Automata and Formal Languages (4)

Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)

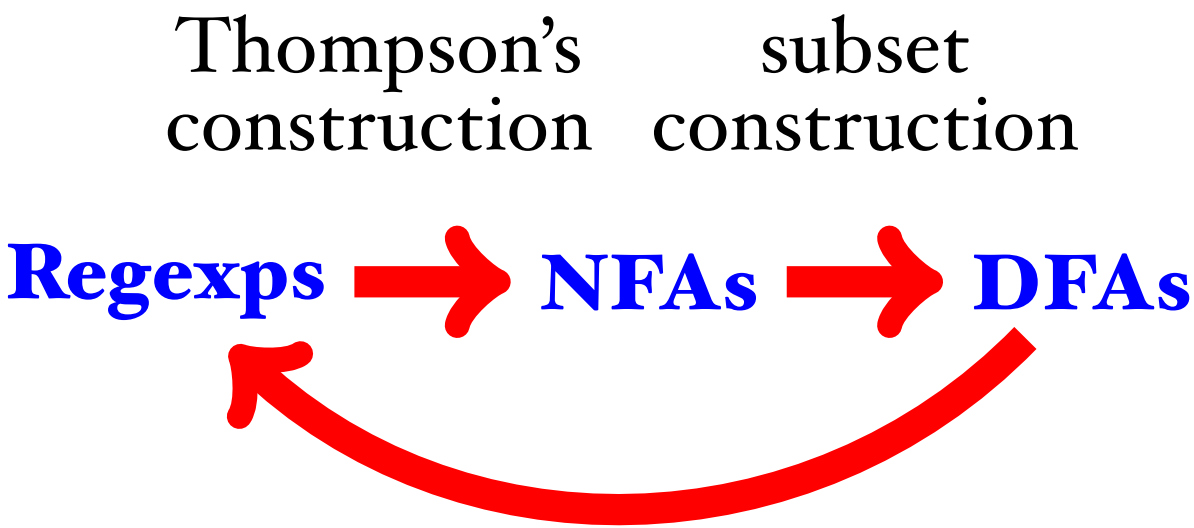
Slides: KEATS (also home work is there)

Regexps and Automata

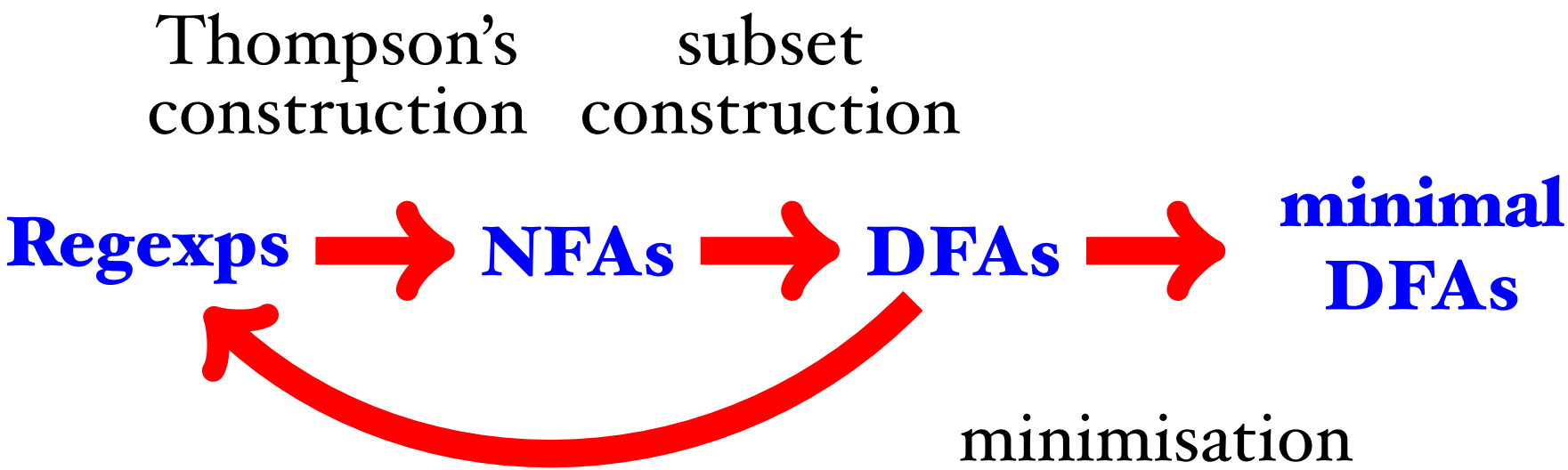
Thompson's construction

subset construction

Regexps and Automata



Regexps and Automata



DFAs

A deterministic finite automaton consists of:

- a finite set of states, Q
- one of these states is the start state, q_0
- there is transition function, δ, and
- some states are accepting states, F

$$A(Q, q_0, \delta, F)$$

State Nodes

```
1  abstract class State
2  type States = Set[State]
3
4  case class IntState(i: Int) extends State
5
6  object NewState {
7    var counter = 0
8
9    def apply() : IntState = {
10     counter += 1;
11     new IntState(counter - 1)
12   }
13 }
```

DFAs

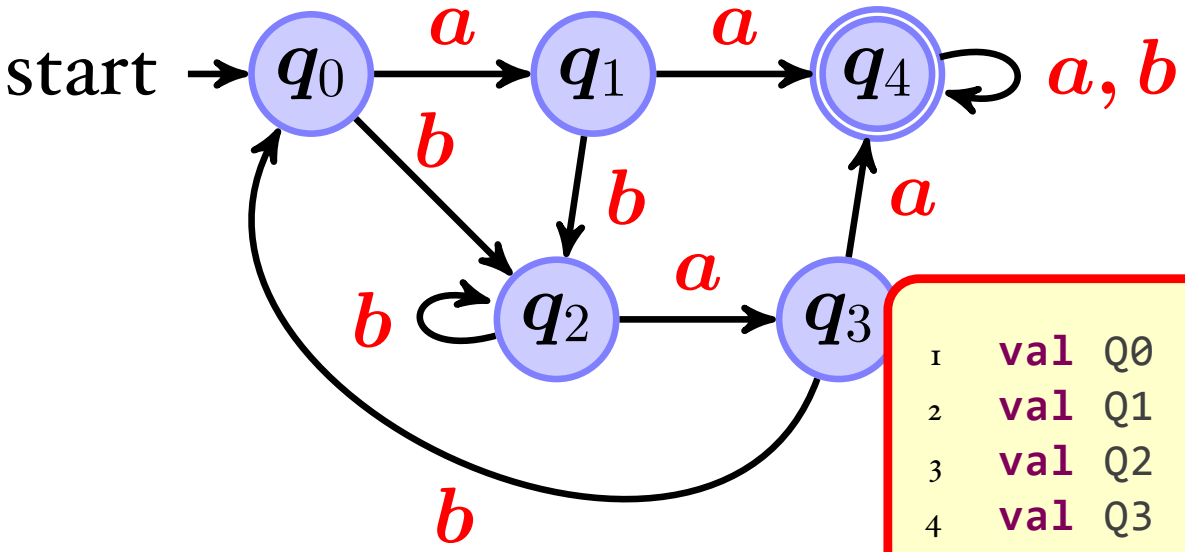
```
1  case class DFA(states: States,
2                 start: State,
3                 delta: (State, Char) => State,
4                 fins: States) {
5
6    def deltas(q: State, s: List[Char]) : State = s match {
7      case Nil => q
8      case c::cs => deltas(delta(q, c), cs)
9    }
10
11   def accepts(s: String) : Boolean =
12     Try(fins contains (deltas(start, s.toList))) getOrElse fal
13 }
```

DF

$$\hat{\delta}(q, "") \stackrel{\text{def}}{=} q$$
$$\hat{\delta}(q, c :: s) \stackrel{\text{def}}{=} \hat{\delta}(\delta(q, c), s)$$

$$\hat{\delta}(q_0, s) \in F$$

```
1  case class DFA(states: States,
2                 start: State,
3                 delta: (State, Char) => State,
4                 fins: States) {
5
6    def deltas(q: State, s: List[Char]) : State = s match {
7      case Nil => q
8      case c::cs => deltas(delta(q, c), cs)
9    }
10
11   def accepts(s: String) : Boolean =
12     Try(fins contains (deltas(start, s.toList))) getOrElse fal
13 }
```




```
1   val Q0 = NewState()
2   val Q1 = NewState()
3   val Q2 = NewState()
4   val Q3 = NewState()
5   val Q4 = NewState()
6
7   val delta : (State, Char) => State =
8   {
9     case (Q0, 'a') => Q1
10    case (Q0, 'b') => Q2
11    case (Q1, 'a') => Q4
12    case (Q1, 'b') => Q2
13    case (Q2, 'a') => Q3
14    case (Q2, 'b') => Q2
15    case (Q3, 'a') => Q4
16    case (Q3, 'b') => Q0
17    case (Q4, 'a') => Q4
18    case (Q4, 'b') => Q4
19  }
```

NFAs

A non-deterministic finite automaton $A(Q, q_0, \delta, F)$ consists of:

- a finite set of states, Q
- one of these states is the start state, q_0
- some states are accepting states, F,
- there is transition relation, δ, and
- there are silent transitions

$$(q_1, a) \rightarrow q_2$$
$$(q_1, a) \rightarrow q_3$$

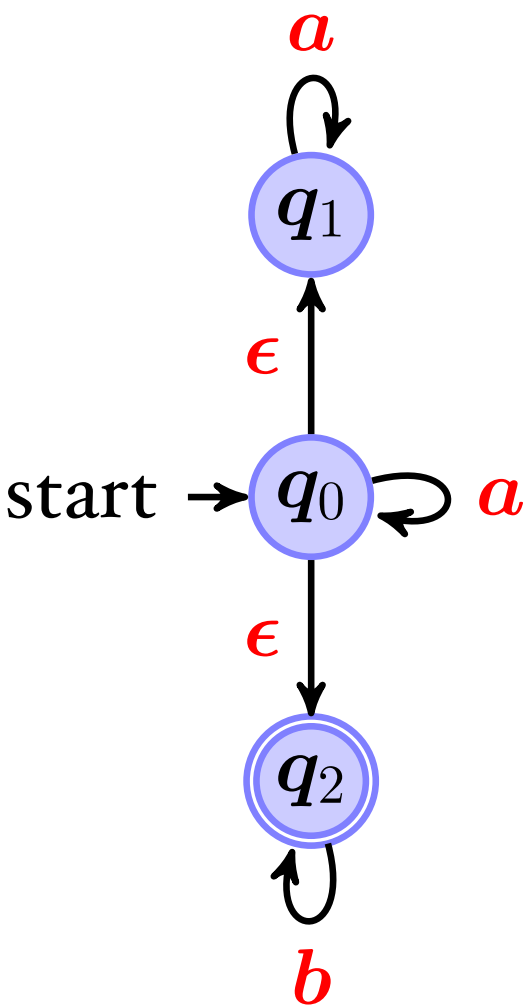
$$(q_1, \epsilon) \rightarrow q_2$$

```
case class NFA(states: States,
               start: State,
               delta: (State, Char) => States,
               eps: State => States,
               fins: States) {

  def epsclosure(qs: States) : States = {
    val ps = qs ++ qs.flatMap(eps(_))
    if (qs == ps) ps else epsclosure(ps)
  }

  def deltas(qs: States, s: List[Char]) : States = s match {
    case Nil => epsclosure(qs)
    case c::cs =>
      deltas(
        epsclosure(
          epsclosure(qs).flatMap(delta (_, c))), cs)
  }

  def accepts(s: String) : Boolean =
    deltas(Set(start), s.toList) exists (fins contains (_))
}
```



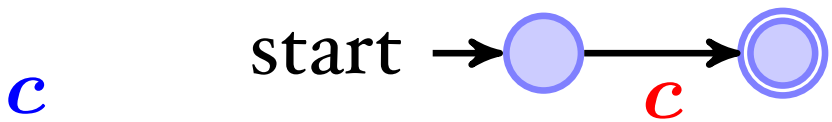

```
1  val Q0 = NewState()
2  val Q1 = NewState()
3  val Q2 = NewState()
4
5  val delta : (State, Char) => States = {
6    case (Q0, 'a') => Set(Q0)
7    case (Q1, 'a') => Set(Q1)
8    case (Q2, 'b') => Set(Q2)
9    case (_, _) => Set ()
10 }
11
12 val eps : State => States = {
13   case Q0 => Set(Q1, Q2)
14   case _ => Set()
15 }
16
17 val NFA1 = NFA(Set(Q0, Q1, Q2),
18                 Q0,
19                 delta,
20                 eps,
21                 Set(Q2))
```

Rexp to NFA

Thompson's construction of a NFA from a regular expression:

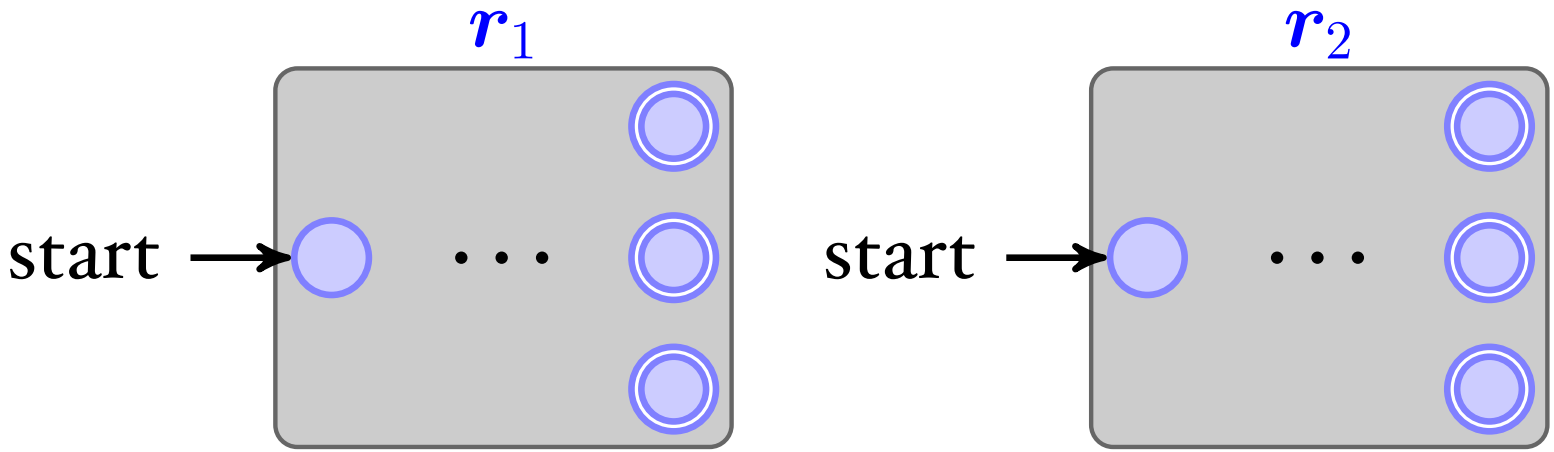
$\varnothing$

ϵ

c 

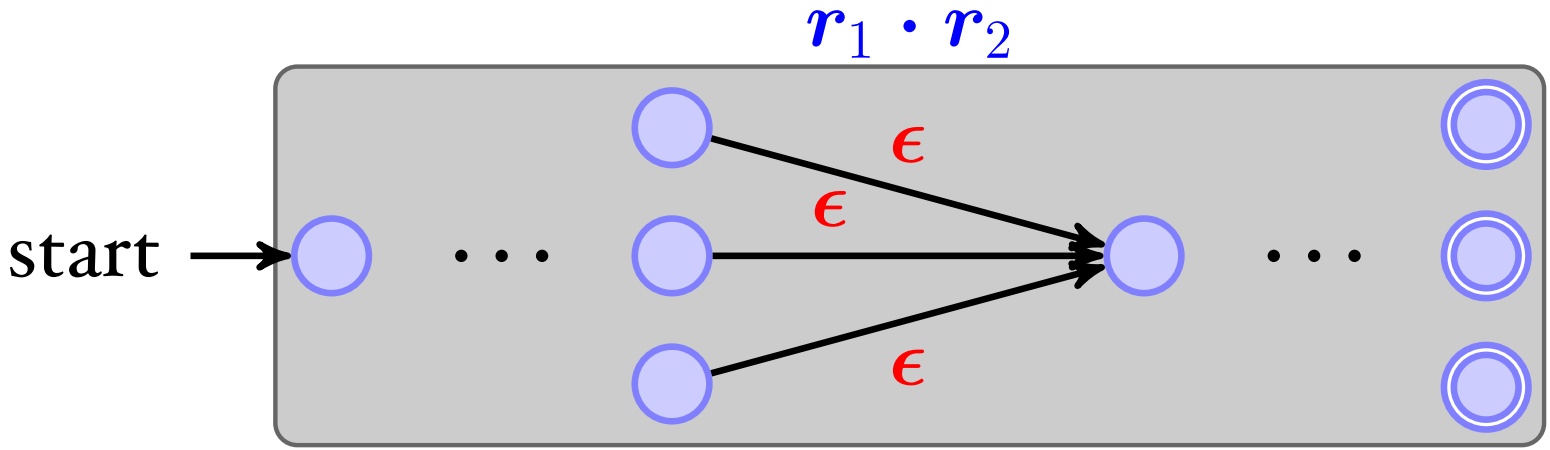
Case $r_1 \cdot r_2$

By recursion we are given two automata:



We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via ϵ-transitions to the starting state of the second automaton.

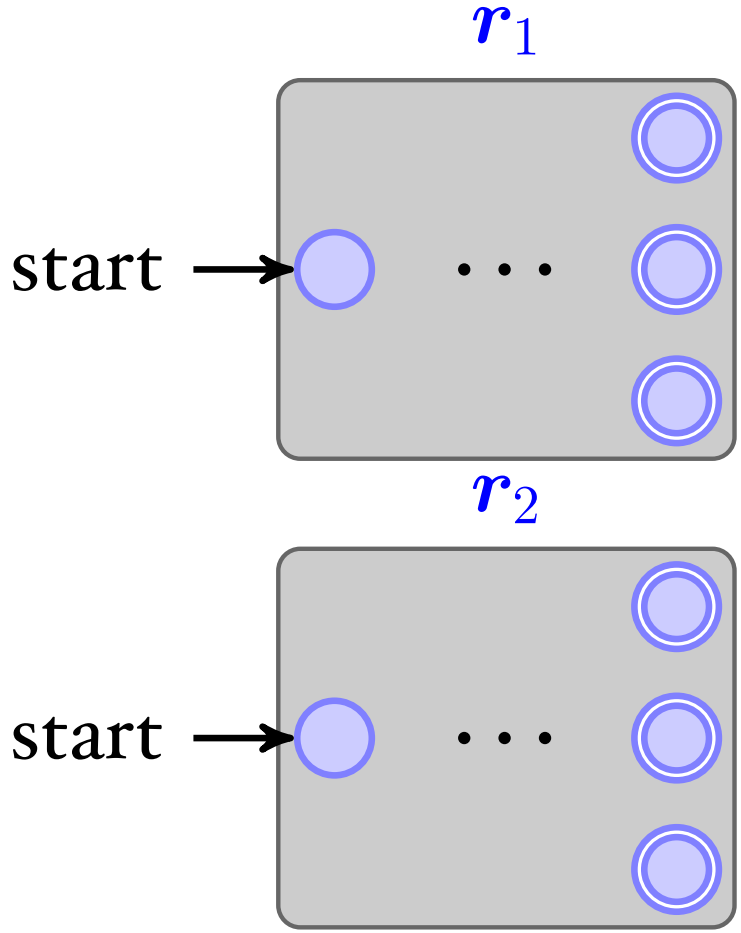
Case $r_1 \cdot r_2$



We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via ϵ-transitions to the starting state of the second automaton.

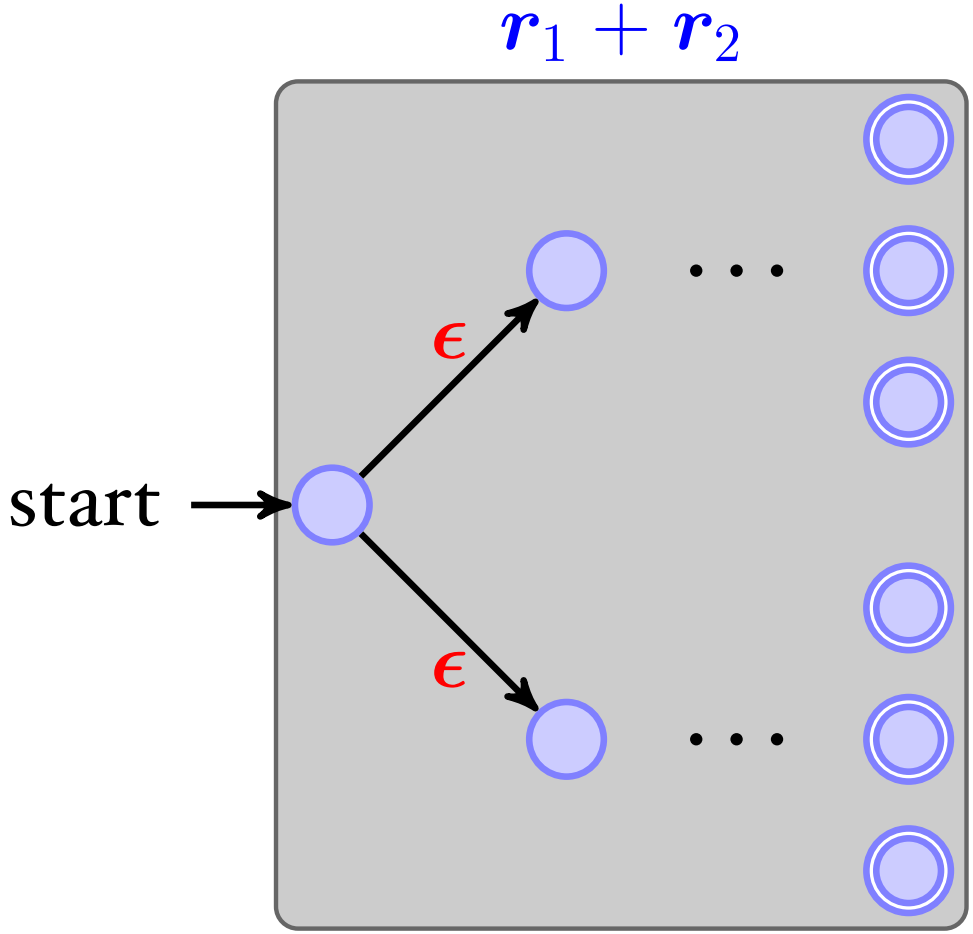
Case $r_1 + r_2$

By recursion we are given two automata:



We (1) need to introduce a new starting state and (2) connect it to the original two starting states.

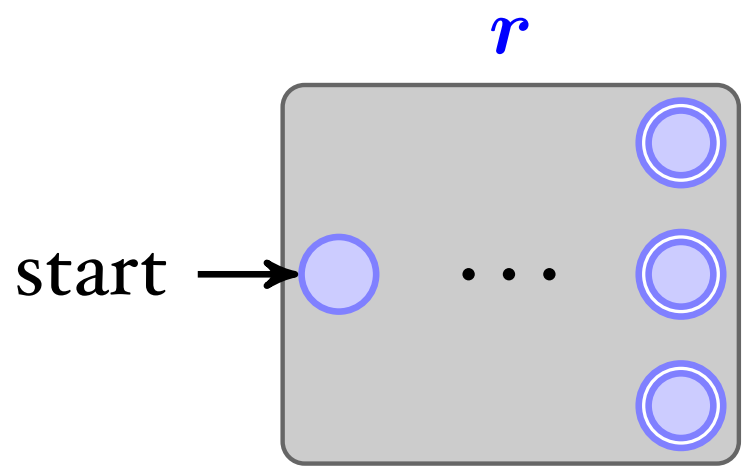
Case $r_1 + r_2$



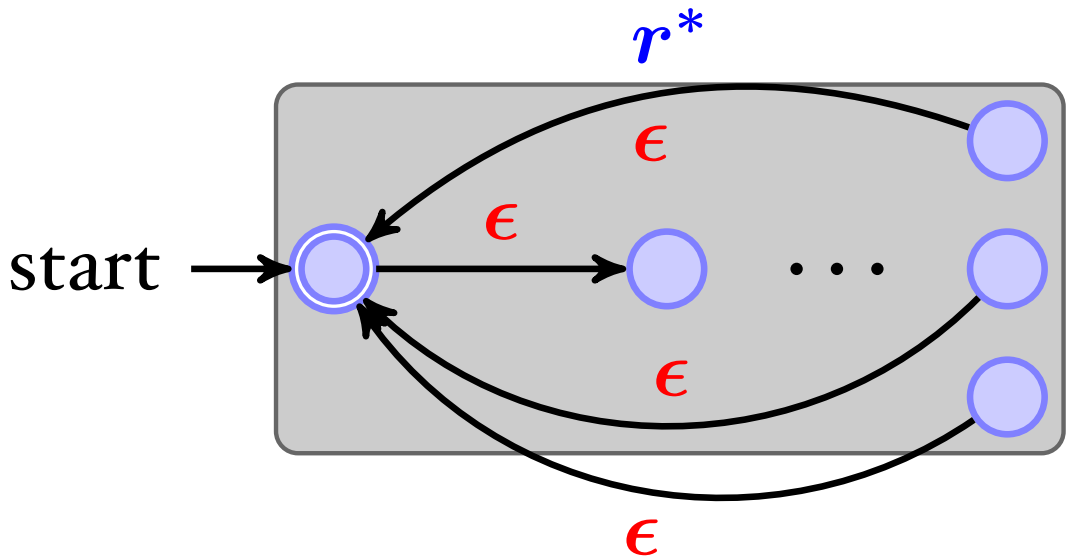
We (1) need to introduce a new starting state and (2) connect it to the original two starting states.

Case r^*

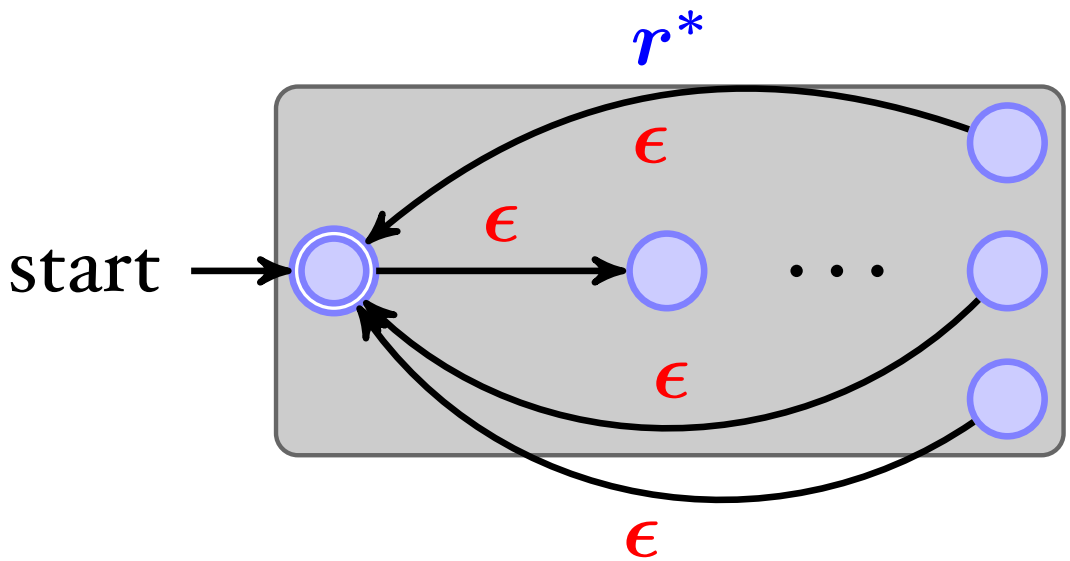
By recursion we are given an automaton for r:



Case r^*

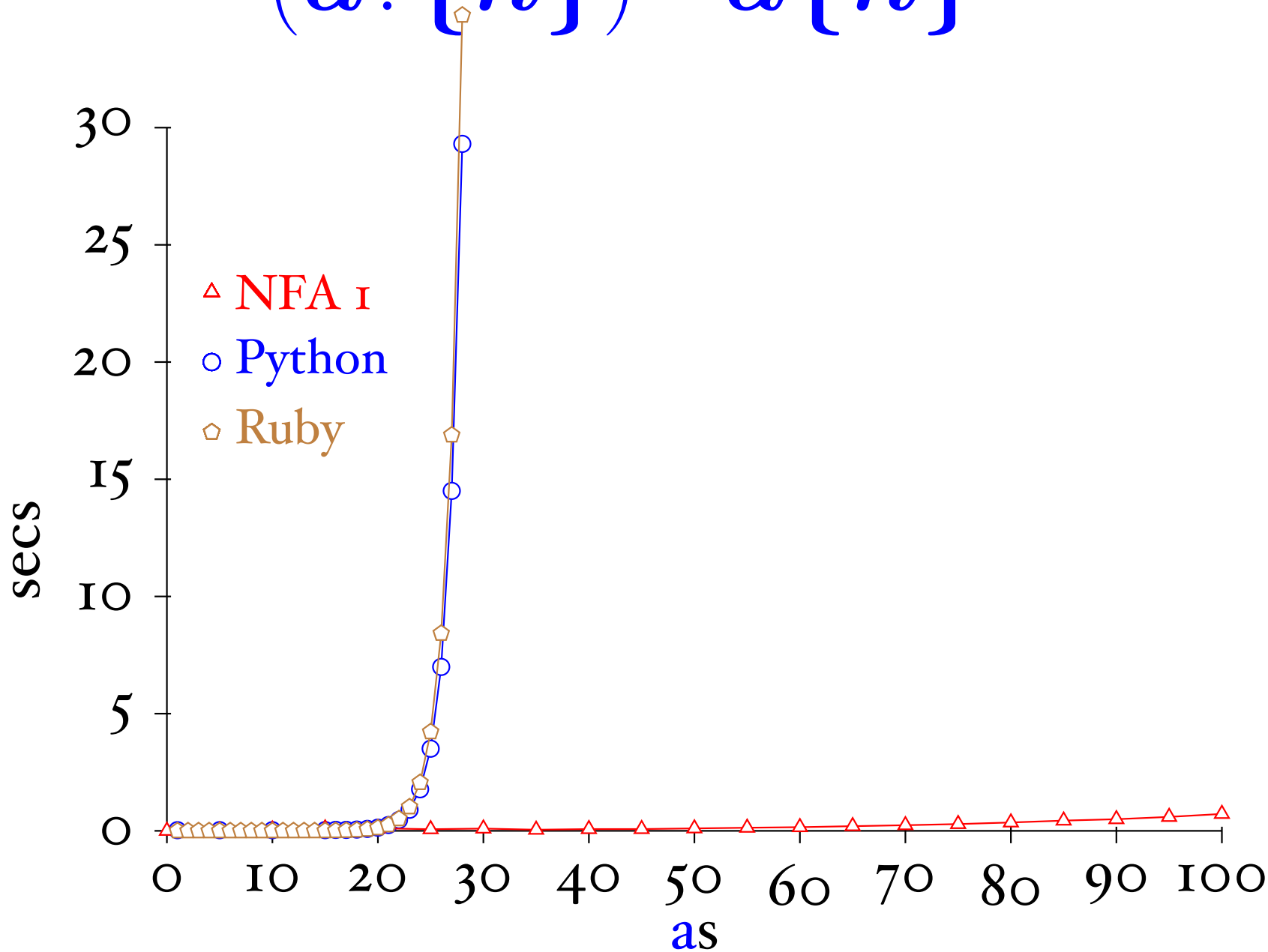


Case r^*



Why can't we just have an epsilon transition from the accepting states to the starting state?

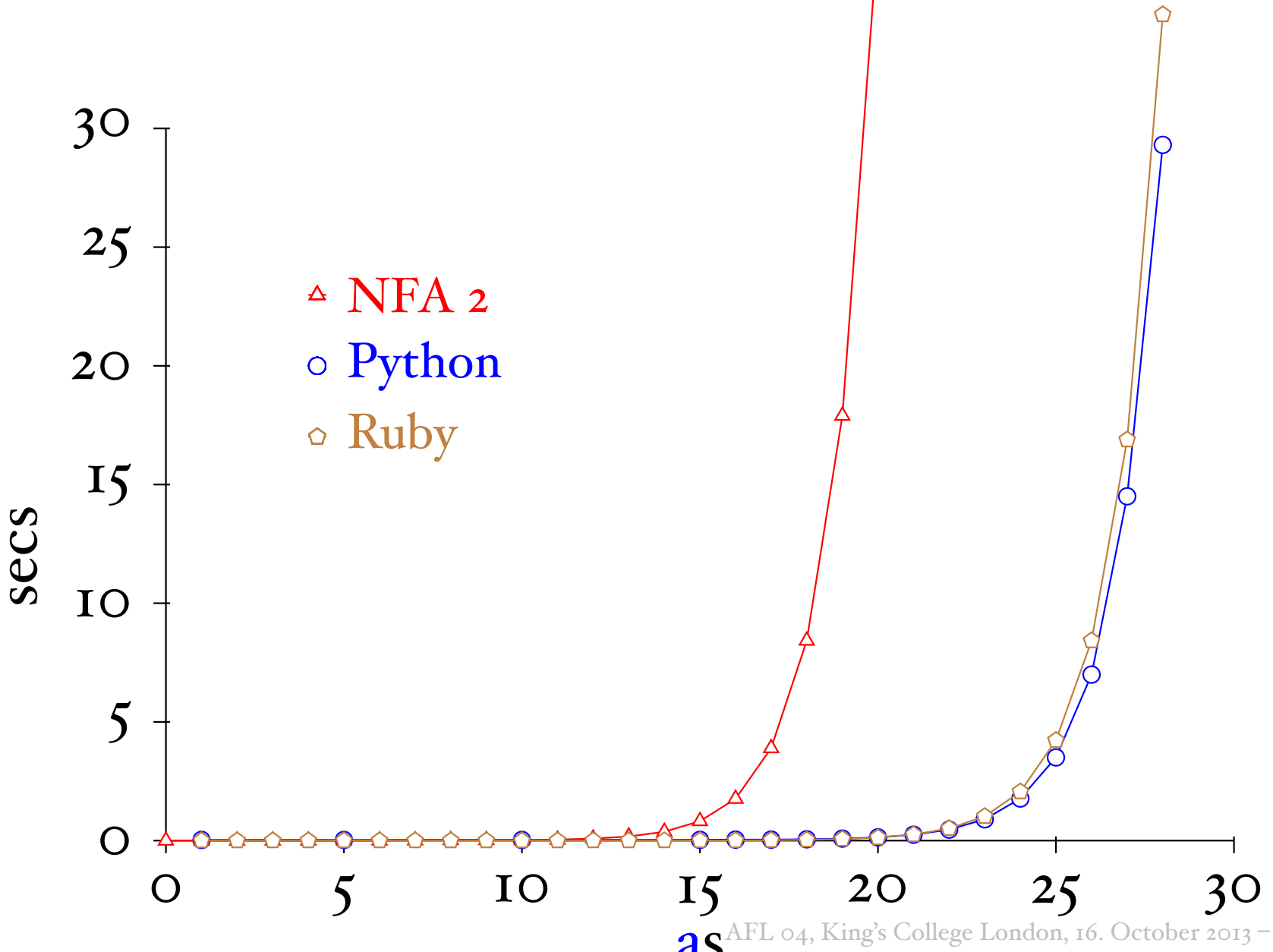
$(a?\{n\}) \cdot a\{n\}$


30
25
20
15
10
5
0
secs
NFA 1
Python
Ruby
0
10
20
30
40
50
60
70
80
90
100
as

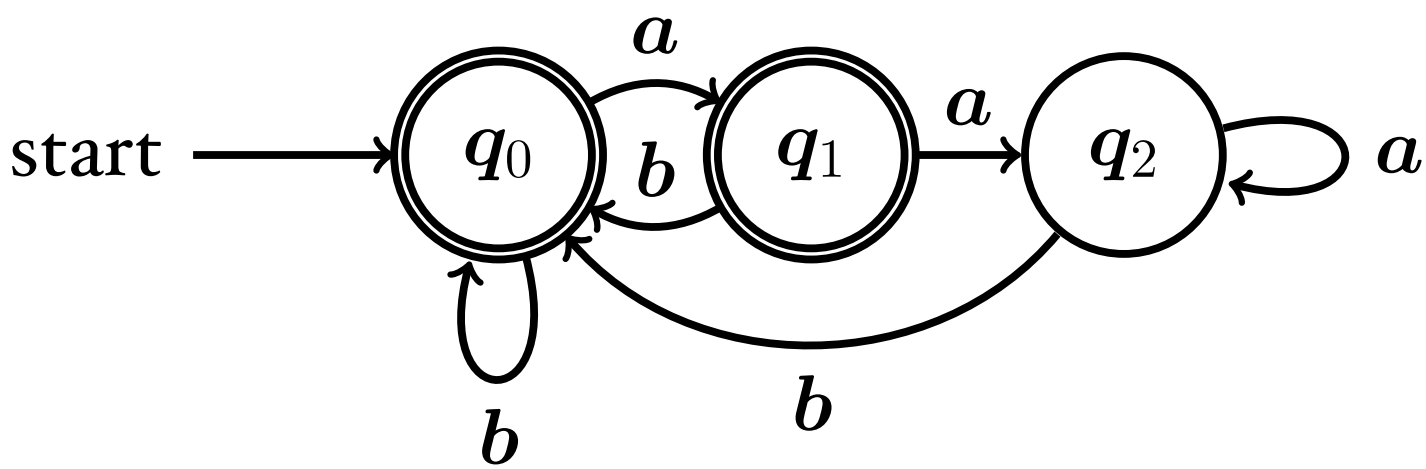
Greedy Depth-First

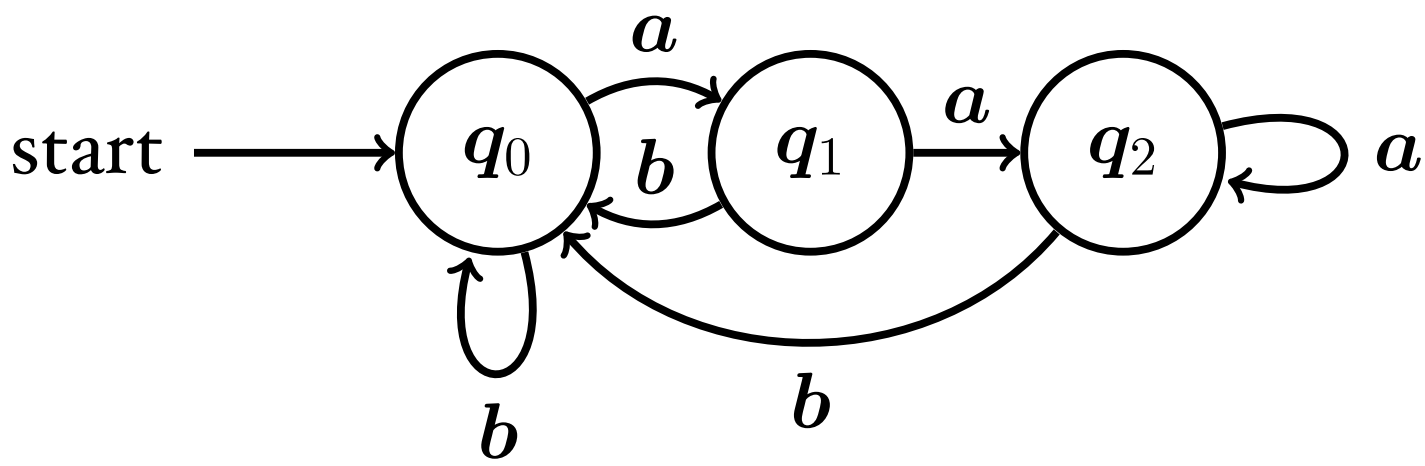
```
1  def accepts2(nfa: NFA, s: String) : Boolean = {
2
3    def search(q: State, s: List[Char]) : Boolean = s match {
4      case Nil => nfa.fins contains (q)
5      case c::cs =>
6        (nfa.delta(q, c) exists (search(_, cs))) ||
7        (nfa.eps(q) exists (search(_, c::cs)))
8    }
9
10   search(nfa.start, s.toList)
11 }
12
13 def matcher2(r: Rexp, s: String) : Boolean =
14   accepts2(thompson(r), s)
```

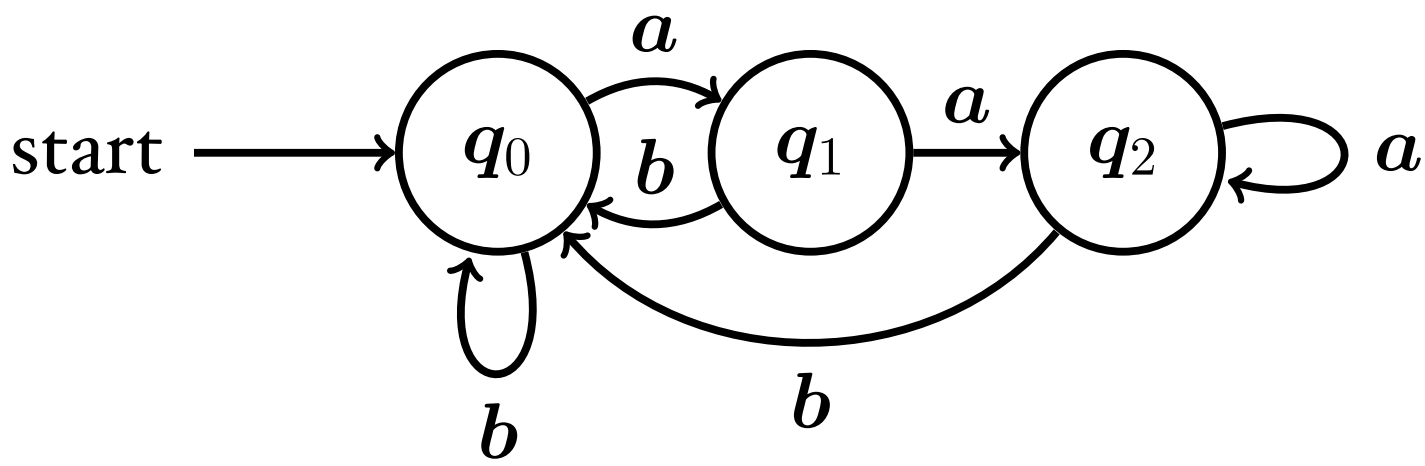
$(a?\{n\}) \cdot a\{n\}$



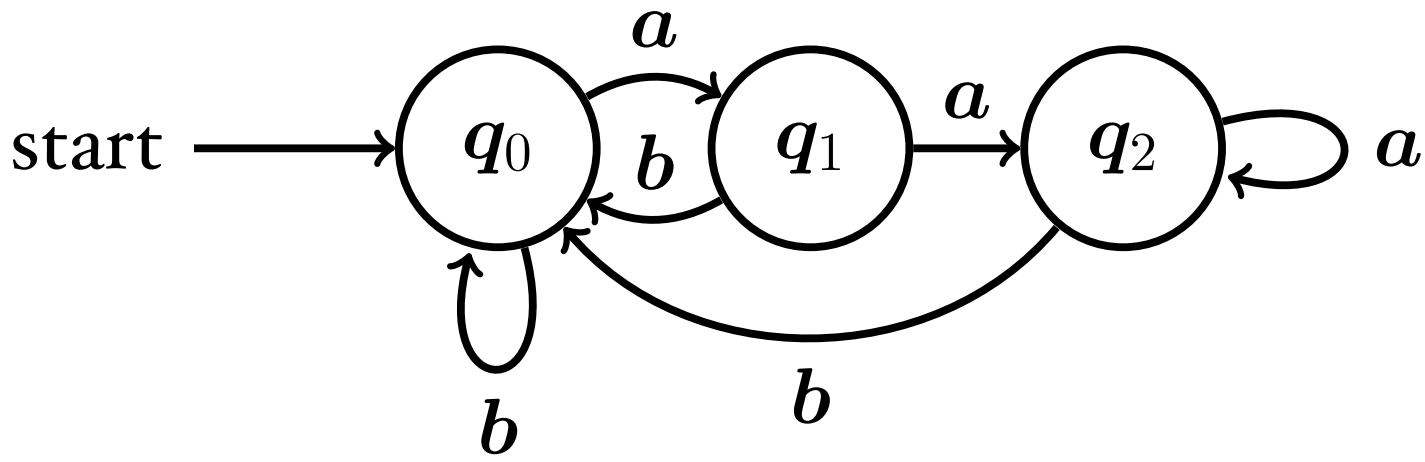
DFA to Rexp

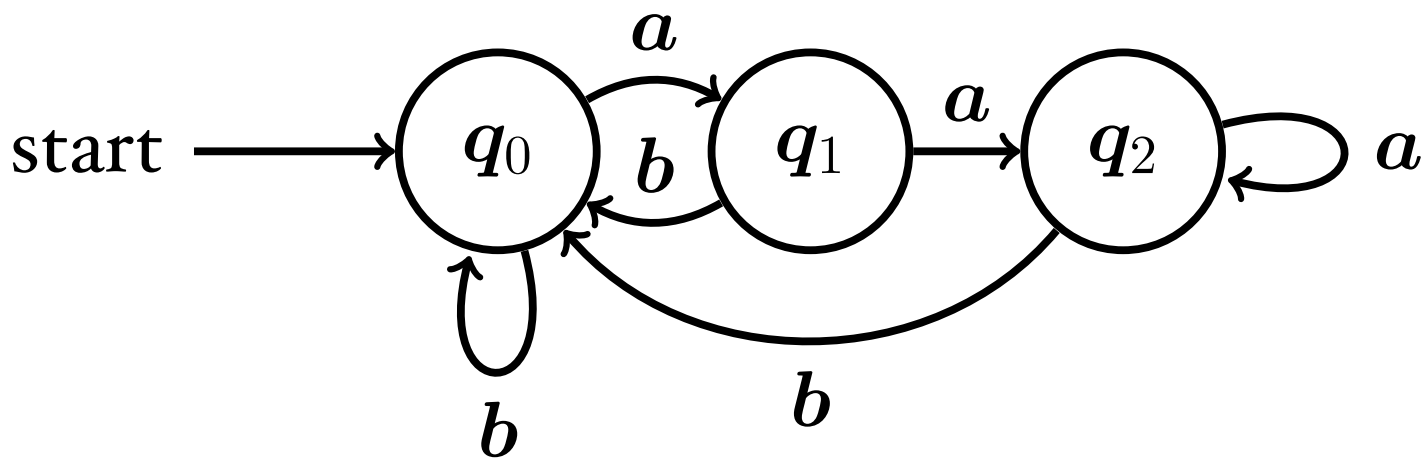



start
q_0
q_1
q_2
a
b
a
a
b
b

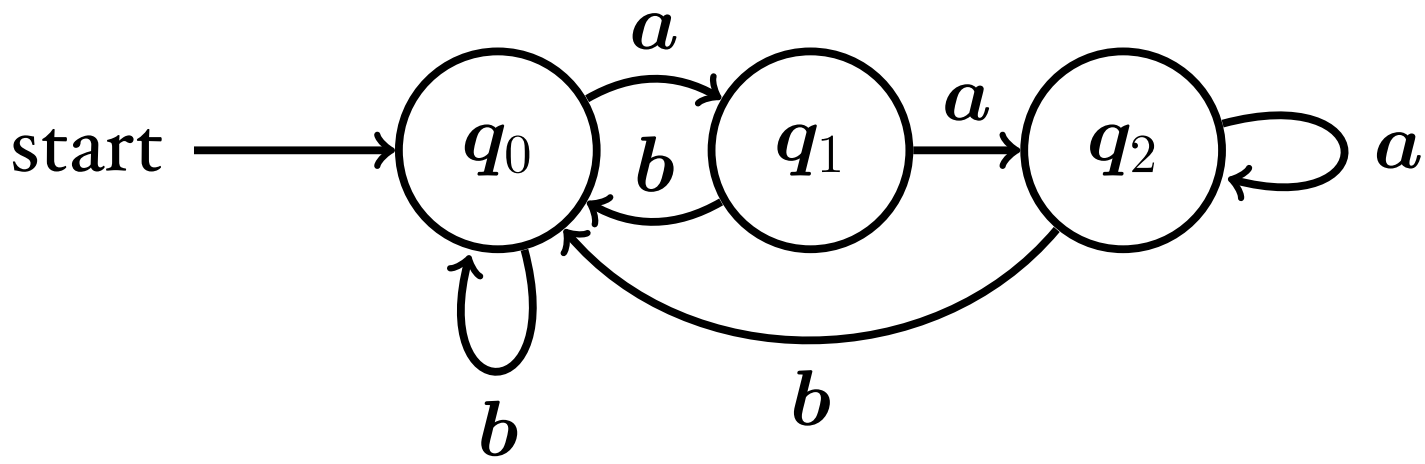


$$
\begin{aligned}
q_0 &= 2\,q_0 + 3\,q_1 + 4\,q_2 \\
q_1 &= 2\,q_0 + 3\,q_1 + 1\,q_2 \\
q_2 &= 1\,q_0 + 5\,q_1 + 2\,q_2
\end{aligned}
$$


start
q_0
q_1
q_2
a
a
a
b
b
b



$$q_0 = \epsilon + q_0\, b + q_1\, b + q_2\, b$$
$$q_1 = q_0\, a$$
$$q_2 = q_1\, a + q_2\, a$$



$$q_0 = \epsilon + q_0\,b + q_1\,b + q_2\,b$$
$$q_1 = q_0\,a$$
$$q_2 = q_1\,a + q_2\,a$$

Arden's Lemma:

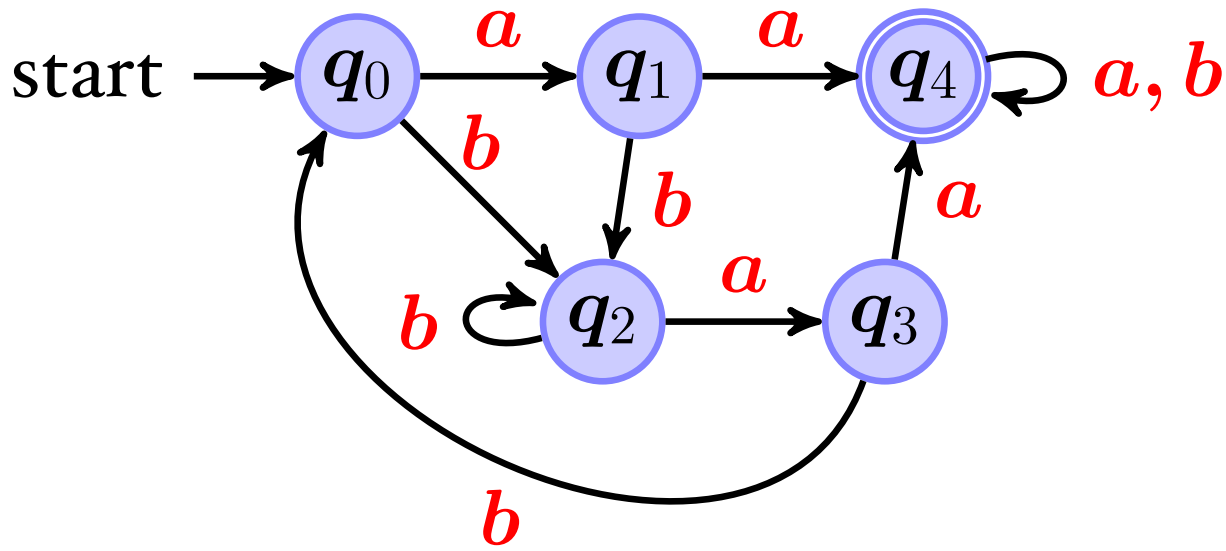
$$\text{If } q = q\,r + s \text{ then } q = s\,r^*$$

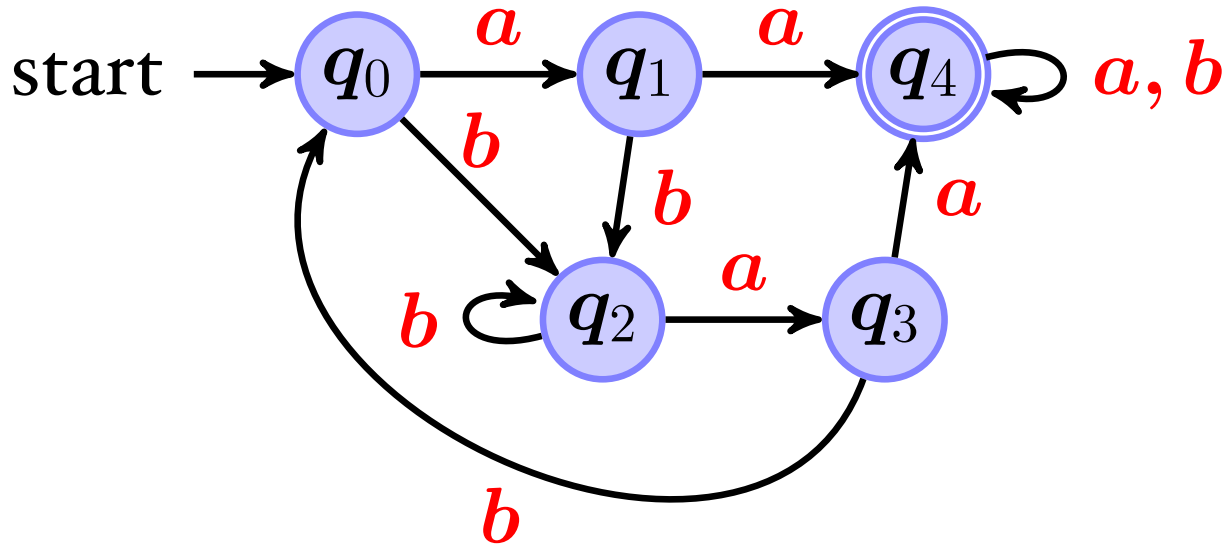
DFA Minimisation

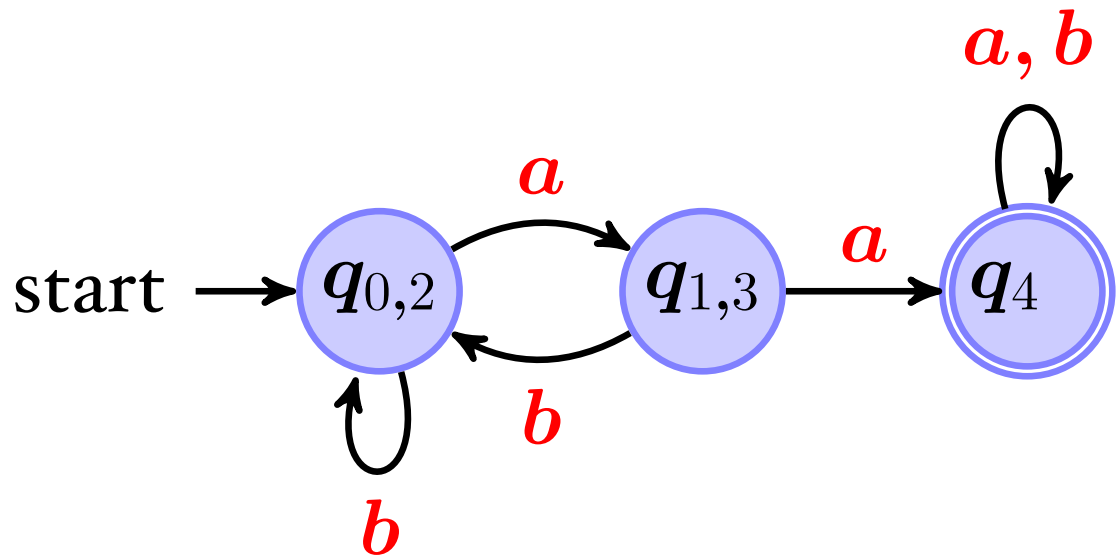
1. Take all pairs (q, p) with $q \neq p$
2. Mark all pairs that accepting and non-accepting states
3. For all unmarked pairs (q, p) and all characters c tests wether

   $$(\delta(q, c), \delta(p, c))$$

   are marked. If yes, then also mark (q, p).
4. Repeat last step until no chance.
5. All unmarked pairs can be merged.


start
q_0
q_1
q_4
q_2
q_3
a
a
a, b
b
b
a
b
a
b





minimal automaton

- Assuming you have the alphabet $\{a, b, c\}$
- Give a regular expression that can recognise all strings that have at least one b.