The University for world-class professionals

Manchester Metropolitan University

Compilers: Constructing Finite Automata

Dr Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

Where we are…

- ~~Admin and overview~~
- **Lexical analysis**
- Parsing
- Semantic analysis
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review

Last week…

Manchester
Metropolitan
University

Counter.java | Main.java

```
package de.vogella.debug.first;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.count();
        System.out.println("We have counted " + counter.getResult());
    }

}
```

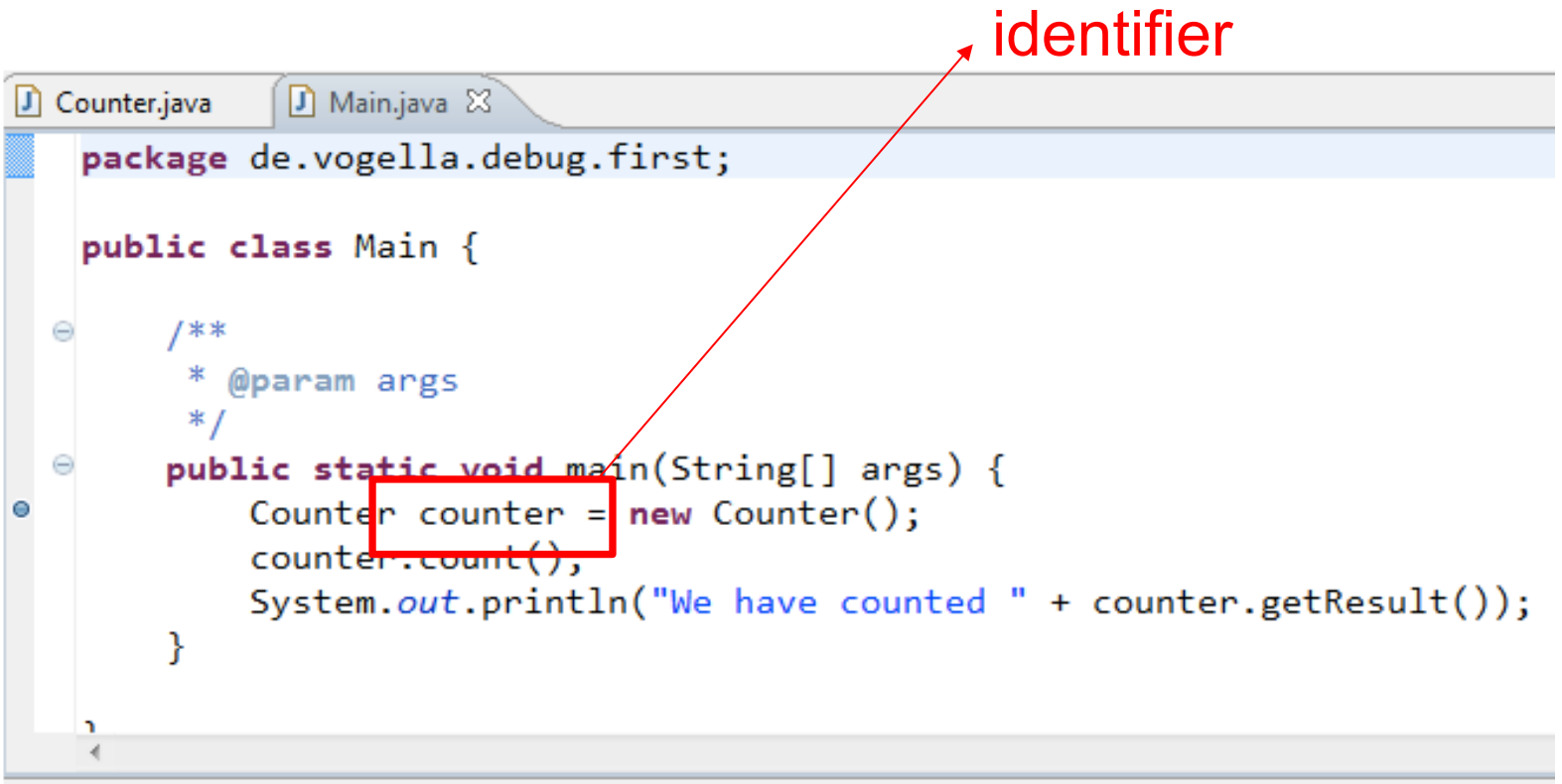
Counter.java | Main.java

```
package de.vogella.debug.first;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.count();
        System.out.println("We have counted " + counter.getResult());
    }

}
```


identifier
Counter.java
Main.java
package de.vogella.debug.first;
public class Main {
/**
* @param args
*/
public static void main(String[] args) {
Counter counter = new Counter();
counter.count();
System.out.println("We have counted " + counter.getResult());
}
}

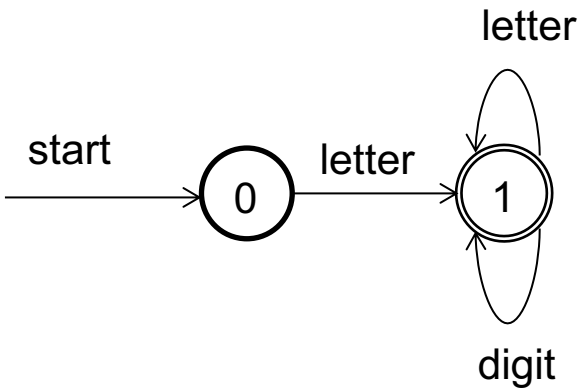
Regular expression for identifier:

Regular expression for identifier:

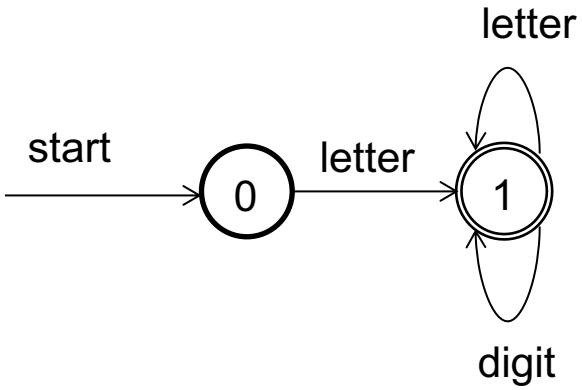
letter = [a-z | A-Z | _]
digit = [0-9]

Identifier = letter(letter | digit)*

Regular expression for a simple identifier: letter(letter | digit)*

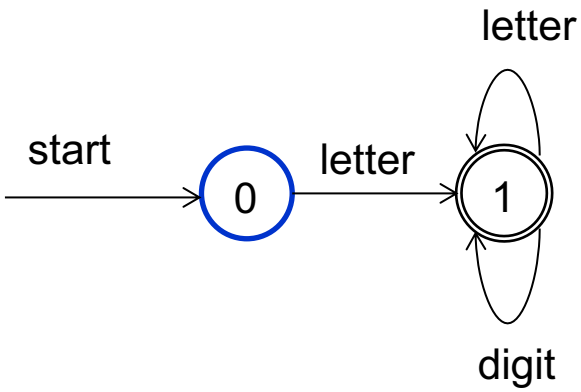


Regular expression for a simple identifier: letter(letter | digit)*



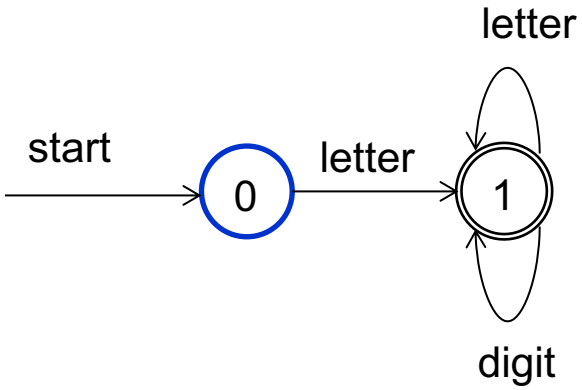
c o u n t e r

Regular expression for a simple identifier: letter(letter | digit)*



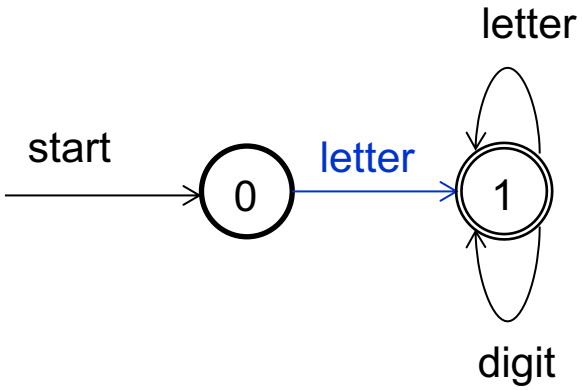
c o u n t e r

Regular expression for a simple identifier: letter(letter | digit)*



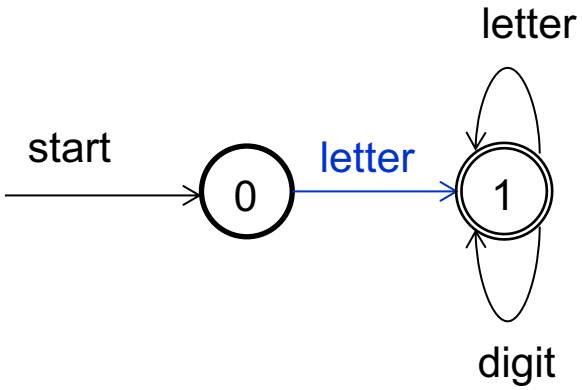
c o u n t e r

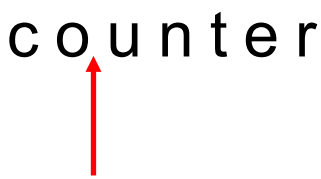
Regular expression for a simple identifier: letter(letter | digit)*



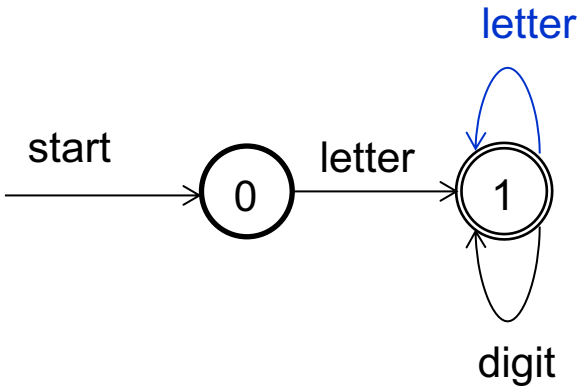
c o u n t e r

Regular expression for a simple identifier: letter(letter | digit)*



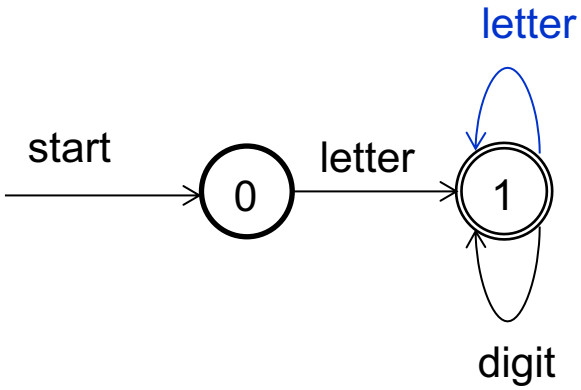


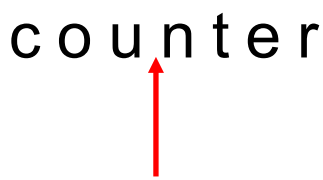
Regular expression for a simple identifier: letter(letter | digit)*



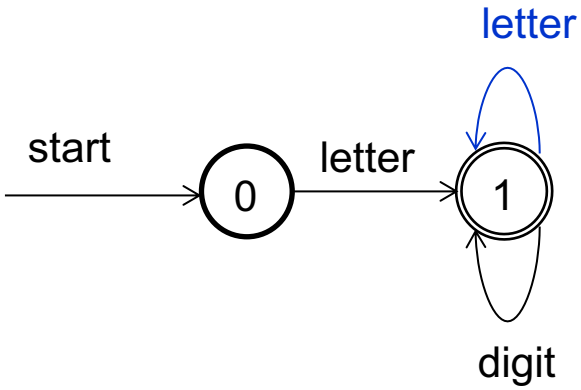
c o u n t e r

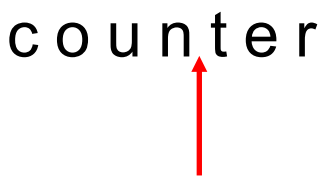
Regular expression for a simple identifier: letter(letter | digit)*



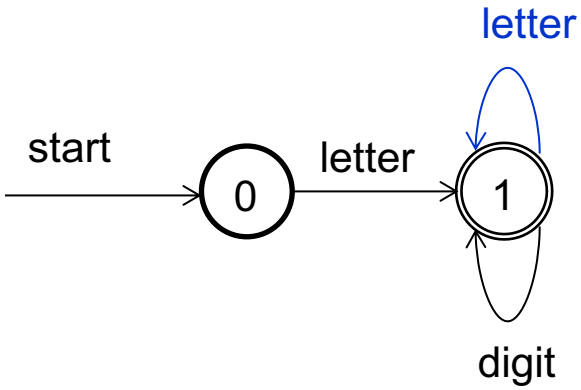


Regular expression for a simple identifier: letter(letter | digit)*

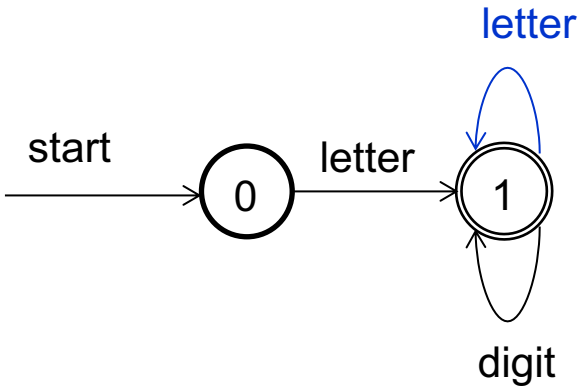




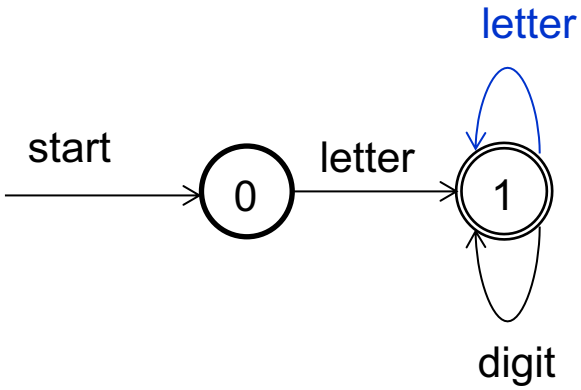
Regular expression for a simple identifier: letter(letter | digit)*



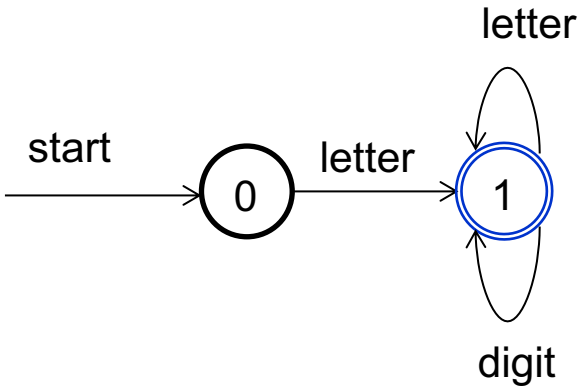
Regular expression for a simple identifier: letter(letter | digit)*



Regular expression for a simple identifier: letter(letter | digit)*



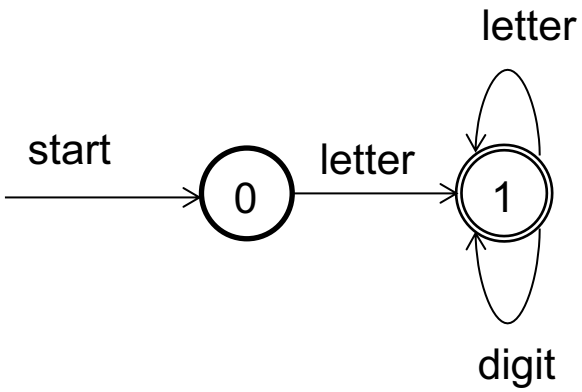
Regular expression for a simple identifier: letter(letter | digit)*



c o u n t e r

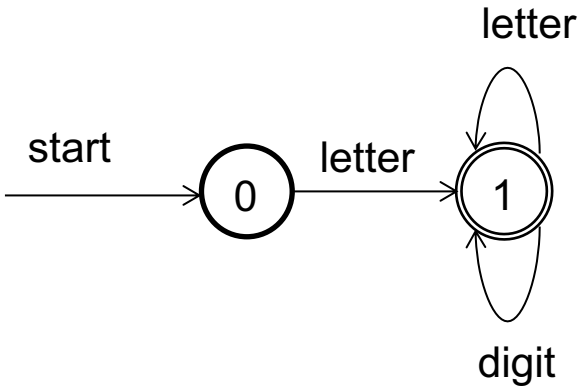
correct identifier

Regular expression for a simple identifier: letter(letter | digit)*



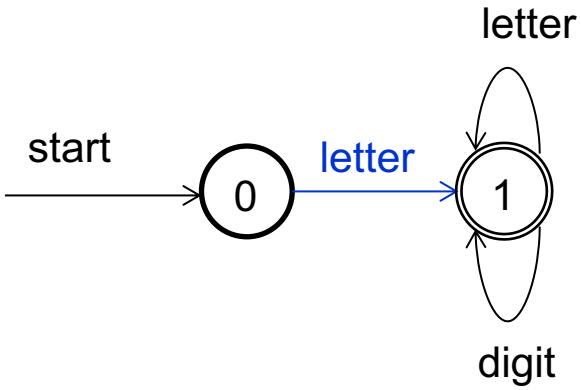
v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



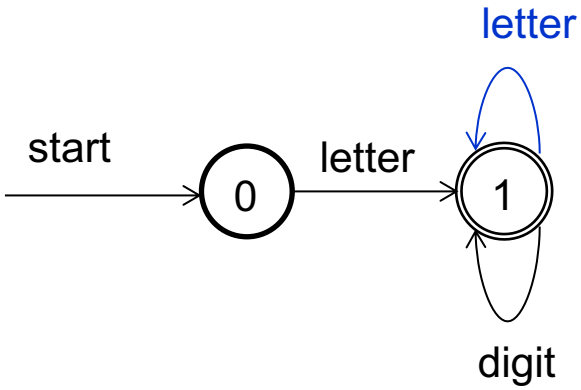
v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



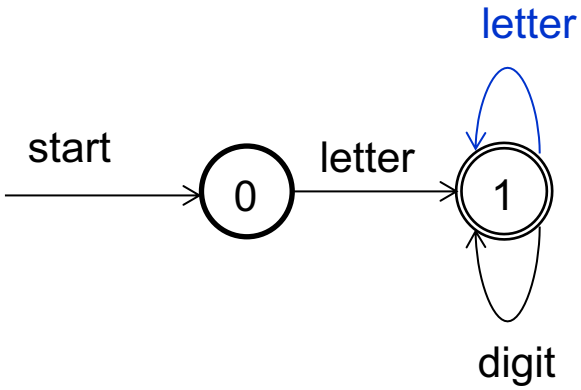
v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



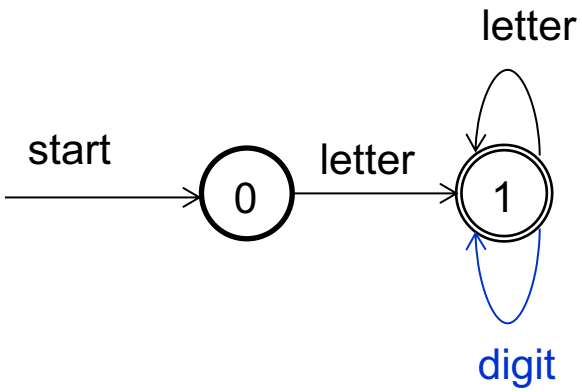
v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



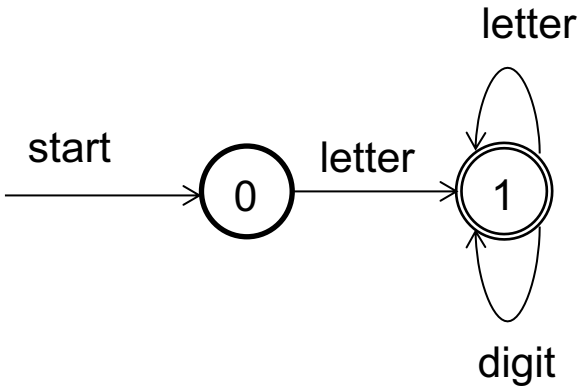
v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



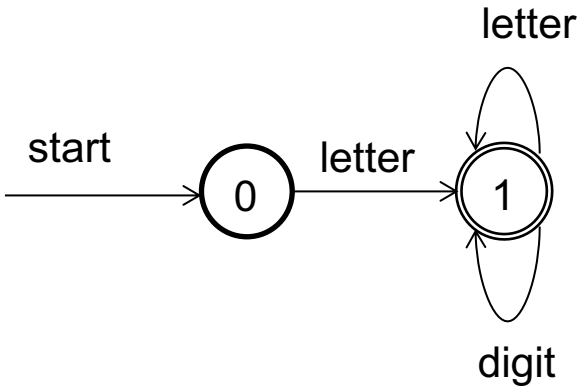
v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



v a r 1 &

Regular expression for a simple identifier: letter(letter | digit)*



v a r 1 &

incorrect identifier

Objectives for today

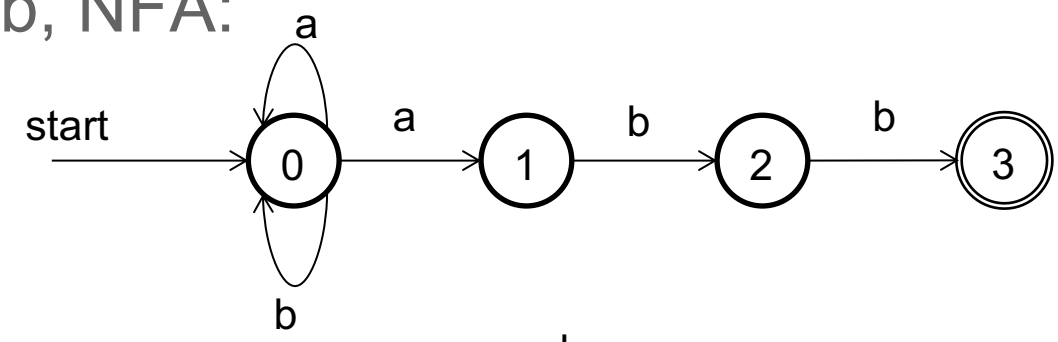
- Introduce rules for constructing NFA
- Demonstrate how to apply rules
- *Practice* constructing NFA
- Demonstrate converting NFA to DFA
- *Practice* converting NFA to DFA

From RE to Implemented Scanner

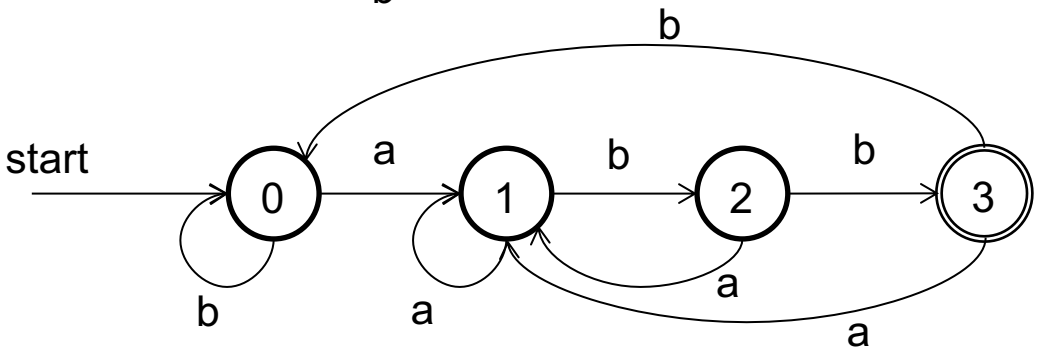
- It is convenient to write the rules for lexemes using regular expressions
- Computers don't understand regular expressions
  - need to an algorithm to convert to something the computer *does* understand
- It's easy to convert a RE into an NFA (using rules that will follow), but computers can't handle nondeterminism
- There is a standard approach to converting from NFA to DFA

Example: NFA vs DFA

- RE: (a|b)*abb, NFA:

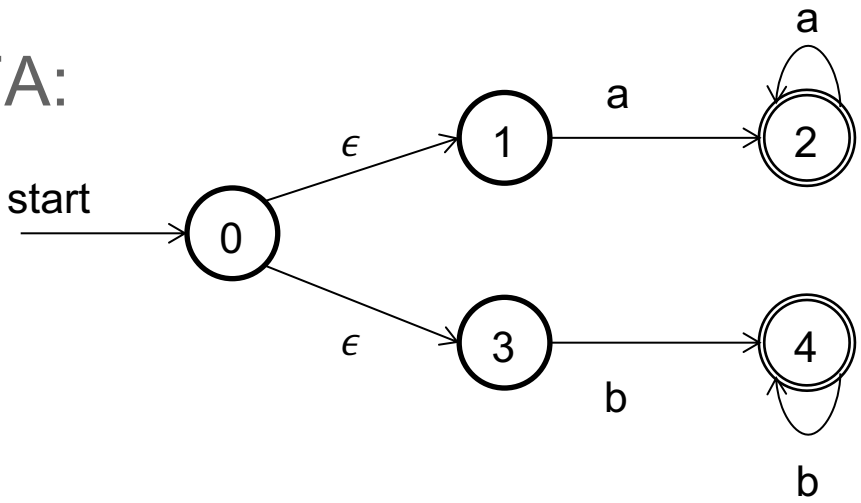


- DFA:

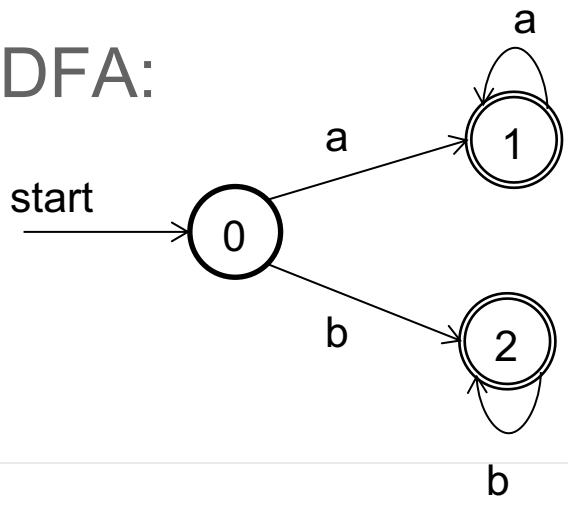


Example: NFA vs DFA

- RE: (aa* | bb*), NFA:

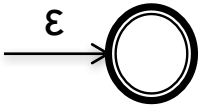


- DFA:



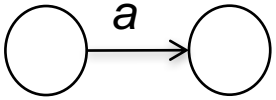
Constructing a NFA from a RE

- Six constructions:
  1. The entire RE is the null string

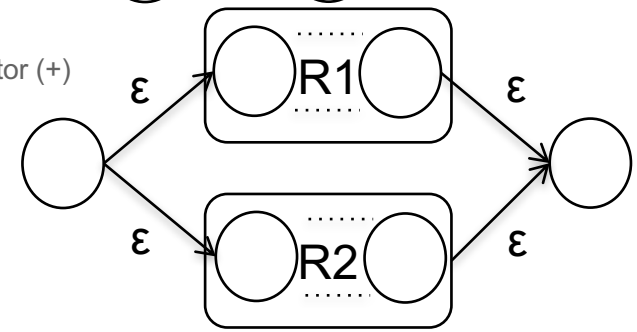


  2. The RE is empty

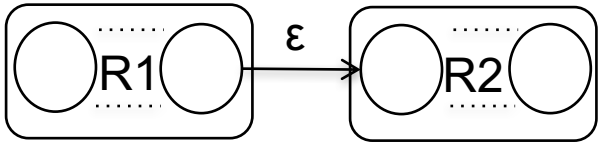
  3. An element *a* of the input alphabet is in the RE



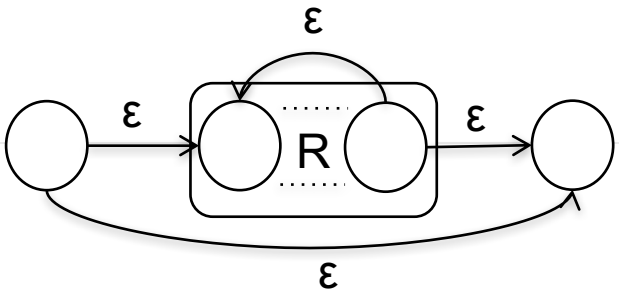
  4. Two REs are joined by the union operator (+)



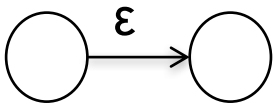
  5. Two REs are concatenated



  6. A RE has a Kleene closure (*) applied to it



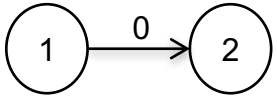
Using these rules

- First convert individual symbols using rule 3
- Then use other rules to combine and build upon these symbols
- Bear in mind that 

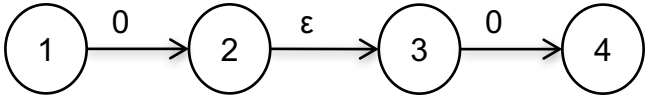
can reduce to a single state (replace with simply )
- Example:
- $(00 + 1)^* 1 (0 + 1)$

$(00 + 1)^* 1 (0 + 1)$

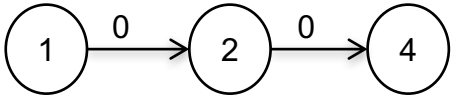
- Start with first 0:



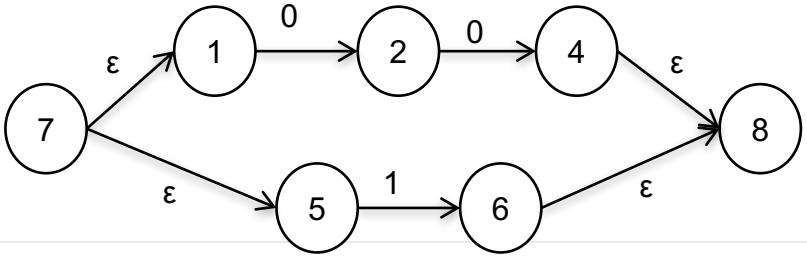
- Then add second 0:



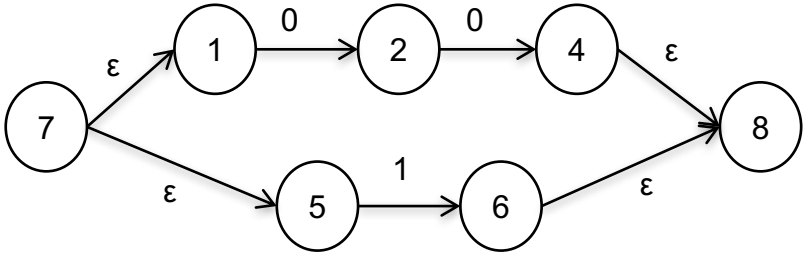
- States 2 & 3 can be combined, removing the ε-transition:



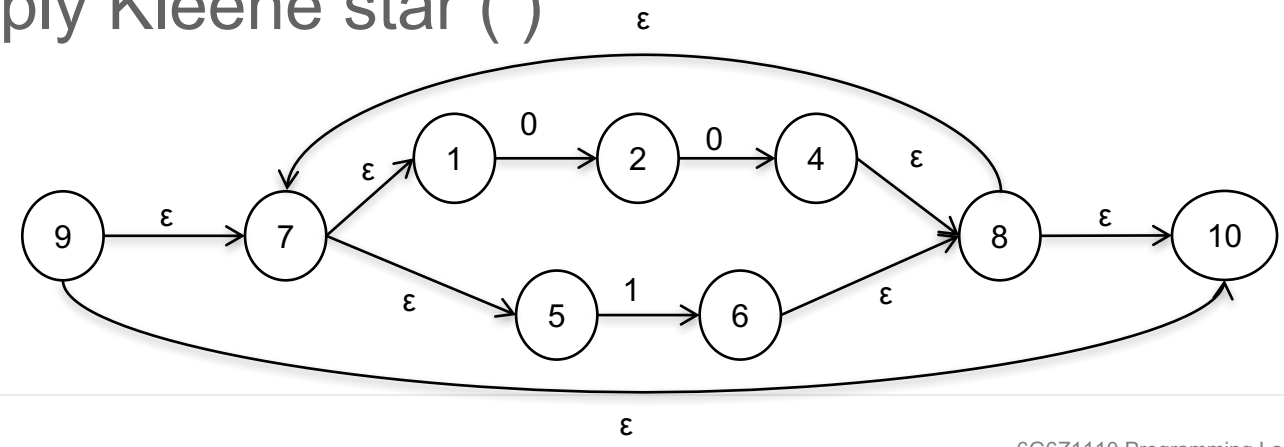
- Now add 1, and the union (+)



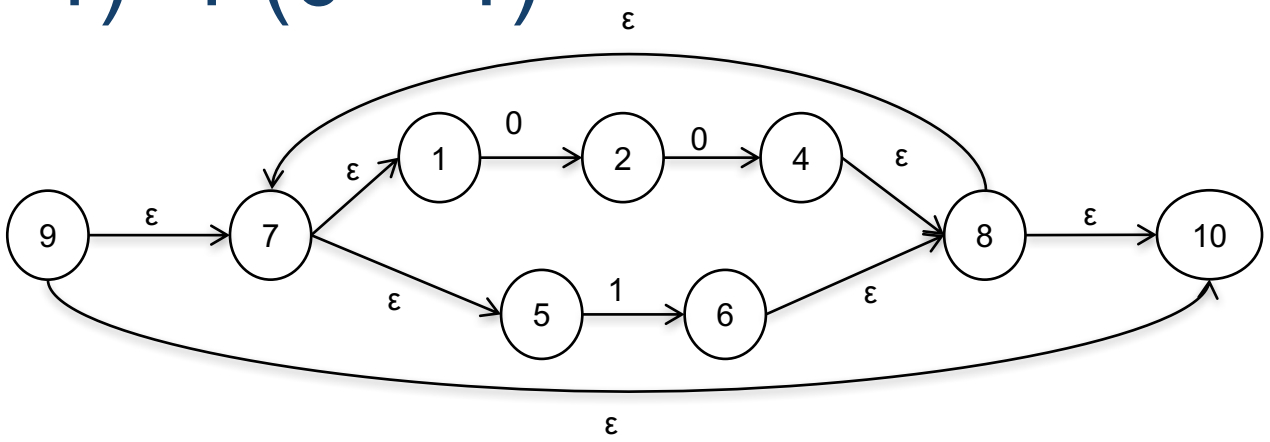
$(00 + 1)^* 1 (0 + 1)$



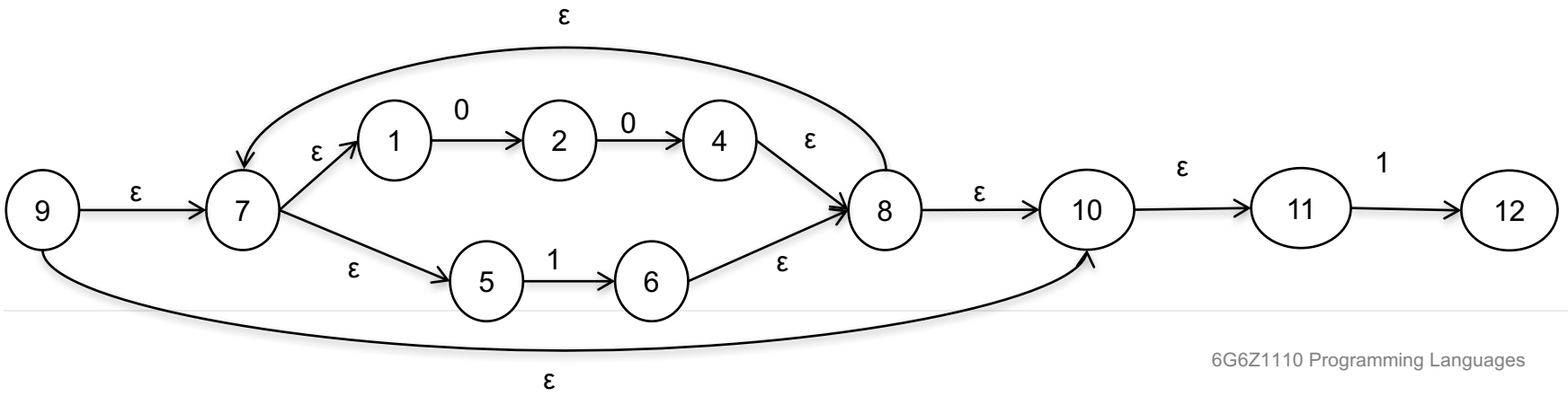
- Now apply Kleene star (*)



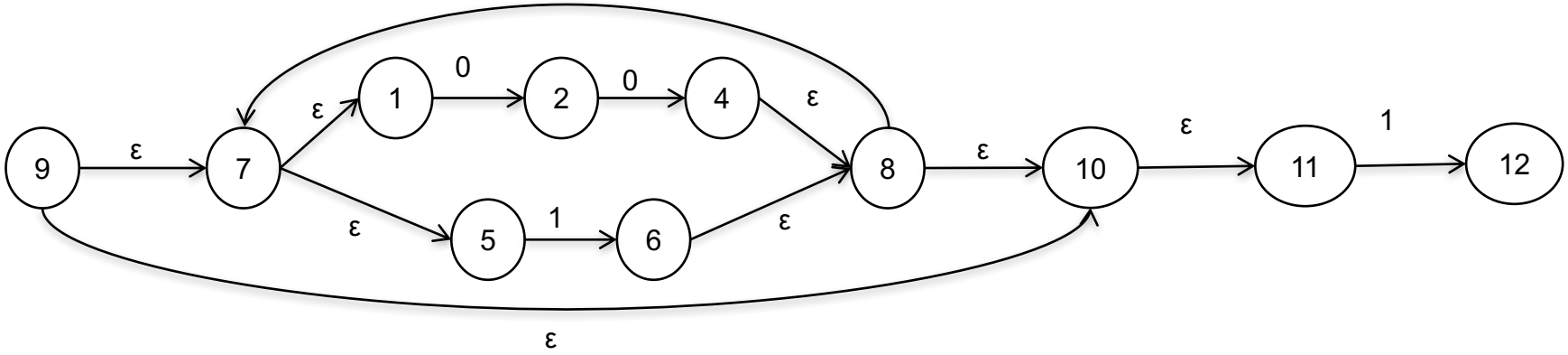
$(00 + 1)^* 1 (0 + 1)$



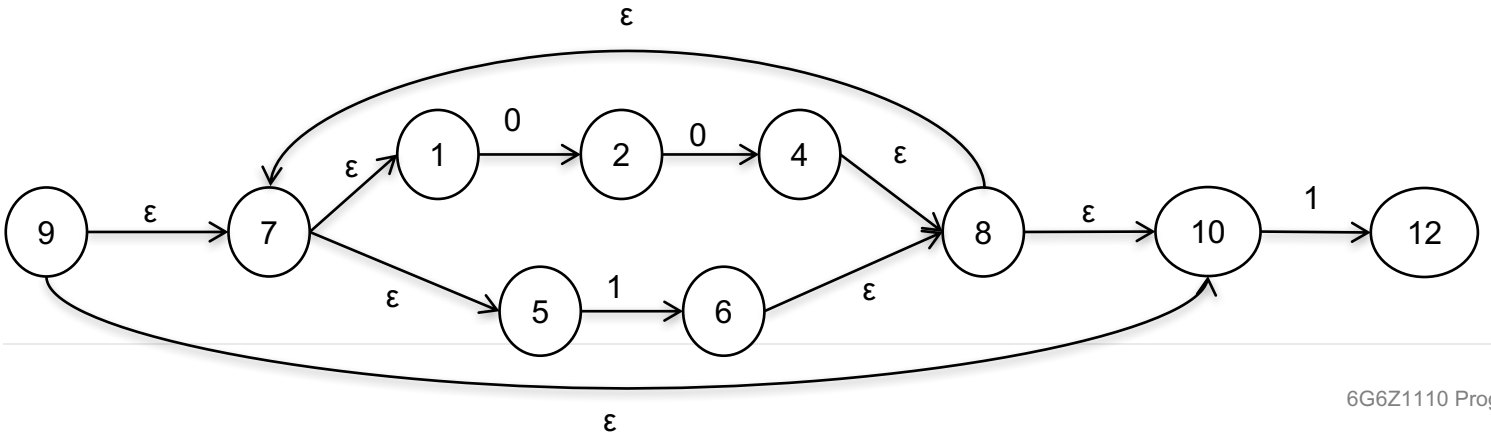
- Concatenate 1:



$(00 + 1)^* 1 (0 +_\varepsilon 1)$

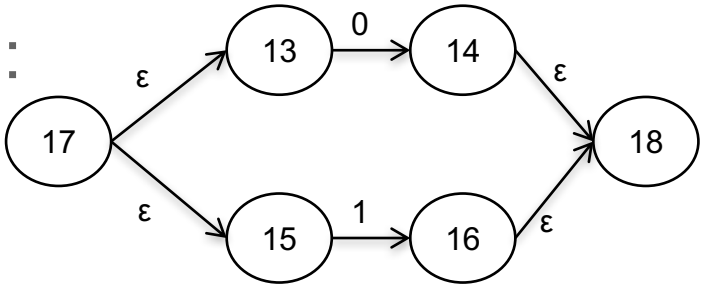


- And remove null-transition:

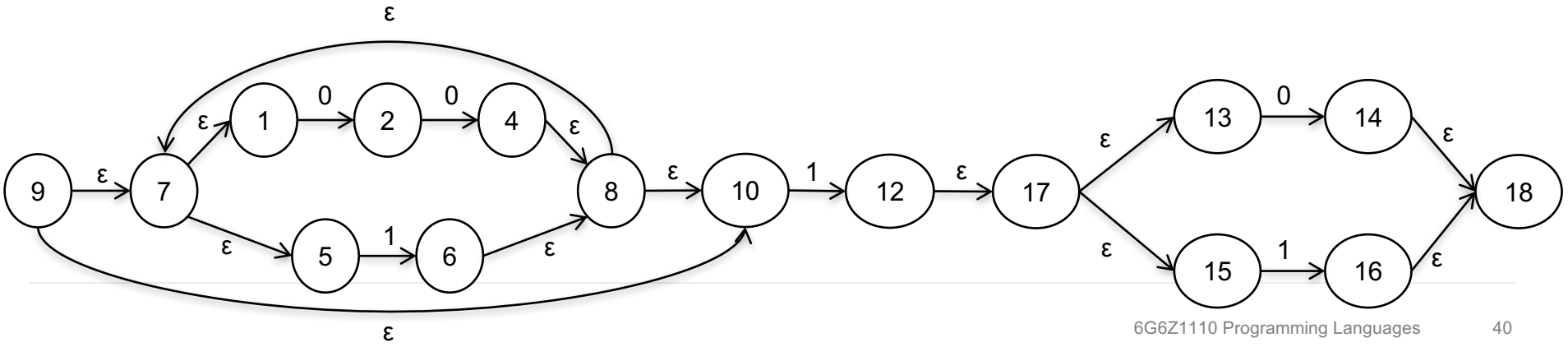


$(00 + 1)^* 1 (0 + 1)$

- Now focus on the last grouping:

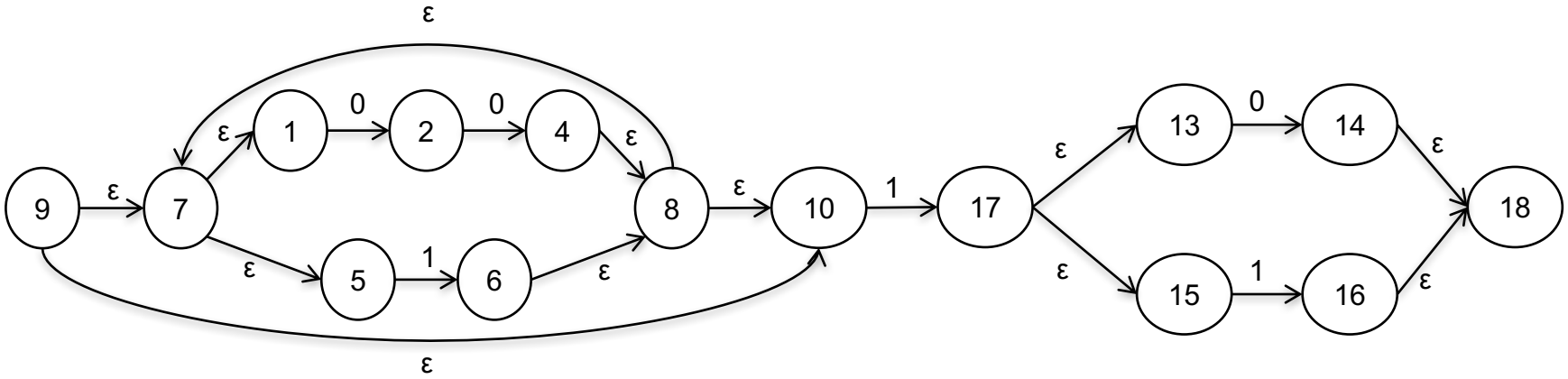


- And concatenate with first part:

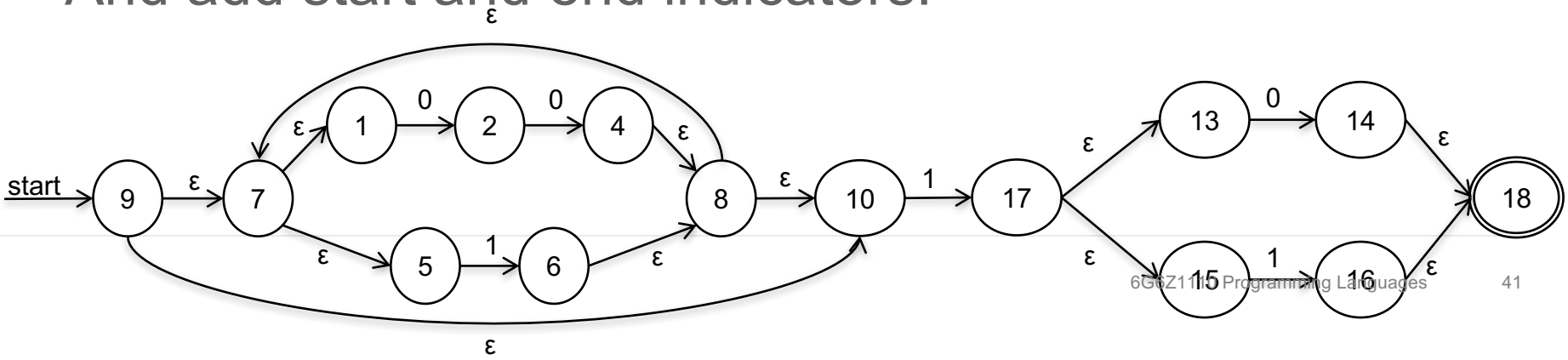


$(00 + 1)^* 1 (0 + 1)$

- Remove redundant null-transition:



- And add start and end indicators:



Similar Problem

- Regular expression: [a-z | _ | 0-9]*-[0-9][0-9]
  - What does it mean (in natural language)?

Similar Problem

- Regular expression: [a-z | _ | 0-9]*-[0-9][0-9]
  - What does it mean (in natural language)?
  - Which of these are valid inputs:
    - test-45
    - apha02--10
    - short_work-0
    - apple_934-34

Similar Problem

- Regular expression: [a-z | _ | 0-9]*-[0-9][0-9]
  - What does it mean (in natural language)?
  - Which of these are valid inputs:
    - test-45
    - apha02--10
    - short_work-0
    - apple_934-34
  - Construct a NFA from this RE. In your NFA, you may represent all lowercase letters by the label *letter*, and all digits by the label *digit*

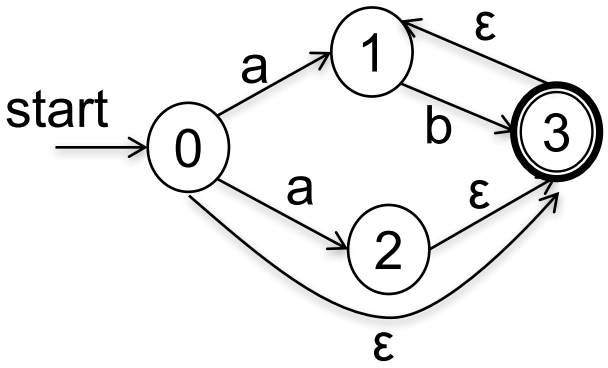
NFA to DFA

- We merge together NFA states by looking at them from the point of view of the input characters:
  - The **ε-closure** function takes a state and returns the set of states reachable from it based on (one or more) ε-transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its ε-closure without consuming any input.
  - The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

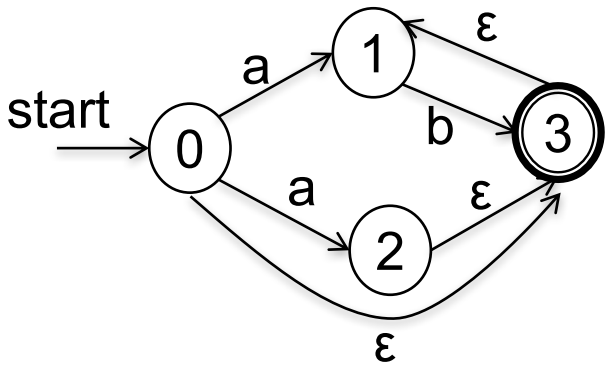
Applying these functions

1. Create the start state of the DFA by taking the ε-closure of the start state of the NFA.
2. Perform the following for the new DFA state:
   For each possible input symbol:
   1. Apply move to the newly-created state and the input symbol; this will return a set of states.
   2. Apply the ε-closure to this set of states, possibly resulting in a new set.
   3. This set of NFA states will be a single state in the DFA.
3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
4. The finish states of the DFA are those which contain any of the finish states of the NFA.

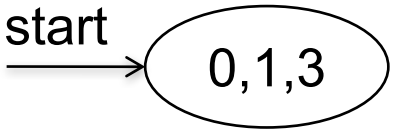
Example

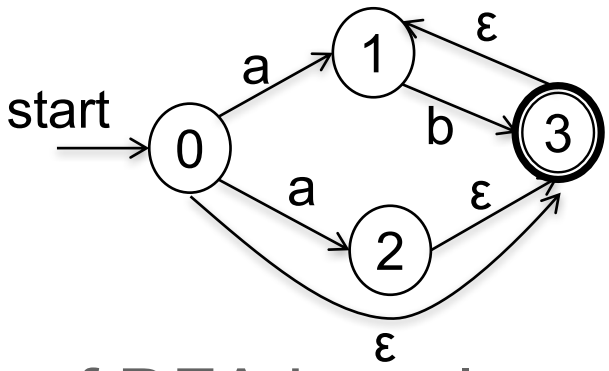


- 4 states, start state 0, accepting state 3

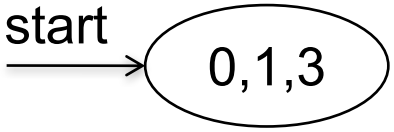


- Start state of DFA is ε-closure of start states of NFA





- Start state of DFA is ε-closure of start states of NFA



DFA state {0,1,3} – if input is **a**

From 0, we have

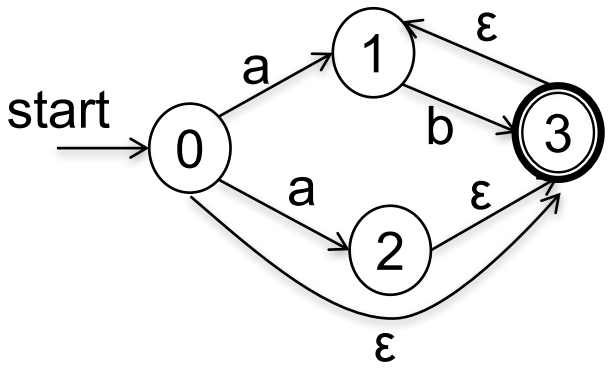
a path from 0 to 1 labeled a: 0 a 1

a path from 0 to 2 labeled a: 0 a 2

a path from 0 to 3 labeled a ε: 0 a 2 ε 3

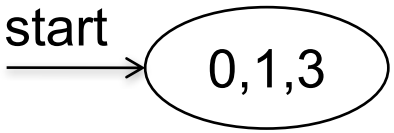
From 1, we have no transitions on input a

From 3 we have no transitions on input a



DFA state {0,1,3} – if input is **b**

From 0, we have



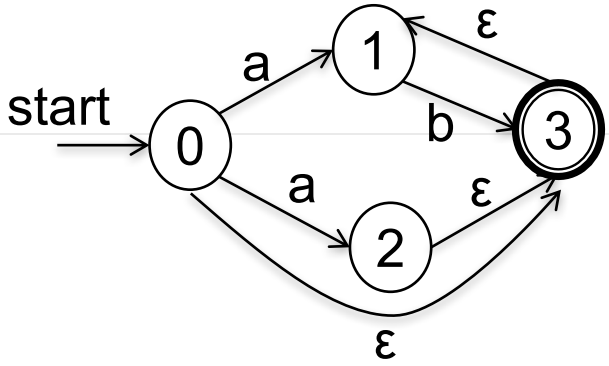
a path from 0 to 3 labelled ε ε b:

0 ε 3 ε 1 b 3

a path from 0 to 1 labelled ε ε b ε:

0 ε 3 ε 1 b 3 ε 1

From 1 and 3 there are no additional paths on input b



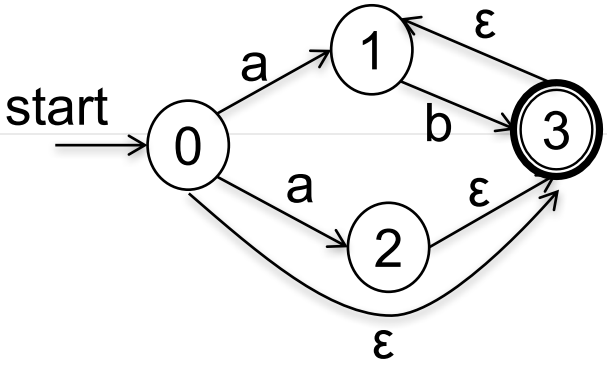
Construct a table

- So build up a table:

DFA state	Input ‘a’	Input ‘b’
{0,1,3}	{1,2,3}	{1,3}

ε-closure of the NFA start state

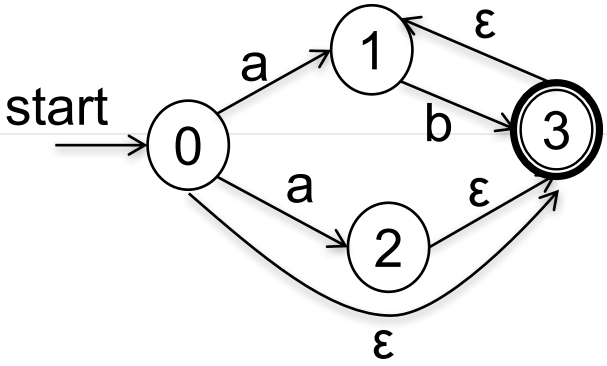
Construct a table



- So build up a table:

DFA state	Input ‘a’	Input ‘b’
{0,1,3}	{1,2,3}	{1,3}
{1,2,3}		

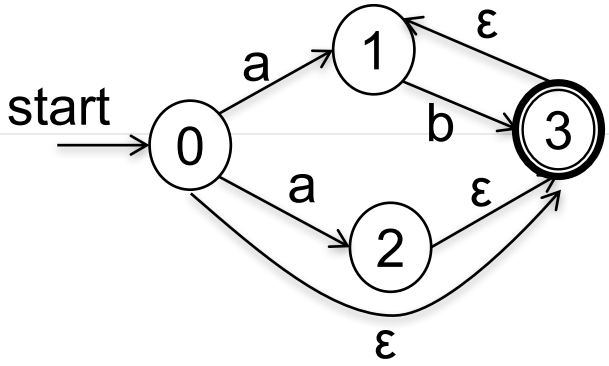
Construct a table



- So build up a table:

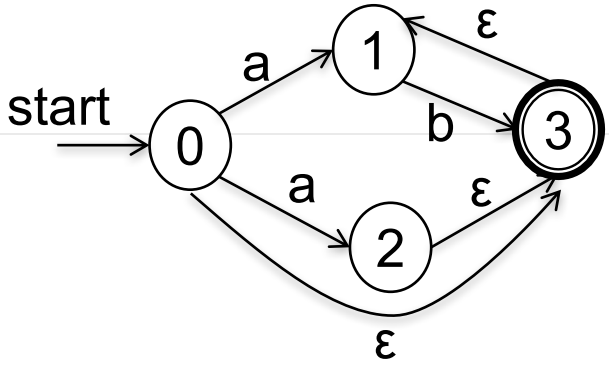
DFA state	Input ‘a’	Input ‘b’
{0,1,3}	{1,2,3}	{1,3}
{1,2,3}	φ	

Construct a table



- So build up a table:

DFA state	Input ‘a’	Input ‘b’
{0,1,3}	{1,2,3}	{1,3}
{1,2,3}	φ	{1,3}



Construct a table

- So build up a table:

DFA state	Input 'a'	Input 'b'
{0,1,3}	{1,2,3}	{1,3}
{1,2,3}	φ	{1,3}
{1,3}	φ	{1,3}

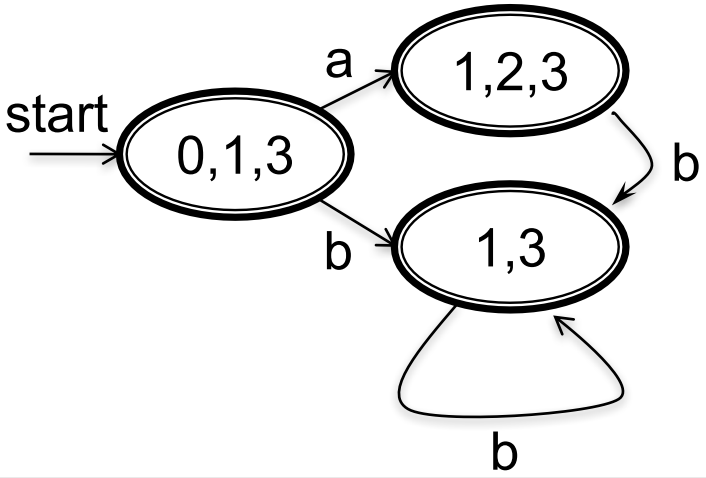
No states here that are not in the DFA state list

STOP

Manchester Metropolitan University

DFA state	Input ‘a’	Input ‘b’
{0,1,3}	{1,2,3}	{1,3}
{1,2,3}	φ	{1,3}
{1,3}	φ	{1,3}

This table can be implemented as a set of if-then-else rules.



Summary

- Computers cannot directly interpret regular expressions
- It is easy to represent a regular expression as a nondeterministic finite automaton (NFA)

  (Learn the six rules)
- Computers cannot handle nondeterminism
- It is possible to convert any NFA to a DFA

  (Easy to implement as if-then-else rules)

Practise constructing NFA and DFA

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- Parsing
- Semantic analysis
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review