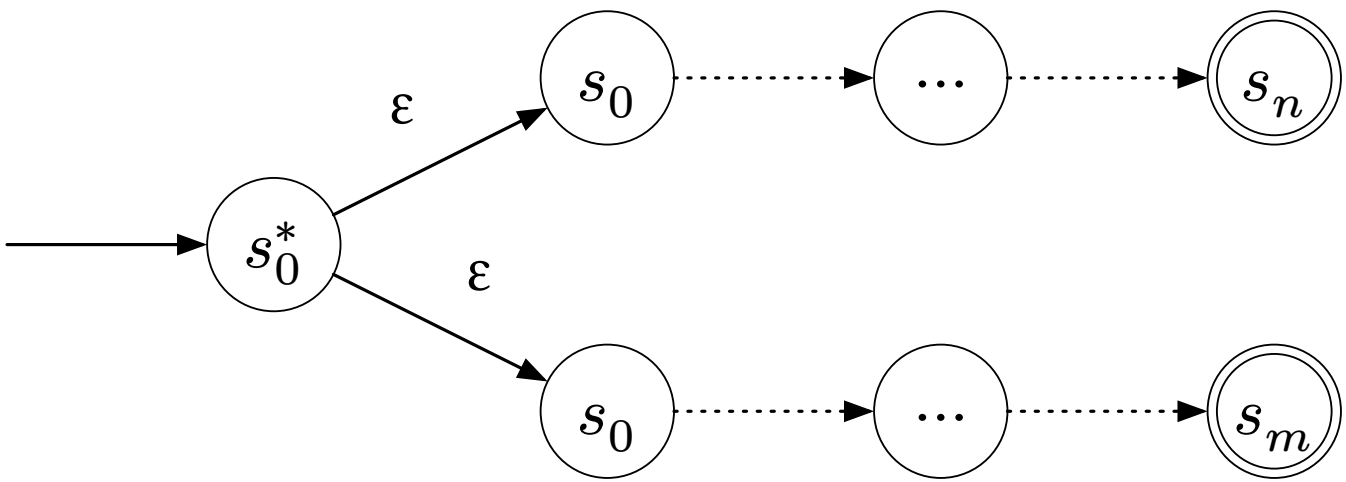

Figure 51: Alternation expressed as an FSA.

Grouping

Grouping is only needed when parsing the regular expression

The parentheses tell us which parts of the NFA under construction to apply quantifier transformations to

E.g. `(a)(b)(c)` and `(abc)` have the same resulting NFA and DFA

Code

Everything discussed so far is implemented in
https://github.com/kdkasad/regex