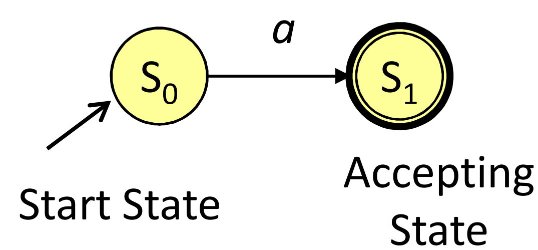
Lecture 4

Lexical Analysis II

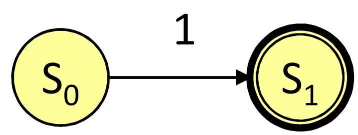
Finite Automata

- Regular Expression = specification
- Finite Automata = implementation
- A Finite Automata consists of
  - An input alphabet (Σ)
  - A finite set of states (S)
  - A start state (say n)
  - A set of accepting states [F ϵ S]
  - A set of transitions



Finite Automata

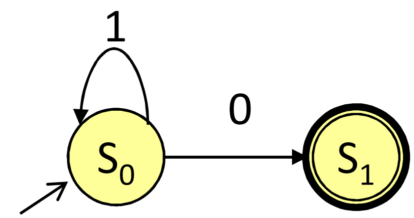
- If end of input and in accepting state → ACCEPT
- Otherwise → REJECT
- Example: A FA that accepts only "1"



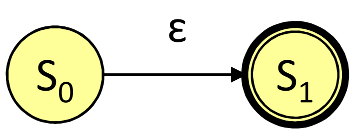
- Input = "1" → Accept but
- Input "0" or "10" → Reject

Finite Automata

- Language of a FA ≡ Set of all accepted strings
  - e.g.: Any number of 1's followed by a '0'
  - Alphabet: {0, 1}

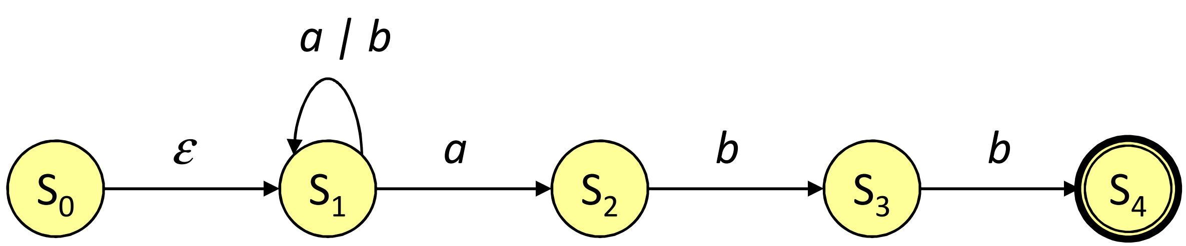


- Another kind of transition: e-*moves*
  - *Control can move to S1 on all input symbols that takes control to S0*



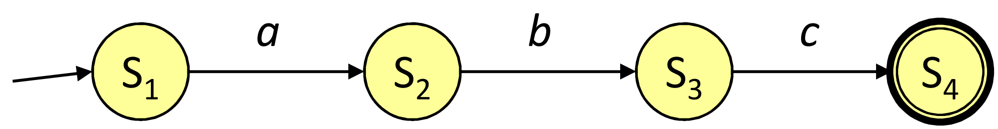
Non-deterministic Finite Automata (NFA)

- Can have multiple transition for one input for a given state
- Can have ε move
- Example: RE for *(a | b)*abb*



Deterministic Finite Automata (DFA)

- A DFA is a special case of an NFA:
  - One transition per input for a state
  - No ε move.



- NFA and DFA recognize the same set of languages (regular languages)
- DFA are faster to execute
  - There are no choice to consider
- NFA, in general, smaller. NFA can choose.
  - Exponentially smaller
- There exists space-time trade of between DRA and NFA

Motivation
Automatic Lexical Analyser Construction

To convert a specification into code:

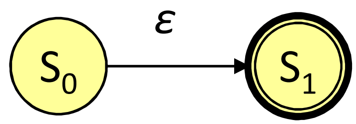
- Write down the RE for the input language.
- Convert the RE to a NFA (Thompson's construction)
- Build the DFA that simulates the NFA (subset construction)
- Shrink the DFA (Hopcroft's algorithm)

  (for the curious: there is a full cycle - DFA to RE construction is all pairs, all paths)

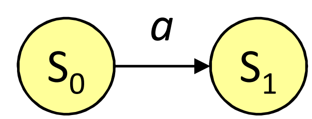
Lexical analyser generators:

- lex or flex work along these lines.
- Algorithms are well-known and understood.
- Key issue is the interface to parser.

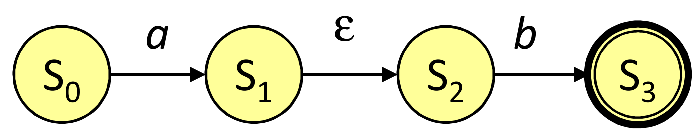
RE to NFA using Thompson's construction



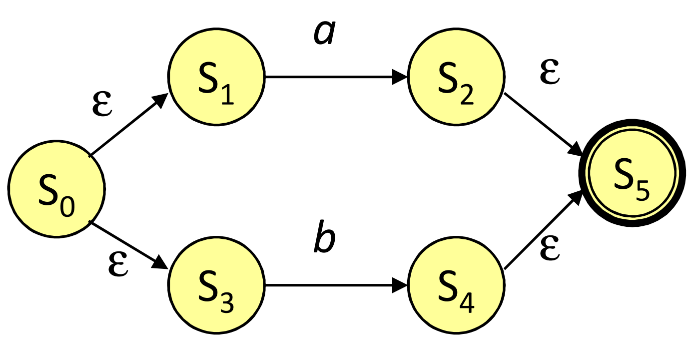
NFA for *ε*



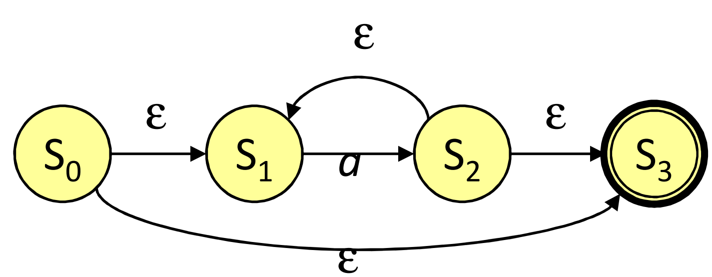
NFA for *a*



NFA for ab (concatenation)



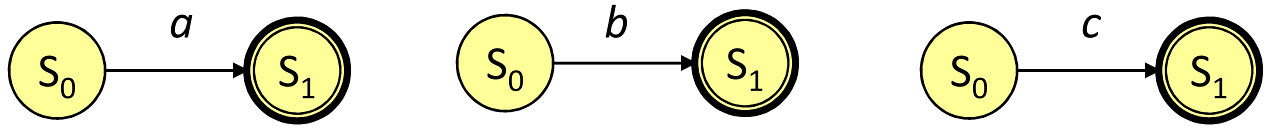
NFA for *a* | *b (union)*

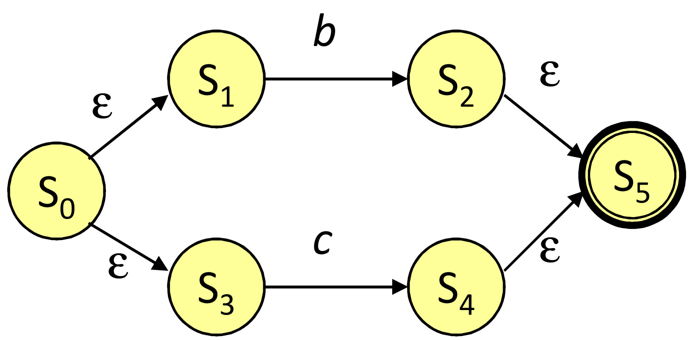


NFA for a (iteration)*

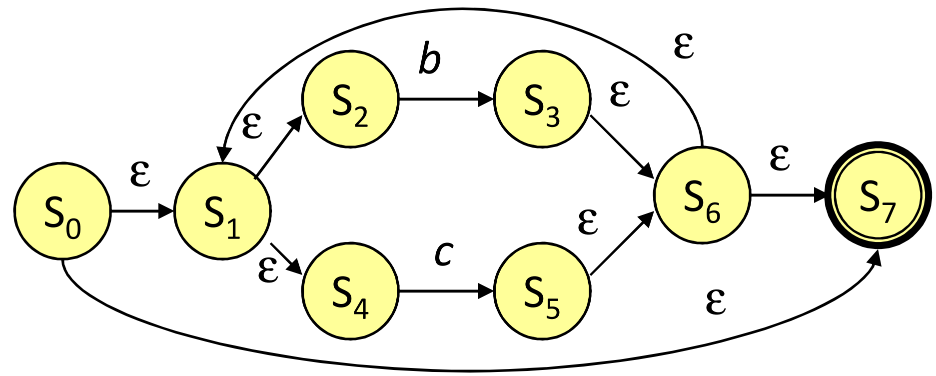
Example: Construct the NFA of *a (b|c)**

First: NFAs for *a, b, c*



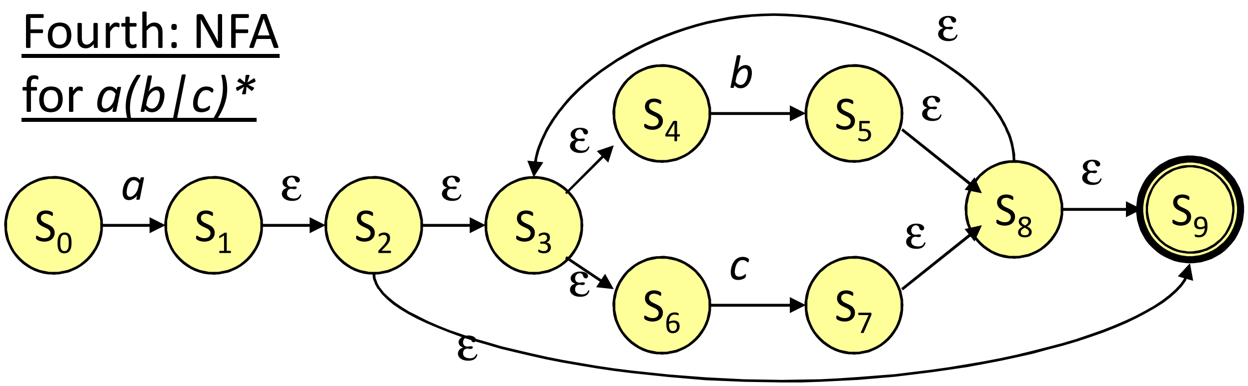


Second: NFA for *b|c*

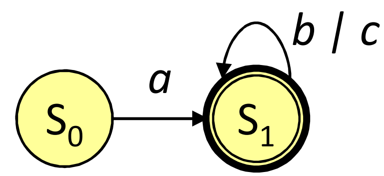


Third: NFA for *(b|c)**

Fourth: NFA for *a(b|c)**



Of course, a human would design a simpler one… But, we can automate production of the complex one...

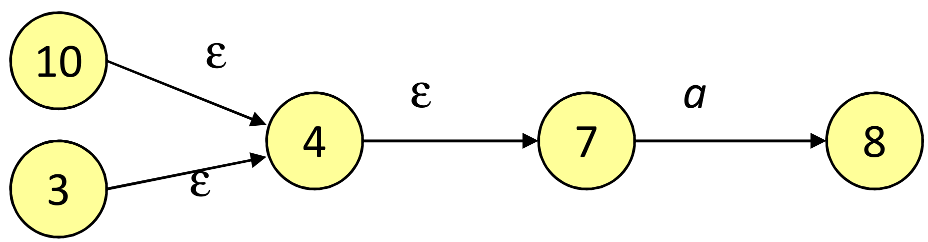


NFA to DFA: two key functions

- **move(s_i,a):** the (union of the) set of states to which there is a transition on input symbol **a** from state $\mathbf{s_i}$
- **ε-closure(s_i):** the (union of the) set of states reachable by ε from $\mathbf{s_i}$.

Example (see the diagram below):

- ε-closure(3)={3,4,7}; ε-closure({3,10})={3,4,7,10};
- move(ε-closure({3,10}),*a*)=8;



The Algorithm:

- start with the ε-closure of s_0 from NFA.
- Do for each unmarked state until there are no unmarked states:
  - for each symbol take their ε-closure(move(state,symbol))

NFA to DFA with subset construction

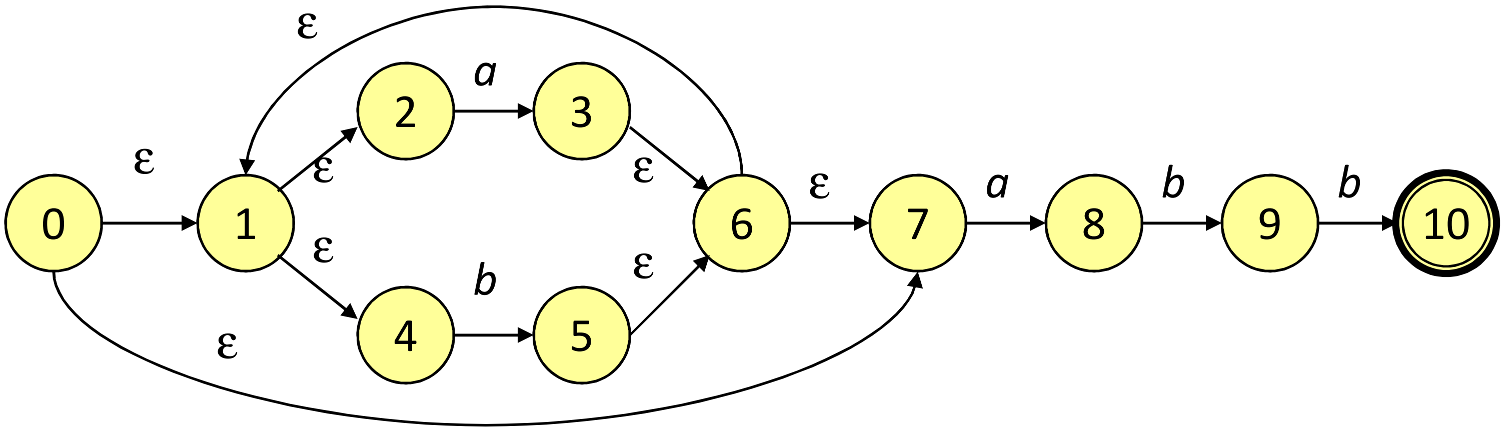
Initially, ε-closure is the only state in Dstates and it is unmarked.

```
while there is an unmarked state T in Dstates
    mark T
    for each input symbol a
        U:=ε-closure(move(T,a))
        if U is not in Dstates then add U as unmarked to Dstates
        Dtable[T,a]:=U
```

- Dstates (set of states for DFA) and Dtable form the DFA.
- Each state of DFA corresponds to a set of NFA states that NFA could be in after reading some sequences of input symbols.
- This is a fixed-point computation.

It sounds more complex than it actually is!

Example: NFA for *(a | b)*abb*

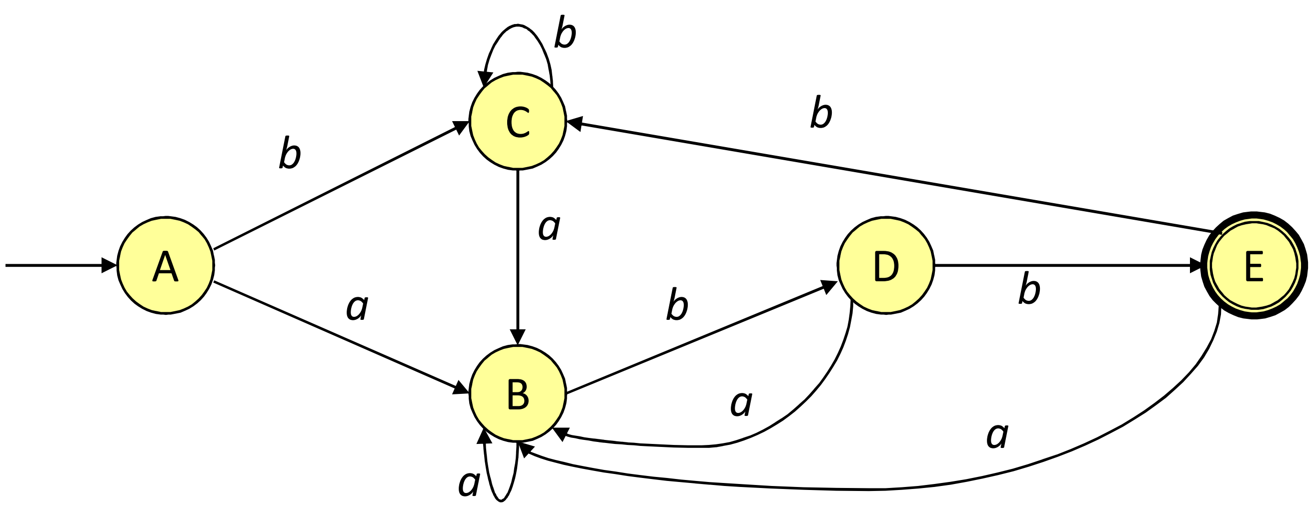


- A=ε-closure(0)={0,1,2,4,7}
- for each input symbol (that is, *a* and *b*):
  - B=ε-closure(move(A,*a*))=ε-closure({3,8})={1,2,3,4,6,7,8}
  - C=ε-closure(move(A,*b*))=ε-closure({5})={1,2,4,5,6,7}
  - Dtable[A,*a*]=B; Dtable[A,*b*]=C
- B and C are unmarked. Repeating the above we end up with:
  - C={1,2,4,5,6,7}; D={1,2,4,5,6,7,9}; E={1,2,4,5,6,7,10}; and
  - Dtable[B,*a*]=B; Dtable[B,*b*]=D; Dtable[C,*a*]=B; Dtable[C,*b*]=C; Dtable[D,*a*]=B; Dtable[D,*b*]=E; Dtable[E,*a*]=B; Dtable[E,*b*]=C; no more unmarked sets at this point!

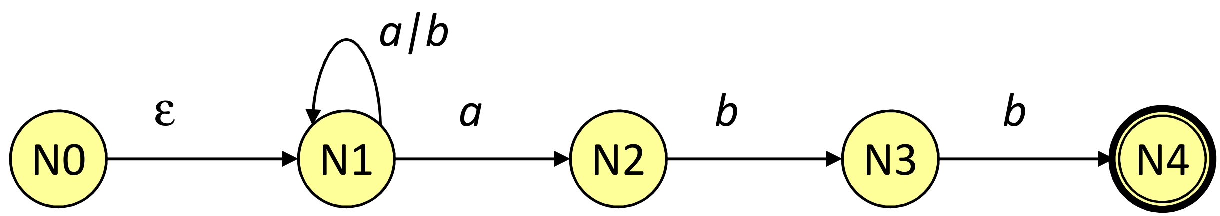
Result of applying subset construction

Transition table:

state	*a*	*b*
A	B	C
B	B	D
C	B	C
D	B	E
E(final)	B	C



Another NFA version of the same RE



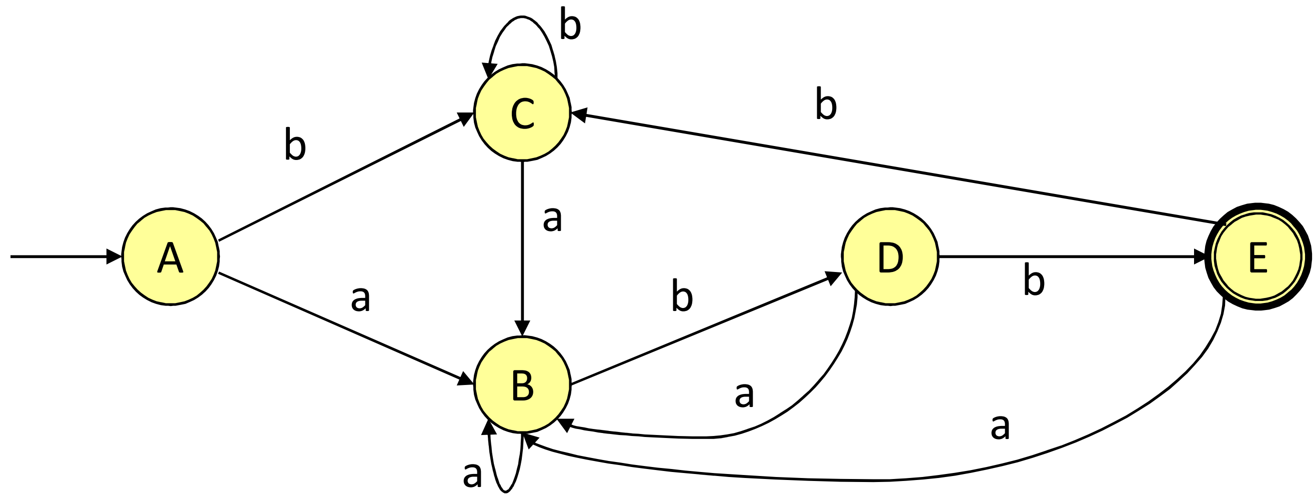
Apply the subset construction algorithm:

Iteration	State	Contains	ε-closure(move(s,*a*))	ε-closure(move(s,*b*))
0	A	N0,N1	N1,N2	N1
1	B	N1,N2	N1,N2	N1,N3
	C	N1	N1,N2	N1
2	D	N1,N3	N1,N2	N1,N4
3	E	N1,N4	N1,N2	N1

Note:

- iteration 3 adds nothing new, so the algorithm stops.
- state E contains N4 (final state)

DFA Minimisation: the problem



- Problem: can we minimize the number of states?
- Answer: yes, if we can find groups of states where, for each input symbol, every state of such a group will have transitions to the same group.

DFA minimisation: the algorithm

(Hopcroft's algorithm: simple version)

Divide the states of the DFA into two groups: those containing final states and those containing non-final states.

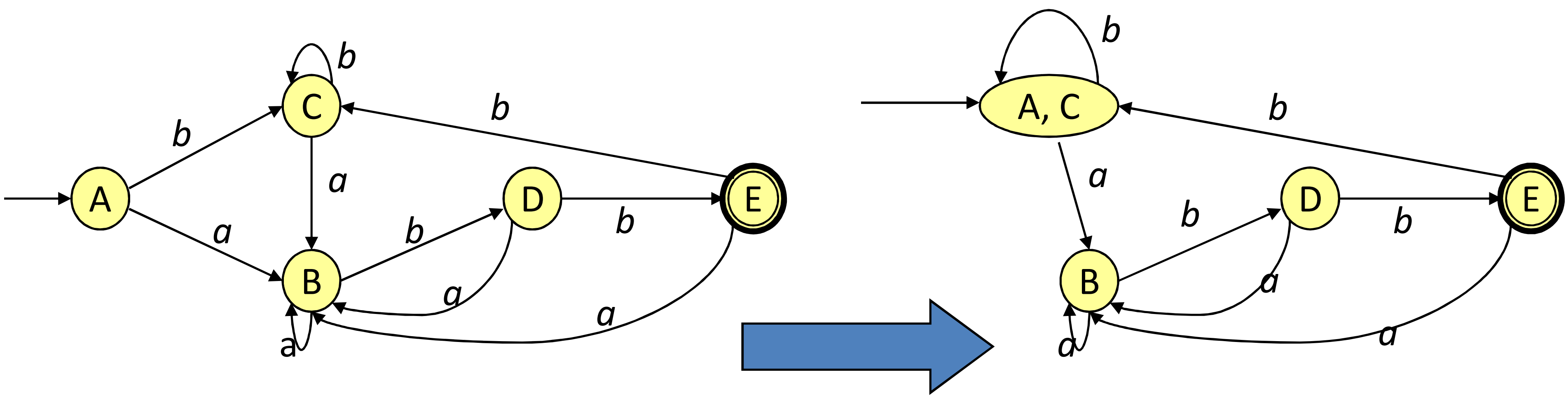
while there are group changes
  for each group
    for each input symbol
      if for any two states of the group and a given input symbol, their transitions do not lead to the same group, these states must belong to different groups.

For the curious, there is an alternative approach: create a graph in which there is an edge between each pair of states which cannot coexist in a group because of the conflict above. Then use a graph colouring algorithm to find the minimum number of colours needed so that any two nodes connected by an edge do not have the same colour (we'll examine graph colouring algorithms later on, in register allocation)

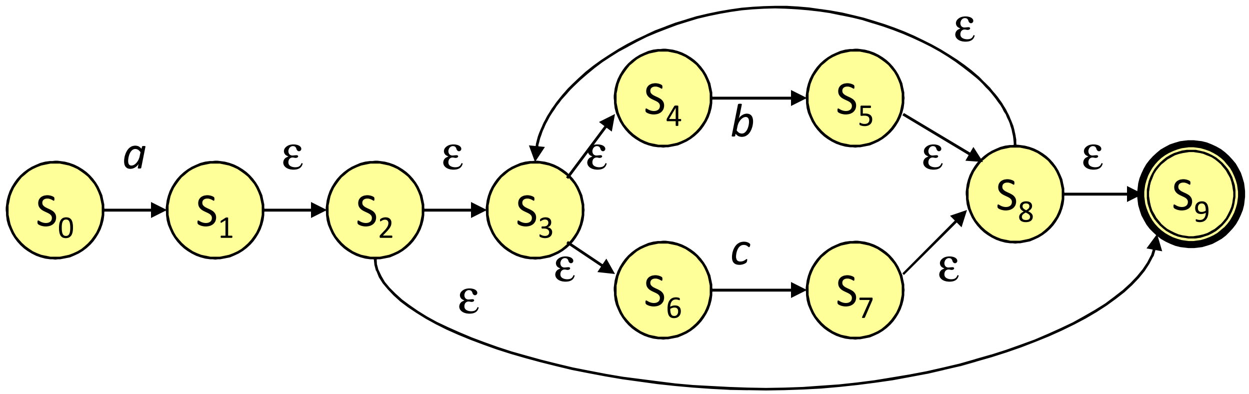
How does it work? Recall *(a | b)* abb*

Iteration	Current groups	Split on a	Split on b
0	{E}, {A,B,C,D}	None	{A,B,C}, {D}
1	{E}, {D}, {A,B,C}	None	{A,C}, {B}
2	{E}, {D}, {B}, {A, C}	None	None

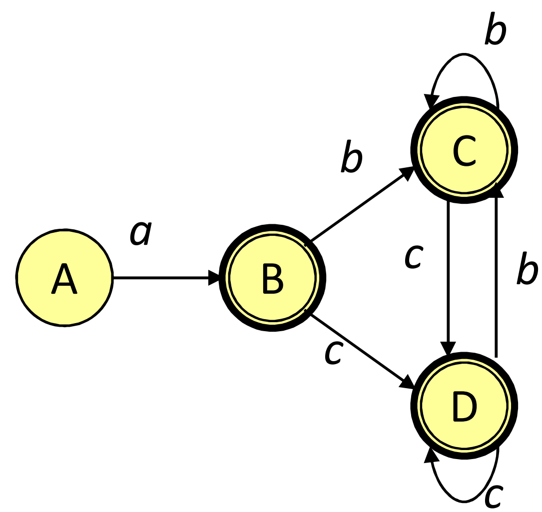
In each iteration, we consider any non-single-member groups and we consider the partitioning criterion for all pairs of states in the group.



From NFA to minimized DFA: recall *a (b | c)**



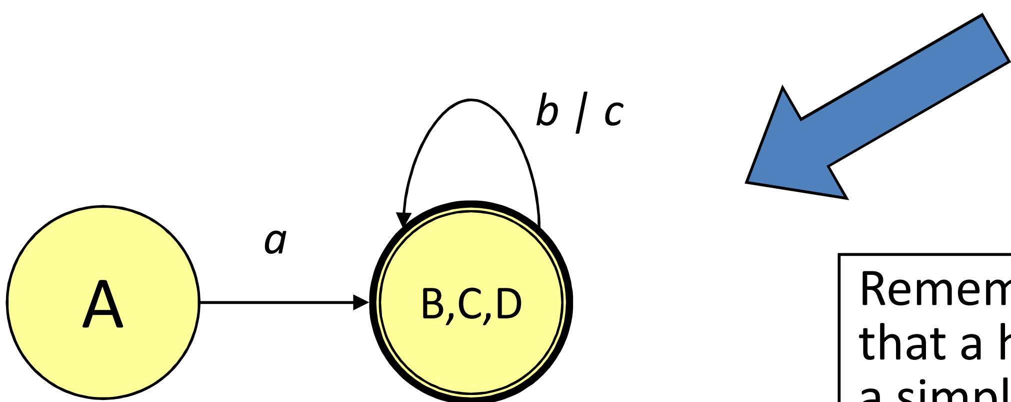
DFA states	NFA states	ε-closure(move(s,*))		
		a	*b*	*c*
A	S0	S1,S2,S3, S4,S6,S9	None	None
B	S1,S2,S3, S4,S6,S9	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6
C	S5,S8,S9, S3,S4,S6	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6
D	S7,S8,S9, S3,S4,S6	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6

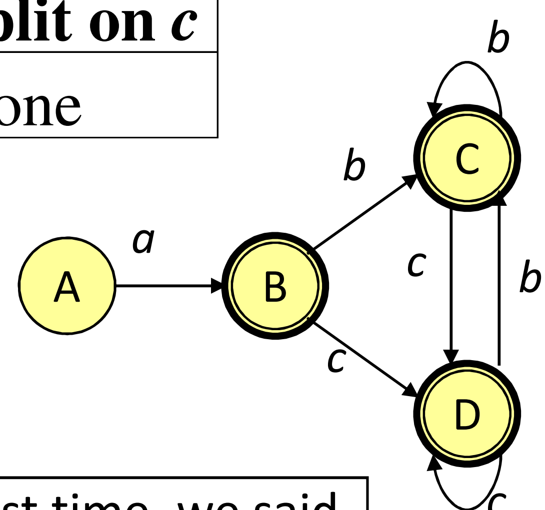


DFA minimisation: recall *a (b | c)**

Apply the minimisation algorithm to produce the minimal DFA:

	Current groups	Split on *a*	Split on *b*	Split on *c*
0	{B, C, D} {A}	None	None	None





Remember, last time, we said that a human could construct a simpler automaton than Thompson's construction? Well, algorithms can produce the same DFA!

THANKS