Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2018

Lecture 5

Ana Bove

March 26th 2018

Recap: Inductive sets, (terminating) recursive functions, structural induction

- To define an inductive set S we
  - state its basic elements
  - and construct new elements in terms of already existing ones;
- To define a recursive function f over an inductively defined set S we
  - define f on the basic elements
  - and define f on the *recursive* elements in terms of the result of f for the *structurally smaller* ones;
- To prove a property P over an inductively defined set S we
  - prove that P holds for the basic elements
  - and assuming that P holds of certain elements in the set, prove that P holds for all ways of constructing new elements from existing ones;
- Using structural induction we prove properties over *all* (finite) elements in an inductive set;
- Mathematical/simple and course-of-values/strong induction, or mutual induction are special cases of structural induction.

Overview of Today's Lecture

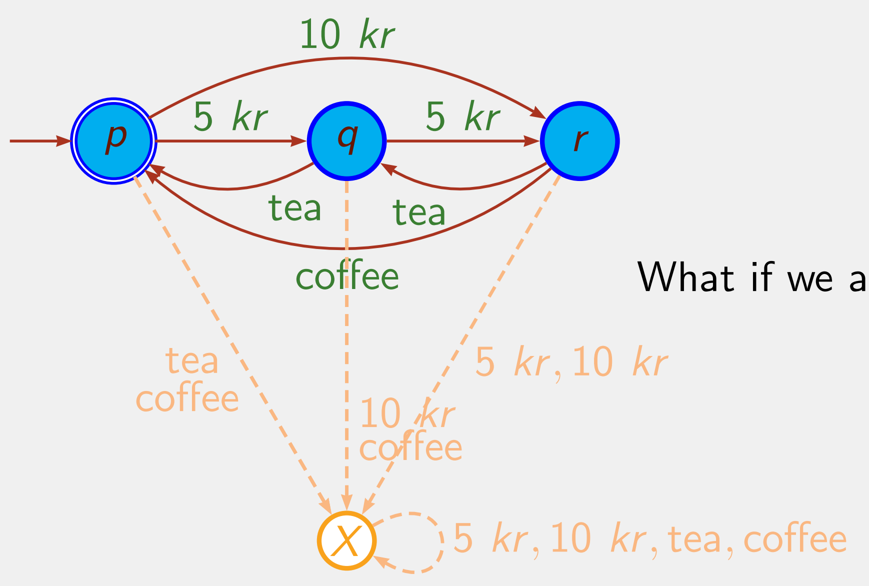
- DFA: deterministic finite automata.

Contributes to the following learning outcome:

- Explain and manipulate the different concepts in automata theory and formal languages;
- Understand the power and the limitations of regular languages and context-free languages;
- Design automata, regular expressions and context-free grammars accepting or generating a certain language;
- Describe the language accepted by an automata or generated by a regular expression or a context-free grammar;
- Determine if a certain word belongs to a language;
- Differentiate and manipulate formal descriptions of languages, automata and grammars.

Deterministic Finite Automata

We have already seen examples of DFA:



What if we ask for coffee in q?

Formally all non-drawn "actions" go to a *dead* state X in a DFA!

We will usually not draw them.

Deterministic Finite Automata: Formal Definition

Definition: A *deterministic finite automaton* (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ consisting of:

1. A finite set Q of *states*;
2. A finite set Σ of *symbols* (alphabet);
3. A total *transition function* $\delta : Q \times \Sigma \to Q$;
4. A *start state* $q_0 \in Q$;
5. A set $F \subseteq Q$ of *final* or *accepting* states.

Example: DFA

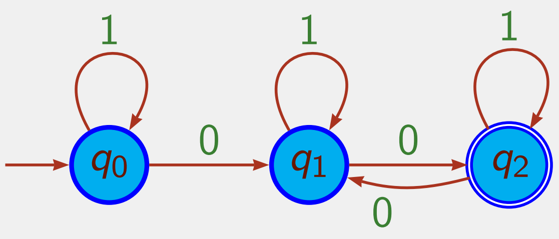
Let the DFA $(Q, \Sigma, \delta, q_0, F)$ be given by:

$$
\begin{array}{l}
Q = \{q_0, q_1, q_2\} \\
\Sigma = \{0, 1\} \\
F = \{q_2\} \\
\delta : Q \times \Sigma \to Q \\
\quad \begin{array}{lll}
\delta(q_0, 0) = q_1 & \delta(q_1, 0) = q_2 & \delta(q_2, 0) = q_1 \\
\delta(q_0, 1) = q_0 & \delta(q_1, 1) = q_1 & \delta(q_2, 1) = q_2
\end{array}
\end{array}
$$

What does it do?

How to Represent a DFA?

Transition Diagram: Helps to understand how it works.



The start state is indicated with $\rightarrow$.
The final states are indicated with a double circle.

Transition Table:

δ	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_1
$*q_2$	q_1	q_2

The start state is indicated with $\rightarrow$.

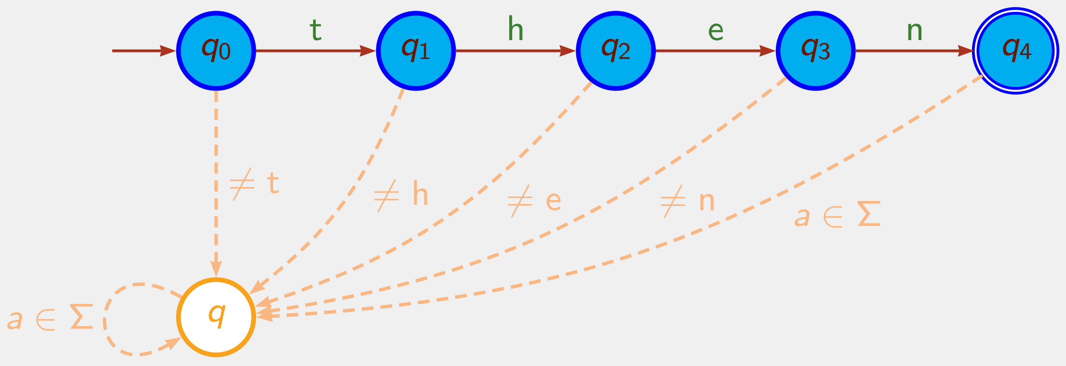
The final states are indicated with a $*$.

When Does a DFA Accept a Word?

When reading the word the automaton moves according to δ.

Definition: One starts reading the word from the start state of the automaton. If when the whole word is read the automaton is on a final state, then the automaton *accepts* the word.

Example:



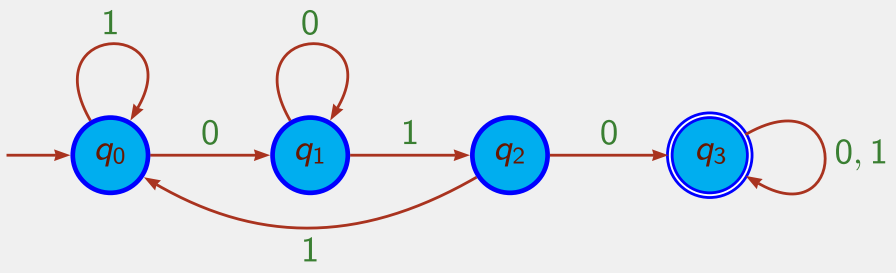
Only the word "then" is accepted.

We have a (non-accepting) *dead* state q.

Example: DFA

Given $\Sigma = \{0,1\}$, we want an automaton accepting the words that contain 010 as a subword, that is, the language $\mathcal{L} = \{x010y \mid x, y \in \Sigma^*\}$.

Solution: $(\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_3\})$ given by



δ	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
q_2	q_3	q_0
$*q_3$	q_3	q_3

Extending the Transition Function to Strings

How can we compute what happens when we read a certain word?

Definition: We extend δ to strings as $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$.

We define $\hat{\delta}(q, x)$ by recursion on x.

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, ax) &= \hat{\delta}(\delta(q, a), x) \end{aligned}$$

Note: $\hat{\delta}(q, a) = \delta(q, a)$ since the string $a = a\epsilon$.

$$\hat{\delta}(q, a) = \hat{\delta}(q, a\epsilon) = \hat{\delta}(\delta(q, a), \epsilon) = \delta(q, a)$$

Example: In the example of slide 8, what are $\hat{\delta}(q_0, 10101)$ and $\hat{\delta}(q_0, 00110)$?

Reading the Concatenation of Two Words

Proposition: *For any words x and y, and for any state q we have that $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$.*

Proof: We prove $P(x) = \forall\, q. \forall\, y. \hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ by (structural) induction on x.

Base case: $\forall\, q. \forall\, y. \hat{\delta}(q, \epsilon y) = \hat{\delta}(q, y) = \hat{\delta}(\hat{\delta}(q, \epsilon), y)$.

Inductive step: Our IH is that $\forall\, q. \forall\, y. \hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ for a given word x.

We should prove that $\forall\, q. \forall\, y. \hat{\delta}(q, (ax)y) = \hat{\delta}(\hat{\delta}(q, ax), y)$, for $a \in \Sigma$.

For a given q and y we have that

$$\begin{aligned}
\hat{\delta}(q, (ax)y) &= \hat{\delta}(q, a(xy)) && \text{by def of concat}\\
&= \hat{\delta}(\delta(q, a), xy) && \text{by def of } \hat{\delta}\\
&= \hat{\delta}(\hat{\delta}(\delta(q, a), x), y) && \text{by IH with state } \delta(q, a)\\
&= \hat{\delta}(\hat{\delta}(q, ax), y) && \text{by def of } \hat{\delta}
\end{aligned}$$

Another Definition of $\hat{\delta}$

Recall that we have 2 descriptions of words: $a(b(c(d\epsilon))) = (((\epsilon a)b)c)d$.

We can define $\hat{\delta}'$ as:
$$\begin{aligned}
\hat{\delta}'(q, \epsilon) &= q\\
\hat{\delta}'(q, xa) &= \delta(\hat{\delta}'(q, x), a)
\end{aligned}$$

Proposition: $\forall\, x. \forall\, q. \hat{\delta}(q, x) = \hat{\delta}'(q, x)$.

Proof: We prove $P(x) = \forall\, q. \hat{\delta}(q, x) = \hat{\delta}'(q, x)$ by (structural) induction on x.

Observe that xa is a special case of xy where $y = a$.

Base case is trivial.

Inductive step: The IH is $\forall\, q. \hat{\delta}(q, x) = \hat{\delta}'(q, x)$ for a given x. Then for $a \in \Sigma$

$$\begin{aligned}
\hat{\delta}(q, xa) &= \hat{\delta}(\hat{\delta}(q, x), a) && \text{by previous prop}\\
&= \delta(\hat{\delta}(q, x), a) && \text{by def of } \hat{\delta}\\
&= \delta(\hat{\delta}'(q, x), a) && \text{by IH}\\
&= \hat{\delta}'(q, xa) && \text{by def of } \hat{\delta}'
\end{aligned}$$

Language Accepted by a DFA

Definition: The *language* accepted by the DFA $(Q, \Sigma, \delta, q_0, F)$ is the set $\mathcal{L} = \{x \mid x \in \Sigma^*, \hat{\delta}(q_0, x) \in F\}$.

Example: In the example on slide 8, 10101 is accepted but 00110 is not.

Note: We could write a program that simulates a DFA and let the program tell us whether a certain string is accepted or not!

Functional Representation of a DFA Accepting $x010y$

```
data Q = Q0 | Q1 | Q2 | Q3
data S = O | I

final :: Q -> Bool
final Q3 = True
final _ = False

delta :: Q -> S -> Q
delta Q0 O = Q1
delta Q0 I = Q0
delta Q1 O = Q1
delta Q1 I = Q2
delta Q2 O = Q3
delta Q2 I = Q0
delta Q3 _ = Q3
```

Functional Representation of a DFA Accepting *x*010*y*

```
delta_hat :: Q -> [S] -> Q
delta_hat q [] = q
delta_hat q (a:xs) = delta_hat (delta q a) xs


accepts :: [S] -> Bool
accepts xs = final (delta_hat Q0 xs)
```

Alternatively, (just for those knowing Haskell :-)

```
run :: Q -> [S] -> Q
run = foldl delta

accepts' :: [S] -> Bool
accepts' = final . run Q0
```

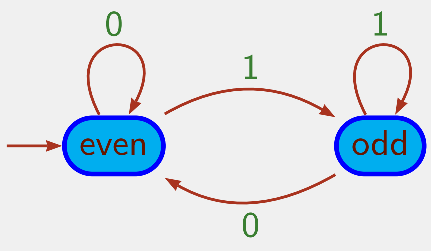
Accepting by End of String

We could use an automaton to identify properties of a certain string.

What is important then is the state the automaton is in when we finish reading the input.

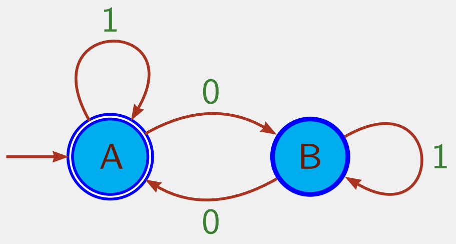
The set of final states is actually of no interest here and can be omitted.

Example: The following automaton determines whether a binary number is even or odd.



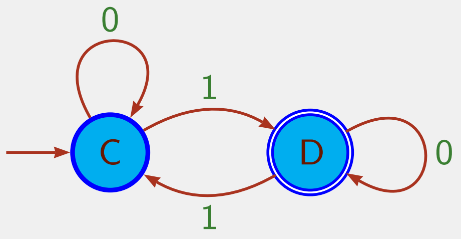
Product of Automata

Given this automaton over $\{0, 1\}$ accepting strings with an even number of 0's:



State A: even number of 0's State
B: odd number of 0's

and this automaton accepting strings with an odd number of 1's:

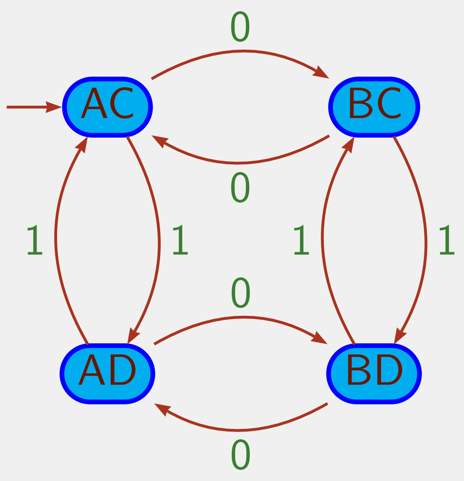


State C: even number of 1's State
D: odd number of 1's

How can we use them to accept the strings with an even nr. of 0's *and* an odd nr. of 1's?

We can *run* them in parallel!

Example: Product of Automata



State AC: even nr. of 0's and 1's

State BC: odd nr. of 0's and
even nr. of 1's

State AD: even nr. of 0's and
odd nr. of 1's

State BD: odd nr. of 0's and 1's

Which is(are) the final state(s)? AD!

Product Construction

Definition: Given two DFA $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ with the *same alphabet* Σ, we can define the *product* $D = D_1 \otimes D_2 = (Q, \Sigma, \delta, q_0, F)$ as follows:

- $Q = Q_1 \times Q_2$;
- $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$;
- $q_0 = (q_1, q_2)$;
- $F = F_1 \times F_2$.

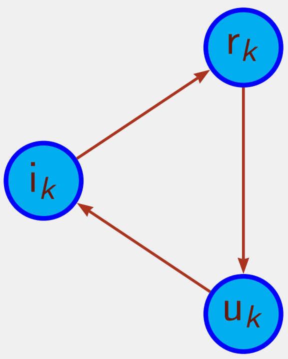
Proposition:
$\forall\, x \in \Sigma^*. \forall\, r_1 \in Q_1. \forall\, r_2 \in Q_2. \hat{\delta}((r_1, r_2), x) = (\hat{\delta}_1(r_1, x), \hat{\delta}_2(r_2, x))$.

Proof: By (structural) induction on x.

Example: Product of Automata

Consider a system where users have three states: *idle*, *requesting* and *using*.

Each user k is represented by a simple automaton:


$$\begin{tikzcd}
 & r_k \arrow[dd] \\
i_k \arrow[ur] & \\
 & u_k \arrow[ul]
\end{tikzcd}$$

If we have only 2 users, how does the whole system look like?

Example: Product of Automata (cont.)

The complete system is represented by the product of these 2 automata and it has 3 * 3 = 9 states.

$$\begin{tikzcd}
\boxed{i_1i_2} \arrow[r] \arrow[d] & \boxed{r_1i_2} \arrow[r] \arrow[d] & \boxed{u_1i_2} \arrow[d] \arrow[ll, bend right] \\
\boxed{i_1r_2} \arrow[r] \arrow[d] & \boxed{r_1r_2} \arrow[r] \arrow[d] & \boxed{u_1r_2} \arrow[d] \arrow[ll, bend right] \\
\boxed{i_1u_2} \arrow[r] \arrow[uu, bend left] & \boxed{r_1u_2} \arrow[r] \arrow[uu, bend right] & \boxed{u_1u_2} \arrow[ll, bend left] \arrow[uu, bend right]
\end{tikzcd}$$

How will it look like with n users?

Language Accepted by a Product Automaton

Proposition: *Given two DFA D_1 and D_2, then* $\mathcal{L}(D_1 \otimes D_2) = \mathcal{L}(D_1) \cap \mathcal{L}(D_2)$.

Proof: $\hat{\delta}(q_0, x) = \hat{\delta}((q_1, q_2), x) = (\hat{\delta}_1(q_1, x), \hat{\delta}_2(q_2, x)) \in F$
iff $\hat{\delta}_1(q_1, x) \in F_1$ and $\hat{\delta}_2(q_2, x) \in F_2$.
That is, $x \in \mathcal{L}(D_1)$ and $x \in \mathcal{L}(D_2)$ iff $x \in \mathcal{L}(D_1) \cap \mathcal{L}(D_2)$.

Note: It can be quite difficult to directly build an automaton accepting the intersection of two languages.

Exercise: Build a DFA for the language that contains the subword *abb* twice and an even number of *a*'s.

Variation of the Product

Definition: We define $D_1 \oplus D_2$ similarly to $D_1 \otimes D_2$ but with a different notion of accepting state:

$$\text{a state } (r_1, r_2) \text{ is accepting iff } r_1 \in F_1 \textit{ or } r_2 \in F_2$$

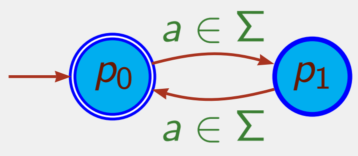
Proposition: *Given two DFA D_1 and D_2, then* $\mathcal{L}(D_1 \oplus D_2) = \mathcal{L}(D_1) \cup \mathcal{L}(D_2)$.

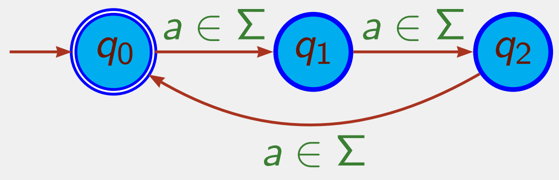
Example: In the automaton in slide 17, which is(are) the final state(s) if we want the strings with an even number of 0's *or* an odd number of 1's?

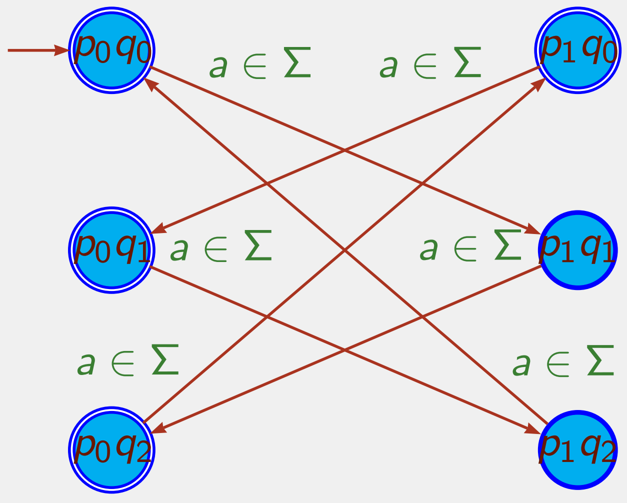
AC, AD and BD!

Example: Variation of the Product

Let us define an automaton accepting strings with lengths multiple of 2 or of 3.





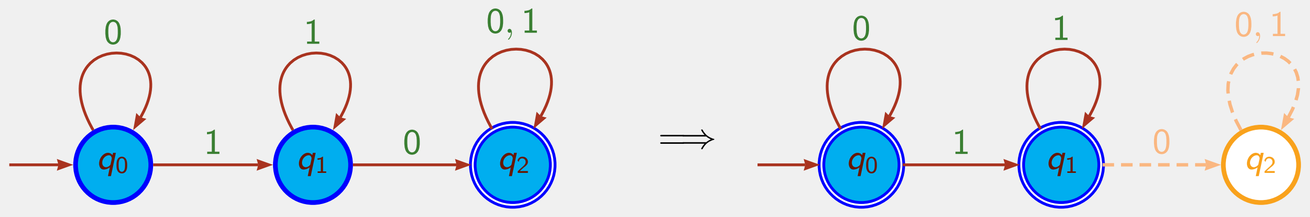


Complement

Definition: Given the automaton $D = (Q, \Sigma, \delta, q_0, F)$ we define the *complement* $\overline{D}$ of D as the automaton $\overline{D} = (Q, \Sigma, \delta, q_0, Q - F)$.

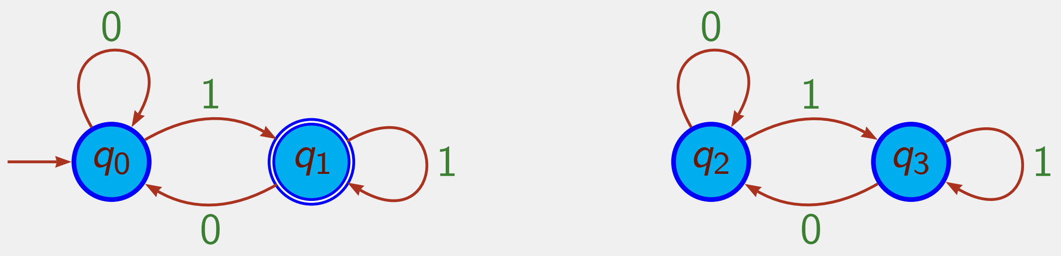
Proposition: *Given a DFA D we have that $\mathcal{L}(\overline{D}) = \Sigma^* - \mathcal{L}(D) = \overline{\mathcal{L}(D)}$.*

Example: We transform an automaton accepting strings containing 10 into an automaton accepting strings *NOT* containing 10.

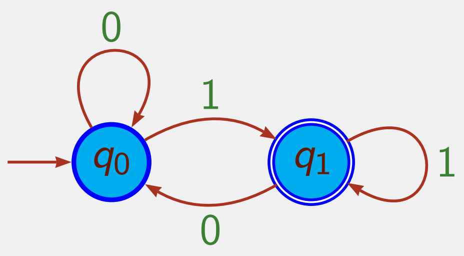


Accessible Part of a DFA

Consider the DFA $D = (\{q_0, \ldots, q_3\}, \{0, 1\}, \delta, q_0, \{q_1\})$ given by



Intuitively, this is equivalent to the DFA



which is the *accessible* part of the D.

q_2 and q_3 are not accessible from the start state and can be removed.

Accessible States

Definition: The set $\text{Acc} = \{\hat{\delta}(q_0, x) \mid x \in \Sigma^*\}$ is the set of *accessible* states (from the state q_0).

Proposition: *If $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA, then $D' = (Q \cap \text{Acc}, \Sigma, \delta|_{Q \cap \text{Acc}}, q_0, F \cap \text{Acc})$ is a DFA such that $\mathcal{L}(D) = \mathcal{L}(D')$.*

Proof: Notice that D' is well defined and that $\mathcal{L}(D') \subseteq \mathcal{L}(D)$.

If $x \in \mathcal{L}(D)$ then $\hat{\delta}(q_0, x) \in F$. By definition $\hat{\delta}(q_0, x) \in \text{Acc}$.
Hence $\hat{\delta}(q_0, x) \in F \cap \text{Acc}$ and then $x \in \mathcal{L}(D')$.

Regular Languages

Recall: Given an alphabet Σ, a *language* $\mathcal{L}$ is a subset of Σ^*, that is, $\mathcal{L} \subseteq \Sigma^*$.

Definition: A language $\mathcal{L} \subseteq \Sigma^*$ is *regular* iff there exists a DFA D on the alphabet Σ such that $\mathcal{L} = \mathcal{L}(D)$.

Recall: Recall that a DFA can only remember a finite amount of things!

Proposition: *If $\mathcal{L}_1$ and $\mathcal{L}_2$ are regular languages then so are $\mathcal{L}_1 \cap \mathcal{L}_2$, $\mathcal{L}_1 \cup \mathcal{L}_2$ and $\Sigma^* - \mathcal{L}_1$.*

Proof: ...

Overview of Next Lecture

Sections 2.3–2.3.5, brief on 2.4:

- NFA: Non-deterministic finite automata;
- Equivalence between DFA and NFA.

Overview of next Week

Mon 9	Tue 10	Wed 11	Thu 12	Fri 12
	Ex 10-12 EA DFA, NFA.		**10-12 EL41 Individual help**	
Lec 13-15 HB3 NFA, FA$\leftrightarrow$NFA.			**Lec 13-15 HB3** NFA, ϵ-NFA.	
Ex 15-17 EA DFA, NFA.		**15-17 EL41 Consultation**		

Assignment 2: DFA, NFA.
Deadline: Sunday 15th April 23:59.