CMSC 330

Finite Automata

Administrative

Administrative

Quiz 2 Today

Midterm next Thursday: PL Concepts, Ruby, OCaml, NFA/DFA/CFG

Project 3 Released

Finite Automata

What Are Finite Automata?

Automata are data structures that accept/reject strings

Automata can be used to implement regular expressions

That’s what you’ll be doing in the project

Formal Definition - What does the example look like?

An automaton is a 5-tuple

Q - A finite set of states

Σ - A set of symbols, the alphabet

□ - A transition function

q_0 - The initial state

F - A set of final states

Example

Q - {0, 1, 2}

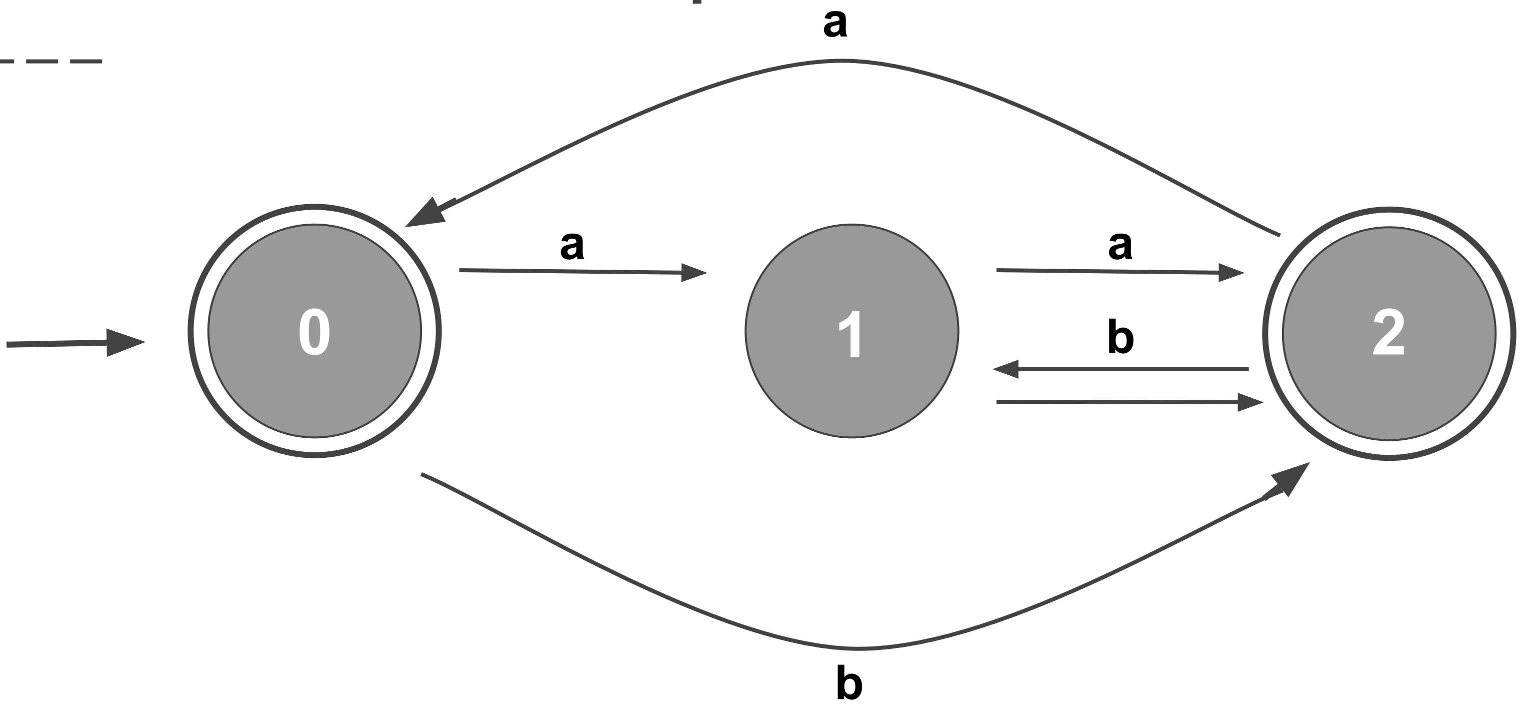
Σ - {a, b}

□ - {(0,a,1); (1,a,2); (2,a,0); (0,b,2); (1,b,2); (2,b,1)}

q_0 - 0

F - {0,2}

Formal Definition - Example



Formal Definition - Notes

Q - {0, 1, 2}

Σ - {a, b}

□ - {(0,a,1); (1,a,2); (2,a,0); (0,b,2); (1,b,2); (2,b,1)}

q_0 - 0

F - {0,2}

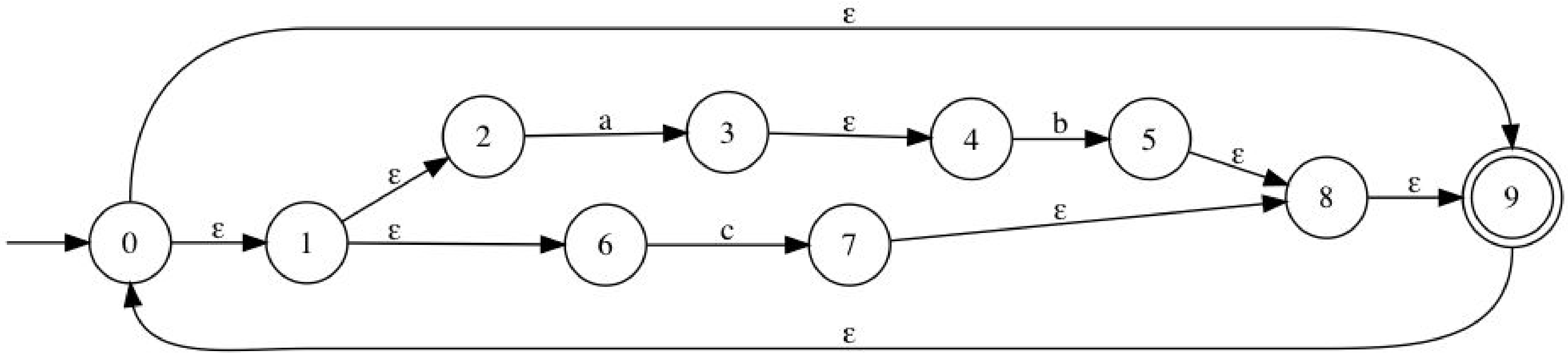
There can only be **one start state**, in this case 0, do not forget to label your start state on quizzes/exams!

There can be **multiple final states**, do not forget to label your final state(s) on quizzes/exams!

A string is **accepted** by the automata if it **ends in a final state**.

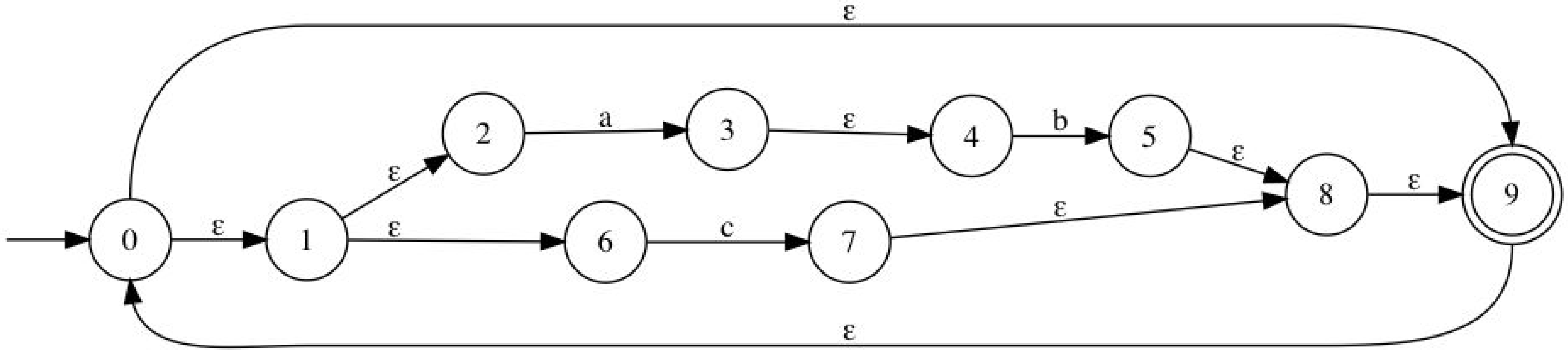
Example

Example



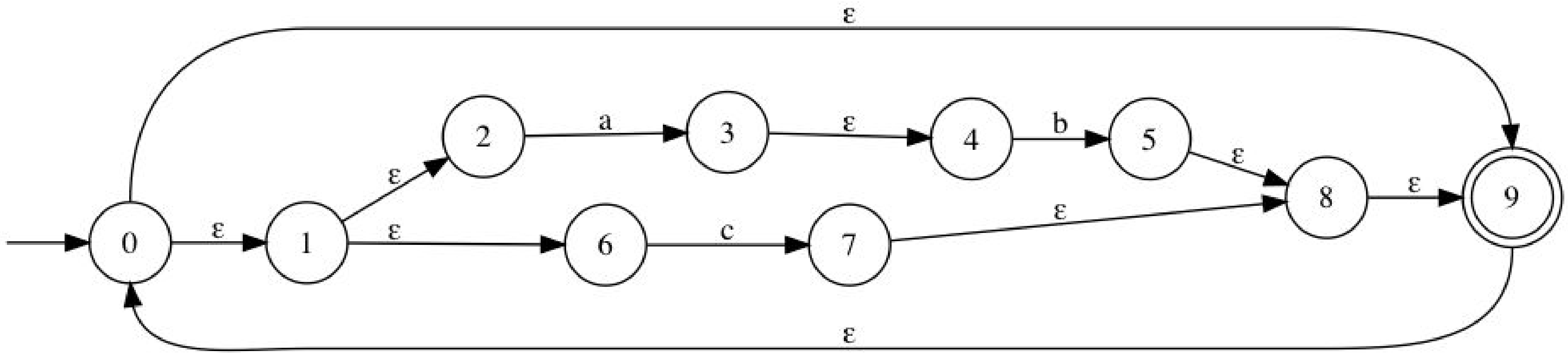
Accepted?   “” (Empty String)

Example



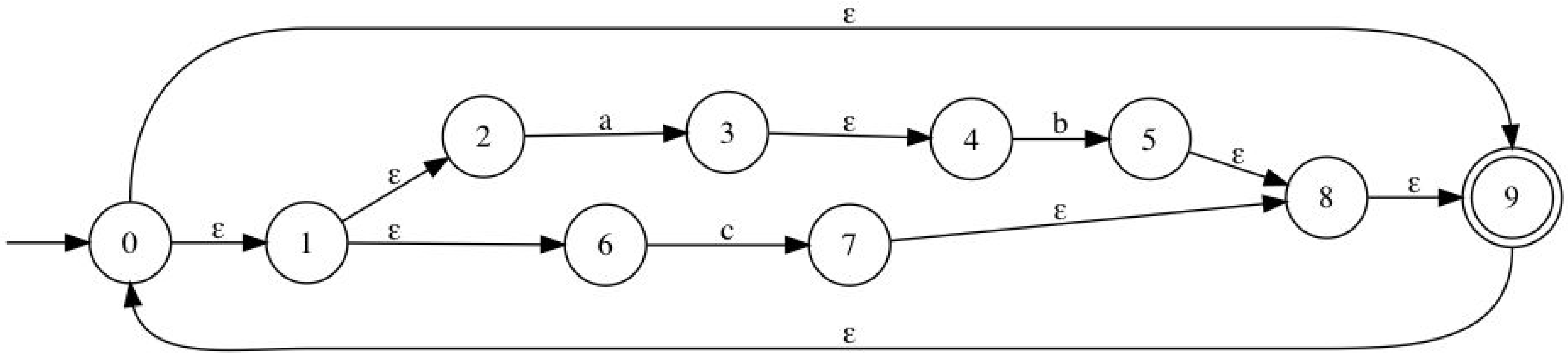
Accepted? ab

Example



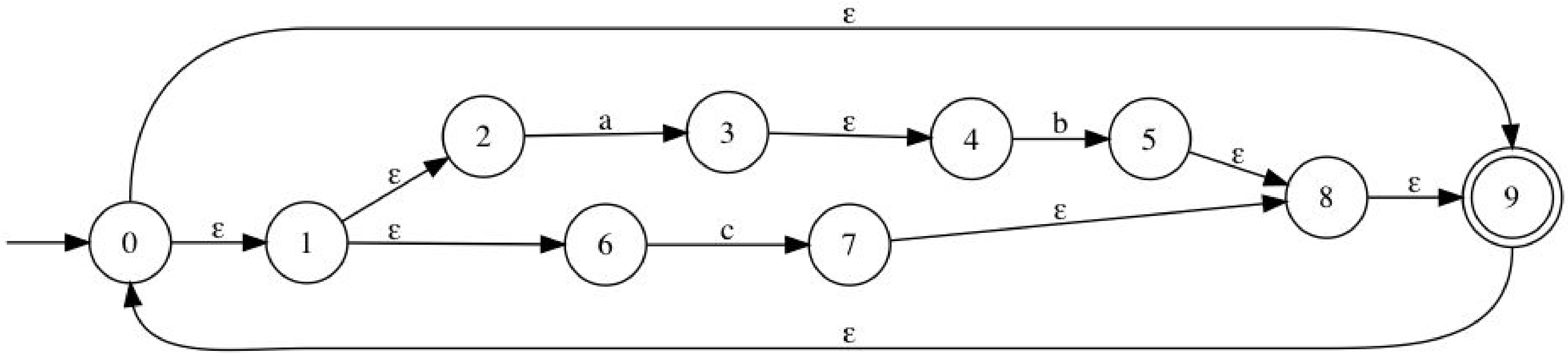
Accepted?   ababab

Example



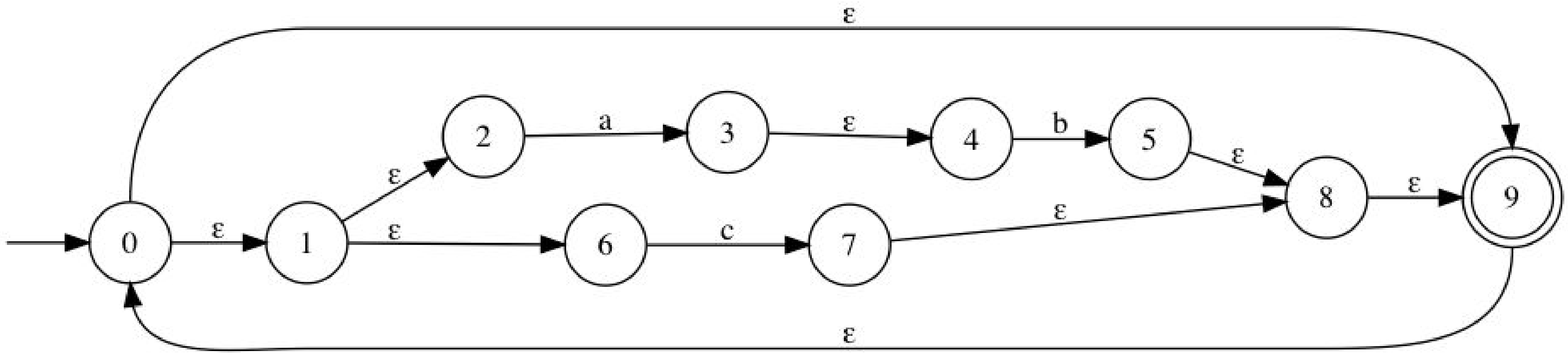
Accepted?   abc

Example



What regular expression matches this automata?

Example



What regular expression matches this automata? (ab|c)*

Types of Finite Automata

Types of Finite Automata

Deterministic Finite Automata (DFA)

- Accepts if the string ends on a final state
- A special type of NFA

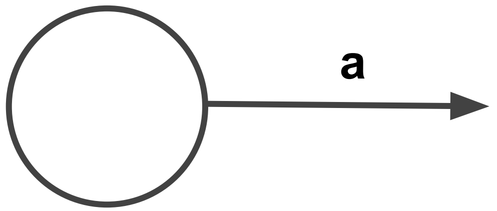
Nondeterministic Finite Automata (NFA)

- 0 or more steps for each character in the string
- Accepts if any valid path ends on a final state

Difference #1: Number of Transitions

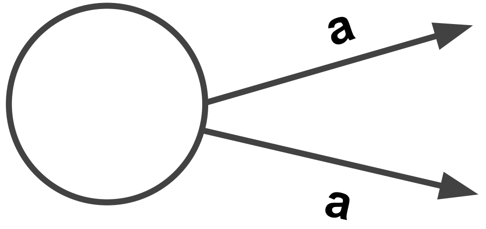
DFA

One transition per symbol



NFA

More than one transition per symbol



Difference #2: Types of Transitions

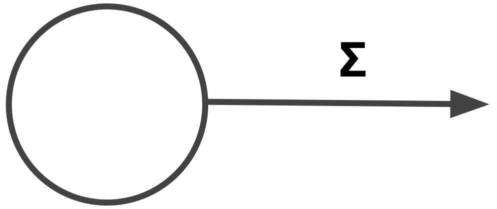
DFA

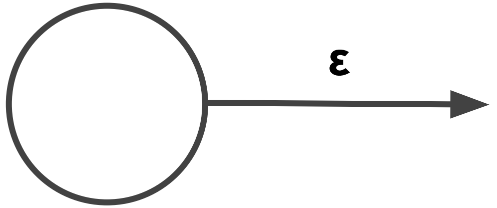
No transitions on empty string

(DFAs can match empty strings, but cannot transition on e)

NFA

May transition on empty string label





Difference #3: Accepting Strings

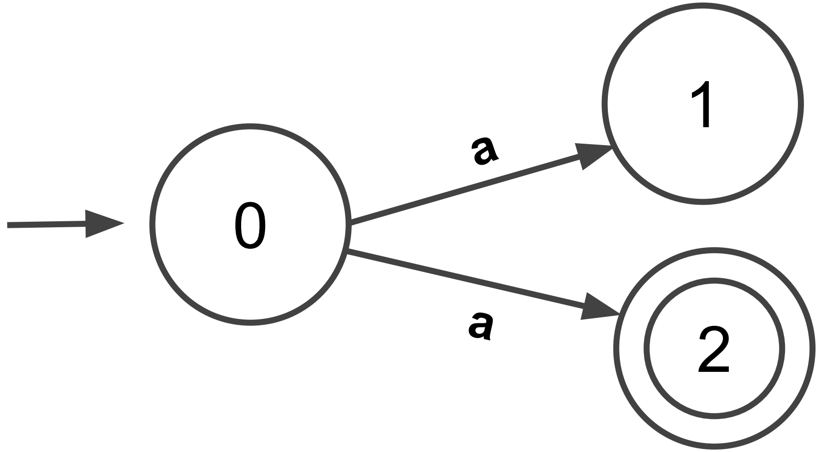
DFA

Accepts if the string ends on a final state

NFA

Accepts if at least one path ends on a final state

“a” would be accept because 0,2



Interlude: Project Talk

Project Talk

In your project, you will implement regular expressions

To do this, you will implement an NFA

(Note: Regex, NFA, DFA accept the same languages)

Project Talk - OCaml NFA

First you make an NFA, recall

Q – A finite set of states

Σ – A set of symbols, the alphabet

□ – A transition function

q_0 – The initial state

F – A set of final states

You will be given

q_0 – The initial state

F – A set of final states

□ – A transition function

How you define the NFA type is up to you, try to make something easy to match on!

Project Talk - E-Closure Function

Remembers that NFAs can transition on the empty string?

This is called an “ε-transition”

The ε-closure(S1) is the set of all states reachable from S1 using only ε-transitions

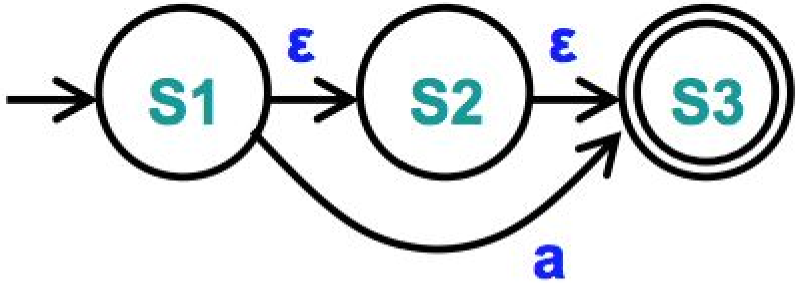
Note: ε-closure(S1) always includes S1

Project Talk - E-Closure 1

ε-closure(S1) =

ε-closure(S2) =

ε-closure(S3) =

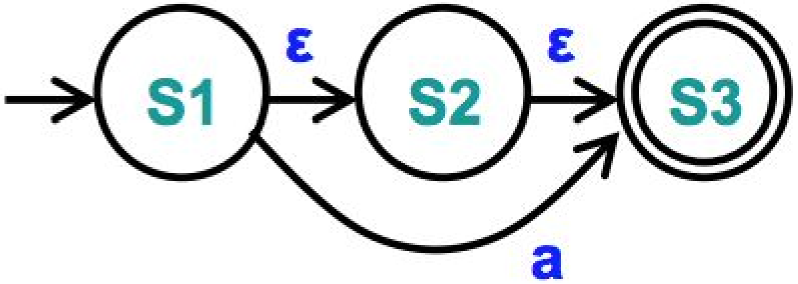


Project Talk - E-Closure 1

ε-closure(S1) = {S1, S2, S3}

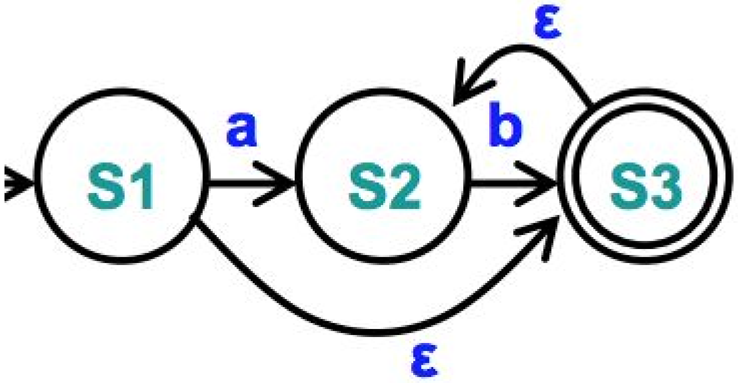
ε-closure(S2) = {S2, S3}

ε-closure(S3) = {S3}



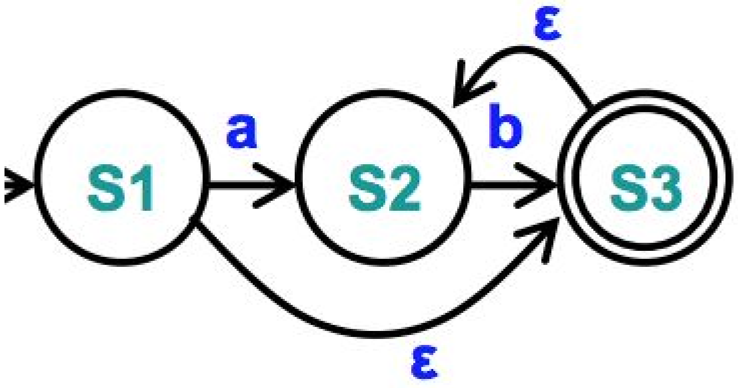
Project Talk - E-Closure 2

```
ε-closure(S1) =

ε-closure(S2) =

ε-closure(S3) =

ε-closure({S2,S3}) =
```



Project Talk - E-Closure 2

```
ε-closure(S1) = {S1, S2, S3}

ε-closure(S2) = {S2}

ε-closure(S3) = {S2, S3}

ε-closure({S2,S3}) = {S2,S3}
```

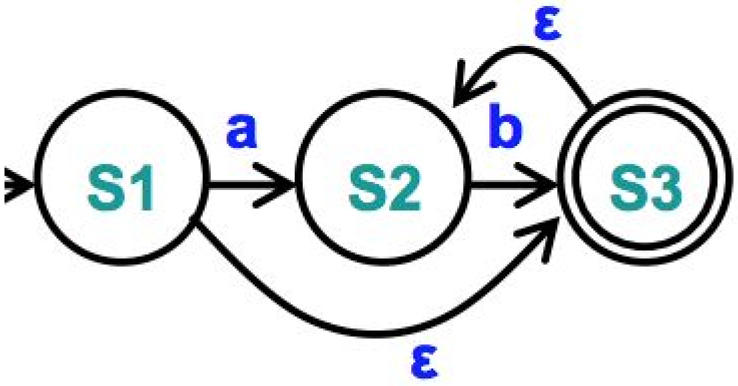


Project Talk - Move

move(S1,symbol)

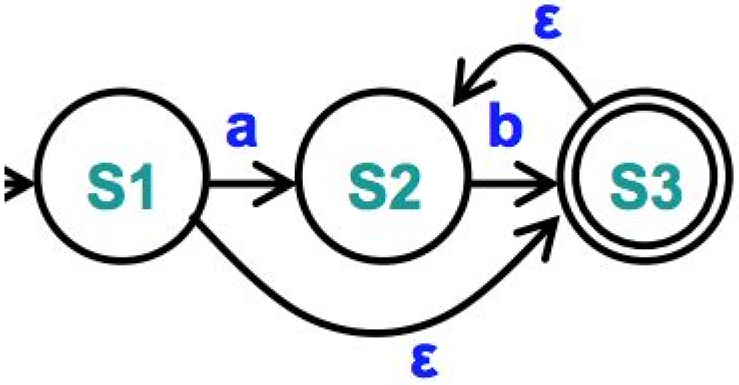
Set of states reachable from S1 using exactly one transition on the symbol.

Note: move(S1,a) only includes S1 if S1 has a transition to itself on a.



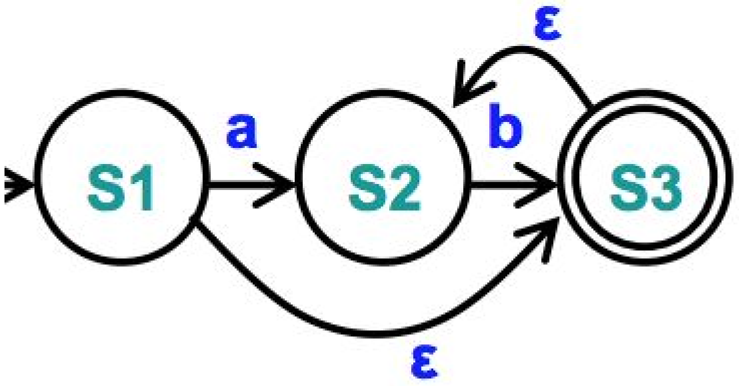
Project Talk - Move

```
move(S1,a) =

move(S1,b) =

move(S2,a) =

move(S2,b) =

move(S3,a) =

move(S3,b) =
```



Project Talk - Move

```
move(S1,a) = {S2}

move(S1,b) = {}

move(S2,a) = {}

move(S2,b) = {S3}

move(S3,a) = {}

move(S3,b) = {}
```



Reductions

Introduction

There are a few operations we care about

- Regex to NFA
- NFA to DFA
- Minimizing a DFA
- Also: DFA to Regex, DFA Complement

Your project will ask you to turn a regex to an NFA

Regex to NFA

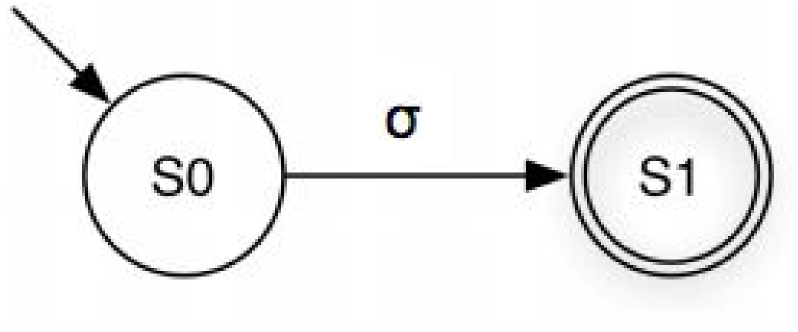
Regular Expressions to NFAs

Regular expressions are defined recursively

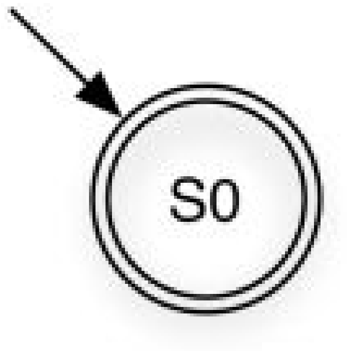
Know a number of base cases and inductive cases

This tells us how to build an NFA given a regex

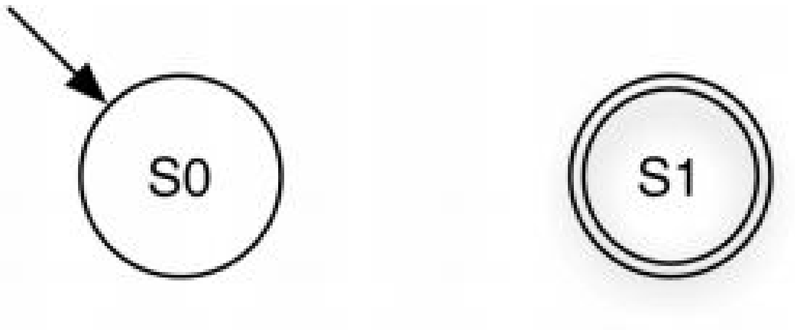
Base Case: A Symbol in the Language (σ in Σ)



Base Case: ε

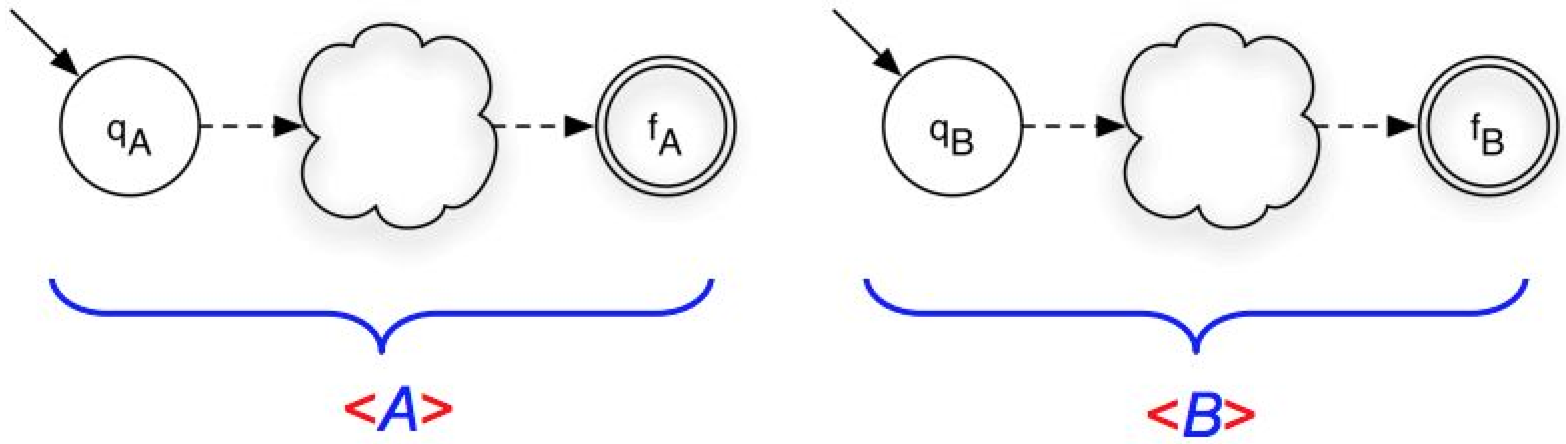


Base Case: ∅



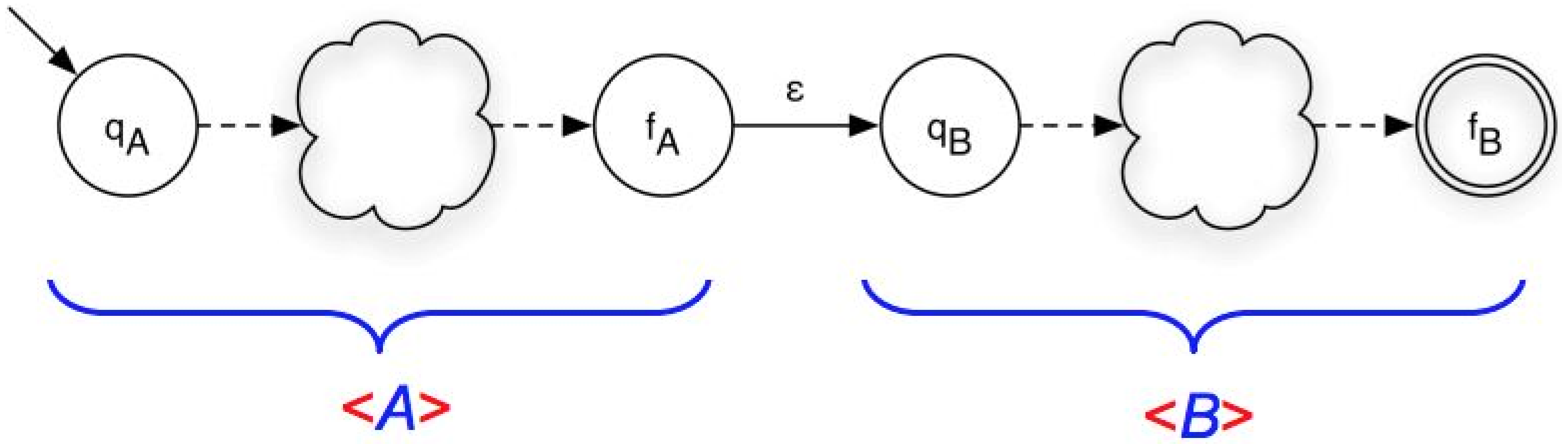
Concatenation: AB

A and B are each represented by an NFA



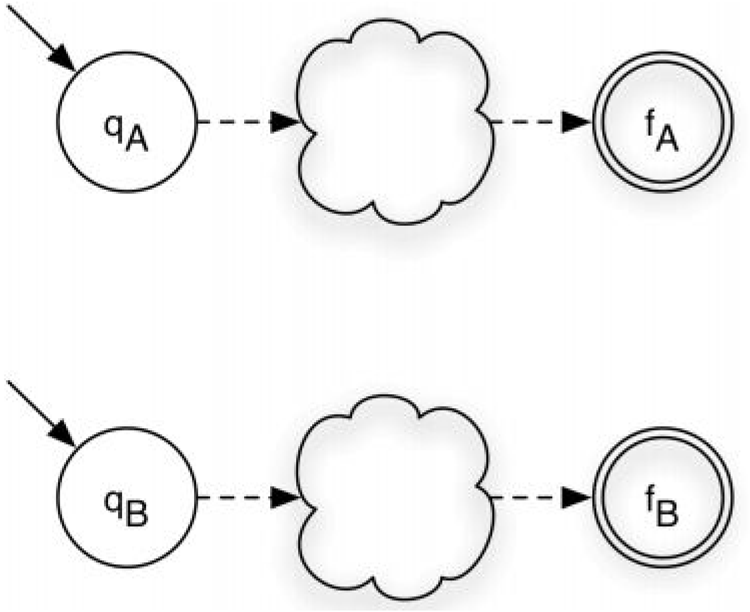
Concatenation: AB

To concatenate A and B, create a new NFA using the start of A, the final states of B, add an ε-transition from the final states of A to the start state of B.



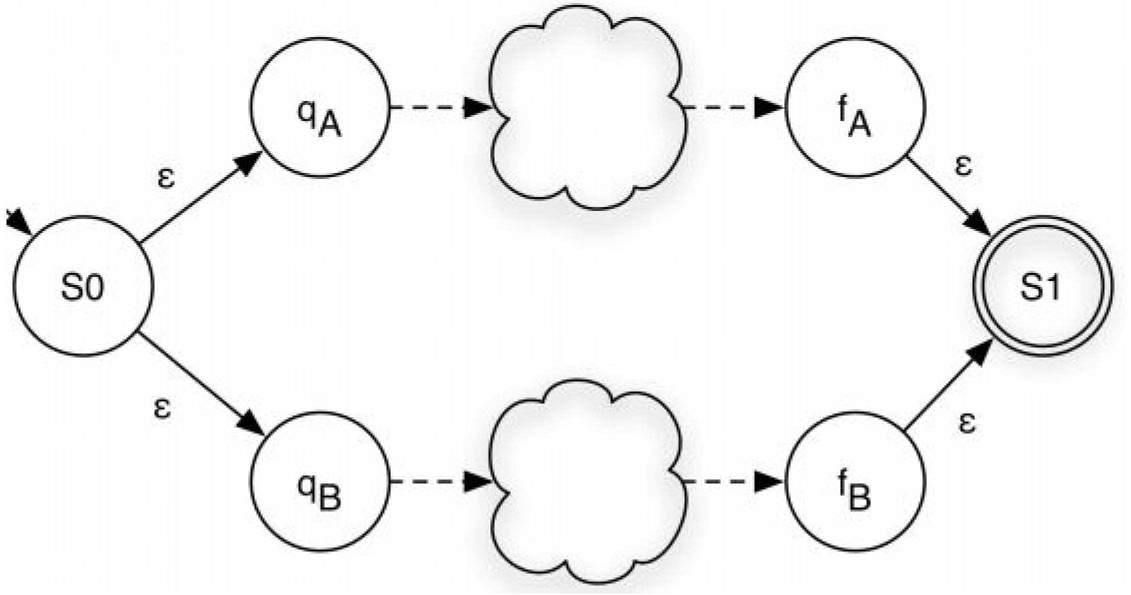
Union: A|B

A and B are each represented by an NFA



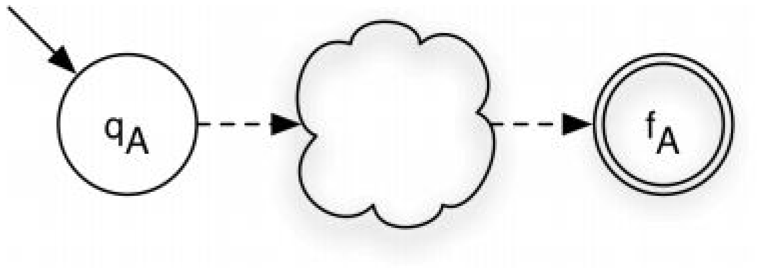
Union: A|B

Add a new start and end, add ε-transitions from new start to old starts, add ε-transitions from old finals to new finals.



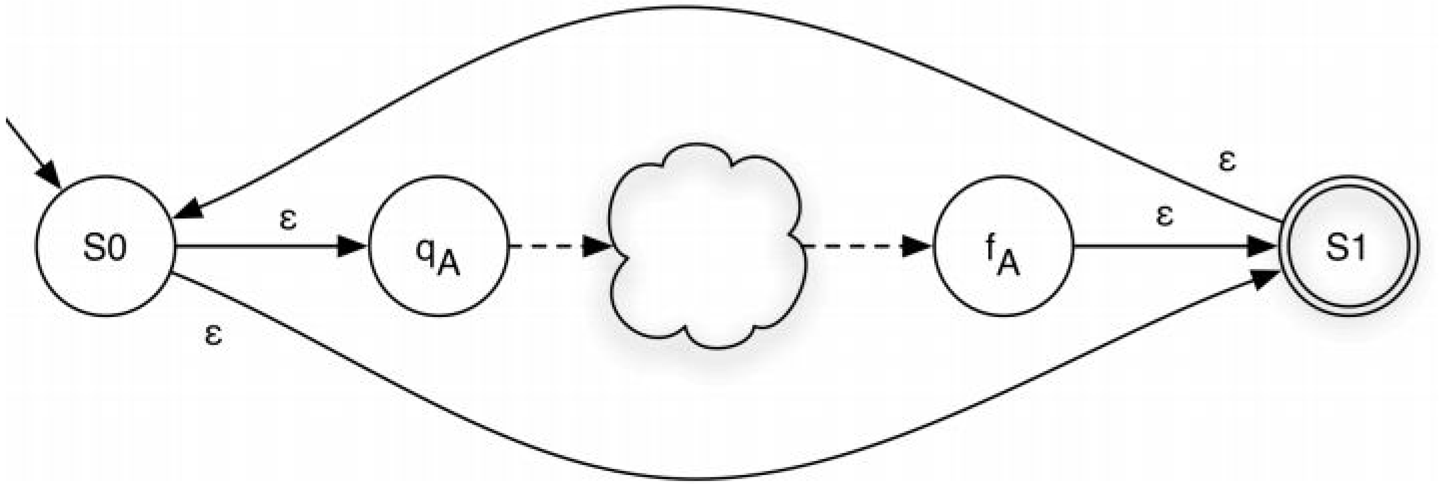
Closure: A*

A is represented by an NFA



Closure: A*

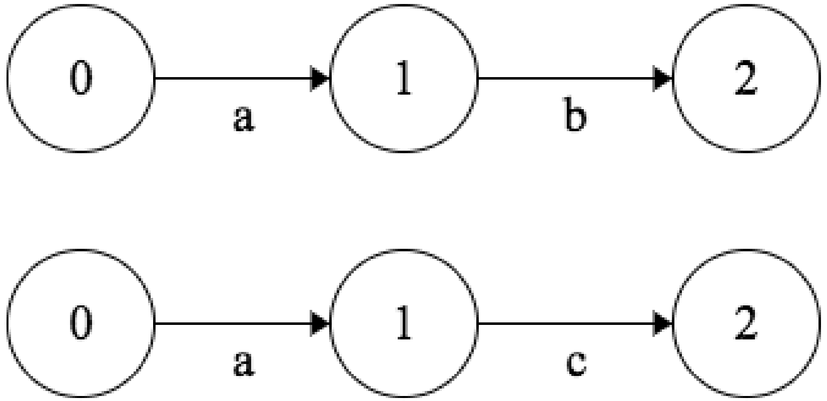
Add a new start and end state. Add ε-transition from new start to old start, old finals to new finals, and from new start to new final, and from new final to new start.



Draw an NFA for (ab|ac)*

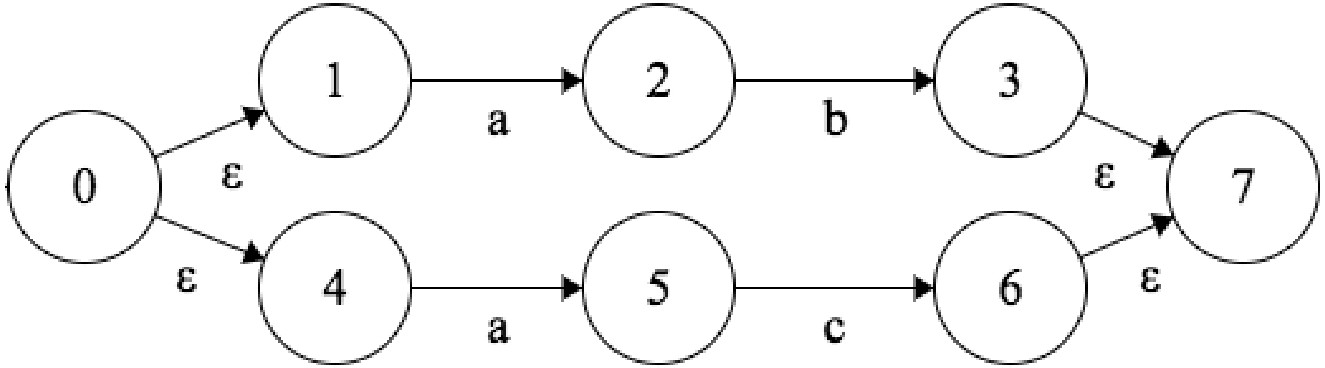
Draw an NFA for (ab|ac)*

Begin by creating an NFA for “ab” and “ac”



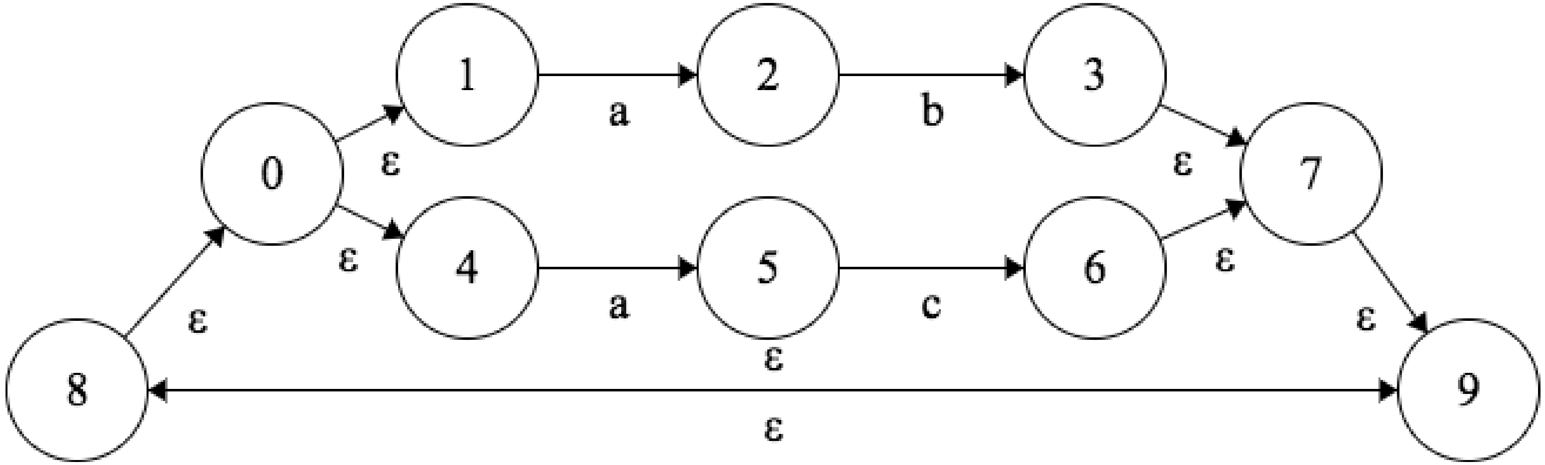
Draw an NFA for (ab|ac)*

Next, find the union of “ab” and “ac”



Draw an NFA for (ab|ac)*

Next, find the closure of (ab|ac).



Regex to NFA Complexity?

If a regular expression “A” has size n, where...

    n = # of symbols + # of operations

Then how many states does the NFA of “A” have?

    Each union and closure operation adds just two states

    O(n), overall, which is pretty good!

Regex to NFA Practice

1. a
2. a*
3. ab
4. (ab)*
5. (a|b)
6. (a|b)*
7. (a|b)*a
8. Binary strings that start and end with 1

Bonus: Consider e_closure and move for each solution

Reducing NFA to DFA

Reducing NFA to DFA

We use the “subset” algorithm to reduce an NFA to a DFA

Any NFA can be reduced to a DFA

Complexity

- Each DFA state will be a subset of the NFA states
- If the NFA has n states, there may be 2^n subsets
- NFA -> DFA may be O(2^n)

Reducing NFA to DFA - Simple Explanation

To reduce and NFA to DFA, we need two procedures

    ε-closure(S) - States reachable with 0+ ε-transitions

    move(S,c) - States reachable with 1 move on symbol c

We discussed these earlier, but let’s review

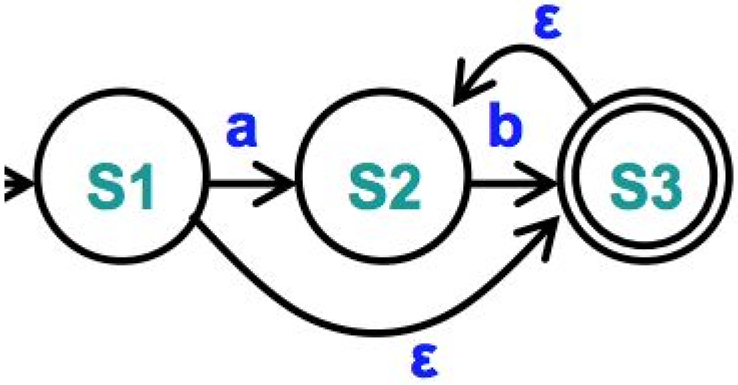
Reducing NFA to DFA: E-Closure(S)

ε-closure(S1) =

ε-closure(S2) =

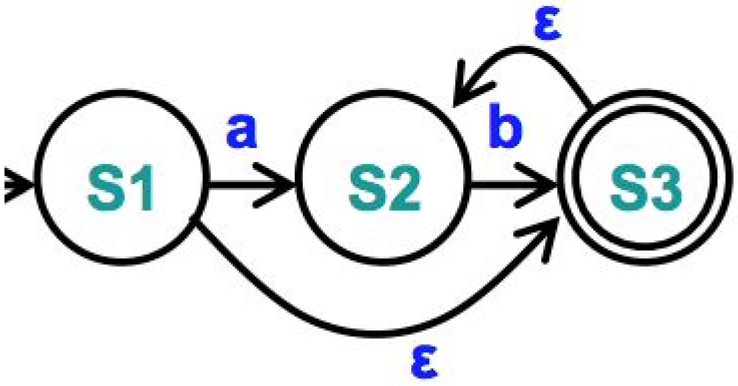
ε-closure(S3) =

ε-closure({S2,S3}) =



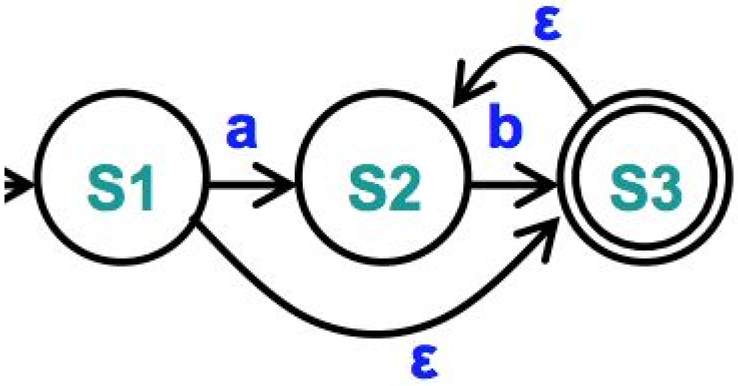
Reducing NFA to DFA: E-Closure(S)

```
ε-closure(S1) = {S1, S2, S3}

ε-closure(S2) = {S2}

ε-closure(S3) = {S2, S3}

ε-closure({S2,S3}) = {S2,S3}
```



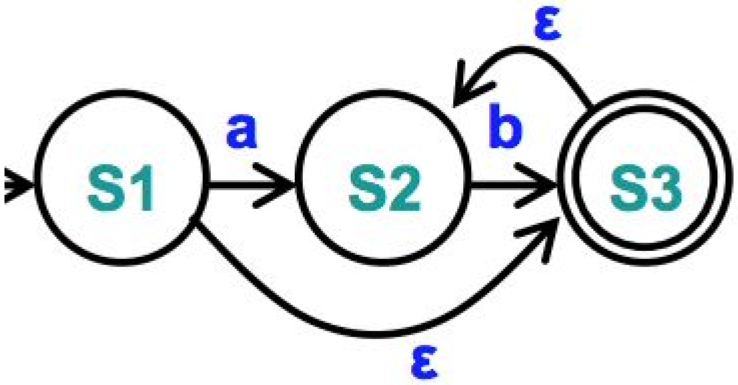
Reducing DFA to NFA: Move

```
move(S1,a) =

move(S1,b) =

move(S2,a) =

move(S2,b) =

move(S3,a) =

move(S3,b) =
```



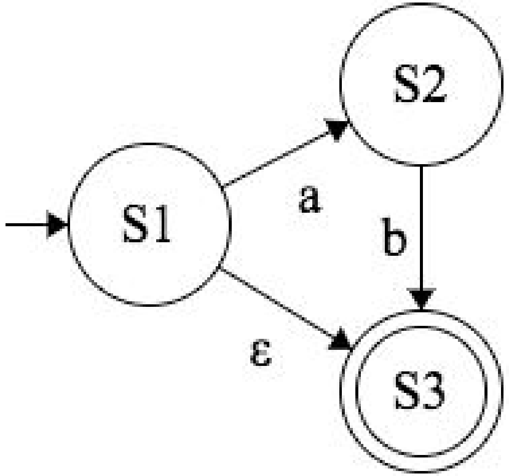
Reducing NFA to DFA: Move

```
move(S1,a) = {S2}

move(S1,b) = {}

move(S2,a) = {}

move(S2,b) = {S3}

move(S3,a) = {}

move(S3,b) = {}
```



Reducing NFA to DFA - Example

Rather than writing out the algorithm, let’s do an example



Step 1: Find the DFA Start State (Explanation)

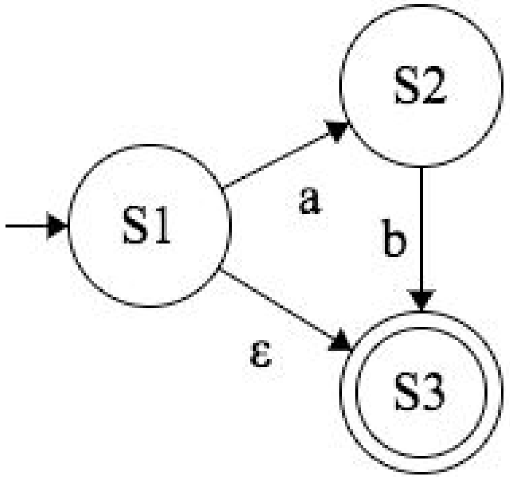
To find the start state of the DFA, we take the e-closure() of the NFA start state.

Why? Any e-transitions from the start node can be taken before we even look at the string, giving us a set of multiple start states.

Step 1: Find the DFA Start State

`new_start = e_closure(NFA Start)`

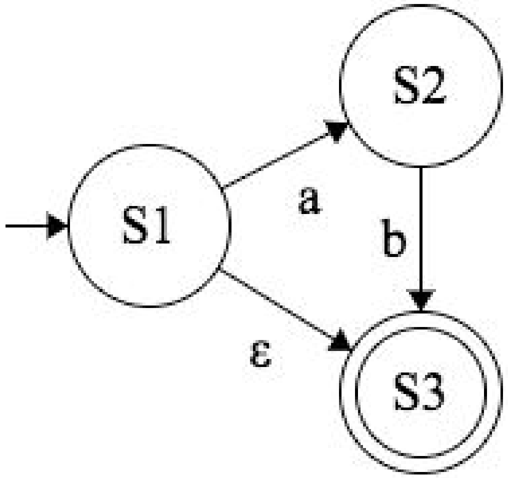
`What is the new start state?`

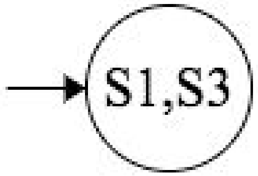


Step 1: Find the DFA Start State

```
new_start = e_closure(NFA Start)
```

```
In this case, we get q0 = {S1, S3}
```





Step 2: Visit, Move, E-Closure, Add, Repeat (Explained)

In the last step, we created a new state named “{S1,S3}”

Let’s at that to our DFA’s list of states

```
    Q = { {S1,S3} }
```

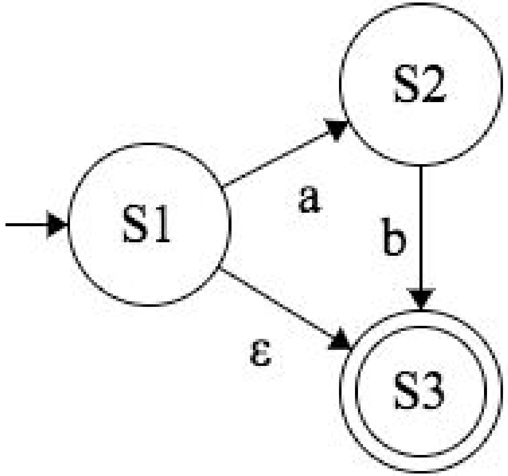
Since it didn’t already exist, we need to visit it

```
    to_visit = { {S1,S3} }
```

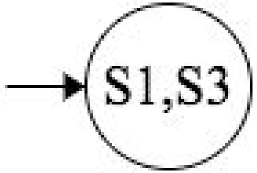
If {S1,S3} was already in Q, we would skip this step

Step 2: Visit, Move, E-Closure, Add, Repeat

```
move({S1,S3}, a) =

move({S1,S3}, b) =
```

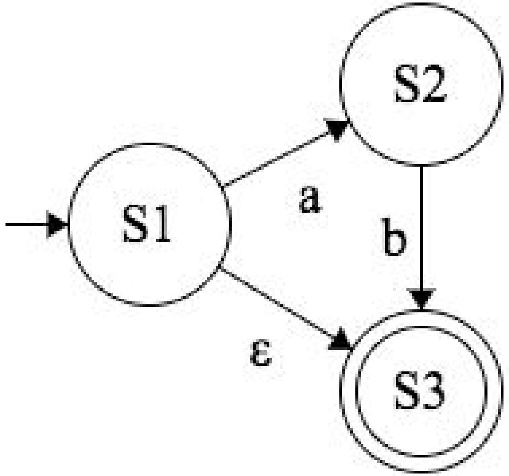


We need to perform the move operation for each letter in our alphabet

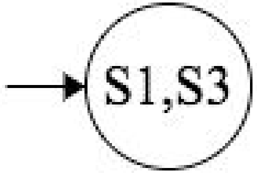


Step 2: Visit, Move, E-Closure, Add, Repeat

```
move({S1,S3}, a) = {S2}

move({S1,S3}, b) = {}
```



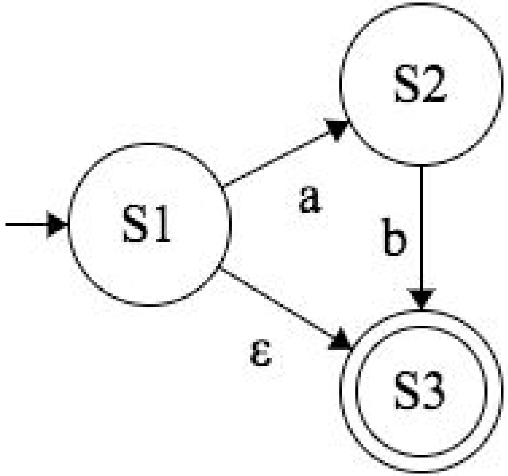
This gives us one new state, and a dead state. Before we add it, we need to perform e-closure to see if we can reach any states “for free”

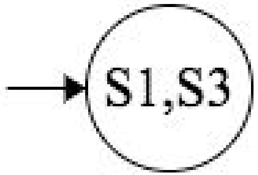


Step 2: Visit, Move, E-Closure, Add, Repeat

e-closure({S2}) = {S2}

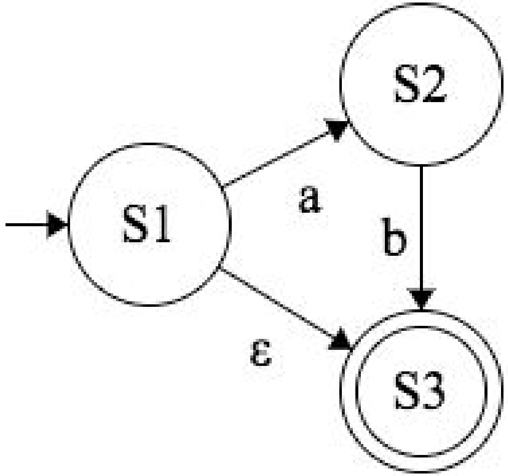
We don’t get any new states, but at least we checked. Now, we can add the new state {S2} to our DFA. We also add it to Q and to_visit.



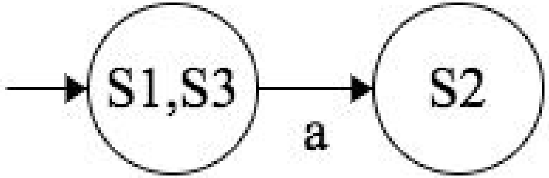


Step 2: Visit, Move, E-Closure, Add, Repeat

```
Q = { {S1,S3}, {S2} }

to_visit = {S2}
```

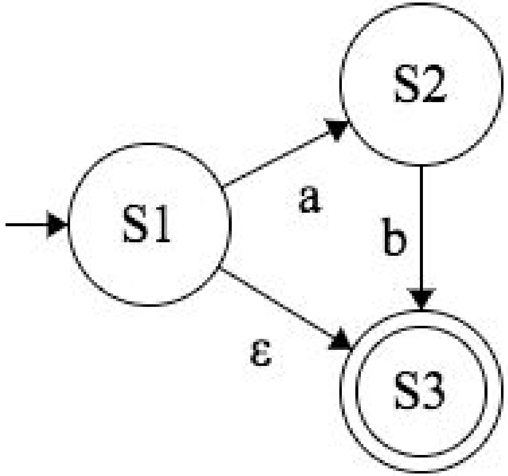


We add {S2}. {S2} is reached by a move on ‘a’, so add that path.

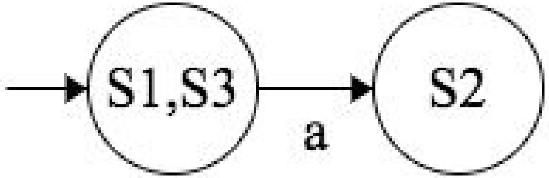


Step 2: Visit, Move, E-Closure, Add, Repeat

```
to_visit = {S2}
```

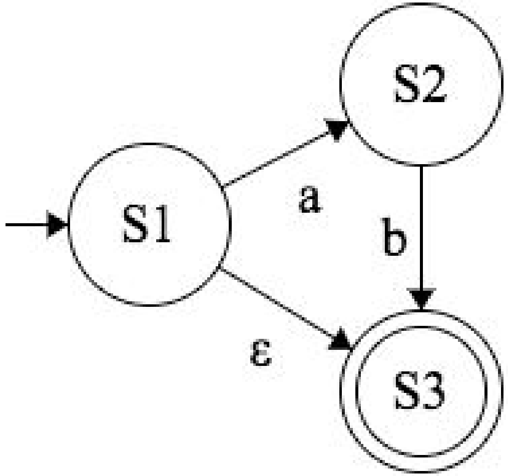


Now we repeat this process until to_visit is empty

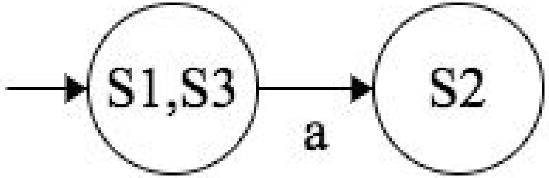


Step 3: Visit, Move, E-Closure, Add, Repeat

```
move({S2},a) =

move({S2},b) =
```

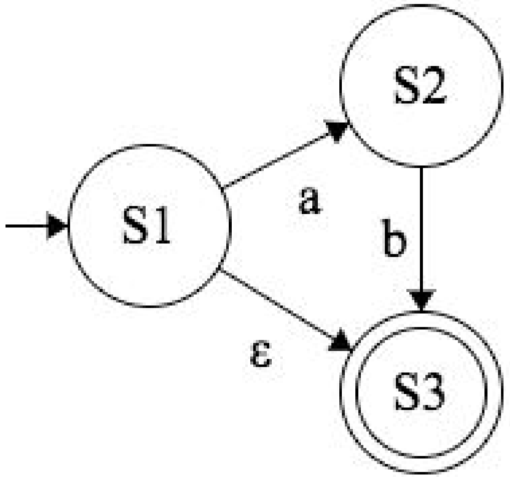


{S2} is the only state in our to_visit list, so let's begin by performing move for each letter in our alphabet

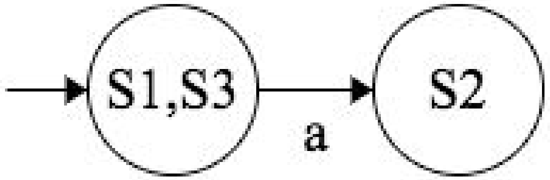


Step 3: Visit, Move, E-Closure, Add, Repeat

```
move({S2},a) = {}

move({S2},b) = {S3}
```



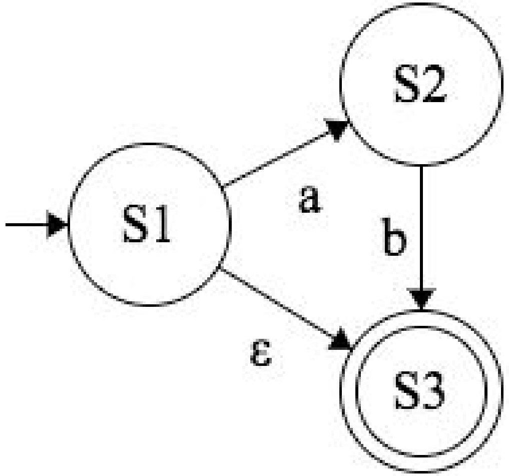
```
Ok, {S3} is new. Let’s perform
e-closure() real quick.
```

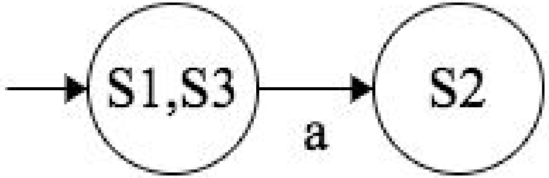


Step 3: Visit, Move, E-Closure, Add, Repeat

```
e-closure({S3}) = {S3}
```

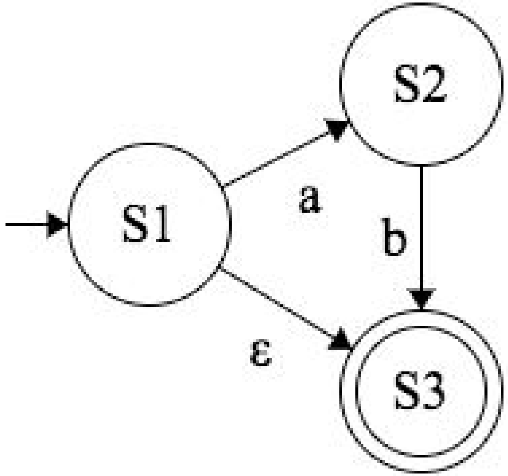
Again, we don’t get any new states. Sometimes you will! Don’t forget this step. Now we can add {S3}.



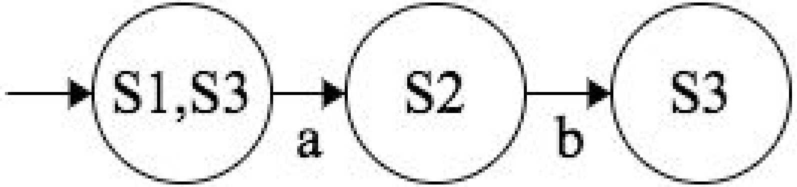


Step 3: Visit, Move, E-Closure, Add, Repeat

```
Q = { {S1,S3}, {S2}, {S3} }

to_visit = {S3}
```



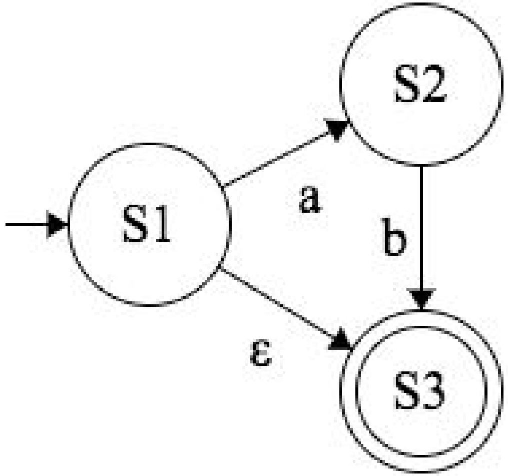
We add {S3}. {S3} is reached by a move on ‘b’, so add that path.

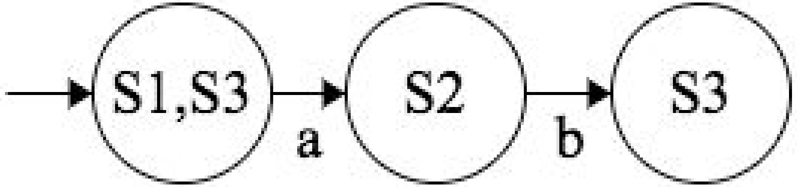


Step 3: Visit, Move, E-Closure, Add, Repeat

```
to_visit = {S3}
```

```
to_visit is not empty, so we repeat
```

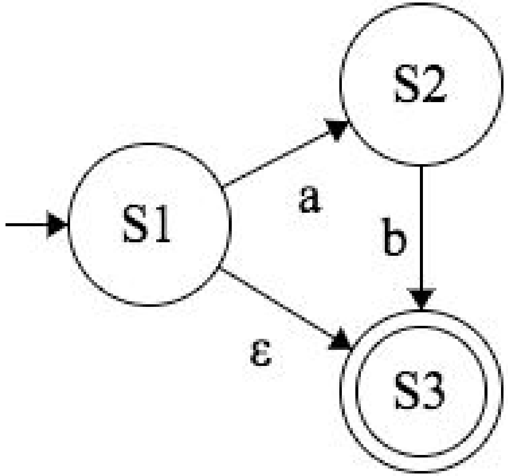


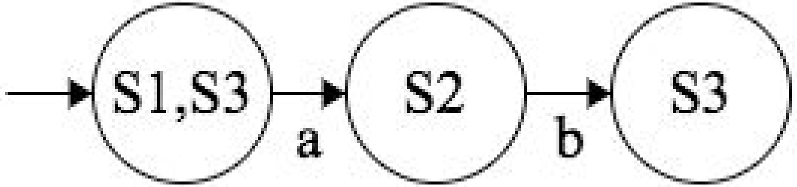


Step 4: Visit, Move, E-Closure, Add, Repeat

```
move({S3},a) =

move({S3},b) =
```

Let’s visit {S3} with move.

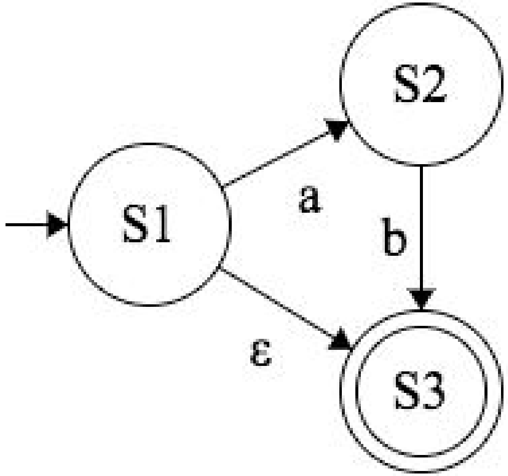


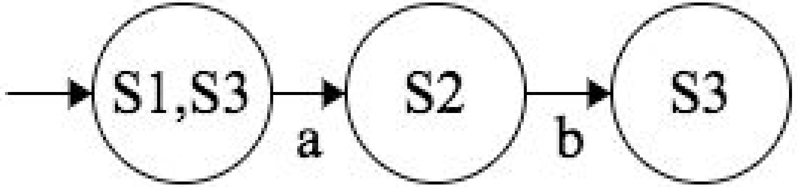


Step 4: Visit, Move, E-Closure, Add, Repeat

```
move({S3}, a) = {}

move({S3}, b) = {}
```

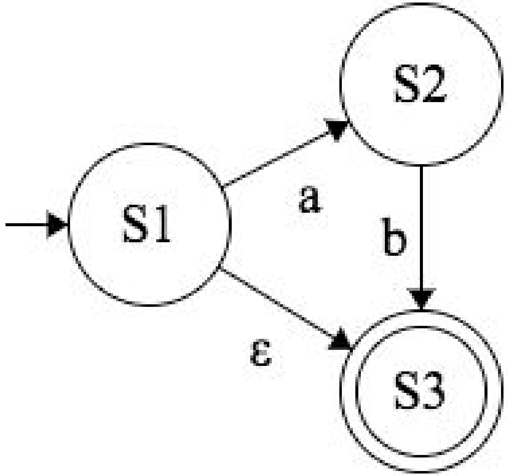
No new states here.



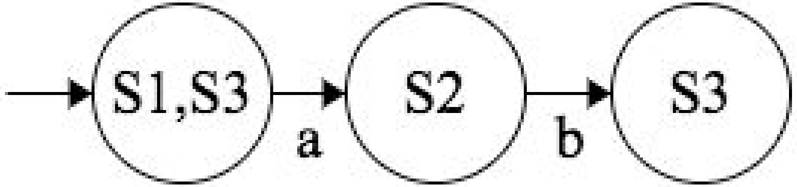


Step 4: Visit, Move, E-Closure, Add, Repeat

```
e-closure({}) = {}
```



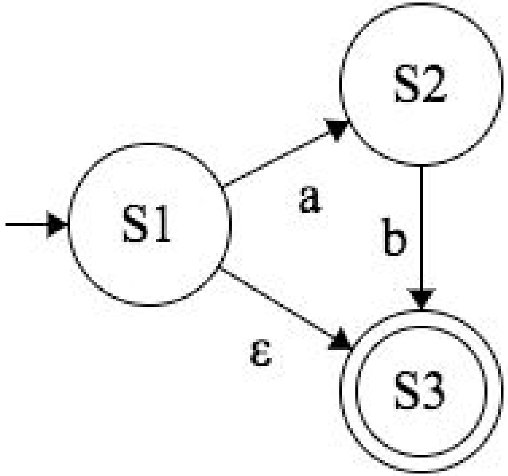
```
Nothing to e-closure()
```

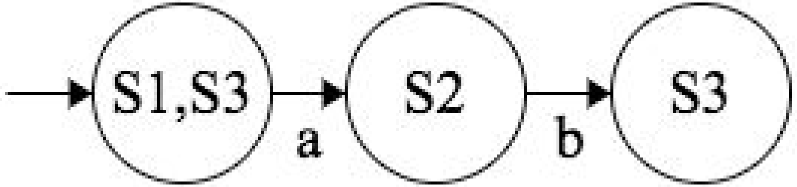


Step 4: Visit, Move, E-Closure, Add, Repeat

```
Q = { {S1,S3}, {S2}, {S3} }

to_visit = {}
```

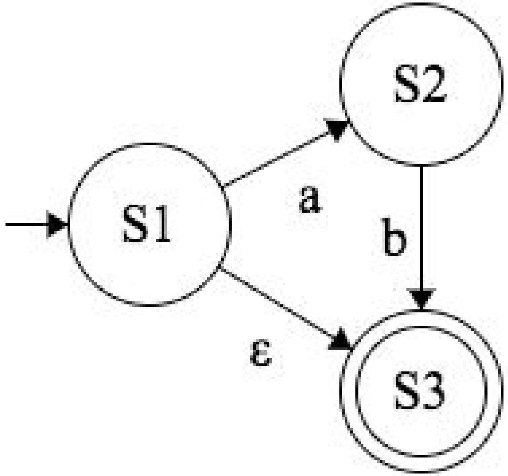
```
Nothing to add
```



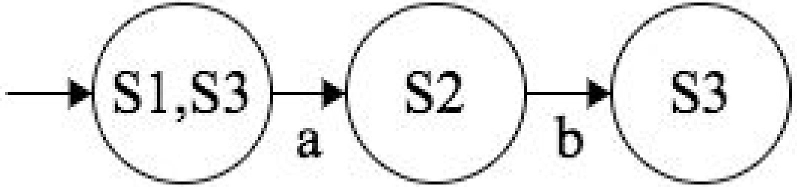


Step 4: Visit, Move, E-Closure, Add, Repeat

```
To_visit = {}
```

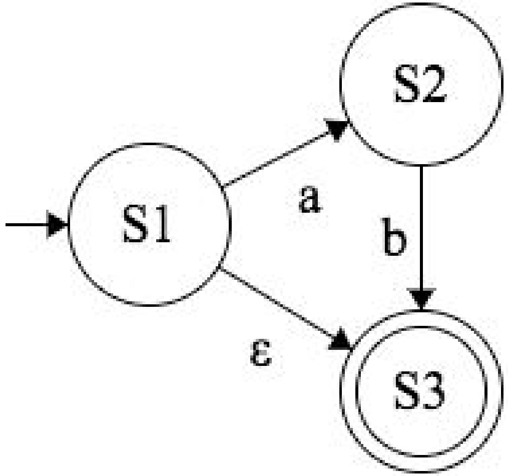


```
to_visit is empty, don't repeat!

Now we perform the final step.
```

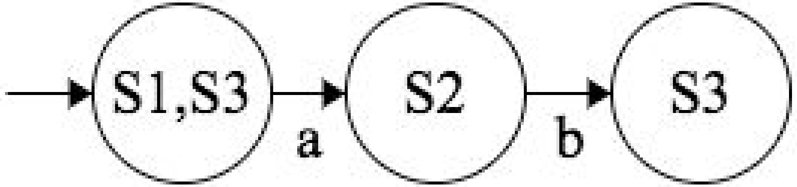


Step 5: Make Final States Final

In our NFA, S3 is a final state

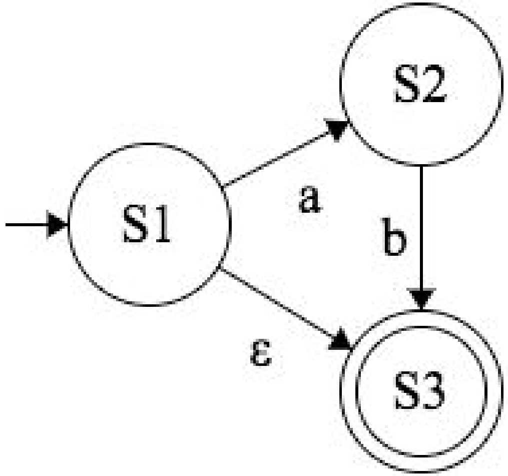


In our DFA, any state that includes S3 is final

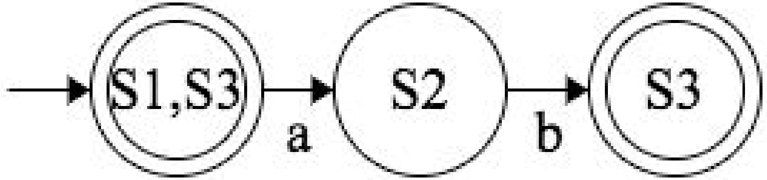


Step 5: Make Final States Final

In our NFA, S3 is a final state



In our DFA, any state that includes S3 is final

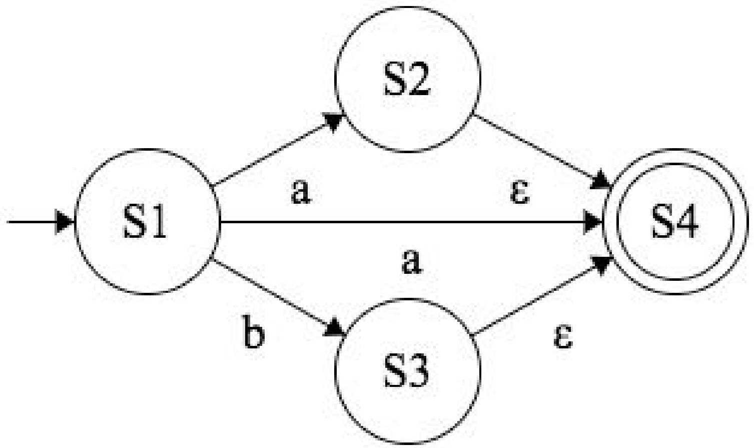


Reducing NFA to DFA: Subset Construction Algorithm

1. Find the new start state using e-closure()
2. For each new state: visit, move, e-closure(), add, repeat
3. When you have no more states to visit, make any state that contains an NFA final state a DFA final state

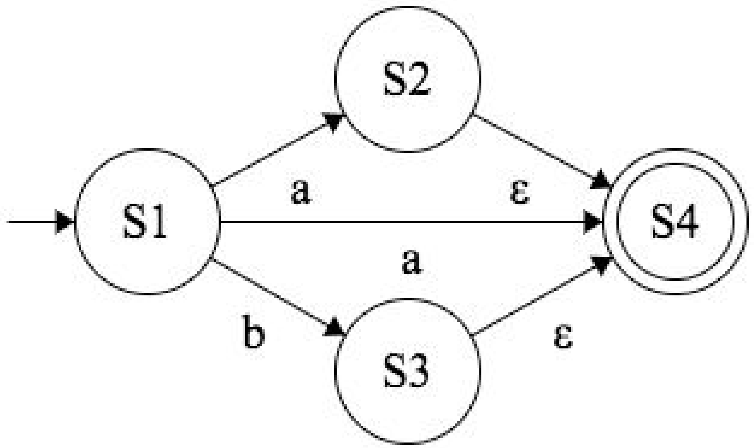
Regex to NFA - Practice

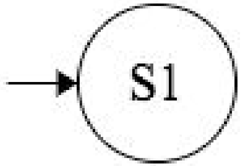
Convert the following DFA to and NFA



Step 1: Create the new start state

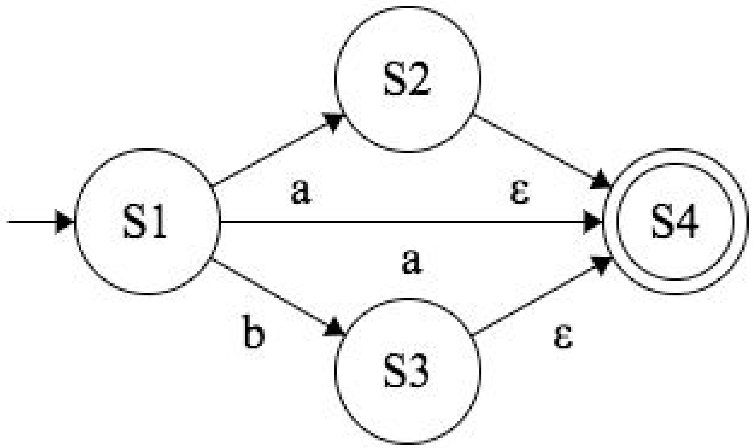
```
Q = { {S1} }; to_visit = {S1}
```

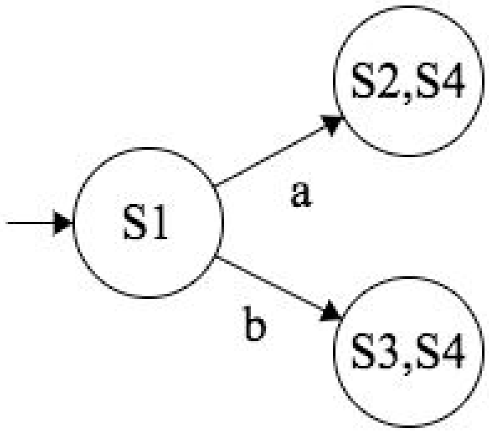




Step 2: Visit, Move, E-Closure, Add, Repeat

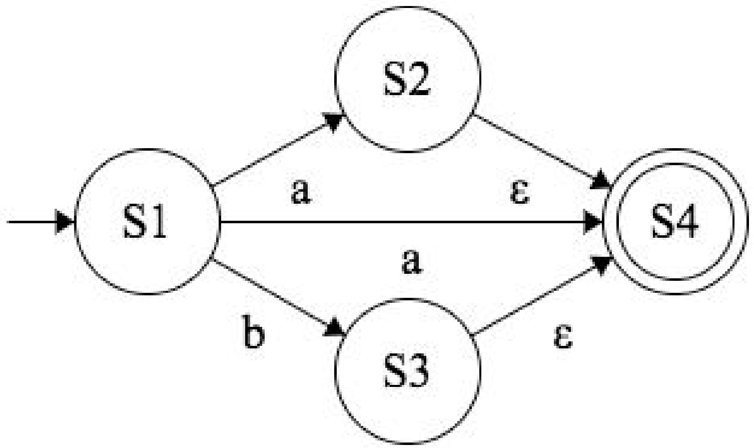
```
Q = {{S1},{S2,S4},{S3,S4}}; to_visit = {{S2,S4}, {S3,S4}}
```

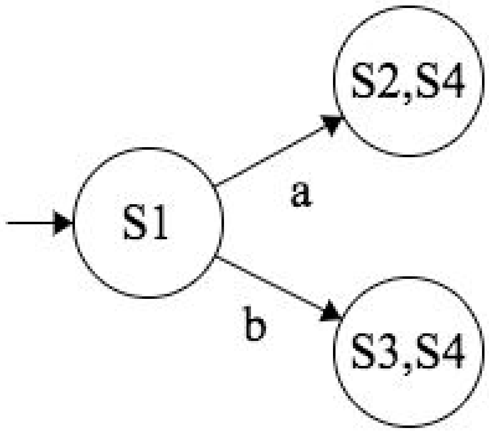




Step 3: Visit, Move, E-Closure, Add, Repeat

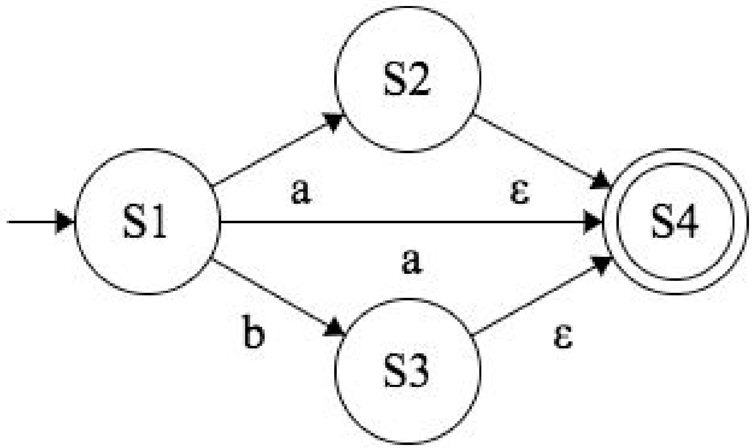
```
Q = {{S1},{S2,S4},{S3,S4}}; to_visit = {}
```

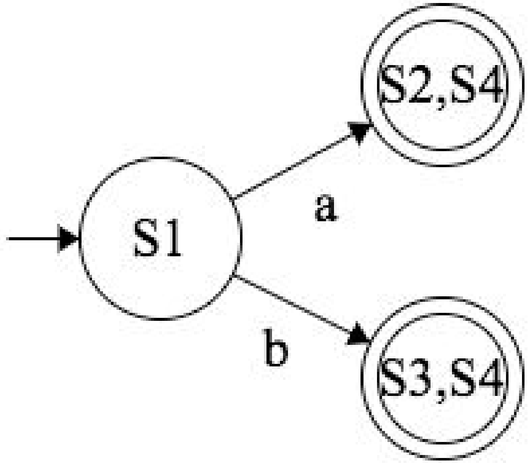




Step 4: Add Final States

```
Q = {{S1},{S2,S4},{S3,S4}}; to_visit = {}
```





NFA to DFA - Table Method

NFA to DFA - Table Method

There's another way to convert an NFA to DFA

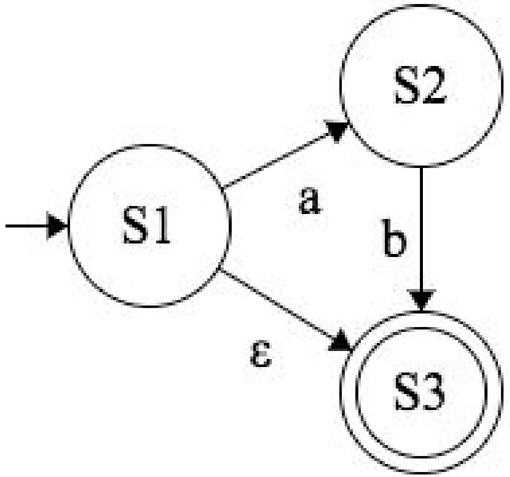
It's the same as the algorithm, but may help you keep track

I think this is why so many students use it

https://youtu.be/taClnxU-nao

Example

Let’s revisit this example



Step 1: First Table

First make a table with all the states and symbols, plus e*

Table 1	a	b	e*
S1			
S2			
S3			

Step 1: First Table

For each symbol, add the move(S,c). Put “-” for empty sets.

Table 1	a	b	e*
S1	S2	-	
S2	-	S3	
S3	-	-	

Step 1: First Table

For the e* column, put e_closure(S)

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

Step 2: Second Table

Create a new table, each state comes from e* of table one, and each symbol now has e* added to it

Table 2	ae*	be*
{S1,S3}		
{S2}		
{S3}		

Step 2: Second Table

For each symbol, do the move, followed by an e-closure.

Table 2	ae*	be*
{S1,S3}		
{S2}		
{S3}		

Step 2: Second Table

move({S1,S3},a) is {S2}, and e-closure({S2}) is {S2}.

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

So, for the {S1,S3} row, column ae*, we add {S2}

Table 2	ae*	be*
{S1,S3}	{S2}	
{S2}		
{S3}		

Step 2: Second Table

`move({S1,S3},b) is {}, and e-closure({}) is {}.`

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

`So, for the {S1,S3} row, column be*, we add {} (or -)`

Table 2	ae*	be*
{S1,S3}	{S2}	-
{S2}		
{S3}		

Step 2: Second Table

move({S2},a) is {}, and e-closure({}) is {}.

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

So, for the {S2} row, column ae*, we add {} (or -)

Table 2	ae*	be*
{S1,S3}	{S2}	-
{S2}	-	
{S3}		

Step 2: Second Table

move({S2},b) is {S3}, and e-closure({S3}) is {S3}.

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

So, for the {S2} row, column be*, we add {S3}

Table 2	ae*	be*
{S1,S3}	{S2}	-
{S2}	-	{S3}
{S3}		

Step 2: Second Table

move({S3},a) is {}, and e-closure({}) is {}.

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

So, for the {S3} row, column ae*, we add {} (or -)

Table 2	ae*	be*
{S1,S3}	{S2}	-
{S2}	-	{S3}
{S3}	-	

Step 2: Second Table

move({S3},b) is {}, and e-closure({}) is {}.

Table 1	a	b	e*
S1	S2	-	{S1,S3}
S2	-	S3	{S2}
S3	-	-	{S3}

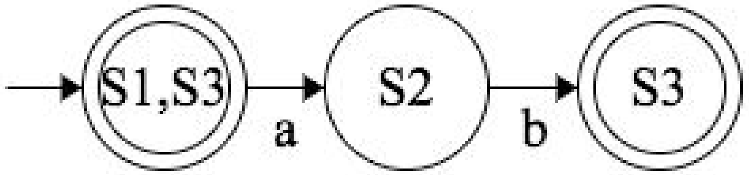
So, for the {S3} row, column be*, we add {} (or -)

Table 2	ae*	be*
{S1,S3}	{S2}	-
{S2}	-	{S3}
{S3}	-	-

Step 3: Create the NFA

Now we can use the table to create the NFA.

Table 2	ae*	be*
{S1,S3}	{S2}	-
{S2}	-	{S3}
{S3}	-	-



Notes

Notice that this gave the same answer as before

Also, after creating table one, we don’t need to look at the NFA anymore. (Notice in Step 2, we fill in the second table just by looking at the first)

If you’re going to use this method, I would recommend watching the video here: https://youtu.be/taClnxU-nao

Reducing DFA to RE

DFA to Regular Expressions

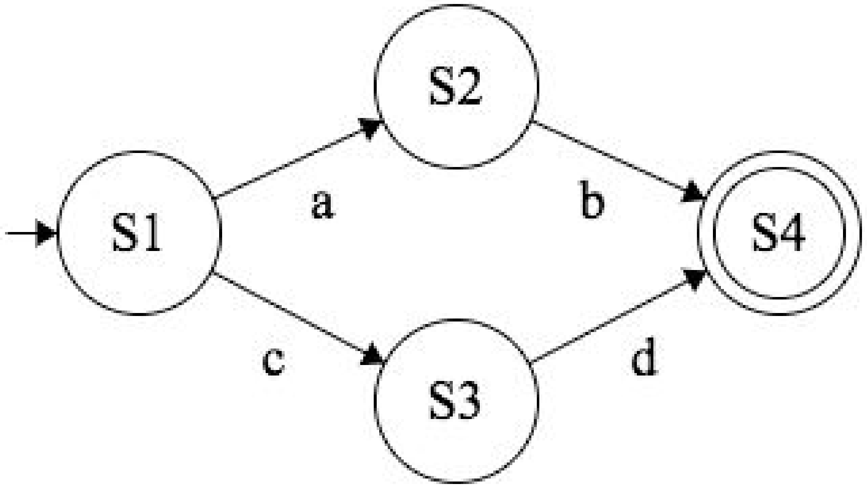
Remove states one by one

When you remove a state, label it’s transition with a regular expression

When the start and final state are left, the transition will be labeled with the regular expression

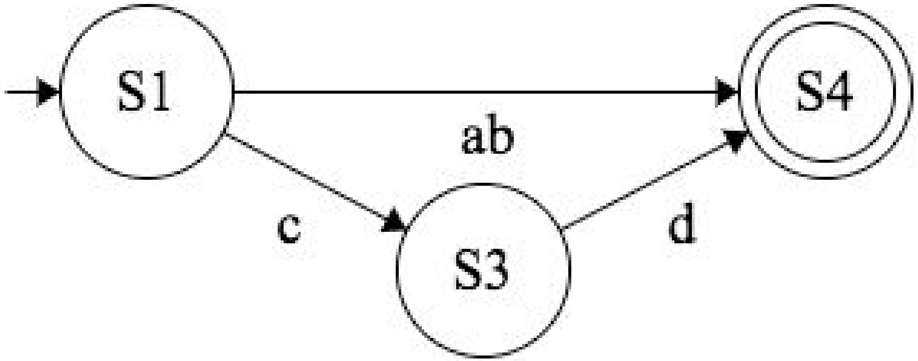
Example

Reduce the following DFA to a regular expression



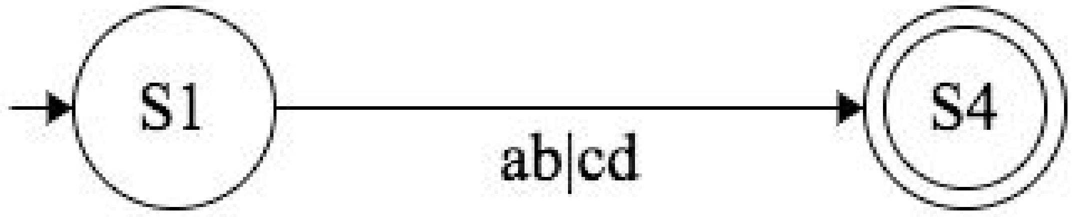
Example

Reduce the following DFA to a regular expression



Example

Reduce the following DFA to a regular expression



Resources

Resources

Creating Finite State Machines

FSM Designer: http://madebyevan.com/fsm/

Regular Expression to NFA

https://youtu.be/RYNN-tb9WxI

NFA to DFA

https://youtu.be/taClnxU-nao