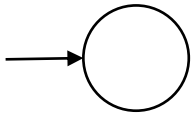
From Regular Languages to Finite Automata

Regular Languages

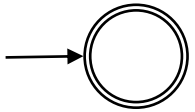
- A language (set of strings) is *regular* if:
  1. It is the empty set (contains no strings).
  2. It is the set containing only the empty string (contains one string).
     - We've been denoting the empty string as "", but another common notation is ε (epsilon).
  3. It is a set containing one string which consists of a single character.
  4. It is the *union* of two regular languages.
  5. It is the *concatenation* of two regular languages.
  6. It is the *Kleene star* of a regular language.
- We are going to show (informally) that *every regular language can be recognized by a DFA.*

From Regular Languages to NFAs

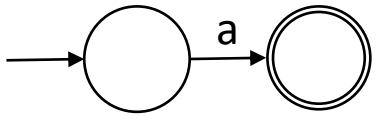
- Let's try to create an NFA for each of the 6 cases in the definition of a regular language.
- The first 3 cases are easy and don't even need knowledge of NFAs.
  1. It is the empty set (contains no strings).



  2. It is the set containing only the empty string (contains one string).



  3. It is a set containing one string which consists of a single character.



From Regular Languages to NFAs

- Let's try to create an NFA for each of the 6 cases in the definition of a regular language.
- The hard part is the "recursive" cases:
    4. It is the *union* of two regular languages.
    5. It is the *concatenation* of two regular languages.
    6. It is the *Kleene star* of two regular languages.
- If we only knew DFAs, it would be difficult to proceed.
- Even with NFAs it might not be obvious what to do.
- The trick is that *ε-transitions* make things easy.

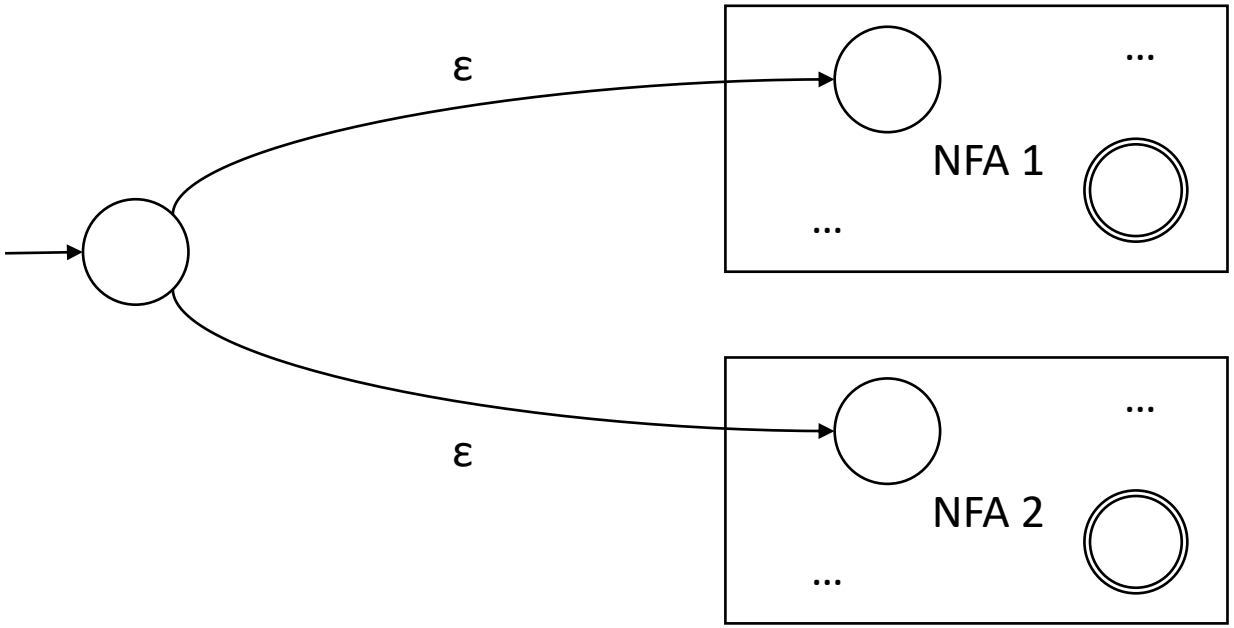
Union of Regular Languages via NFAs

- Suppose we have two NFAs representing regular languages.

- We have represented them in an abstract way above. (They do not actually just have two disconnected states.)
- How would we create an NFA for the *union* of the two regular languages by using ε-transitions?

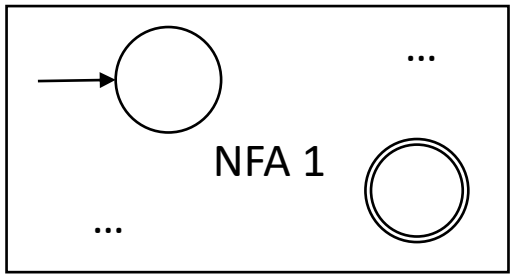
Union of Regular Languages via NFAs

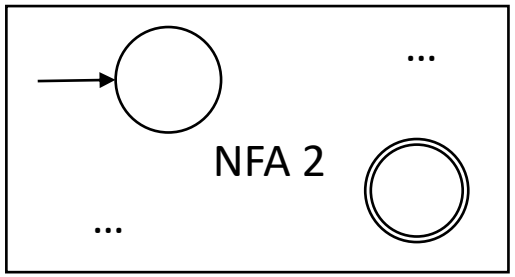
- Add a new starting state, and connect it to the old starting states using ε-transitions.



Concatenation of Regular Languages via NFAs

- Suppose we have two NFAs representing regular languages.

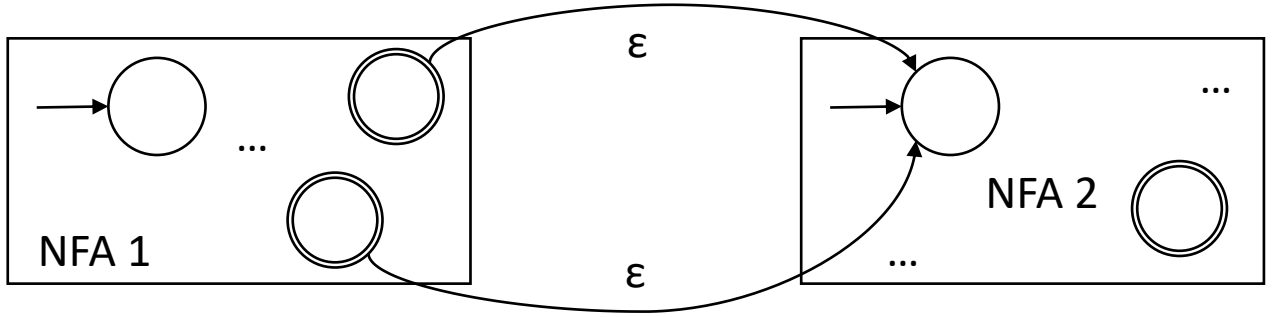




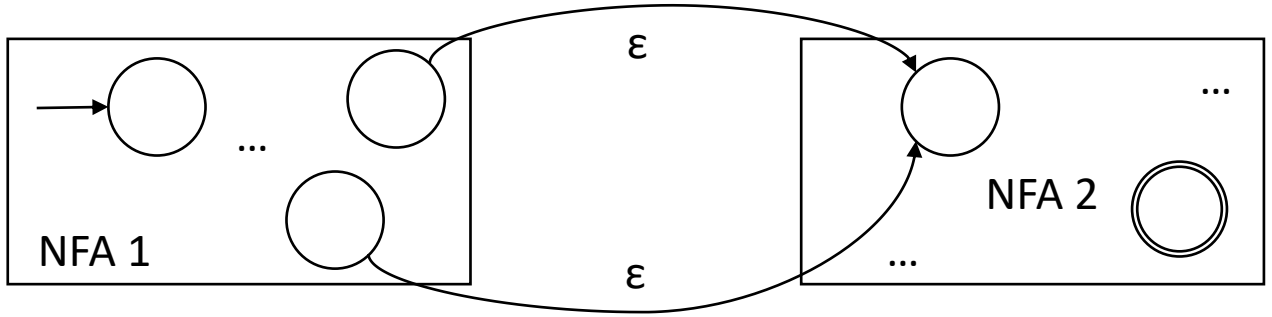
- How would we create an NFA for the *concatenation* of the two regular languages by using ε-transitions?

Concatenation of Regular Languages via NFAs

- Step 1: Connect **all** accepting states of NFA 1 to initial state of NFA 2 with ε-transitions.

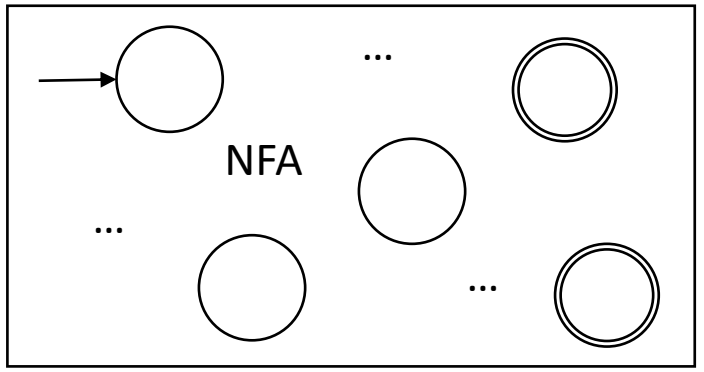


- Step 2: Make the accepting states of NFA 1 non-accepting, and change the initial state of NFA 2 into a normal state.



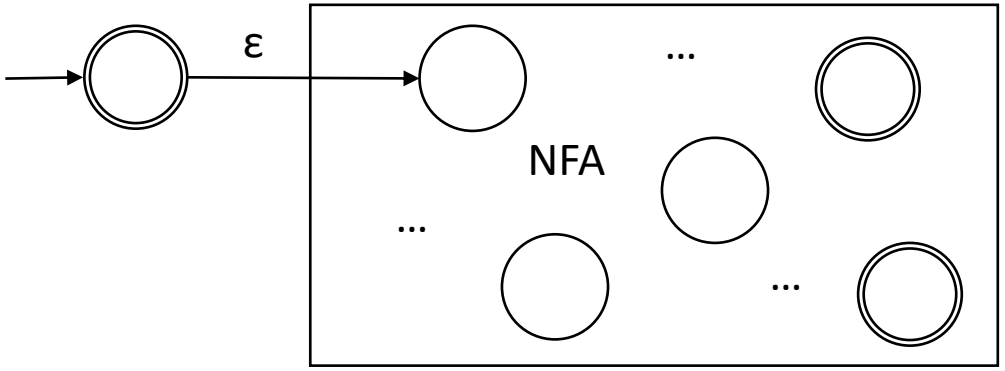
Kleene Star of Regular Languages via NFAs

- How do we take an NFA for a regular language and create an NFA for the Kleene star of the same language?
- Recall: The Kleene star of a language consists of all strings formed by concatenating *zero or more* strings from that language.
- It always includes the empty string (formed by "concatenating zero strings").

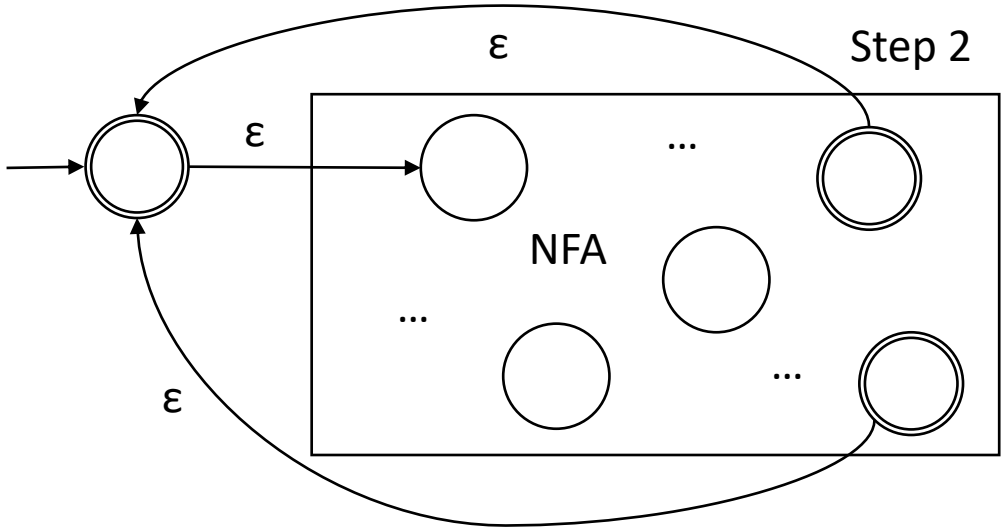


Kleene Star of Regular Languages via NFAs

- Step 1: Add a new initial state that is also accepting, to accept the empty string. Connect it to the old initial state via an ε-transition.



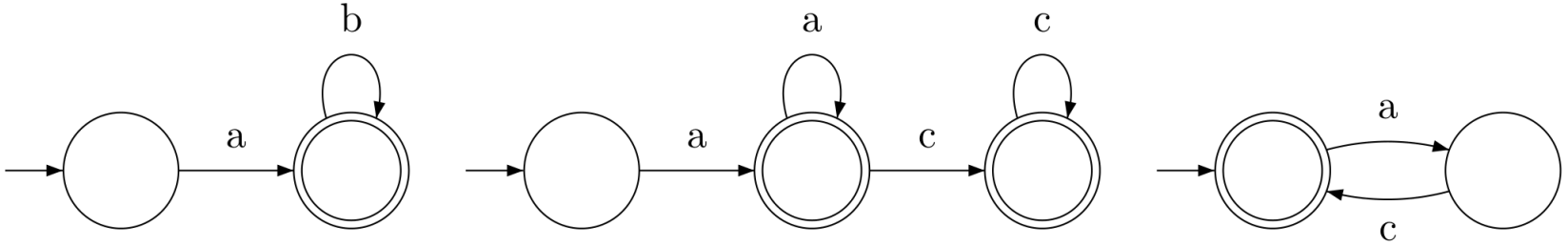
Step 1



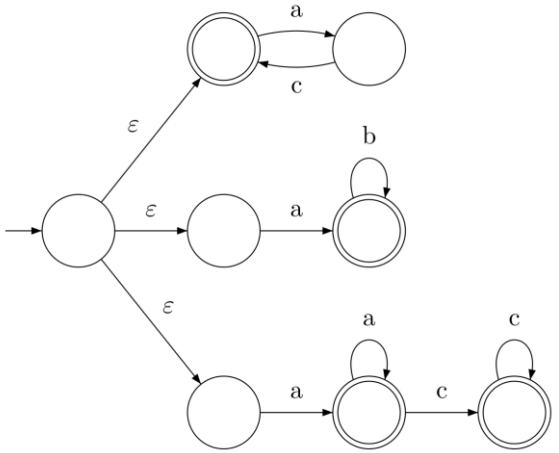
- Step 2: Connect all the original accepting states back to the new initial state with ε-transitions.

Examples

- Construct an NFA for the regular expression ab*|aa*c*|(ac)*.
- We create DFAs for each of the smaller expressions:



- Then use ε-transitions:

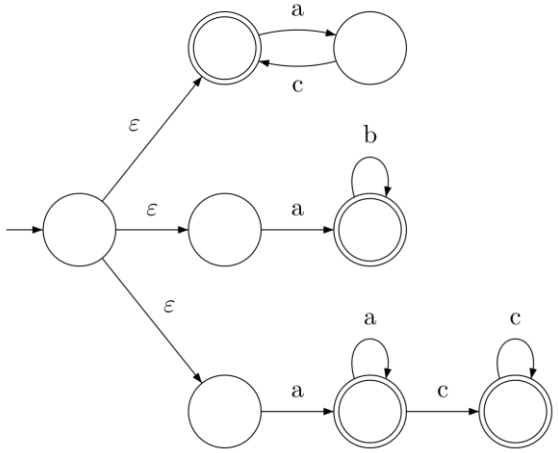


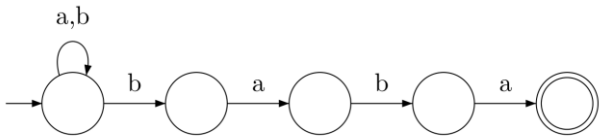
Examples

- Construct an NFA for the regular expression:

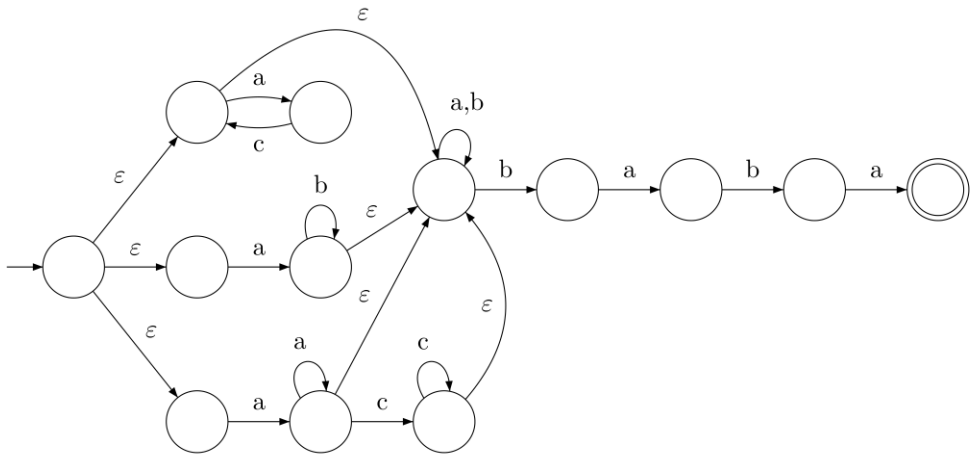
$$(ab^*|aa^*c^*|(ac)^*)(a|b)^*baba$$

- This is a concatenation of two NFAs we have already seen:





- Using the ε-transition construction:

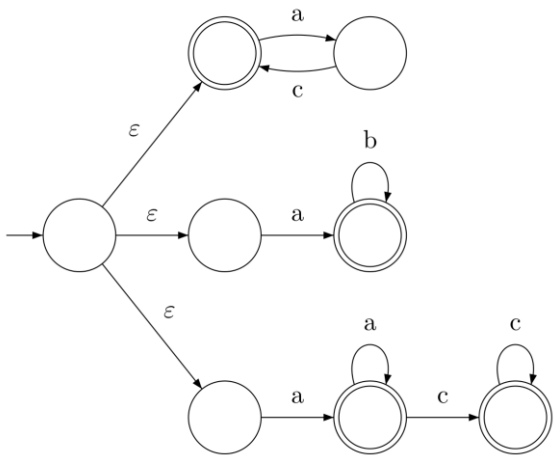


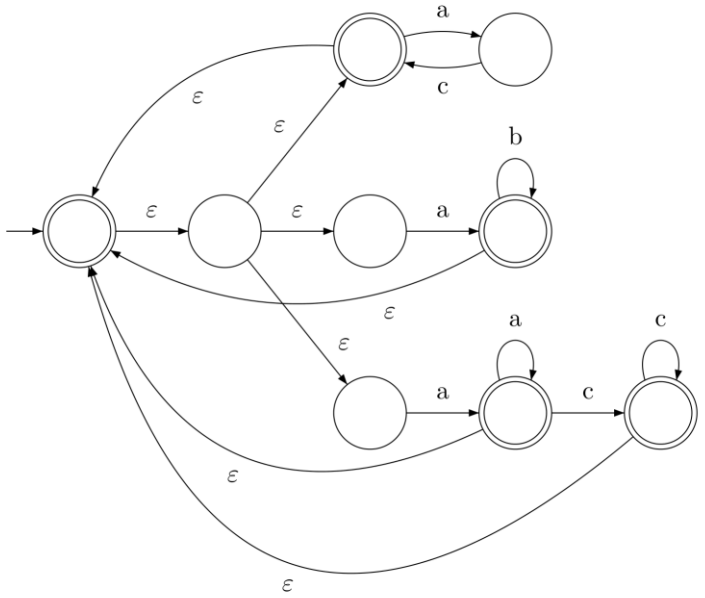
Examples

- Construct an NFA for the regular expression:

$$(ab^*|aa^*c^*|(ac)^*)^*$$

- Using the ε-transition construction for star:



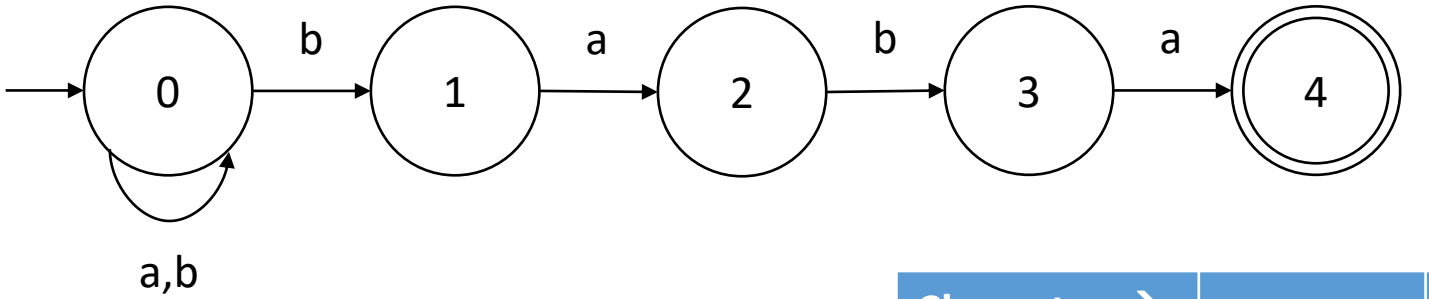


From NFAs to DFAs

- We've now seen that every regular expression can be represented by a NFA (or at least seen the idea behind the proof).
- If we can show every language recognized by an NFA can be recognized by a DFA, this shows every regular expression can be represented by a DFA (our ultimate goal!)
- Recall that we traced through NFAs by thinking about the possible "set of NFA states" we can be in.
- There are *finitely many* possible sets!
- For each NFA, we can create a DFA that simulates it, where the **DFA states** will be **sets of NFA states**.

NFA Transition Tables

- Here is the NFA for (a|b)*baba (with the states renamed to numbers).

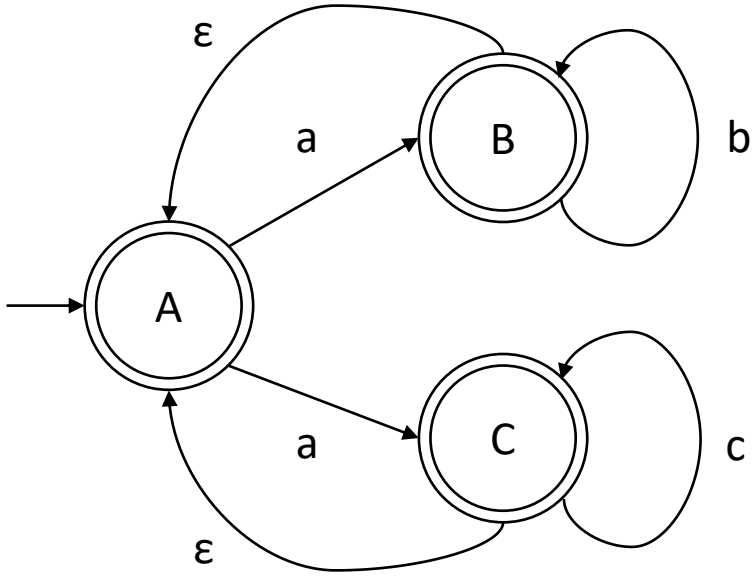


- A convenient way to represent the transitions (arrows) is with a *lookup table*, where the rows are states, the columns are characters, and the table entries are *sets of states*.

Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

NFA Transition Tables

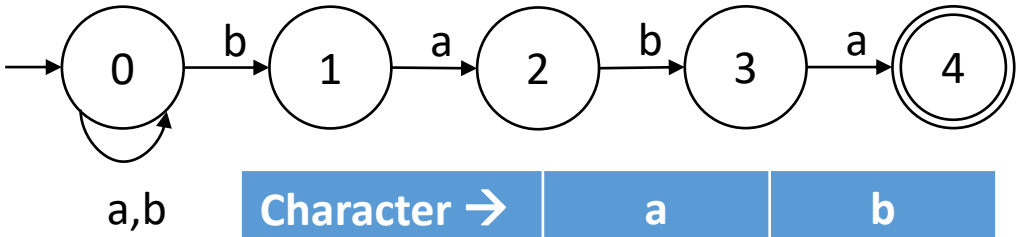
- Here's an example with ε-transitions (an extra column is added for ε).



Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

The Subset Construction

- Let's use this to create a new table. The rows will be *sets of states* and we start out with a set just containing the initial state.

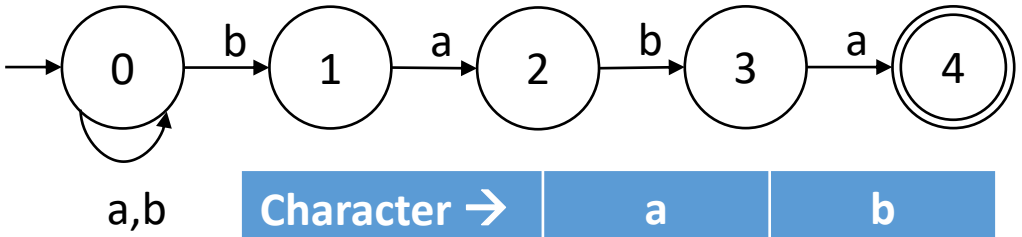


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}		

The Subset Construction

- We use the NFA table to fill in the "set table".

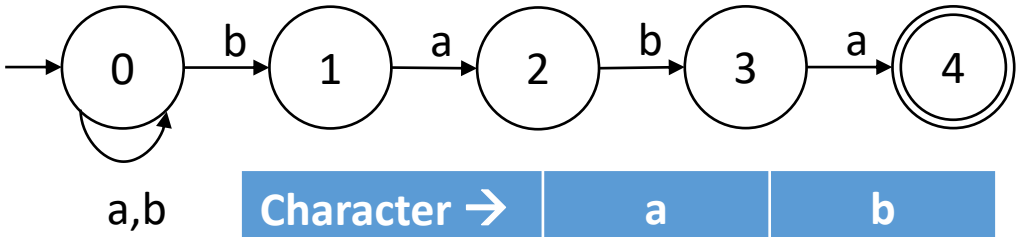


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}

The Subset Construction

- Once all rows are filled, we add a new row for **each new set we reached** that isn't already in the table.

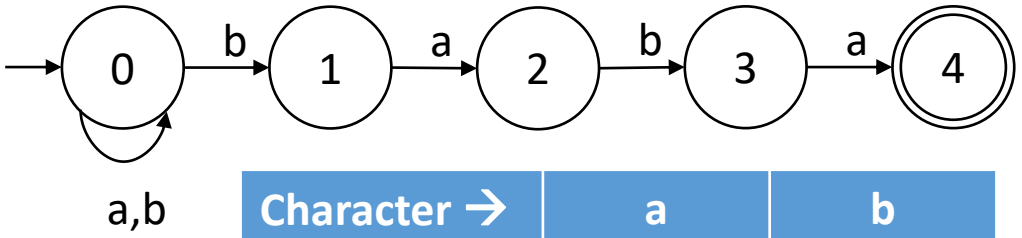


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}		

The Subset Construction

- For rows that contain a *set of NFA states*, we look at the rows of the NFA table for *each state in the set* and *take the union*.

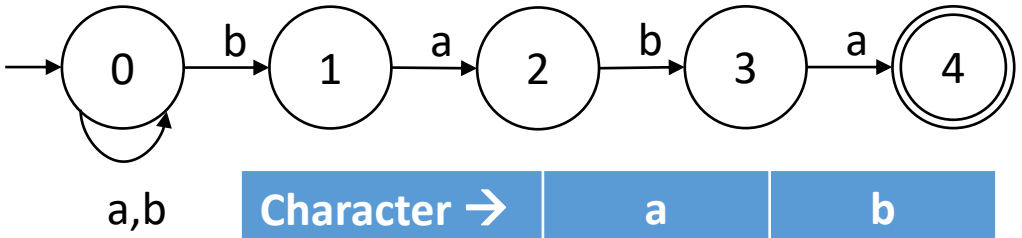


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	

The Subset Construction

- For rows that contain a *set of NFA states*, we look at the rows of the NFA table for *each state in the set* and *take the union*.

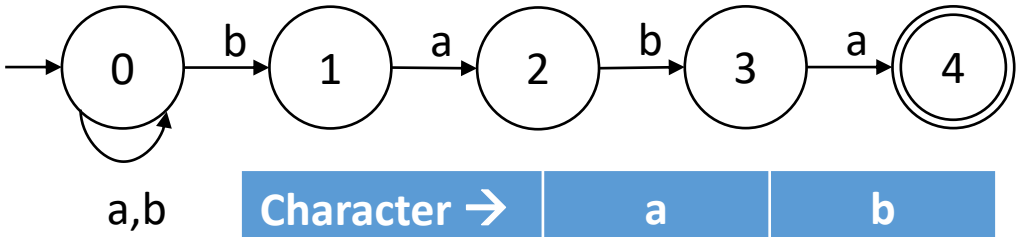


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	{0, 1}

The Subset Construction

- We continue until we stop finding new rows.

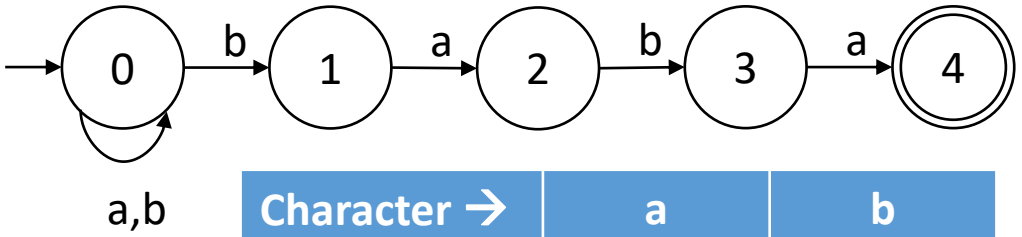


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	{0, 1}
{0, 2}		

The Subset Construction

- We continue until we stop finding new rows.

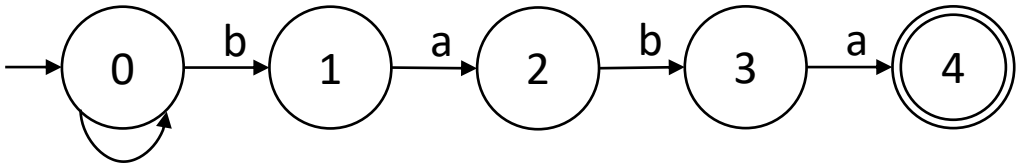


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	{0, 1}
{0, 2}	{0}	{0, 1, 3}

The Subset Construction

- We continue until we stop finding new rows.

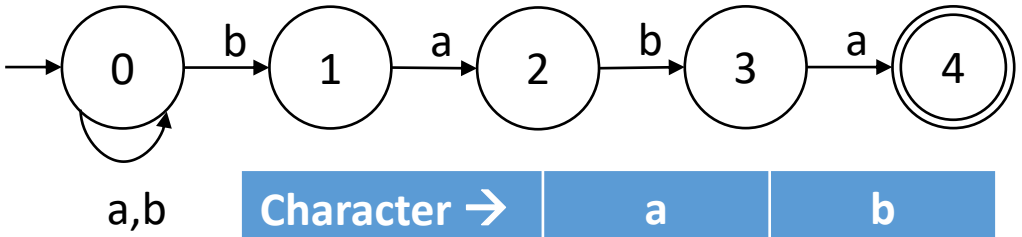


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	{0, 1}
{0, 2}	{0}	{0, 1, 3}
{0, 1, 3}	{0, 2, 4}	{0, 1}

The Subset Construction

- We continue until we stop finding new rows.

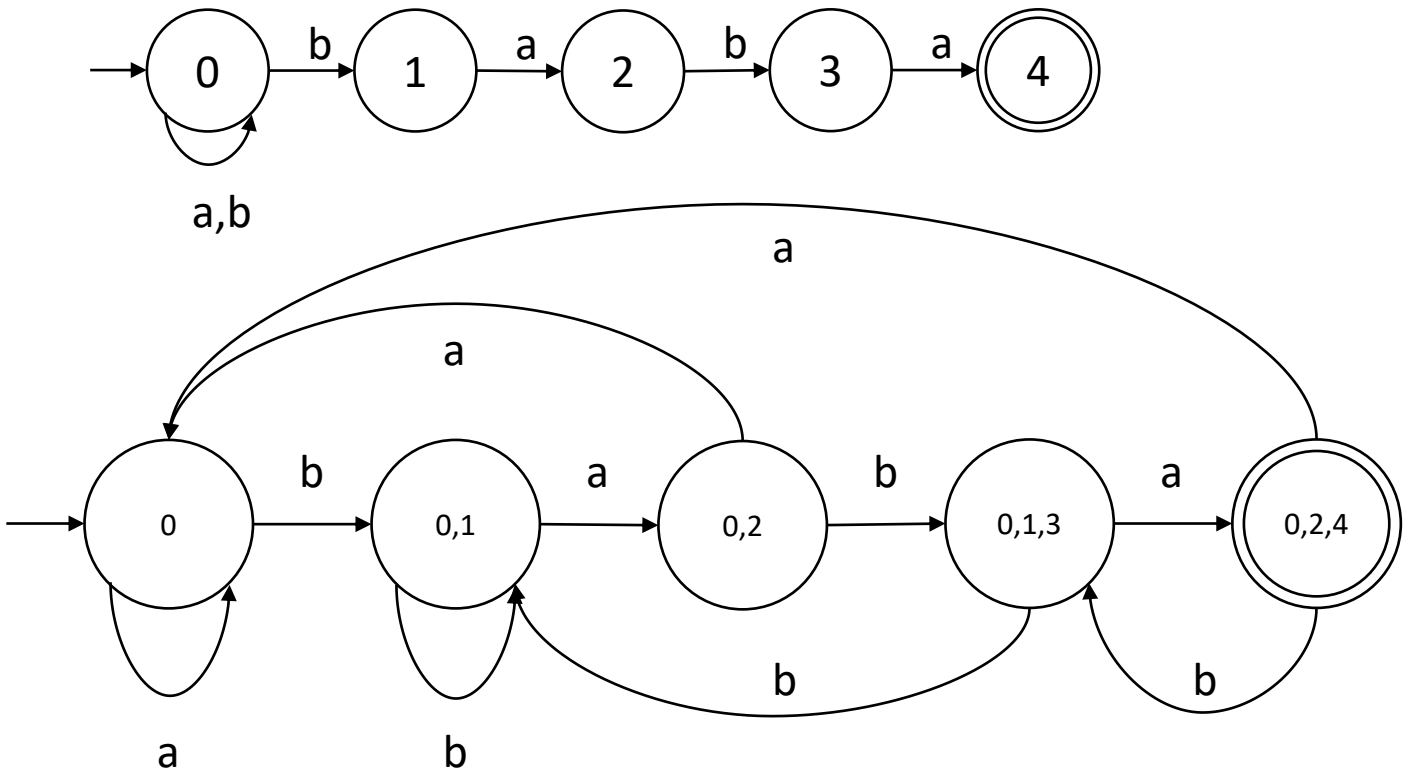


Character → State ↓	a	b
0	{0}	{0, 1}
1	{2}	{ }
2	{ }	{3}
3	{4}	{ }
4	{ }	{ }

Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	{0, 1}
{0, 2}	{0}	{0, 1, 3}
{0, 1, 3}	{0, 2, 4}	{0, 1}
{0, 2, 4}	{0}	{0, 1, 3}

The Subset Construction

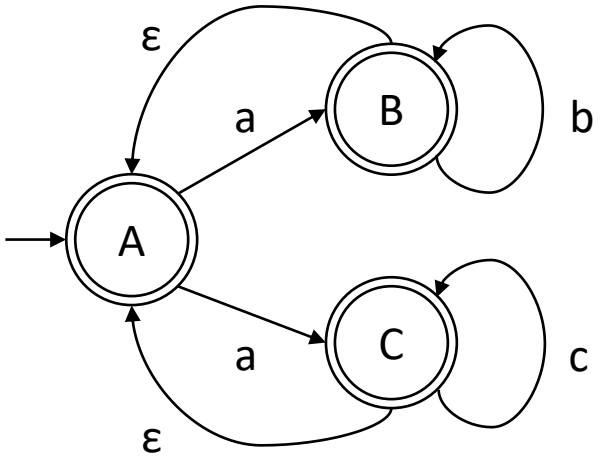
- Now we turn the new table into a **DFA** with **one state for each row**. Accepting DFA states are sets which contain an accepting NFA state.



Character → State ↓	a	b
{0}	{0}	{0, 1}
{0, 1}	{0, 2}	{0, 1}
{0, 2}	{0}	{0, 1, 3}
{0, 1, 3}	{0, 2, 4}	{0, 1}
{0, 2, 4}	{0}	{0, 1, 3}

The Subset Construction with ε-transitions

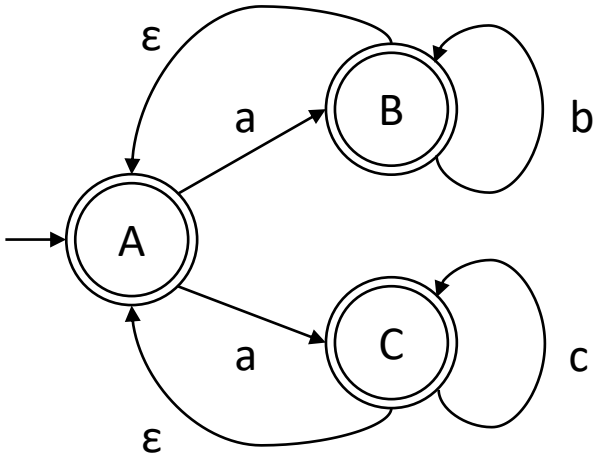
- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



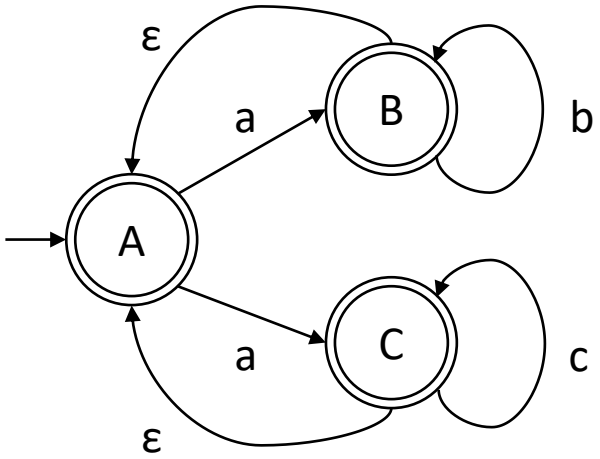
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c

First row is ε-closure({A})

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



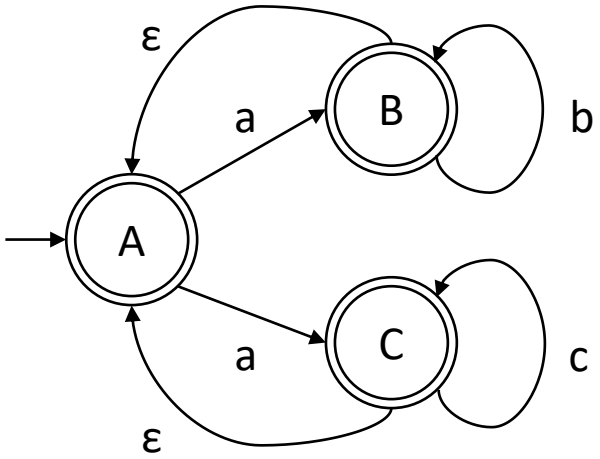
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}			

ε-closure({A}) = {A} ∪ { } = {A}

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



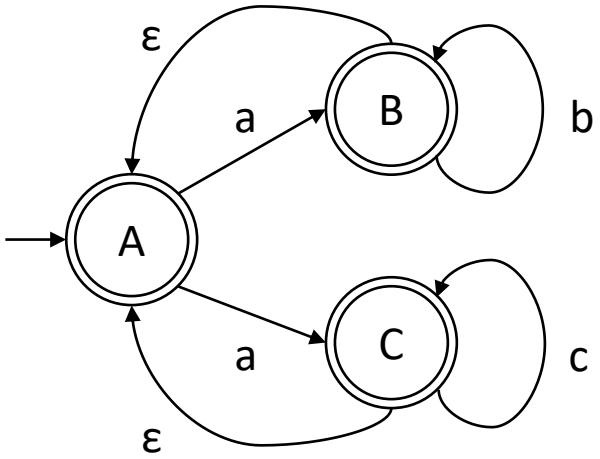
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{B,C}		

Is this correct?

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



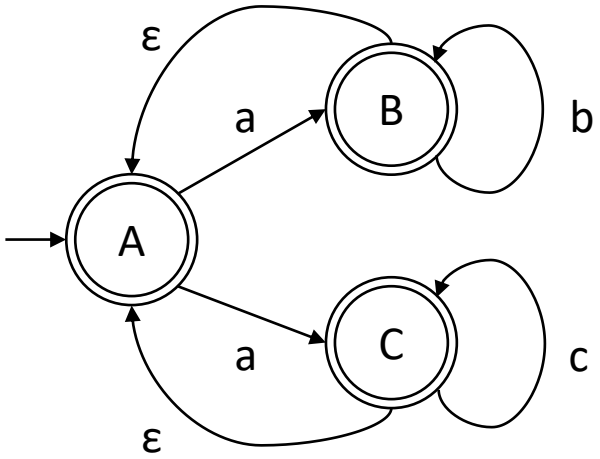
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}		

Don't forget the ε-closure!

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)

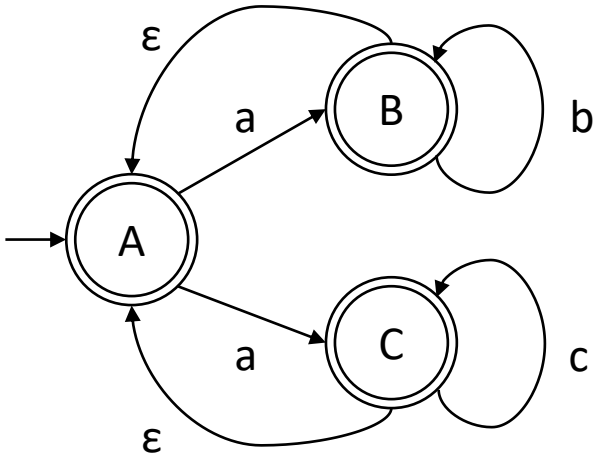


Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



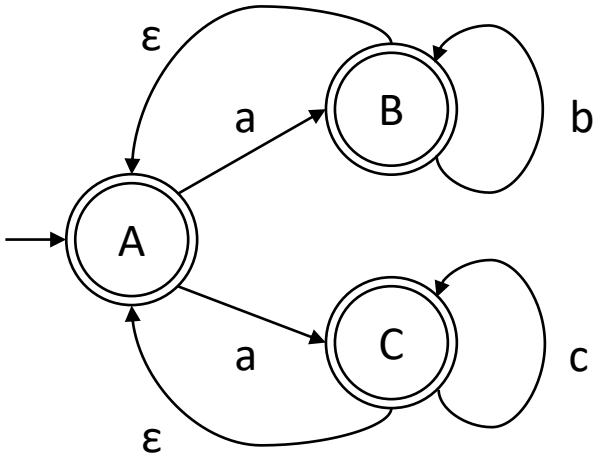
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}			
{ }	{ }	{ }	{ }

We'll count the empty set as a row too.

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



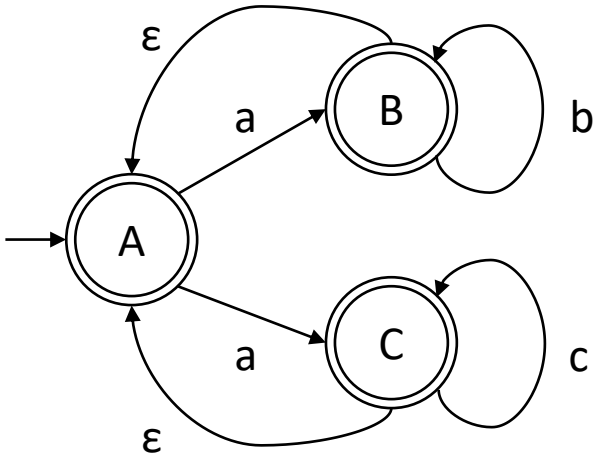
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{B,C}		
{ }	{ }	{ }	{ }

…

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



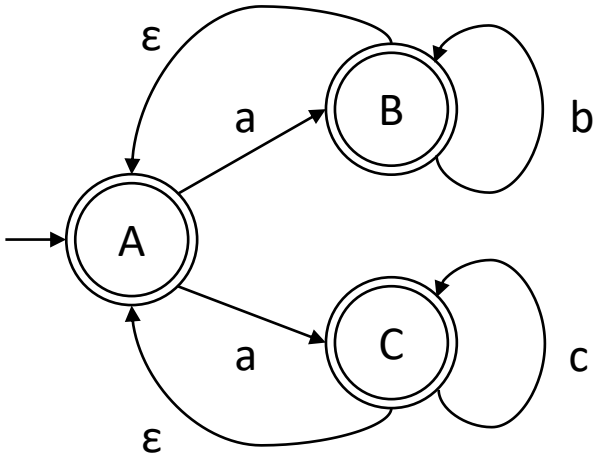
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}		
{ }	{ }	{ }	{ }

Don't forget the ε-closure.

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



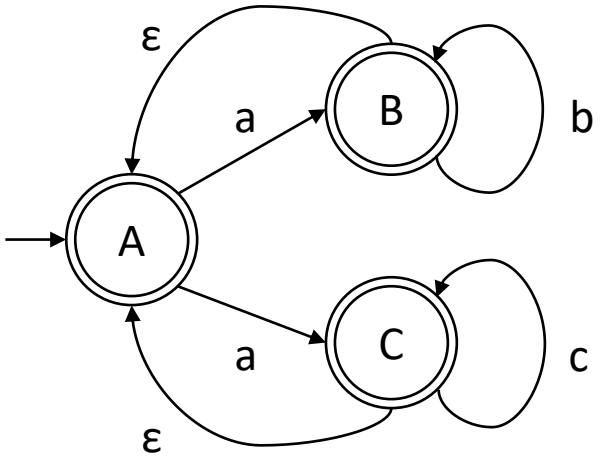
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	
{ }	{ }	{ }	{ }

Don't forget the ε-closure.

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)



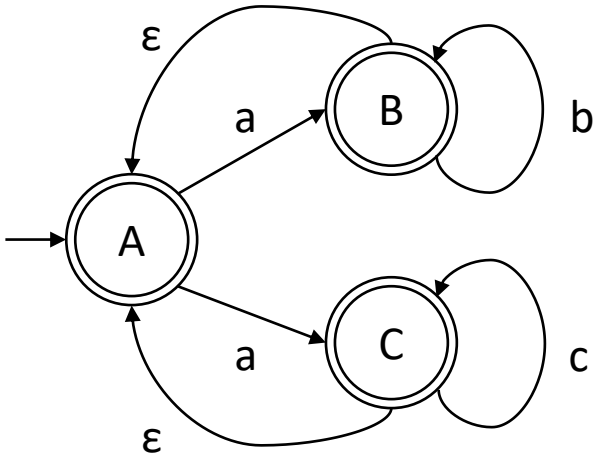
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	{A,C}
{ }	{ }	{ }	{ }

Don't forget the ε-closure.

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)

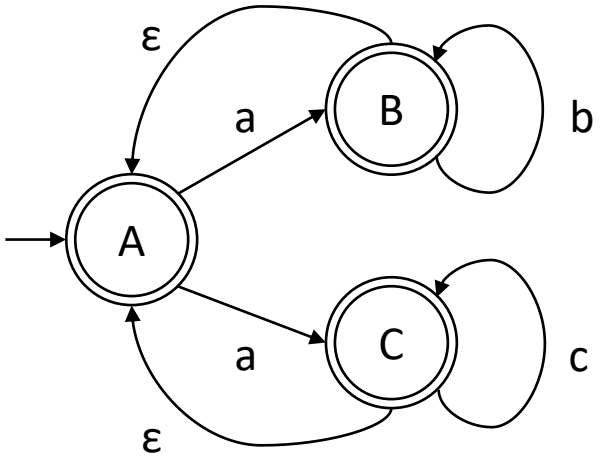


Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	{A,C}
{A,B}			
{A,C}			
{ }	{ }	{ }	{ }

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)

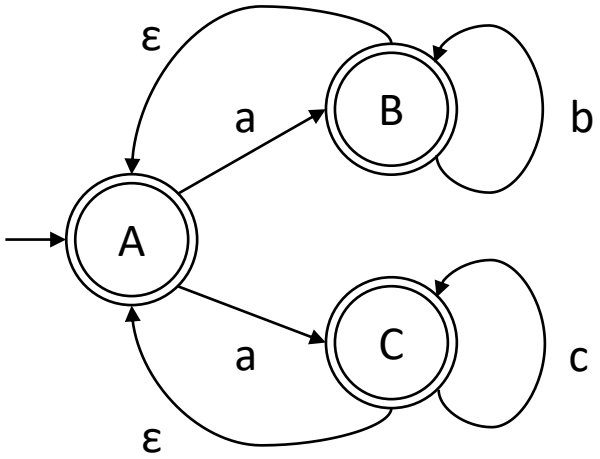


Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	{A,C}
{A,B}	{A,B,C}	{A,B}	{ }
{A,C}			
{ }	{ }	{ }	{ }

The Subset Construction with ε-transitions

- The process is similar, except for two details.
  - The DFA table does not have a column for ε.
  - *When you write a set in the DFA table, you must take the **ε-closure** of the set.* This includes the set you write in the very first row (i.e., you must start with the *ε-closure* of the set containing the initial state.)

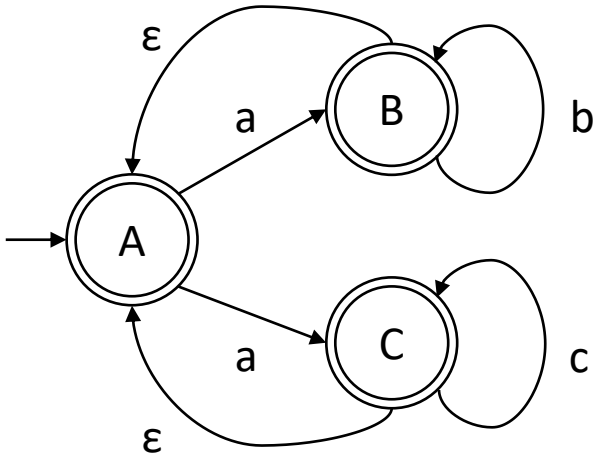


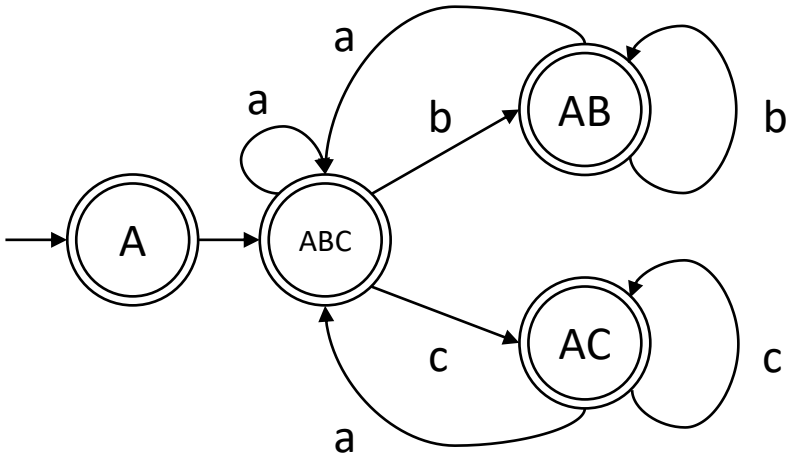
Character → State ↓	a	b	c	ε
A	{B,C}	{ }	{ }	{ }
B	{ }	{B}	{ }	{A}
C	{ }	{ }	{C}	{A}

Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	{A,C}
{A,B}	{A,B,C}	{A,B}	{ }
{A,C}	{A,B,C}	{ }	{A,C}
{ }	{ }	{ }	{ }

The Subset Construction with ε-transitions

- You don't really need to draw the state for the empty set row (just leave out the corresponding transitions), so you technically don't need to add it to the table either if you don't want to.

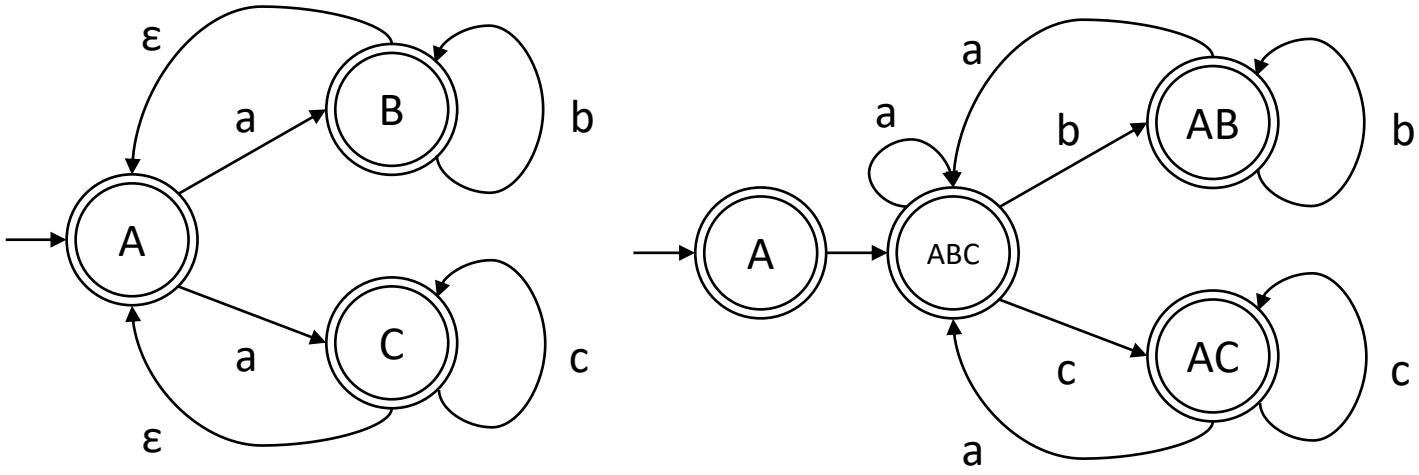




Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	{A,C}
{A,B}	{A,B,C}	{A,B}	{ }
{A,C}	{A,B,C}	{ }	{A,C}
{ }	{ }	{ }	{ }

The Subset Construction with ε-transitions

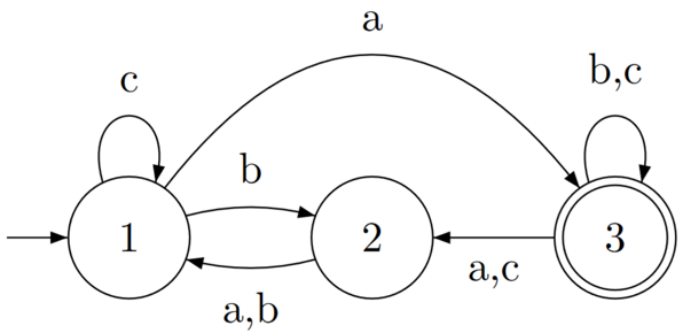
- There is an algorithm (not covered) to remove ε-transitions from an NFA. Sometimes this is less annoying than doing the subset construction directly with ε-transitions.



Character → State ↓	a	b	c
{A}	{A,B,C}	{ }	{ }
{A,B,C}	{A,B,C}	{A,B}	{A,C}
{A,B}	{A,B,C}	{A,B}	{ }
{A,C}	{A,B,C}	{ }	{A,C}
{ }	{ }	{ }	{ }

Downsides

- We can simulate any NFA using a DFA whose states represent *sets of NFA states*, so DFAs have the same "recognition power" as NFAs.
- But if the NFA has *n* states, there are 2^n distinct sets of NFA states.
- Sometimes the DFA *requires* all 2^n possible states, meaning the DFA could require exponentially more space!
- Try the subset construction yourself with this innocent-looking NFA.



Consequences

- If *every regular language can be recognized by a DFA*, then DFAs can specify any language we can specify with a regular expression.
- Regular expressions are sometimes easier or more concise, like the example of (a|b)*baba, but DFAs have the same "power".
- A DFA is essentially a simple computer with finite memory:
  - The states can be thought of as "possible states of the computer's memory".
  - Reading a character updates memory in some way.
  - Instead of a monitor showing different images based on the contents of memory, a DFA can only say whether or not the memory state is "accepting".
- All regular languages can be recognized using finite memory!

Kleene's Theorem

- It is also true that *every language recognized by a DFA is regular*.
  - This is a little unintuitive. Proved in CS 360 or CS 365.
- **Kleene's Theorem** is the combined statement that *a language is regular if and only if it is recognized by a DFA.*
- A DFA is equivalent in language recognition "power" to a computer (more formally, a Turing machine) with finite memory.
  - Sort of intuitive, but tricky to prove. Proved in CS 462.
- This leads to the fact that *a language is regular if and only if it has recognition program that uses a fixed constant amount of memory!*

Languages That Need Infinite Memory

- What kinds of languages can only be recognized with an infinite amount of memory?
- Most high-level programming languages!

```
if (condition) {
  if (condition) {
    ...
  }
}
```

- Unless there is a hardcoded limit on *nesting depth*, to track whether the curly braces { } match up, we need to count *arbitrarily high*.
  - Real computers have finite memory, so there's always a limit in practice.

Finite Memory and Nesting

- Because all real-world computers have a finite amount of memory, they can technically *only* recognize regular languages.
- But regular languages are not always the best *model* to use.
- **Example:** Write a program that reads a string of a's and b's and determines whether the number of a's and b's are equal.
  - This problem is impossible to solve. It requires infinite memory.
  - Most people would be happy with a solution that counts the a's and b's using e.g. a 32-bit int variable. But, this only solves the problem for *some* strings.
  - Also, a DFA representation of this solution would need around 2^{32} states to keep track of all the different counter values!!

Beyond Regular Languages

- Everything in the real world is *technically* regular, but some problems are cumbersome to solve with regular language methods like DFAs.
- In particular, for anything involving *nested structures*, which are extremely common in high-level programming languages, a DFA requires at least *n* states to handle *n* levels of nesting depth.
- Regular expressions have a similar problem.
- We typically want a very high limit on nesting depth that no practical program will reach, so DFAs and regular expressions are cumbersome for specifying high-level languages.
  - Once you get beyond the *scanning/tokenization* phase, at least.

Conclusions

- We've seen that DFAs and regular languages are useful for specifying **valid tokens** and for **scanning** a string into tokens.
- In our assembler, the next phase was **parsing**, which was fairly straightforward and just involved reading the tokens in sequence.
- Parsing for high-level languages, which do not have a simple line-based structure, is an extremely complicated problem, which regular languages are ill-suited for due to the presence of **nested structures**.
- Next, we'll study a new, larger class of languages called **context-free languages** that are more suitable for describing the syntax of high-level programming languages.