DFAs vs. languages

- So far we discussed how to determine the language of a DFA.

$$A \rightsquigarrow L(A)$$

$$\text{program} \rightsquigarrow \text{specification}$$

- There is the 'reverse' task:

  Given a language L, design a DFA A such that the language $L(A)$ of A is L.

$$L \rightsquigarrow A$$

$$\text{specification} \rightsquigarrow \text{program}$$

Designing deterministic finite automata

Task: *Design a DFA* A *such that* $L(A)$ *consists of*

all *the strings of* 0*s and* 1*s ending with* 11*.*

How? creative task, needs thinking

Slogan: states = memory

(1) Determine what to remember about the input string, while the head is reading through it, from left to right.
Assign a state to each of the possibilities:

- state "waiting for 11"
- state "waiting for 1"
- state "OK at the moment"

(2) Add the transitions telling how the possibilities rearrange, and select the initial state and the favourable states.

(3) Test the automaton.

Designing deterministic finite automata (cont.)

It is sufficient to remember three 'states' of the **string read so far:**

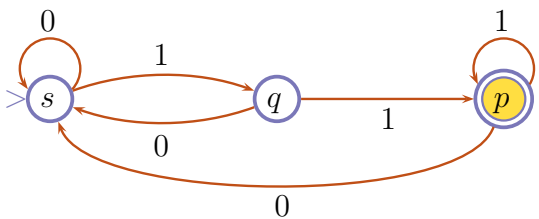
- The string does not end in 1. (Either it is ε or it ends in 0.)

  $\rightsquigarrow$ initial state s ("waiting for 11")

- The string is 1 or ends in 01. $\rightsquigarrow$ state q ("waiting for 1")

- The string ends in 11.

  $\rightsquigarrow$ favourable state p ("OK at the moment")



Designing deterministic finite automata II

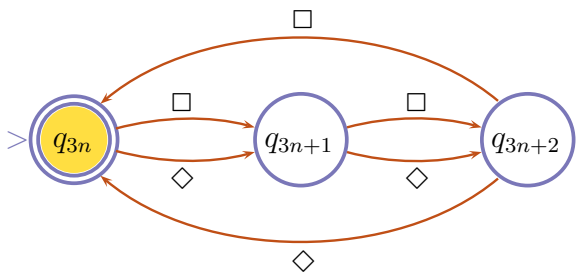
Task: *Design a DFA* A *such that* $L(A)$ *consists of* **all** *the strings of* $\square$*s and* $\diamond$*s whose length is divisible by* 3 *(that is, either* 0*, or* 3*, or* 6*, or . . . , or* $6033, \ldots$*)*

(1) Determine what to remember about the input string, while the head is reading through it, from left to right.

Assign a state to each of the possibilities: q_{3n}, q_{3n+1}, q_{3n+2}.

(2) Add the transitions telling how the possibilities rearrange, and select the initial state and the favourable states.

(3) Test the automaton.

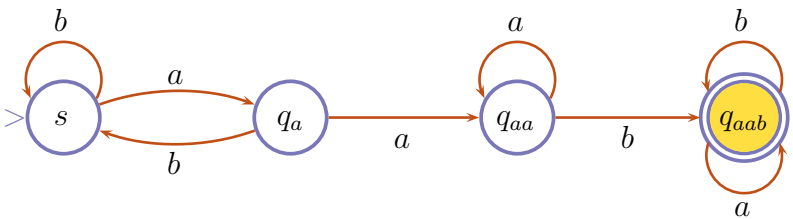


Pattern matching

Task: *Design a DFA* A *such that* $L(A)$ *consists of* **all** *the strings of* a*s and* b*s that contain* $\boxed{aab}$ *as a substring.*

There are four possibilities to remember:

- 'haven't seen any symbols of the pattern:' (initial) state s
- 'have just seen an a:' state q_a
- 'have just seen aa:' state q_{aa}
- 'have just seen the entire pattern aab:' (favourable) state q_{aab}



DFA as a theoretical model for programming

– unusual (visual), different from conventional programming languages

\+ major concepts of imperative programming (e.g. sequences, branching, loops) can be simulated

\+ elegant, simple to use (visual), has a thoroughly developed theory

\+ all pattern matching (=first step in WWW search engines) can be described

– limited expressive power (e.g. only constant amount of memory is available)

What about `method-calling'?

Combining two automata

Task: *Design an automaton A such that $L(A)$ consists of* **all** *the strings of as and bs that have either two consecutive as or two consecutive bs.*

- On Slide 193 there is a DFA A_4 accepting
  all the strings that have two consecutive as.

  This automaton can easily be transformed into an automaton A'_4 that accepts all the strings that have two consecutive bs.

- Since we already have the 'methods' that solve the problem for aas and, respectively, bbs, an obvious idea would be to design a 'main program' that would just 'call' the given methods.

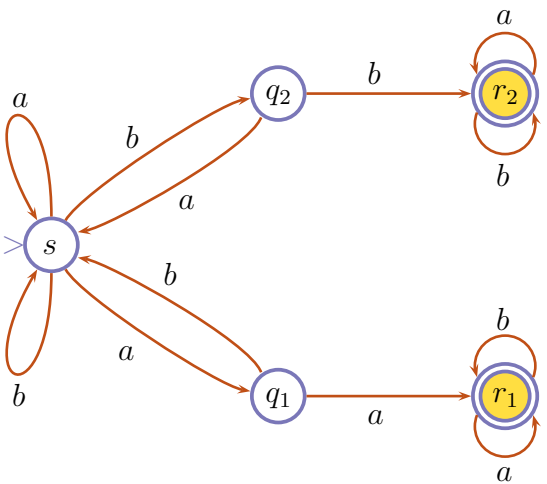
  A possible solution would be to 'combine' the initial states of A_4 and A'_4 (see next slide).

Combining two automata (cont.)

Given: $L(A_4) =$ all the words having two consecutive as

$L(A'_4) =$ all the words having two consecutive bs

Needed: automaton A such that $L(A) = L(A_4) \cup L(A'_4)$



Nondeterminism

Observe that the state transition diagram on the previous slide no longer represents a **DFA:**

- there is more than one a-arrow leaving state s,
- there is more than one b-arrow leaving state s.

We say that this automaton is **nondeterministic:**

Because, say, being in state s and reading a from the input tape, the device now has a choice: it can choose to move **either** to state q_1 **or** to state s.

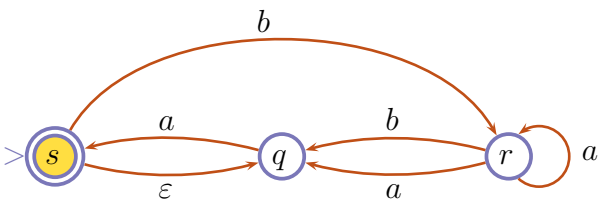
In other words, the next state is *not determined* by the previous state and the symbol read.

But then what does it mean that a word is accepted??

More nondeterminism

There are other permitted features in nondeterministic finite automata (**NFA**):

- There can be states and symbols such that **no** arrow labelled by that symbol leaves the state in question: reading that symbol in that state the device gets stuck (say, in the example below, there is no a-arrow leaving s).
- We also allow the machine to '**jump**' from a state to another state, without 'consuming' any input symbol from the tape (that is, the head does not move when 'jump' happens). The allowed 'state-jumps' will be indicated on the state transition diagrams by arrows labelled by the empty word ε.



Observe that transition tables are no longer adequate to describe NFA.

Acceptance of words by NFAs

In an NFA, there can be *more than one* computation on an input word.

Some of these may end up in a favourable state, while some others either get stuck, or may end up in a non-favourable state.

So, what is it supposed to mean that a word is **accepted** by an NFA ?

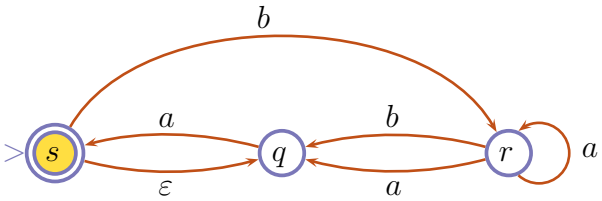
A word w is **accepted** by NFA A if *there is* a computation of A on input w that ends up in some *favourable state.*

- watch out: there can be many other, 'bad' computations on w
- watch out: 'getting stuck' in a favourable state does not mean acceptance
  (ends up = all symbols of the input word have been read)

Otherwise, A **rejects** the input word.
So an NFA can reject an input word w because *every* computation on w

- is either 'stuck'
- or ends up in a non-favourable state

Example: how an NFA works



- Computations on input $baba$:

  $(s, baba), (q, baba)$ (stuck)

  $(s, baba), (r, aba), (r, ba), (q, a), (s, \varepsilon)$ $\rightsquigarrow$ accepted ...

- Computations on input aa:

  $(s, aa), (q, aa), (s, a), (q, a), (s, \varepsilon)$ $\rightsquigarrow$ accepted

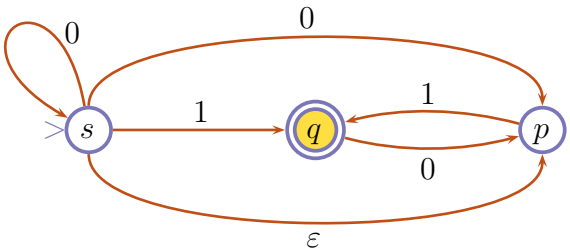
- Computations on input $babba$:

  $(s, babba), (q, babba)$ (stuck)

  $(s, babba), (r, abba), (r, bba), (q, ba)$ (stuck)

  $(s, babba), (r, abba), (q, bba)$ (stuck, no more) $\rightsquigarrow$ rejected

NFA: another example



- Computations on input 11:

  $(s, 11), (q, 1)$ (stuck)

  $(s, 11), (p, 11), (q, 1)$ (stuck, no more) $\rightsquigarrow$ rejected

- Computations on input 00:

  $(s, 00), (s, 0), (s, \varepsilon)$

  $(s, 00), (s, 0), (p, \varepsilon)$

  $(s, 00), (s, 0), (p, 0)$ (stuck)

  $(s, 00), (p, 0)$ (stuck)

  $(s, 00), (p, 00)$ (stuck, no more) $\rightsquigarrow$ rejected

- Computations on input 001:

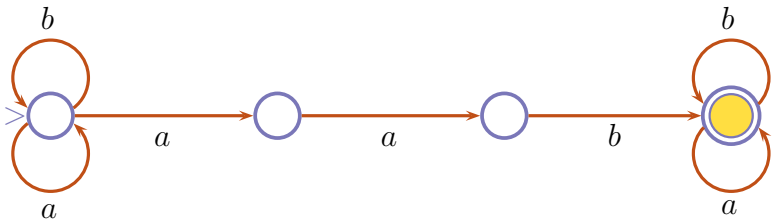
  $(s, 001), (s, 01), (s, 1), (q, \varepsilon)$ $\rightsquigarrow$ accepted ...

NFAs: what the whole fuss is about?

- Nondeterminism can be viewed as a kind of parallel computation model wherein several 'processes' can be running concurrently ('threads').
- We will see:
  Nondeterministic finite automata are much **easier to design** than deterministic ones.
- We will also see:
  Whatever can be done by a nondeterministic finite automaton, it can also be done (though usually in a bit more complicated way) by a deterministic one. In other words, nondeterminism **does not increase the computational power** of finite automata.
- NFAs are used in hardware (chip) design:
  - Given a task, an NFA is designed.
  - Then this NFA is turned to a DFA doing the same thing.
  - Then the resulting DFA is 'minimised'.
  - The obtained minimal DFA is hard-wired.

Pattern matching revisited

Task: *Design a finite automaton* A *such that* $L(A)$ *consists of* **all** *the strings of* a*s and* b*s that contain* $\boxed{aab}$ *as a substring.*



- Given the same task, it is much easier to design an NFA than a DFA for it: We don't have to worry about all computations. We just need to ensure that
  - if a word should be accepted,
    then there is at least one 'good' computation for it
  - if a word should be rejected, then there is no 'good' computation for it

Equivalence of DFAs and NFAs

- And it really does not matter:
  We will now see that every NFA can be turned to a DFA
  accepting precisely the same words.

Two automata A and A' that accept the same language are said to be **equivalent**.

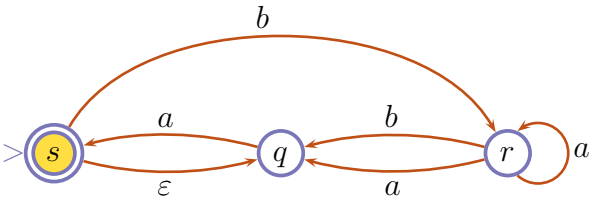
For every NFA A, there is a DFA A' equivalent to A.

Moreover, there is a 'mechanical method' (an *algorithm*) that carries out this conversion from a nondeterministic automaton to its deterministic equivalent:

$\rightsquigarrow$ the 'Subset Construction'

The Subset Construction

Task: Given an NFA



turn it to an equivalent **D**FA (**deterministic !**)

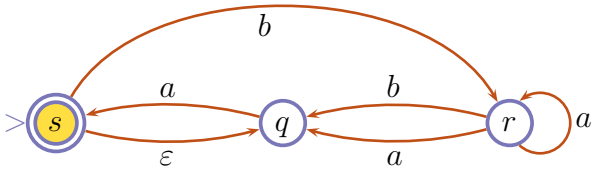
- watch out: in a DFA **no** choice, **no** 'getting stuck', **no** ε-jumps are allowed

Things to define:

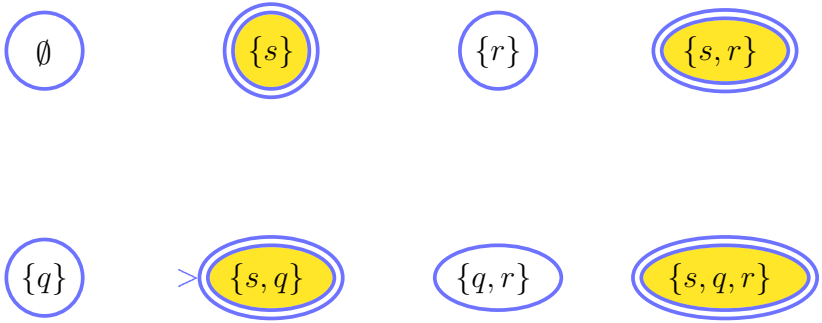
- new states ?
- new initial state ?
- new favourable states ?
- new transitions (arrows) ?

Subset construction: the new states

NFA:



- The new states: all the **subsets** of the NFA's states
- The new initial state: the set containing the NFA's initial state plus all those states that are reachable from it by some ε-jumps
- The new favourable states: those subsets that contain at least one of the NFA's favourable states

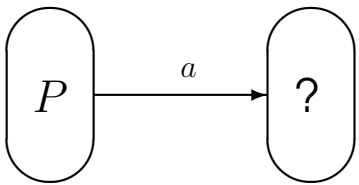


Subset construction: the new transition function

Recall: Being in a state and reading an input symbol, the transition function tells the automaton's next state.

And the new states are *subsets* of the 'old' states now.

General rule:



for each symbol a
in the input alphabet

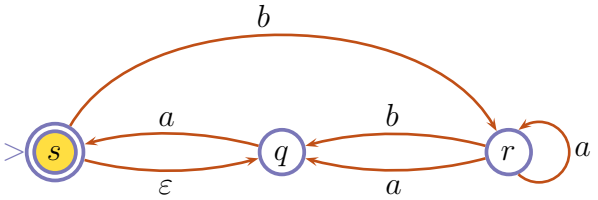
↑

the set of all those states that are reachable from some state in the set P

- either by an a-arrow,
- or by an a-arrow followed by possibly one or more ε-jumps

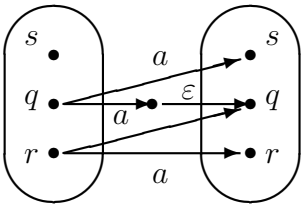
Example: computing the new transition function

NFA:



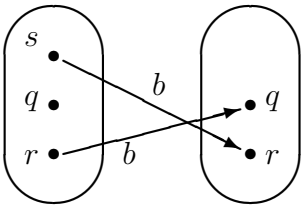
DFA: a-arrow leaving state $\{s, q, r\}$

'goes' to $\{s, q, r\}$



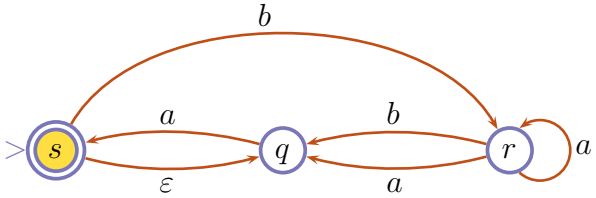
b-arrow leaving state $\{s, q, r\}$

'goes' to $\{q, r\}$

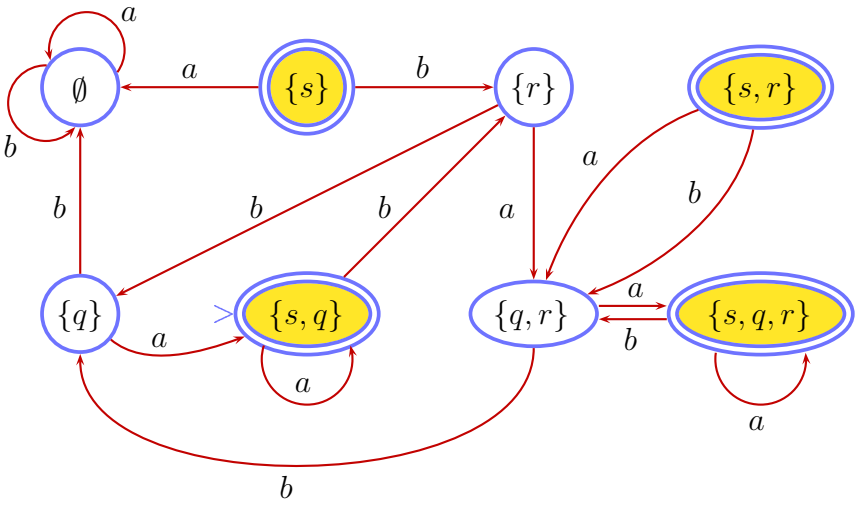


Subset construction: the new transition function

NFA:

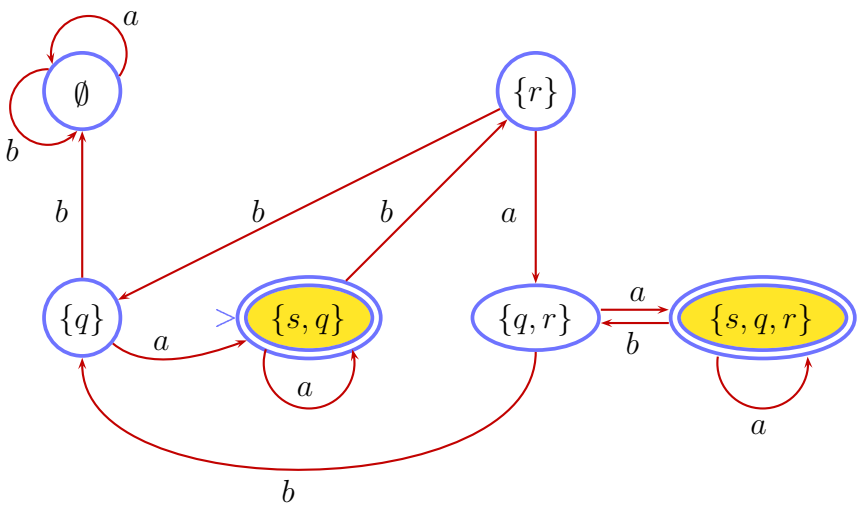


DFA:



Removing the unreachable states

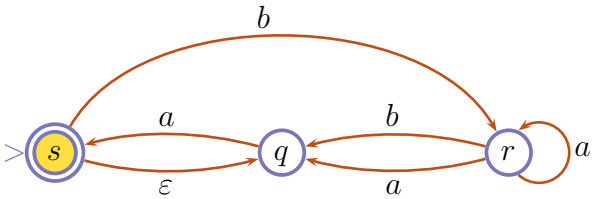
Observe that there is no computation going through the states $\{s\}$ and $\{s, r\}$, so we can safely remove them together with their outgoing arrows:



The initial state is never removable! (all computations start there)

Example: the new transition function with transition table

NFA:



DFA:

	a	b
$\{s\}$	$\emptyset$	$\{r\}$
$\{q\}$	$\{s,q\}$	$\emptyset$
$\{r\}$	$\{q,r\}$	$\{q\}$
$\{s,q\}$	$\{s,q\}$	$\{r\}$
$\{s,r\}$	$\{q,r\}$	$\{q,r\}$
$\{q,r\}$	$\{s,q,r\}$	$\{q\}$
$\{s,q,r\}$	$\{s,q,r\}$	$\{q,r\}$
$\emptyset$	$\emptyset$	$\emptyset$

State minimisation problem

We saw that it is very easy to design nondeterministic finite automata for, say, pattern matching problems. However, an NFA cannot be directly implemented as a computer program. One must convert it into a DFA. But the number of states of the resulting DFA may increase significantly:

$$2^{\text{number of states of the original NFA}} \quad \text{at the worst case}$$

An important field where finite automata are being used is *hardware design*: some components of hardware are based on simulation of DFA. An important objective here is to use finite automata that have a *minimum* possible number of states. There are methods for doing this (see recommended literature).
The 'subset construction' and the 'deletion of unreachable states' are just the initial phases of design.