LECTURE 5

Scanning

SYNTAX ANALYSIS

We know from our previous lectures that the process of verifying the syntax of the program is performed in two stages:

- Scanning: Identifying and verifying tokens in a program.
- Parsing: Identifying and verifying the patterns of tokens in a program.

LEXICAL ANALYSIS

In the last lecture, we discussed how valid tokens may be specified by regular expressions.

In this lecture, we will discuss how we can go about identifying tokens that are specified by regular expressions.

RECOGNIZING TOKENS

- A *recognizer* for a language is a program that takes a string x as input and answers "yes" if x is a sentence of the language and "no" otherwise.
  - In the context of lexical analysis, given a string and a regular expression, a recognizer of the language specified by the regular expression answers "yes" if the string is in the language.
  - How can we recognize a regular expression (*int*) ? What about (*int* | *for*)?

    We could, for example, write an ad hoc scanner that contained simple conditions to test, the ability to peek ahead at the next token, and loops for numerous characters of the same type.

RECOGNIZING TOKENS

For example, to recognize (int | for):

```
if cur_char == 'i' :
    peak at next character
    if it is 'n':
        peak at next character
        if it is 't':
            return int
if cur_char == 'f':
    peak at next character
    if it is 'o':
        peak at next character
        if it is 'r':
            return for
else error
```

FINITE AUTOMATA

A set of regular expressions can be compiled into a recognizer automatically by constructing a *finite automaton* using scanner generator tools (lex, for example).

The advantage of using an automatically generated scanner over an ad-hoc implementation is ease-of-modification.

- If there are ever any changes to my token definitions, I only need to update my regular expressions and regenerate my scanner.

Where performance is a concern, hand-written scanners or optimized scanners may be necessary but for development scanner generators are most useful.

FINITE AUTOMATA

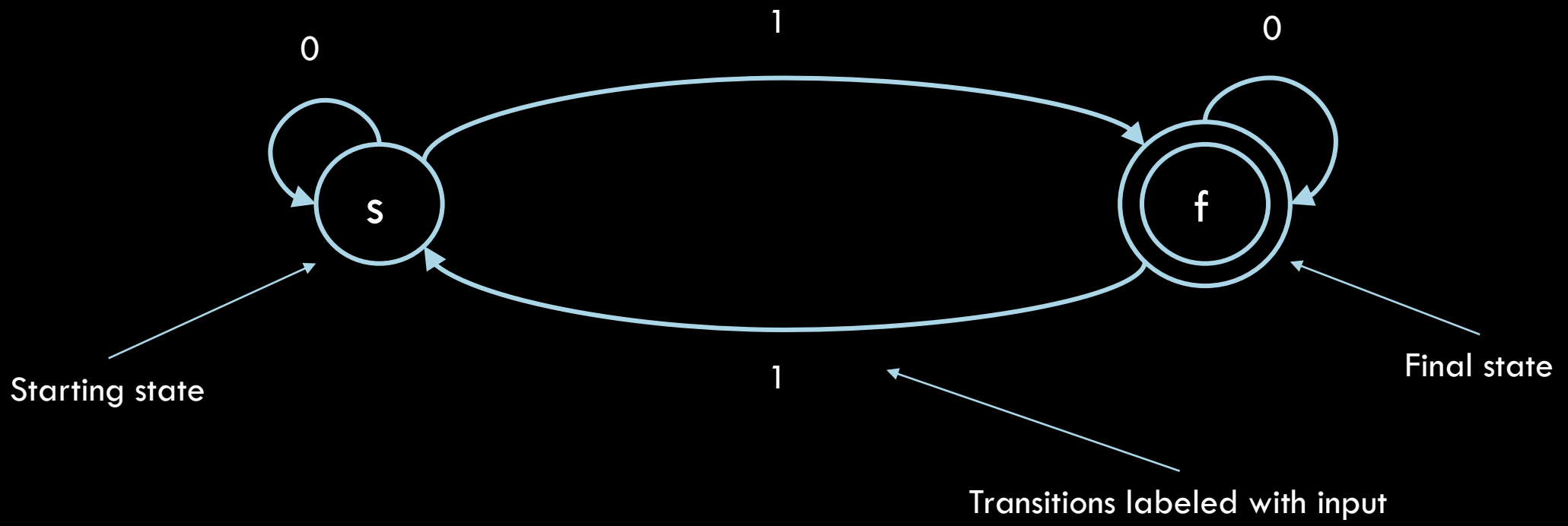
A finite automaton is a simple idealized machine that is used to recognize patterns within some input.

- A finite automaton will accept or reject an input depending on whether the pattern defined by the finite automaton occurs in the input.

- The elements of a finite automaton, given a set of input characters, are
  - A finite set of states (or nodes).
  - A specially-denoted start state.
  - A set of final (accepting) states.
  - A set of labeled transitions (or arcs) from one state to another.

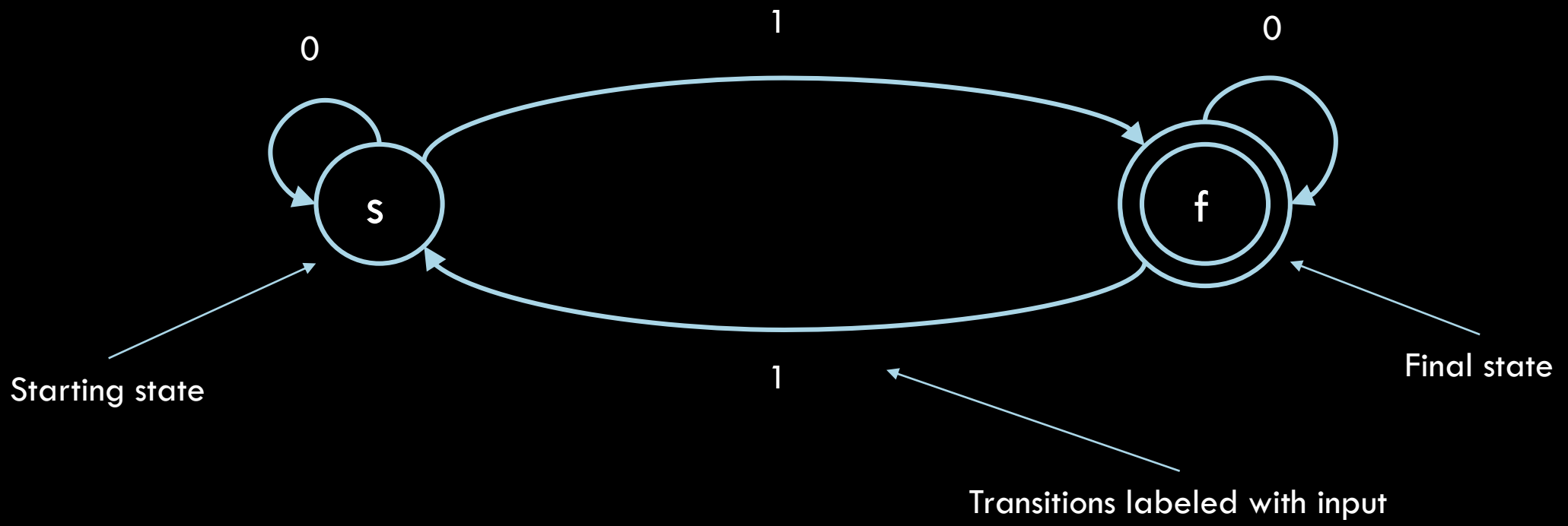
FINITE AUTOMATA

Here's an example finite automaton that accepts any sequence of 1's and 0's which has an odd number of 1's.



FINITE AUTOMATA

We accept a string only if its characters provide a path from the starting state to some final state.



FINITE AUTOMATA

Finite automata come in two flavors.

- Deterministic
  - Never any ambiguity.
  - For any given state and any given input, only one possible transition.
- Non-deterministic
  - There may be more than one transition from any given state for any given character.
  - There may be epsilon transitions – transitions labeled by the empty string.

There is no obvious algorithm for converting regular expressions to DFAs.

FINITE AUTOMATA

Typically scanner generators create DFAs from regular expressions in the following way:

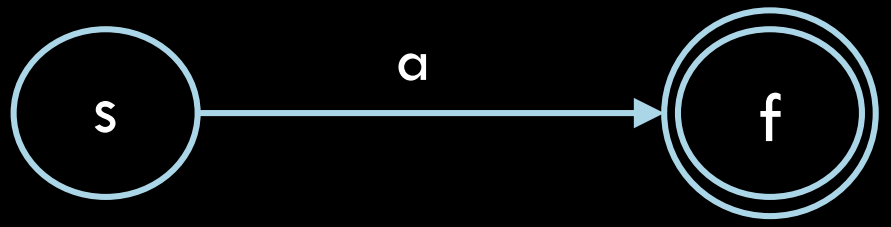
- Create NFA equivalent to regular expression.
- Construct DFA equivalent to NFA.
- Minimize the number of states in the DFA.

CONSTRUCTING NFAS

Let's say we have some regular expression that specifies the tokens we allow in our programming language. How do we turn this into an NFA?

Well, there are a couple of basic building blocks we can use.

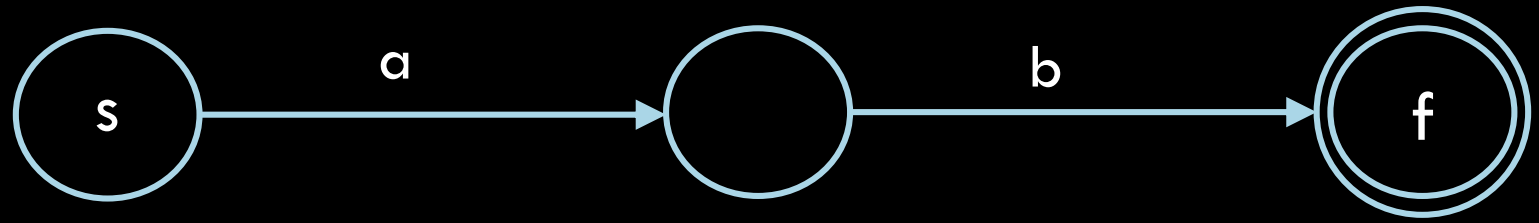
Let's say our regular expression is simply 'a', meaning we only accept one instance of the character 'a' as a token.



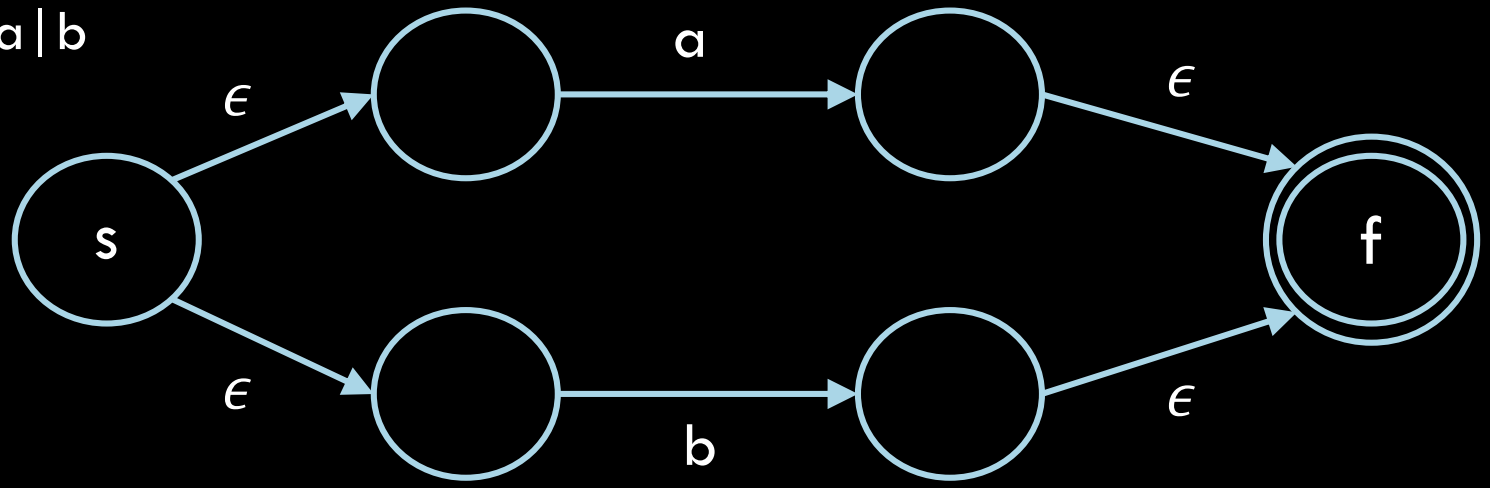
Similarly, we could have a two state NFA that accepts the empty string ϵ.

CONSTRUCTING NFAS

- Concatenation: ab

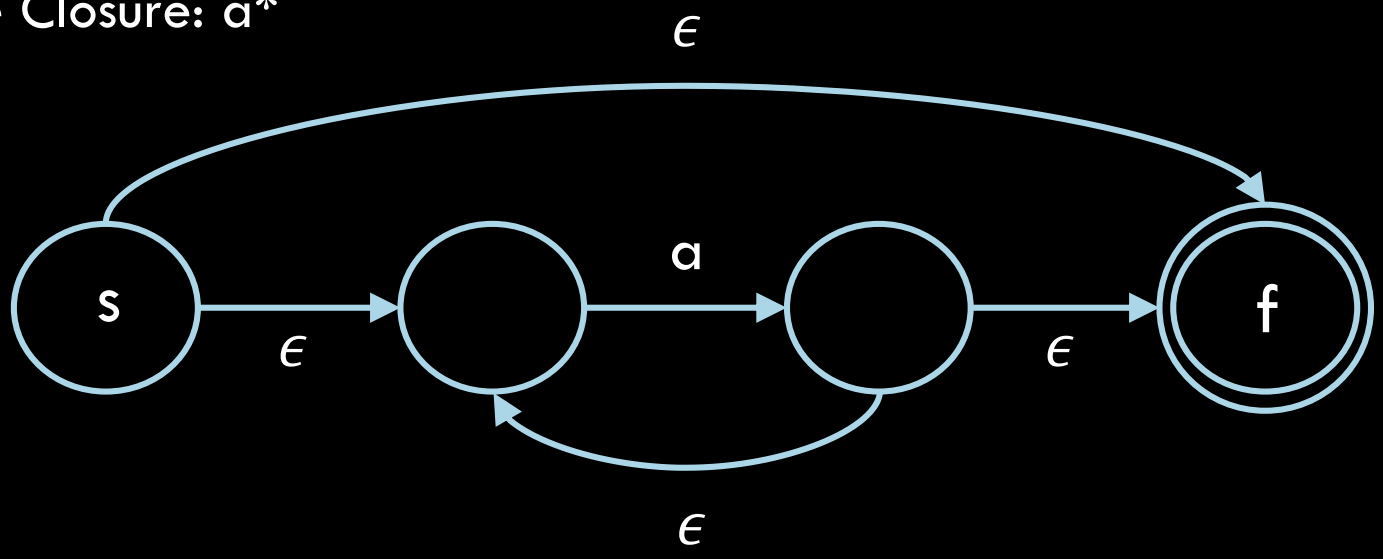


- Alternation: a | b



CONSTRUCTING NFAS

- Kleene Closure: a*



CONSTRUCTING NFAS

There are three important properties of these basic NFAs that we should take note of:

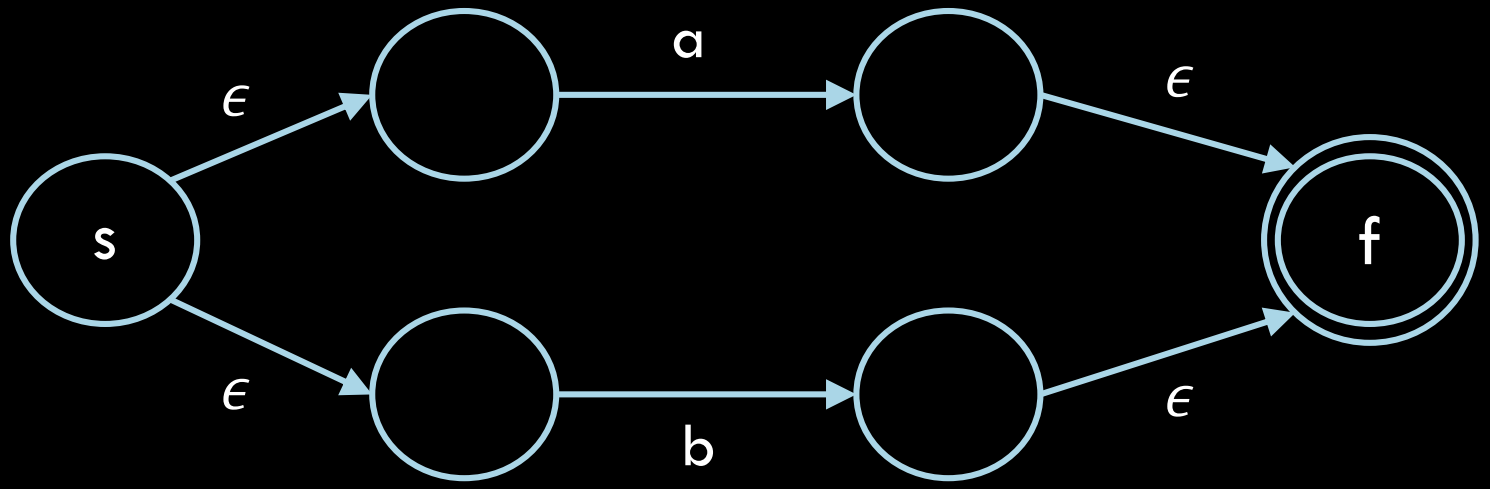
- There are no transitions back into the initial state.
- There is a single final state.
- There are no transitions out of the final state.

Because of these invariant properties, we can combine smaller NFAs to create larger NFAs. When translating a regular expression into an NFA, start with small components of the regular expression and use them to create larger constructions.

CONSTRUCTING NFAS

Let's put this all together. Let's say I have the following regular expression:

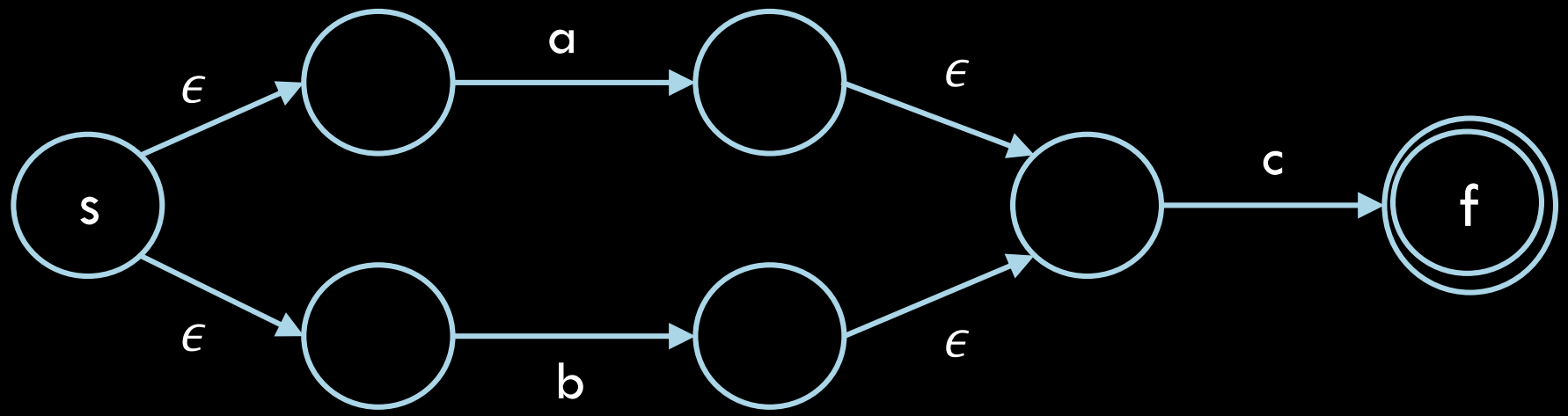
((a | b) c)* which describes the set {ϵ, ac, bc, acac, acbc, bcac, bcbc, acacac….}
Let's start with (a | b). We already know the corresponding NFA is:



CONSTRUCTING NFAS

Let's put this all together. Let's say I have the following regular expression:

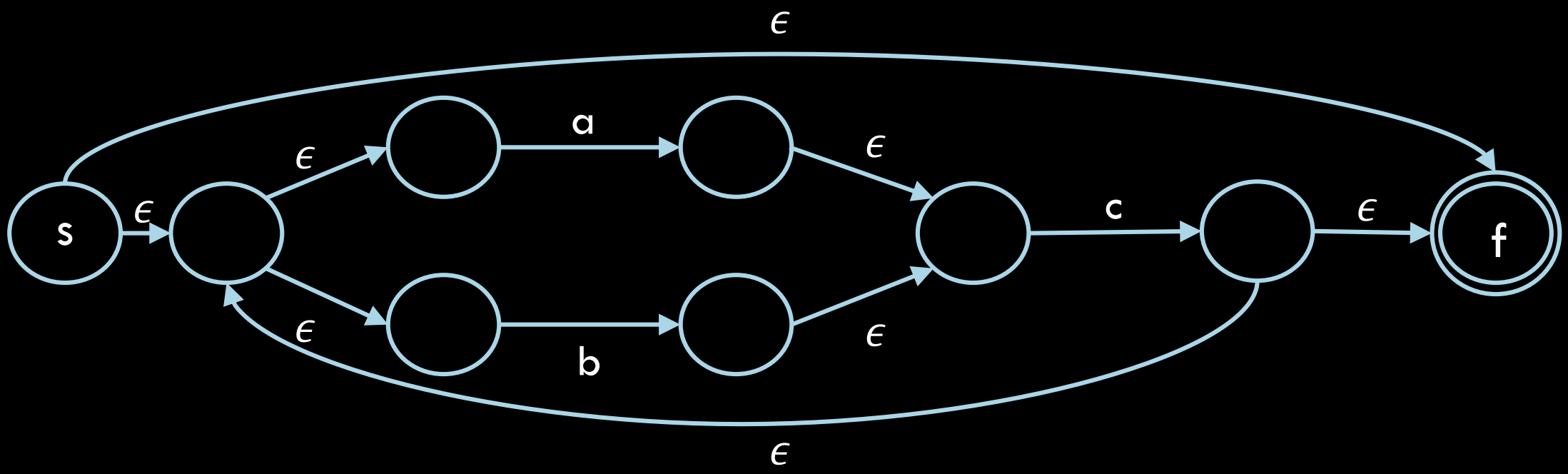
((a | b) c)* which describes the set {ϵ, ac, bc, acac, acbc, bcac, bcbc, acacac….}
Concatenating (a | b) with c gives us:



CONSTRUCTING NFAS

Let's put this all together. Let's say I have the following regular expression:

((a | b) c)* which describes the set {ϵ, ac, bc, acac, acbc, bcac, bcbc, acacac….}
Adding in the Kleene closure gives us the NFA for ((a | b) c)*:



FROM NFAS TO DFAS

If we can easily convert our regular expressions into an NFA, then why do we need to further convert it to a DFA?

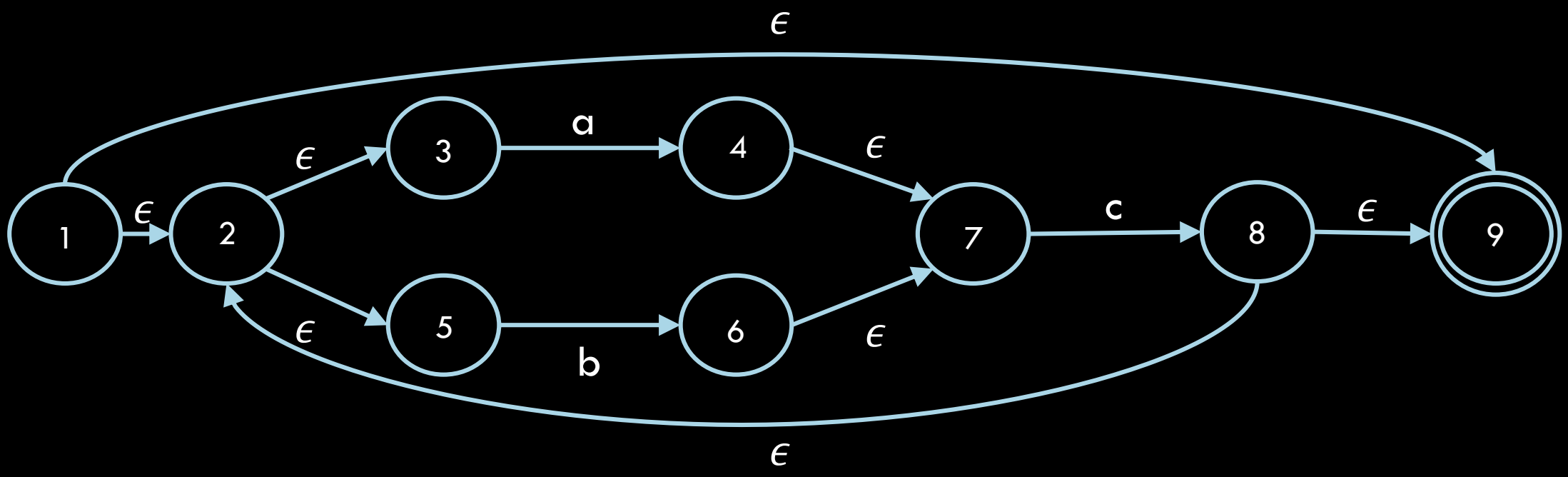
Well, NFAs admit a number of transitions from a single state for the same input. For example, in the last NFA, we can move to a state that admits ‘a’ or a state that admits ‘b’ on an empty string.

To implement an NFA, we’d need to explore all possible transitions to find the right path.

Instead, we implement a DFA which, for a given input, moves to a state that represents the *set of states* we could arrive at in an equivalent NFA.

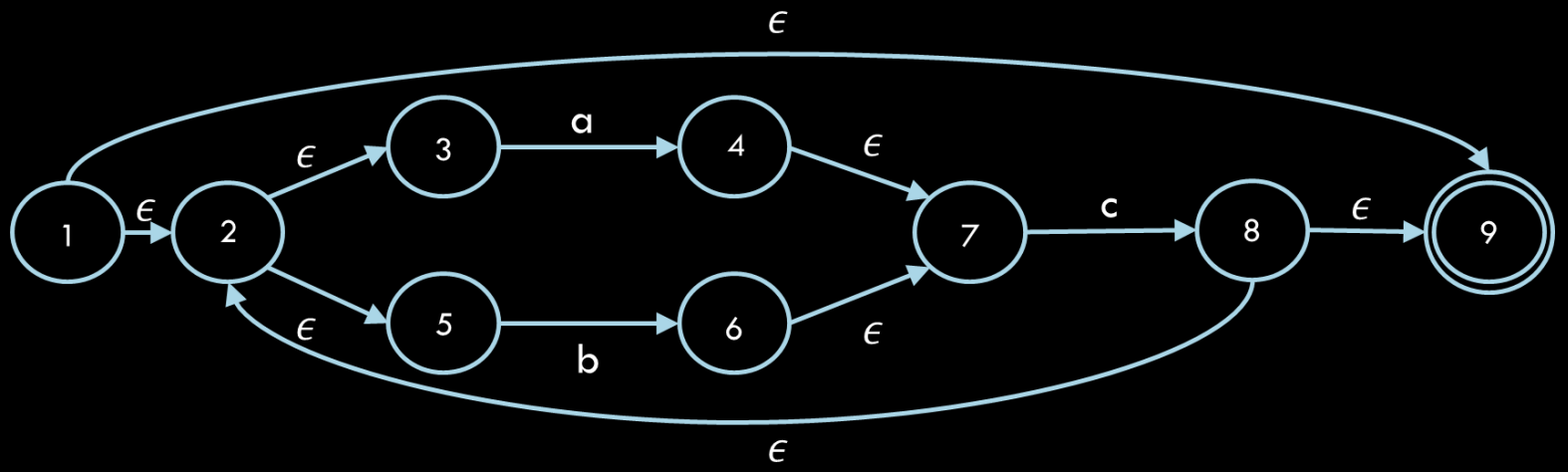
FROM NFAS TO DFAS

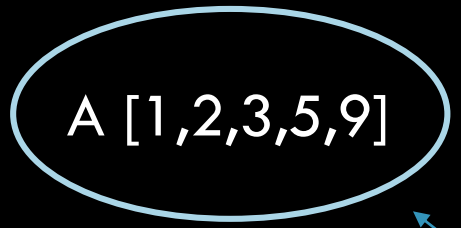
Let's look at our NFA again. Notice we've added labels to identify individual states.



FROM NFAS TO DFAS

Before consuming any input, we could be in the state 1. But we could also be in the states 2, 3, 5 and 9 via epsilon transitions.

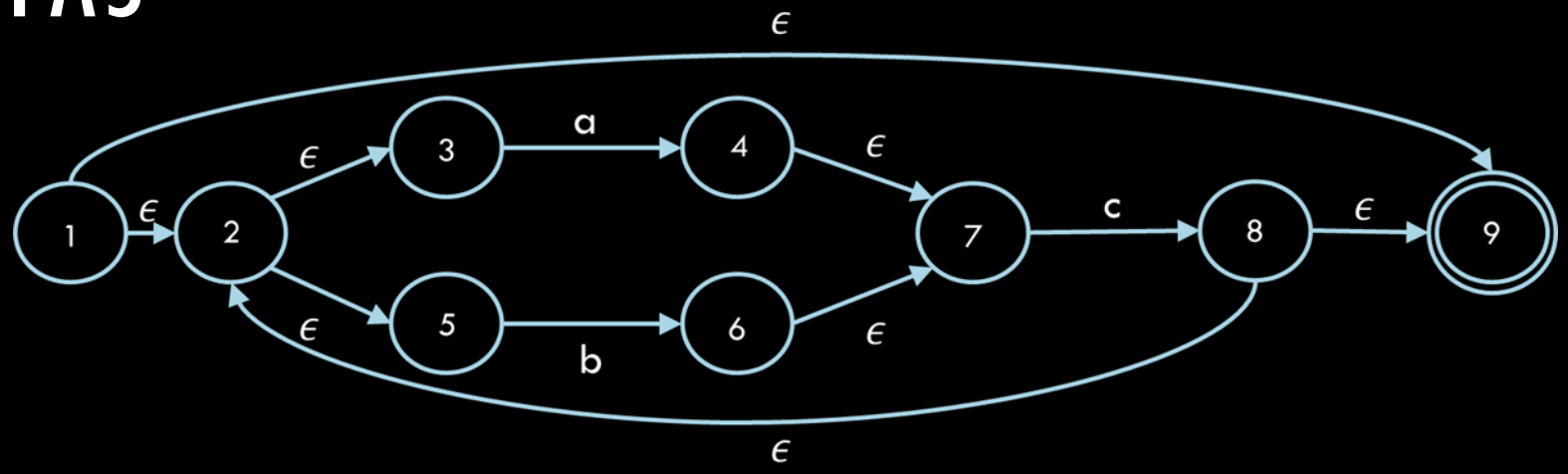


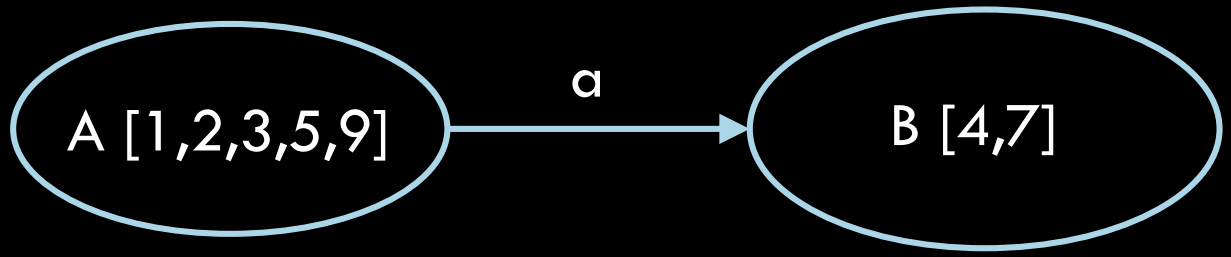


Lets make a starting state called A which is the set of all possible starting states.

FROM NFAS TO DFAS

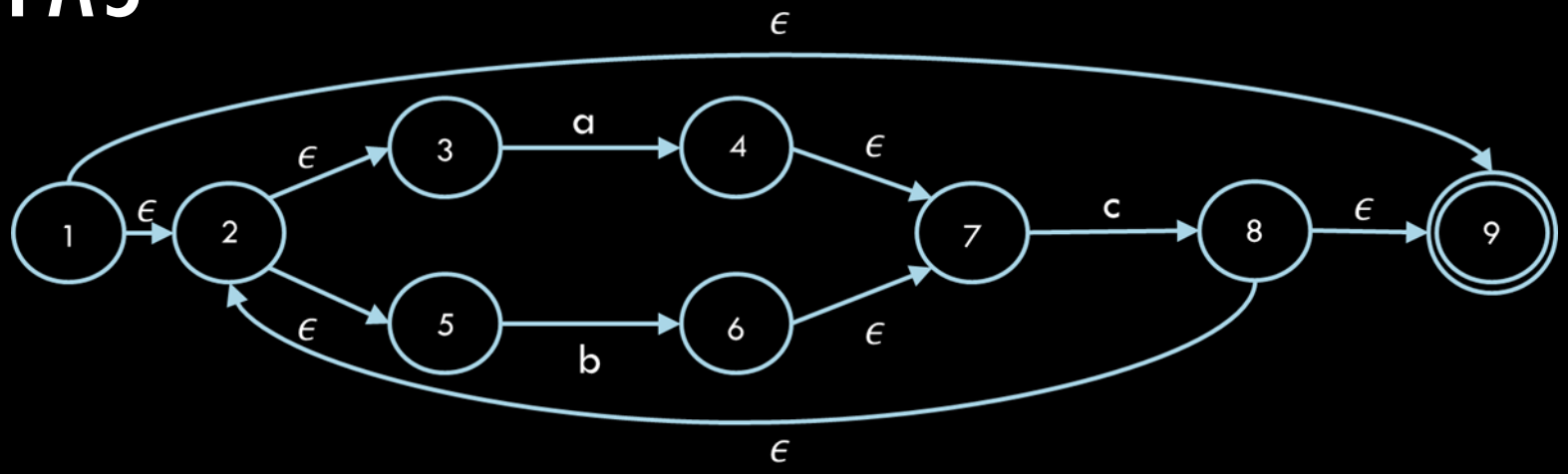
If we receive an input ‘a’, from A we could transition to 4 or 7 (via epsilon transitions).

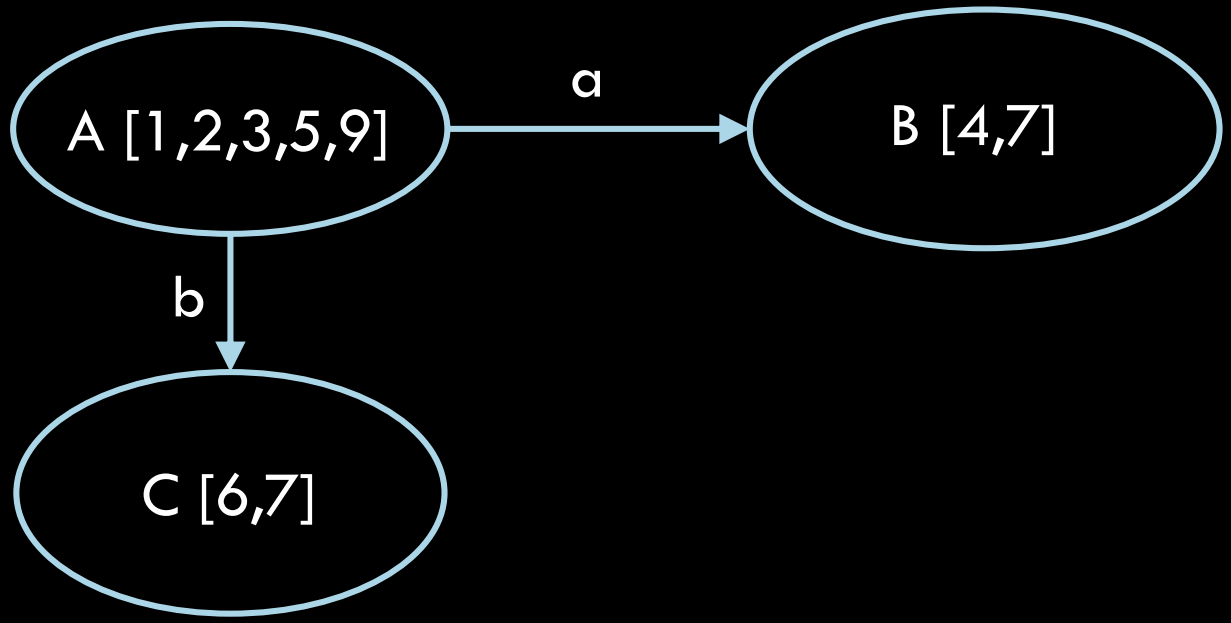




FROM NFAS TO DFAS

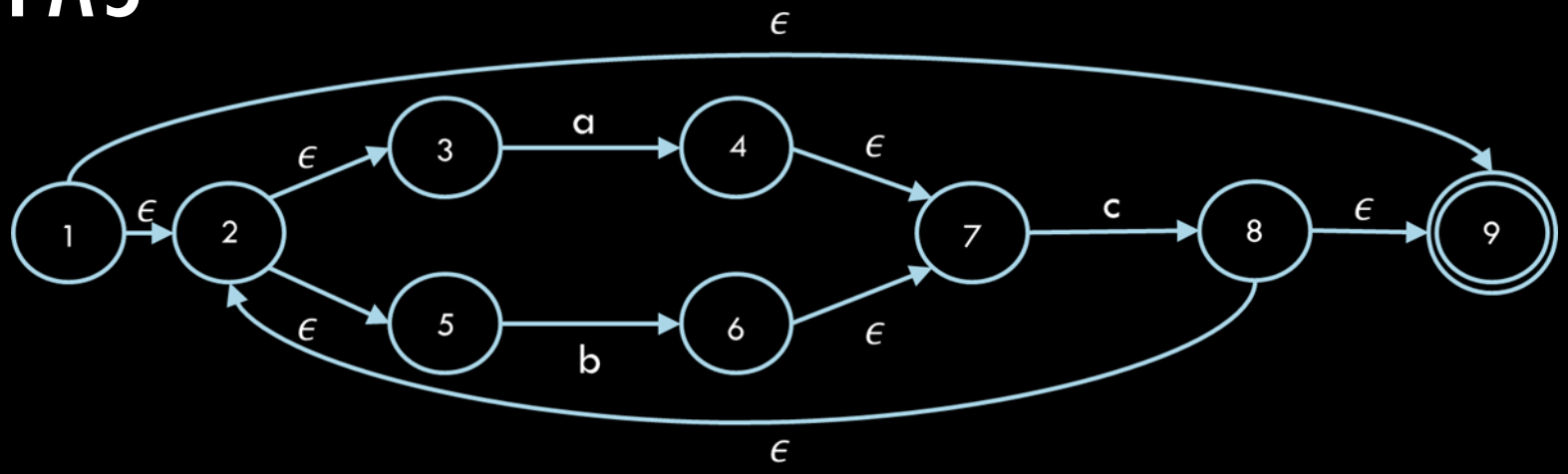
If we receive an input ‘b’, from A we could transition to 6 or 7 (via epsilon transitions).

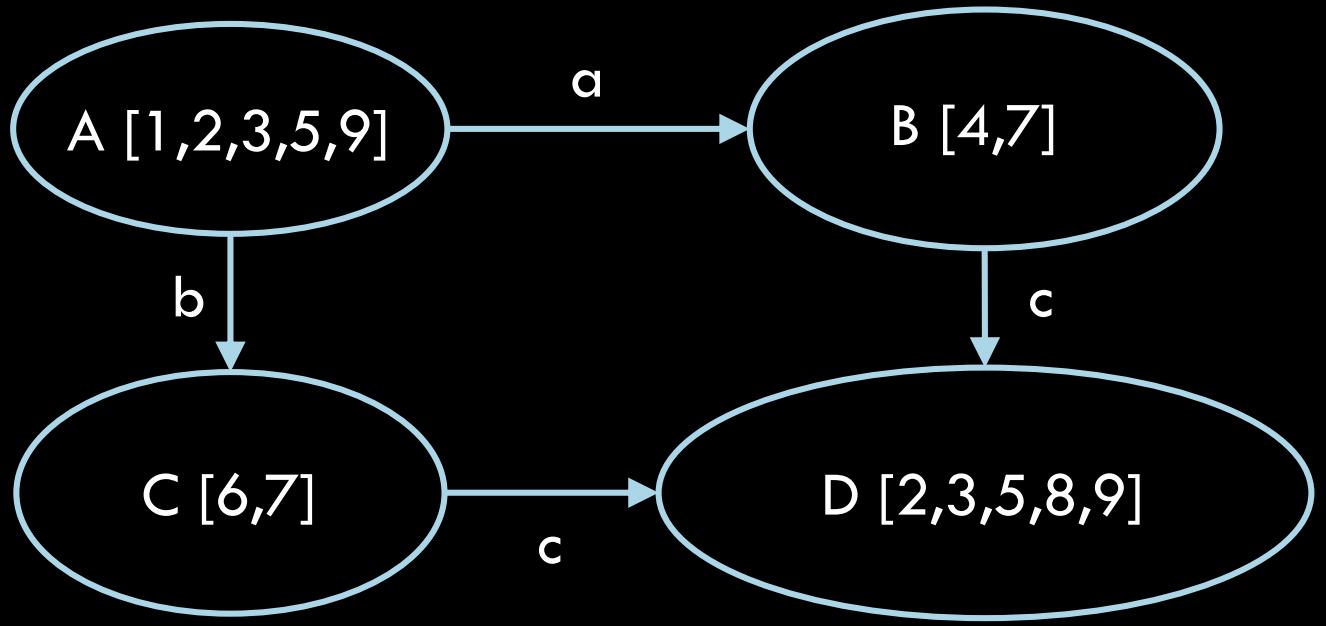




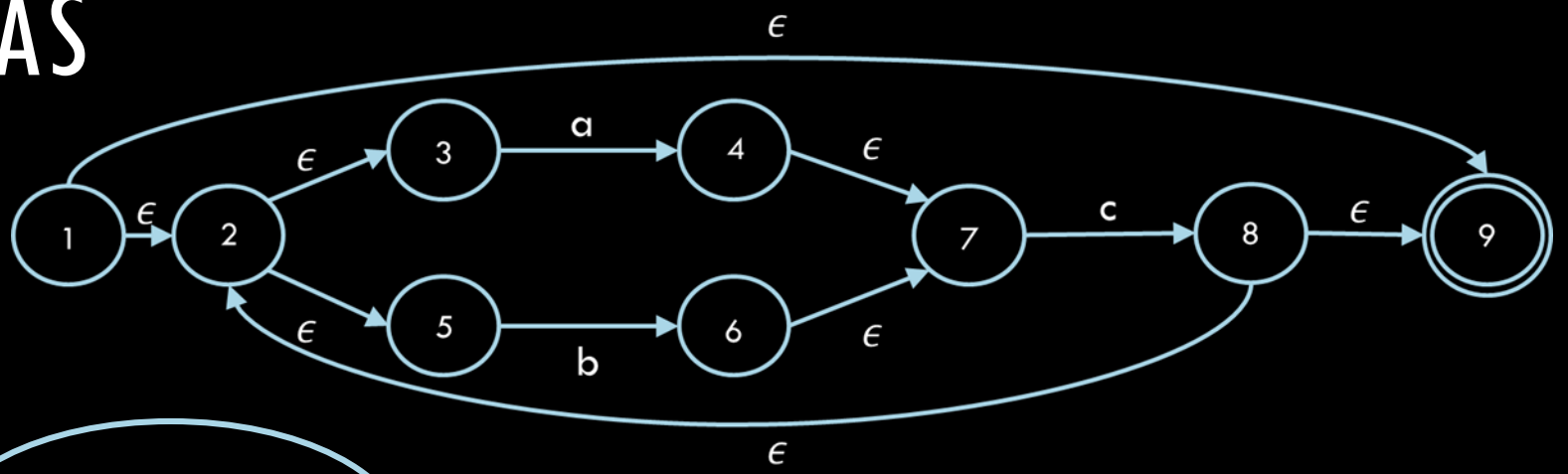
FROM NFAS TO DFAS

If we receive an input ‘c’,
we cannot go anywhere from A, but
both B and C can go to 2,3,5,8,9.

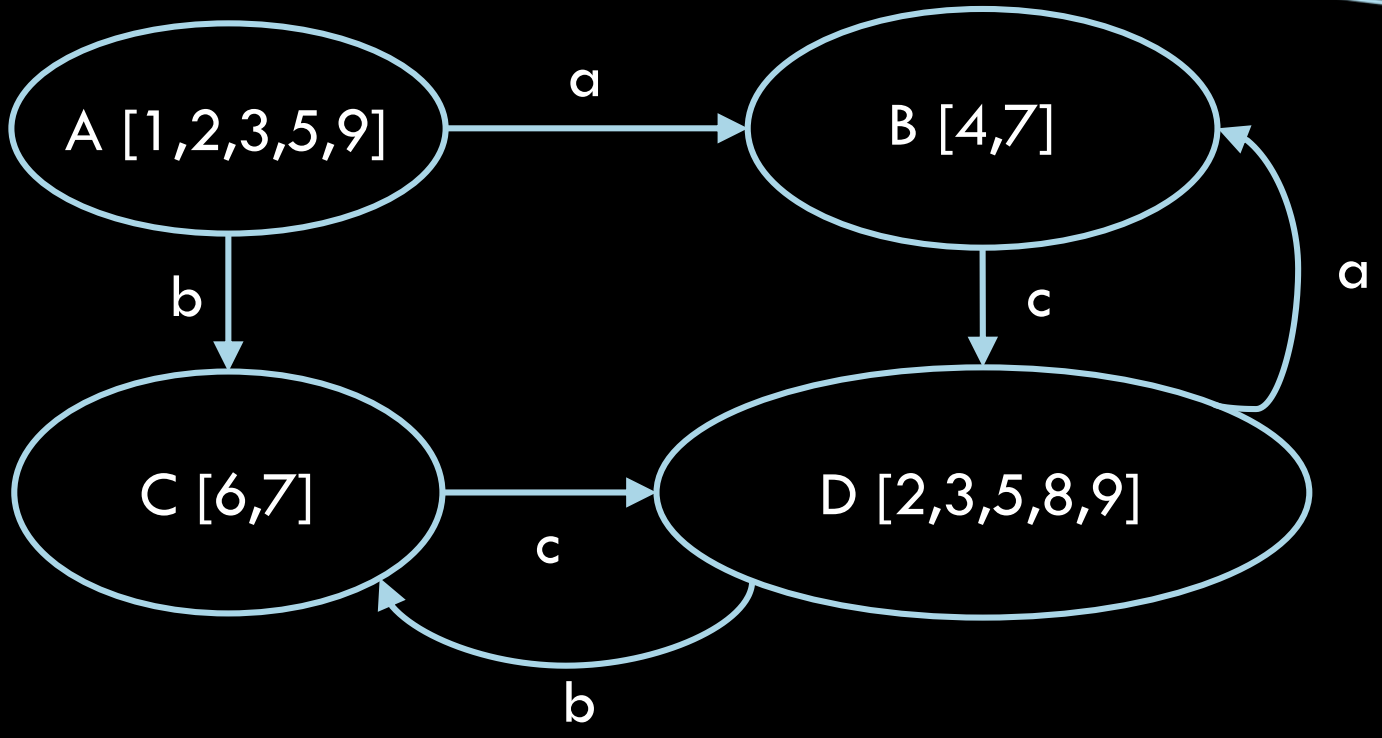




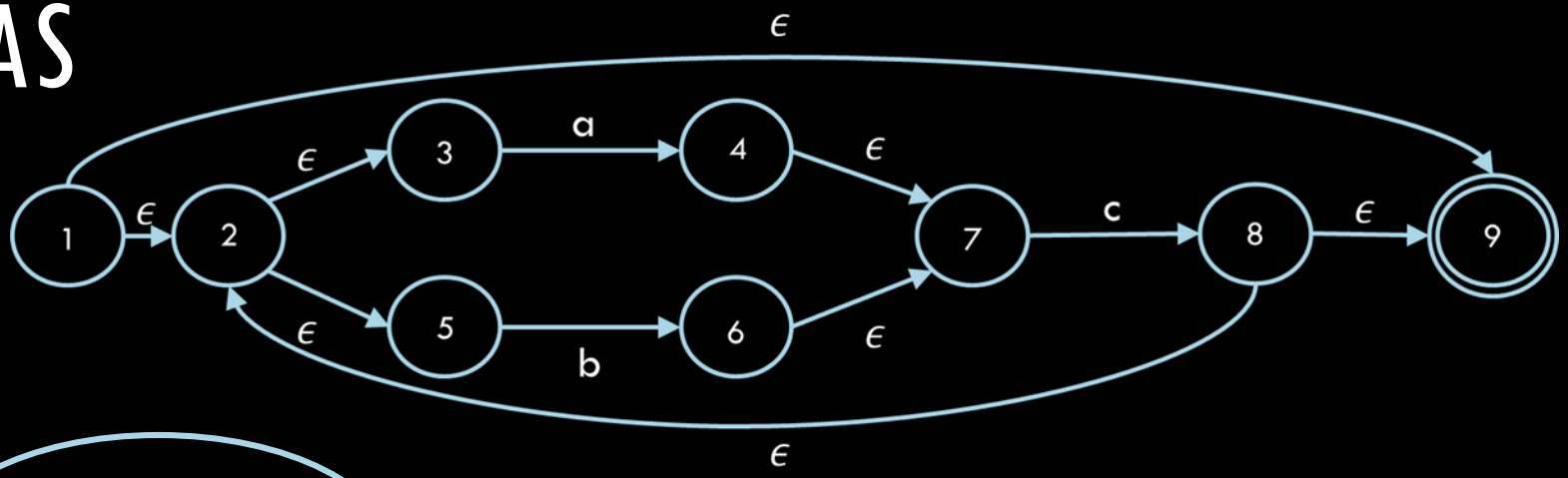
FROM NFAS TO DFAS



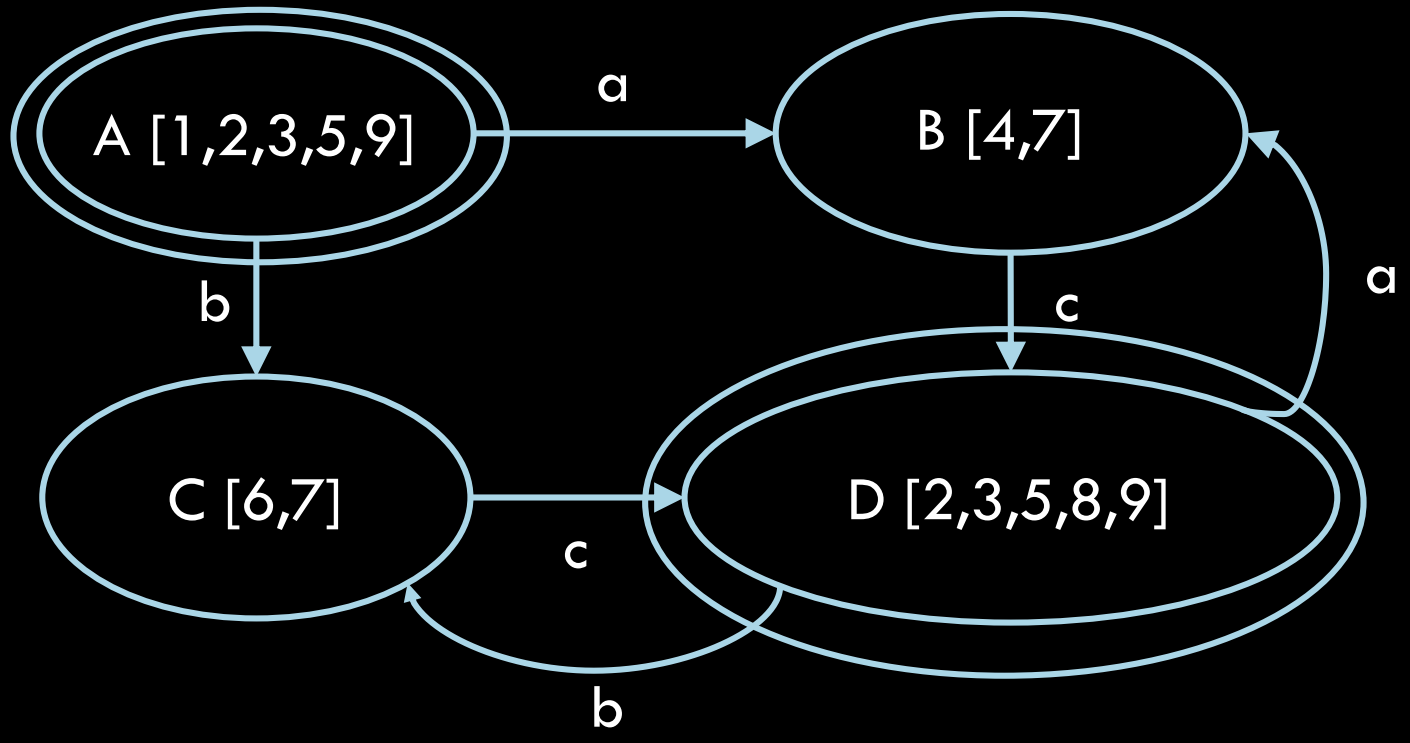
From D, we can go to B on an input of ‘a’ or go to C on an input of ‘b’.



FROM NFAS TO DFAS

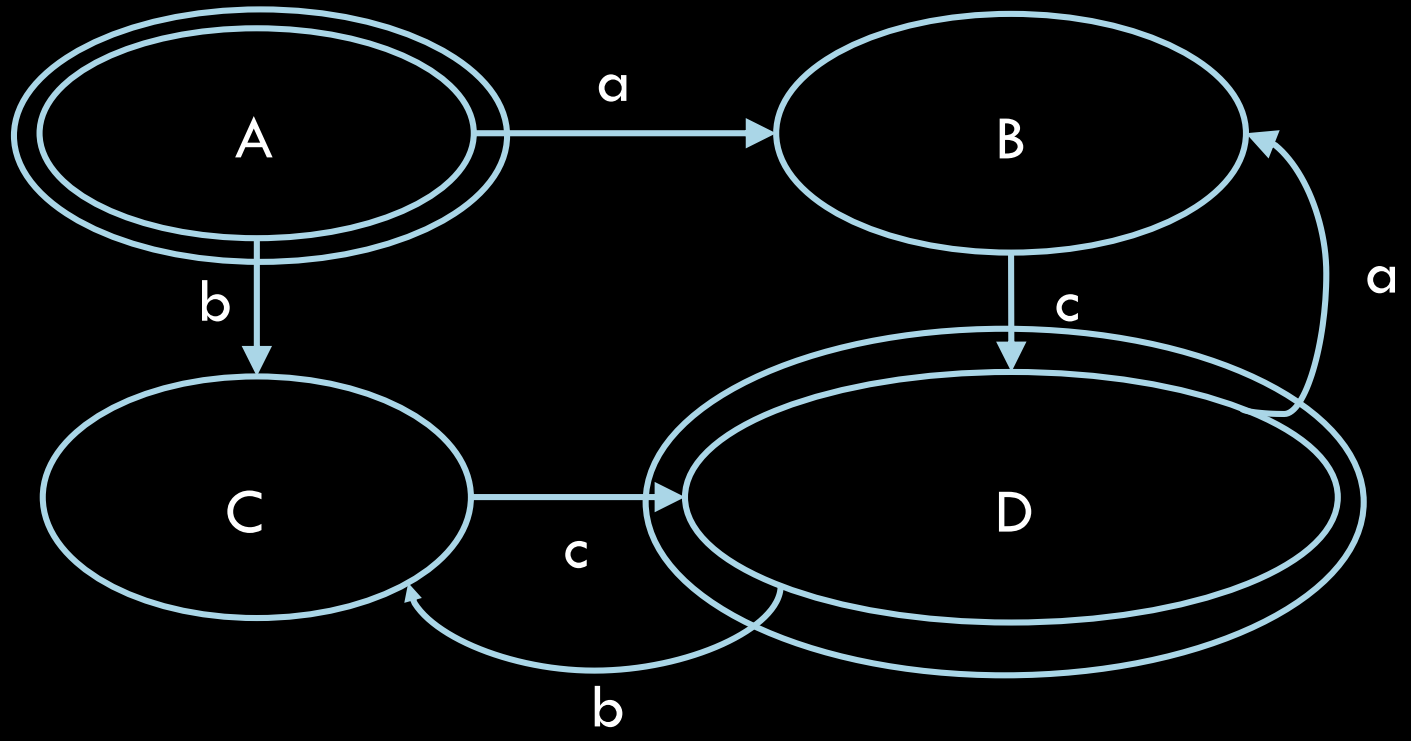


Finally, we've graphed all possible scenarios. Let's circle the final states.



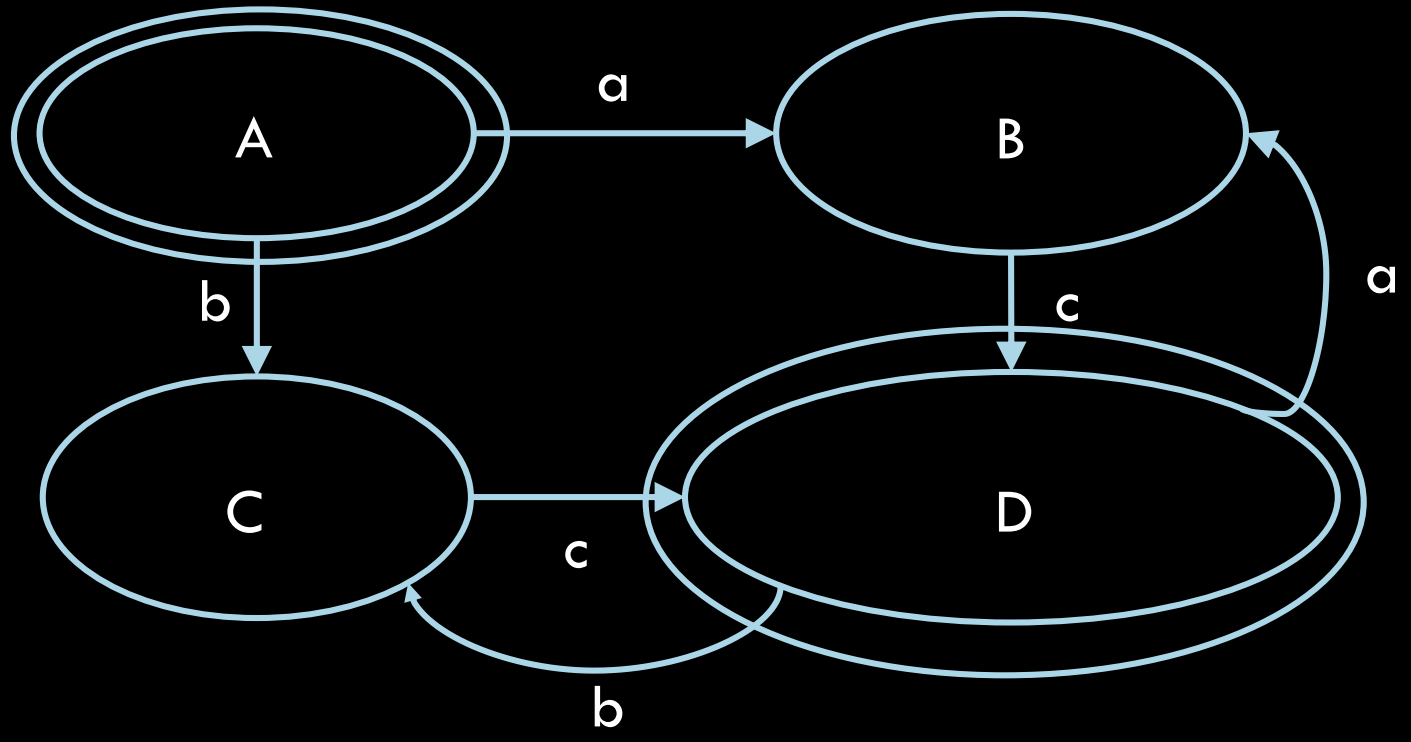
FROM NFAS TO DFAS

Here's our final DFA. For reference, our regular expression was ((a | b) c)* which describes the set {ϵ, ac, bc, acac, acbc, bcac, bcbc, acacac….}.



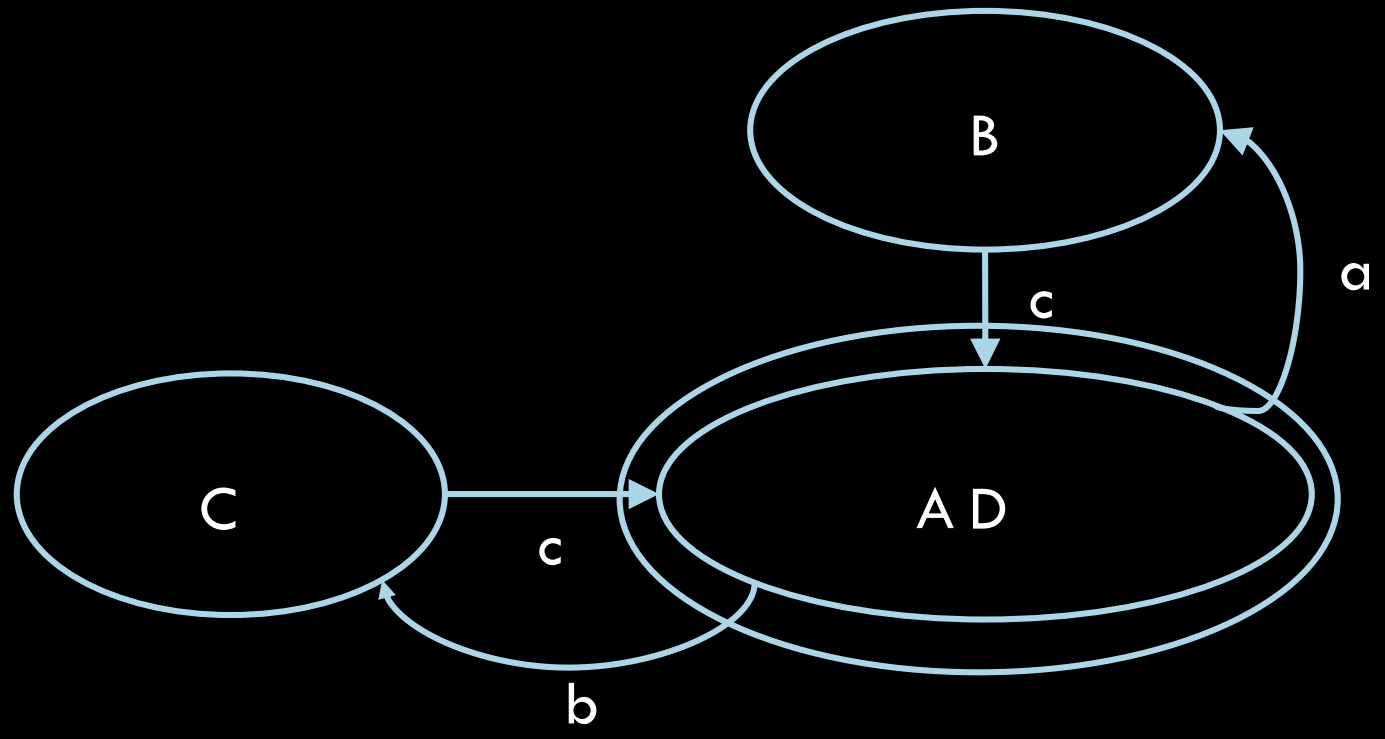
FROM NFAS TO DFAS

It is possible that the resulting DFA can be minimized further. Start by combining all non-final states together and all final states together.



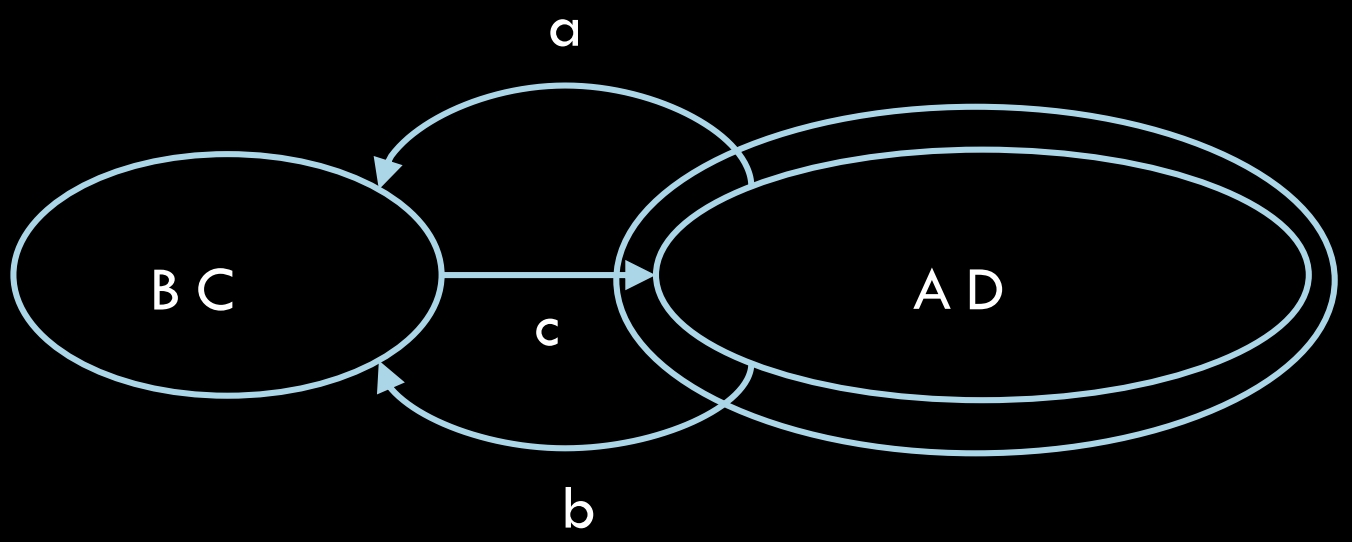
FROM NFAS TO DFAS

Now combine all of the non-final states together.



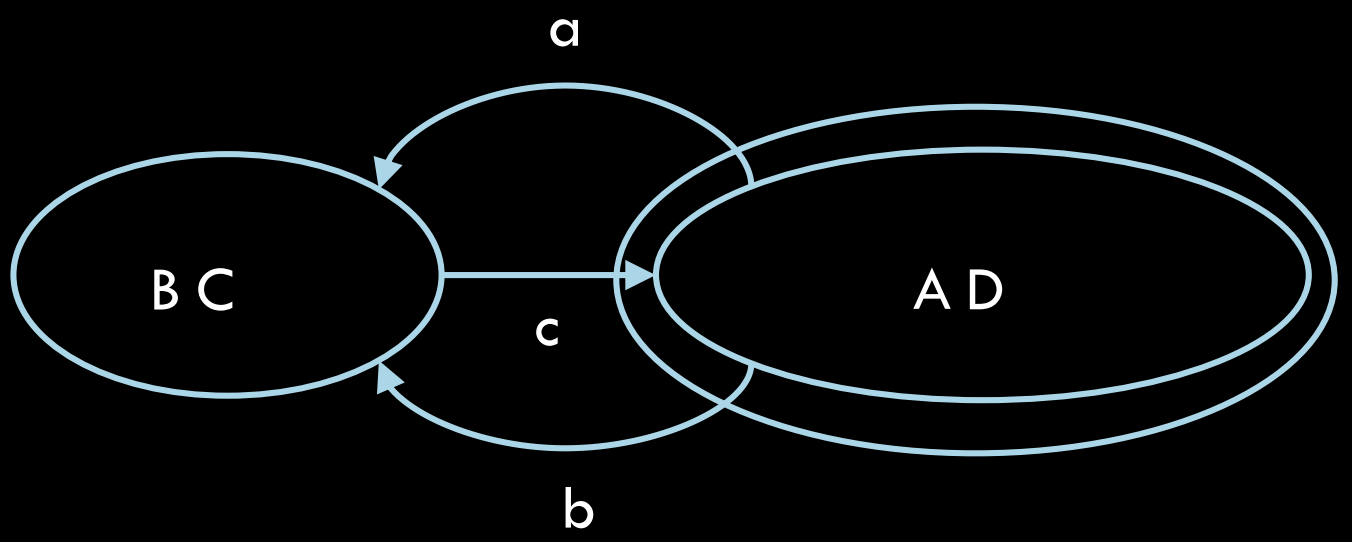
FROM NFAS TO DFAS

Now combine all of the non-final states together.



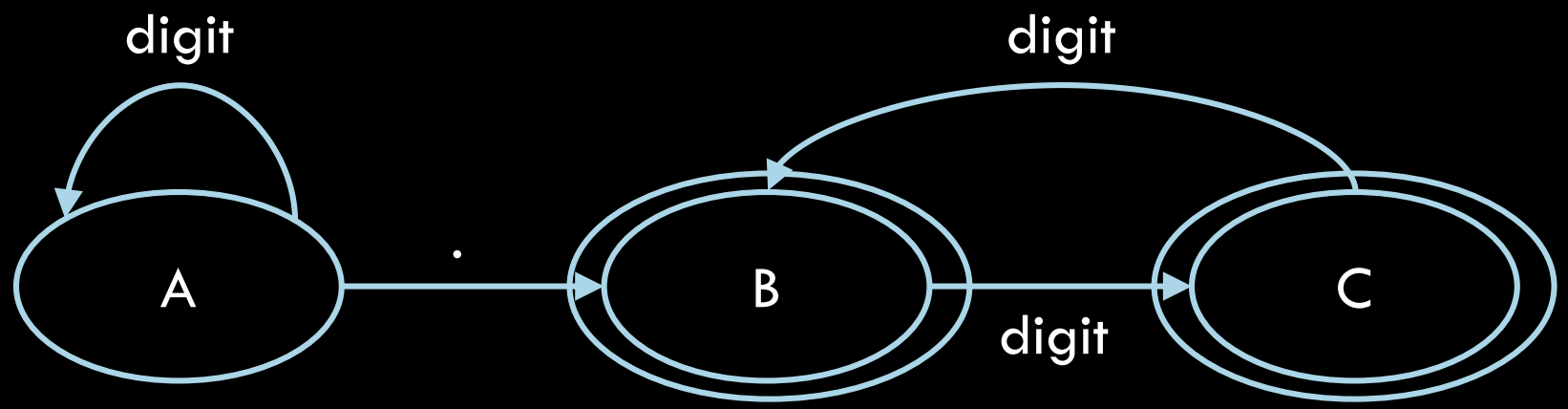
FROM NFAS TO DFAS

Note that AD is our starting state as well as an accepting state. Note that, in some cases, we may have to pull apart some states to avoid ambiguity, but here there is none.



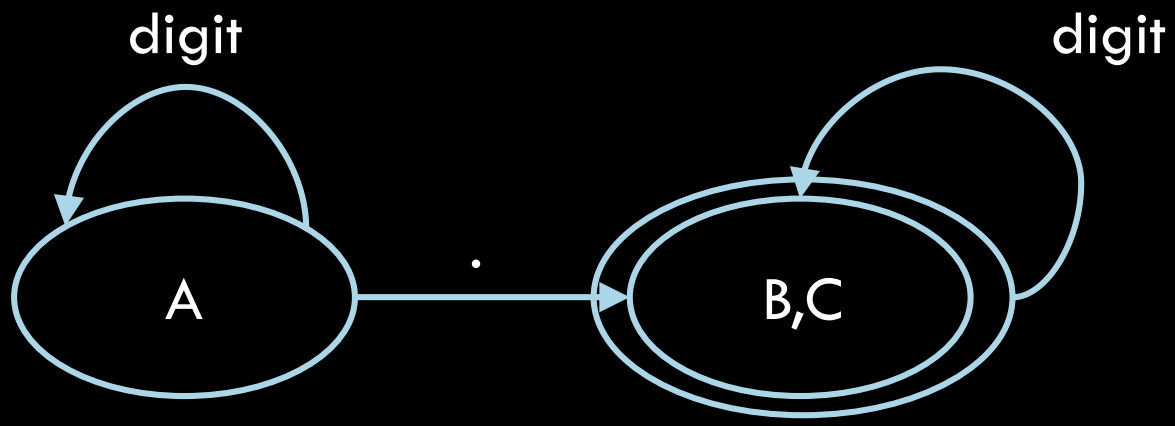
FROM NFAS TO DFAS

Here's another simple example of a DFA that can be minimized:



FROM NFAS TO DFAS

This is a simplistic example, but we can minimize this DFA in the following way:



Be careful that your minimization doesn't introduce ambiguity! Split states as needed until none remain.

FROM DFAS TO SCANNERS

Now that we have our DFA, we can actually create a scanner.

In the next lecture, we'll talk about creating a scanner that models the structure of the DFA we created.