0.1 Nondeterministic Automata

0.1.1 Motivation

In the previous lecture we have investigated the **semantics** of regular expressions and saw that we can determine the language accepted by, e.g. $(A \cup B)(a \cup b)*(0 \cup 1)*$. However, given a regular expression e. we are missing a **computational means** for determining if a given word w is a member of $L(e)$, and this is precisely the task of the **lexical stage**.
In more formal terms, we have a *generator* for languages, but we lack a means for *accepting* (the words for) languages.
We shall informally illustrate an algorithm for verifying the membership $w \in L((A \cup B)(a \cup b) * (0 \cup 1)*)$.

- **input:** a word `w=c1c2 ... cn`.
- define a *set of integers* `s` . Set `s={0}`
- for each `ci` in `w`:
  - if `s=={0}` and `ci==A` or `ci==B` then `s=={1,2}`
  - if `1∈s` and `ci==a` or `ci==b` then `s=={1}`
  - if `1∈s` and `ci==0` or `ci==1` then `s=={2}`
  - if `2∈s` and `ci==0` or `ci==1` then `s=={2}`
  - otherwise return `false`
- if `2∈s` or `1∈s` then return `true`
  - otherwise return `false`

The idea underlying the algorithm is the *state variable* `s`.

- Initially, `s={0}`, which means that we are at the beginning of the word. If this is so, the first symbol must be `A` or `B`, otherwise the word is not accepted by the regexp.
- After the correct first-symbol was read, we might expect a sequence of `a`s and `b`s or a sequence of `0`s and `1`s. We do not know that in advance, hence the state variable is `{1,2}`, modelling this incomplete knowledge.
- If `1` is a possible current state and we have read `a` or `b` then the current state is surely `1`. As long as this is so, we continue to process symbols.

- If `2` is a possible current state and we have read `0` or `1` then the current state is surely `2`. As long as this is so, we continue to process symbols.
- If the end of the word has been found while on state `2`, we stop and report `true`. In any other situation, we report `false`.

0.1.2 Nondeterministic automata

The key idea behind the previous algorithm can be generalised to **any** regular expression. In order to do that, we require the concept of **nondeterministic finite automaton** (NFA). We will soon discover some similarities between NFAs and the previous algorithm.

Definition (NFA)**.** *A* ***non-deterministic finite automaton*** *is a tuple* $M = (K, \Sigma, \Delta, q_0, F)$ *where:*

- K *is a finite set of* ***states***
- Σ *is an alphabet*
- Δ *is a* ***subset*** *of* $K \times \Sigma^* \times K$ *and is called a* ***transition relation***
- $q_0 \in K$ *is* ***the initial state***
- $F \subseteq K$ *is* ***the set of final states***

As an example, consider:

- $K = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- $\Delta = \{(q_0, 0, q_0), (q_0, 1, q_0), (q_0, 0, q_1), (q_1, 1, q_2)\}$
- $F = \{q_2\}$

Notice that the NFA gets stuck for certain inputs, i.e. it **does not accept**.
Graphical notation

Definition (Configuration)**.** *A* ***configuration*** *of an NFA, is a* ***member*** *of* $K \times \Sigma^*$.

Informally, configurations capture a **snapshot** of the execution of an NFA. The snapshot consists of the:

- **current state** of the automaton and

- **the rest of the word** from the input.

For instance, $(q_0, 0001)$ is the **initial configuration** of the automaton from our example, on input 0001.

Definition (Transition)**.** *We call* $\vdash_M \subseteq (K \times \Sigma^*) \times (K \times \Sigma^*)$ *a* ***one-step*** *move relation of automaton* M*. The relation describes how the automaton* ***must behave*** *to reach one configuration from another. Formally:*

- $(q, w) \vdash_M (q', w')$ *if and only if there exists* $u \in \Sigma^*$*, such that* $w = uw'$ *(*u *is a prefix of* w*) and* $(q, u, q') \in \Delta$*: from state* q *on input* u *we reach state* q'*.*

We call $\vdash_M^*$*, the* ***reflexive and transitive closure of*** $\vdash_M$*, i.e. the* ***zero-or-more step(s)*** *move of automaton* M*.*

For instance, in our previous example, $(q_0, 0001) \vdash_M (q_0, 001)$ and also $(q_0, 0001) \vdash_M (q_1, 001)$. At the same time, $(q_0, 0001) \vdash_M^* (q_2, \epsilon)$. Can you figure out the sequence of steps?

Preposition (Acceptance)**.** *A word* w *is accepted by an NFA* M *iff* $(q_0, w) \vdash_M^* (q, \epsilon)$ *and* $q \in F$*. In other words, after the word* w *was processed by the automaton, we reach a* ***final state****.*

Notice that the word 0001 is indeed accepted by the automaton M from our example.

Definition (Language accepted by an NFA)**.** *Given an NFA* M*, we define* $L(M) = \{w \mid w \text{ is accepted by} M\}$ *as the language* ***accepted*** *by* M*. We say* M *accepts the language* $L(M)$*.*

Execution tree for Nondeterministic Finite Automata

Illustration of an AFN for $(A \cup B)(a \cup b) * (0 \cup 1)*$.
There are two ways of writing this automaton:

- one that follows exactly our previous algorithm sketch.
- one that employs **epsilon transitions**.

Epsilon transitions are a means for jumping from a state to another without consuming the input. It is a useful way of defining automata, because it empowers us to **combine** multiple automata procedures.

0.1.3 Nondeterminism as imperfect information

Notice that **nondeterminism** actually refers to our imperfect information regarding the current state of the automaton. **Nondeterminism** means that, after consuming some part (prefix) of a word, *several concrete states may be possible current states*.

0.1.4 From Regular Expressions to NFAs

While Regular Expressions are a natural instrument for declaring (or generating) tokens, NFAs are a **natural instrument for accepting** tokens (i.e. their respective language).
The following theorem shows how this can be achieved.
For every language $L(E)$ defined by the regular expression E, there exists an NFA M, such that $L(M) = L(E)$.
This theorem is particularly important, because it also provides an **algorithm** for constructing NFAs from regular expressions.

Proof. Let E be a regular expression. We construct an NFA, with:

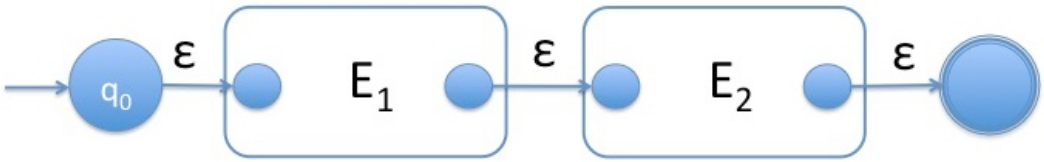
- **exactly one initial state**.
- **exactly one final state**.
- **no transitions from the final state**.

The proof is by **induction** over the expression structure.
Basis case $E = \emptyset$
We construct the following automaton:

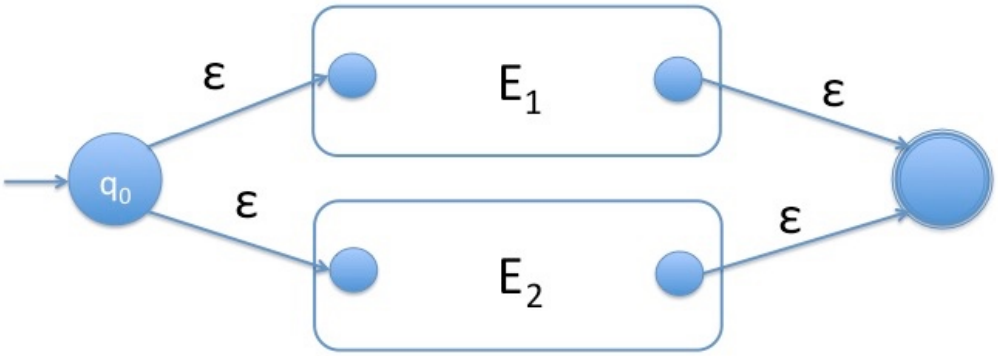
It is clear that this automaton accepts no word, and obeys the three aforementioned conditions.
Basis case $E = \epsilon$
We construct the following automaton:
hich only accepts the empty word.

Basis case $E = c$ where c is a symbol of the alphabet.
We construct the following automaton:

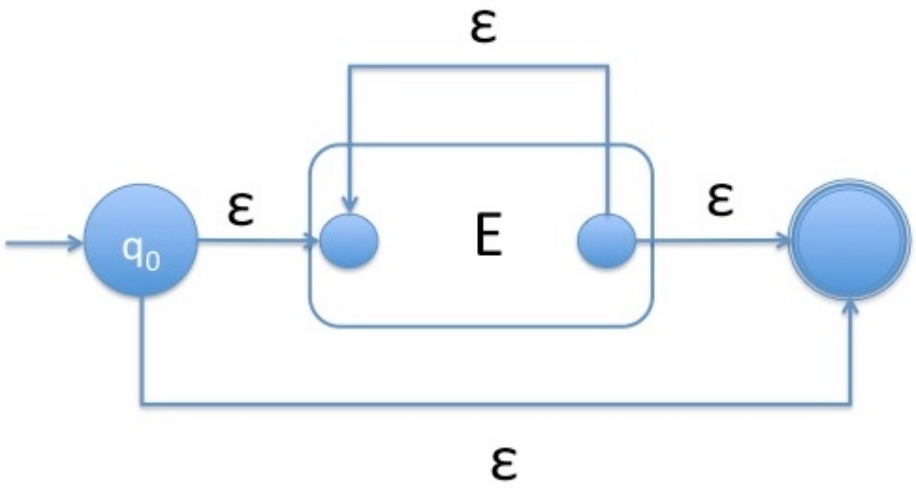
Since regular expressions have three *inductive rules* for constructing regular expressions (union, concatenation and Kleene-star), we have to treat three induction steps:
Induction step $E = E_1E_2$ **(concatenation)**
Suppose E_1 and E_2 are regular expressions for which NFAs can be built (**induction hypothesis**). We build the following NFA which accepts all words generated by the regular expression E_1E_2.



Induction step $E = E_1 \cup E_2$ **(union)**
Suppose E_1 and E_2 are regular expressions for which NFAs can be built (**induction hypothesis**). We build the following NFA which accepts all words generated by the regular expression $E_1 \cup E_2$.
Induction step E^* **(union)**

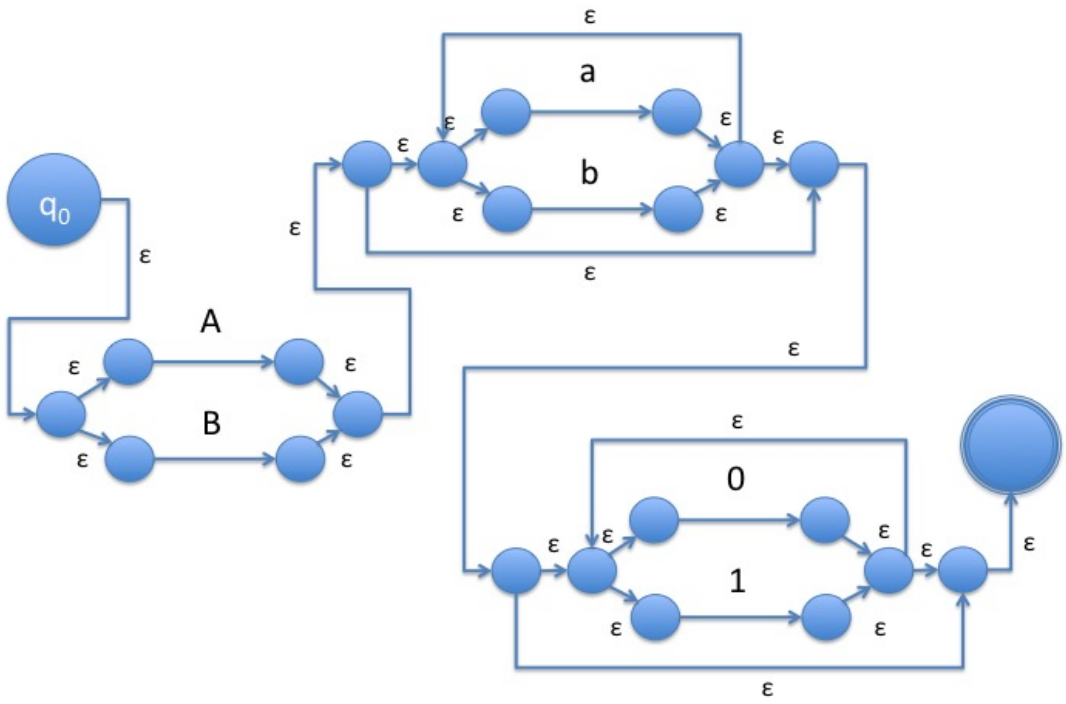


Suppose E is regular expression for which an NFA can be built (**induction hypothesis**). We build the following NFA which accepts all words generated by the regular expression $E*$.



□

We illustrate the algorithmic procedure on our regular expression $(A \cup B)(a \cup b) * (0 \cup 1)*$. The result is shown below:


q0
ε
A
B
a
b
0
1