Automata

Automata in computer science

In computer science:

automaton = abstract computing device, or "machine"

Automata in this lecture

- **Turing machines** (1937) and **abacus machines** (1960s): have all capabilities of today's computers. Used to study the boundary between **computable** and **uncomputable**.
- **Finite automata** (also called **finite state machines**, emerged during the 1940's and 1950's): useful e.g. text search, protocol verification, compilers, descriptions of certain **formal grammars** (N. Chomsky, 1950's).

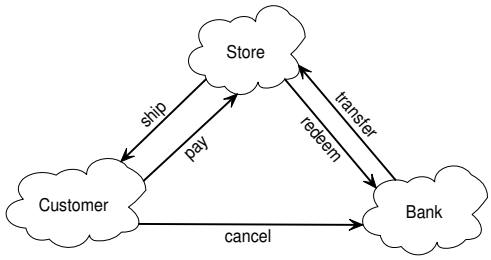
Finite automata

We shall study finite automata first, because they can be seen as a first step towards Turing machines and abacus machines.

Uses of finite automata

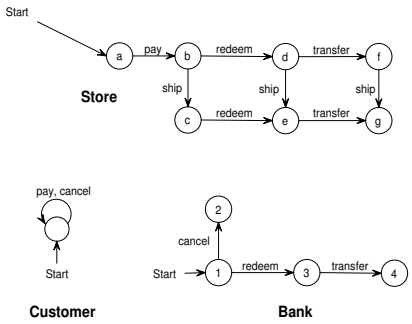
- Used in software for verifying all kinds of systems with a finite number of states, such as communication protocols
- Used in software for scanning text, to find certain patterns
- Used in "Lexical analyzers" of compilers (to turn program text into "tokens", e.g. identifiers, keywords, brackets, punctuation)
- Part of Turing machines and abacus machines

Example: comm. protocol



Customer, Store, and Bank will be finite automata.

Communication protocol



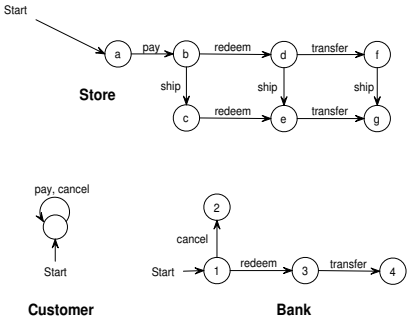
Simulating the whole system

- Idea: running Customer, Store, and Bank "in parallel".
- Initially, each automaton is in its start position.
- The system can move on for every action that is possible in **each** of the three automata.

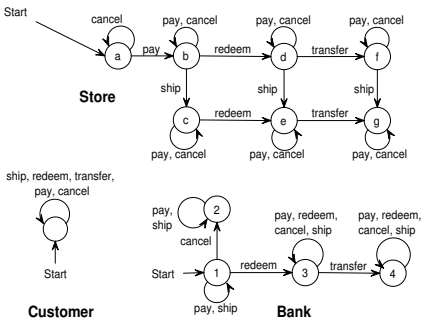
The missing irrelevant actions

- Problem: Bank gets stuck during the pay action, although paying is only between Customer and Store.
- Solution: we need to add a loop labeled “pay” to state 1 of Bank.
- More generally, we need loops for all such irrelevant actions.
- But illegal actions should remain impossible. E.g. Bank should not allow “redeem” after “cancel”.

Adding irrelevant actions



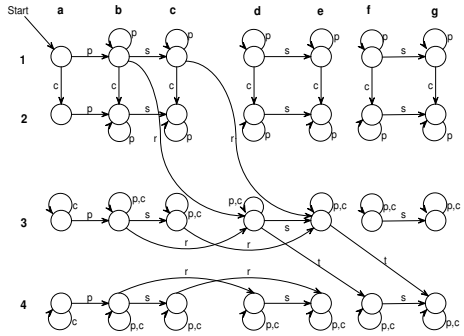
Adding irrelevant actions



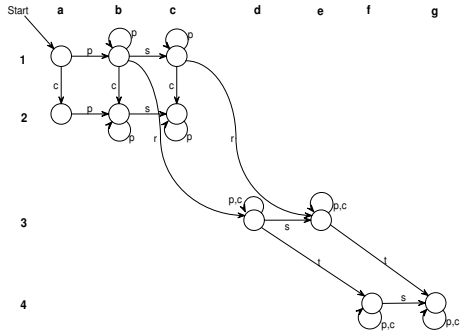
Simulating the whole system

- Simulation by **product automaton**.
- Its states are pairs $(StoreState, BankState)$, e.g. (a,1) or (c,3). (Because Customer has only one state and allows every action, it can be neglected.)
- It has a transition $(StoreState, BankState) \xrightarrow{action} (StoreState', BankState')$ whenever Store has a transition $StoreState \xrightarrow{action} StoreState'$ and Bank has a transition $BankState \xrightarrow{action} BankState'$.

Product automaton



Without unreachable states



Usefulness for protocol verification

- We can now answer all kinds of interesting questions, e.g. “Can it happen that Store ships the product and never receives the money transfer?”
- Yes! If Customer has indicated to pay, but sent a cancellation message to the Bank, we are in state (b,2). If Store ships then, we make a transition into (c,2), and the Store will never receive a money transfer!
- So store should never ship before redeeming.

Automata in computer science

- In computer science, an **automaton** is an abstract computing machine.
- “Abstract” means here that it need not exist in physical form, but only as a precisely-described idea.

- We saw an example of a communication protocol that we could simulate with finite state automaton.
- We drew the finite state automaton: circles represented states, labelled arrows represented transitions. We labelled a start state and we said that a machine would "die" if there was no transition to follow from the current state to another.
- We are going to give the formal definition of a deterministic finite state automaton.

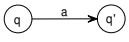
Formal definition of DFA's

Definition. A **deterministic finite automaton** (**DFA**) consists of

- a finite set of **states**, often denoted Q,
- a finite set Σ of **input symbols**,
- a total **transition function** $\delta : Q \times \Sigma \to Q$,
- a **start state** $q_0 \in Q$, and
- a set $F \subseteq Q$ of **final** or **accepting states**.

Remark: we require the transition function to be total, but some people allow it to be partial.

Terminology and intuitions

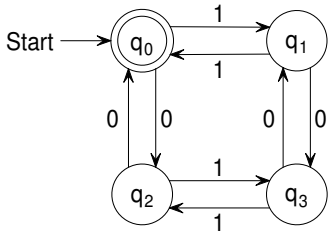
- The transition graph we used before is an informal presentation of the transition function δ. We have 

if $\delta(q, a) = q'$.
- "Deterministic" means that for every state q and input symbol a, there is a **unique** (i.e. exactly one) following state, $\delta(q, a)$.
- Later, we shall also see **non-deterministic finite automata** (NFA's), where (q, a) can have any number of following states.
- FA's are also called "finite state machines".

Useful notations for DFA's

- Transition graph, like that for Customer, Store, or Bank.
- Transition table, which is a tabular listing of the δ function.

Transition graph: example



$Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $\delta(q_3, 0) = q_1 \ldots$, $F = \{q_0\}$.

Meaning of the transition graph

- The nodes of the graph are the states.
- The labels of the arrows are input symbols.
- The labeled arrows describe the transition function.
- The node labeled "Start" is the start state q_0.
- The states with double circles are the final states.

Transition table: example

	0	1
$* \to q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

This is the transition table for the transition graph given above.

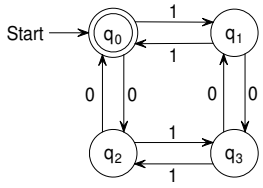
Meaning of the transition table

- The symbols in the leftmost column are the states.
- The symbols in the top row are the input symbols.
- The symbols "inside" the table describe the transition function.
- The arrow in the leftmost column marks the start symbol.
- The symbol $*$ in the leftmost column marks the final states.

How a DFA processes strings

- Let $a_1 a_2 \cdots a_n$ be a string of input symbols.
- Initially, the DFA is in its start state q_0.
- Let q be the state reached after the first i symbols $a_1 a_2 \cdots a_i$ of the input string. Upon reading the next symbol a_{i+i}, the DFA makes a transition into the new state $\delta(q, a_{i+1})$.
- Repeated until the last symbol a_n.
- The DFA said to **accept** the input string if the state reached after the last symbol a_n is in the set F of final states.

Accepted strings: example



- This DFA accepts 1010, but not 1110
- It accepts those strings that have an even number of 0's and an even number of 1's.
- Therefore, we call this DFA "parity checker".

Formal approach to accepted strings

We define the **extended transition function** $\hat{\delta}$. It takes a state q and an input **string** w to the resulting state. The definition proceeds by **induction** over the length of w.

- Induction basis (w has length 0): in this case, w is the **empty string**, i.e. the string of length 0, for which we write ϵ. We define

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

Formal approach to accepted strings

- Induction step (from length l to length $l+1$): in this case, w, which has length $l+1$, is of the form va, where v is a string of length l and a is a symbol. We define

$$\hat{\delta}(q, va) = \delta(\hat{\delta}(q, v), a).$$

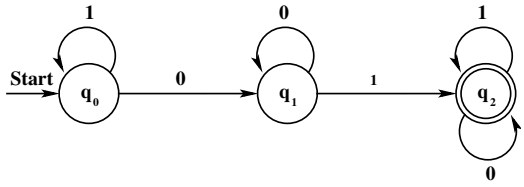
This works because, by induction hypothesis, $\hat{\delta}(q, v)$ is already defined.

The language of a DFA

- Intuitively, the language of a DFA A is the set of strings w that take the start state to one of the accepting states.
- Formally, the language $L(A)$ accepted by the DFA A is defined as follows:

$$L(A) = \{w \,|\, \hat{\delta}(q_0, w) \in F\}.$$

Transition graph: example



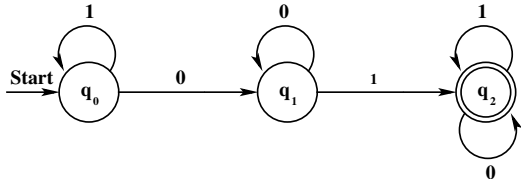
$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $\delta(q_0, 0) = q_1 \ldots$, $F = \{q_2\}$.

Transition table: example

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

This is the transition table for the same automaton. Let us call this machine A. What language does this automaton accept?

Accepted strings: example



- This DFA accepts 0011, but not 1110 for instance. Try some other strings.
- To find out what language $L(A)$ is accepted by A you need to work out what strings A will accept. Try and describe $L(A)$ in English.

Exercises

Give DFA's accepting the following languages over the alphabet $\{0, 1\}$. (Note that you can choose between giving a transition table, a transition graph, or a formal presentation of Q, Σ, q_0, δ, and F.)

1. The set of all strings ending in 00.
2. The set of all strings with two consecutive 0's (not necessarily at the end).
3. The set of strings with 011 as a substring.

Exercise

For the alphabet $\{a, b, c\}$, give a DFA accepting all strings that have abc as a substring.

Exercises

(More advanced; do not worry if you need tutor's help to solve this.) Give DFA's accepting the following languages over the alphabet $\{0, 1\}$.

1. The set of all strings such that each block of five consecutive symbols contains at least two 0's.
2. The set of all strings whose tenth symbol from the right is a 1.
3. The set of strings such that the number of 0 is divisible by five, and the number of 1's is divisible by three.

Exercise

Consider the DFA with the following transition table:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

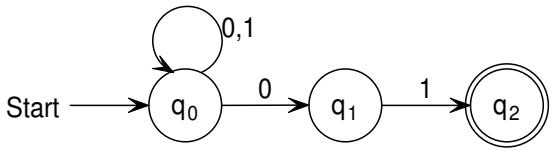
(1) Informally describe the language accepted by this DFA; (2) prove by induction on the length of an input string that your description is correct. (Don't worry if you need tutor's help for (2).)

Non-deterministic FA (NFA)

- An NFA is like a DFA, except that it can be in several states at once.
- This can be seen as the ability to guess something about the input.
- Useful for searching texts.

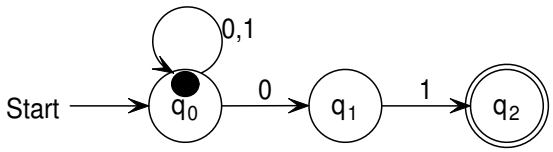
NFA: example

An NFA accepting all strings that end in 01:



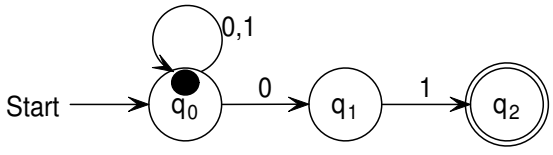
It is non-deterministic because input 0 in state q_0 can lead to both q_0 and q_1.

Using the NFA



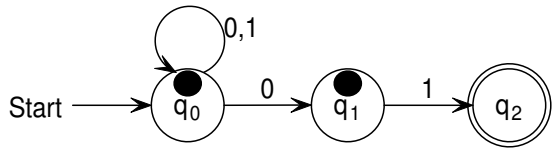
Suppose the input string is 100101. The NFA starts in state q_0, as indicated by the token.

Using the NFA



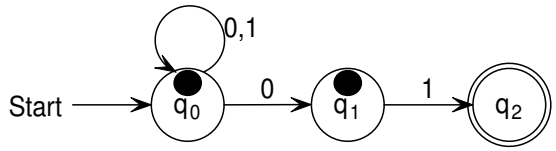
The remaining input string is 100101. The NFA reads the first symbol, 1. It remains in state q_0.

Using the NFA



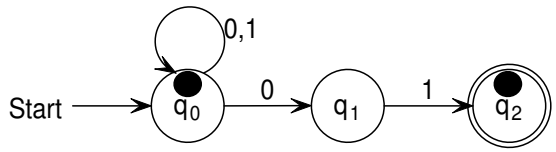
The remaining input string is 00101. The NFA reads the next symbol, 0. The resulting possible states are q_0 or q_1.

Using the NFA



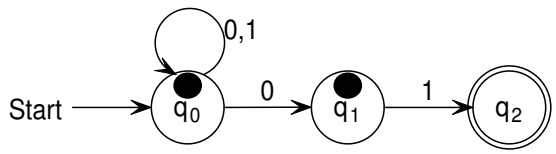
The remaining input string is 0101. The NFA reads the next symbol, 0. The resulting possible states are still q_0 or q_1.

Using the NFA



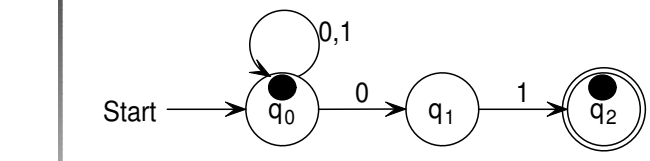
The remaining input string is 101. The NFA reads the next symbol, 1. The resulting possible states are q_0 and q_2. (Because q_2 is a final states, this means that the word so far, 1001, would be accepted.)

Using the NFA



The remaining input string is 01. The NFA reads the next symbol, 0. There is no transition for 0 from q_2, so the token on q_2 **dies**. The resulting possible states are q_0 or q_1.

Using the NFA



The remaining input string is 1. The NFA reads the next symbol, 1. The possible states are q_0 and q_2. Because q_2 is final, the NFA accepts the word, 100101.

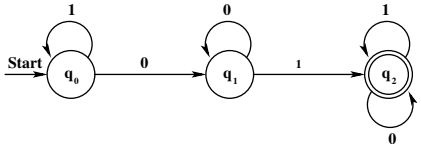
Deterministic Finite State Automaton (DFA).

Formal definition of DFA's

We saw the formal definition of a DFA:
Definition. A **deterministic finite automaton** (**DFA**) consists of

- a finite set of **states**, often denoted Q,
- a finite set Σ of **input symbols**,
- a total **transition function** $\delta : Q \times \Sigma \to Q$,
- a **start state** $q_0 \in Q$, and
- a set $F \subseteq Q$ of **final** or **accepting states**.

Accepted strings: example



- 01 is accepted.
- Strings of length 3 that are accepted: 101, 011, 010, 001. Strings not accepted: 111, 110, 100, 000
- Clearly 01 is going to be a substring of any string accepted. Is anything else required?

Formal approach to accepted strings

We define the **extended transition function** $\hat{\delta}$. It takes a state q and an input **string** w to the resulting state. The definition proceeds by **induction** over the length of w.

- Induction basis (w has length 0): in this case, w is the **empty string**, i.e. the string of length 0, for which we write ϵ. We define

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

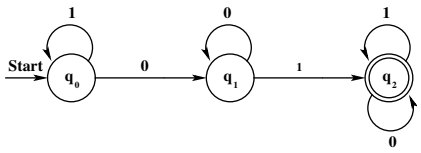
Formal approach to accepted strings

- Induction step (from length l to length $l+1$): in this case, w, which has length $l+1$, is of the form va, where v is a string of length l and a is a symbol. We define

$$\hat{\delta}(q, va) = \delta(\hat{\delta}(q, v), a).$$

This works because, by induction hypothesis, $\hat{\delta}(q, v)$ is already defined.

Accepted strings: example



- It accepts the language of all strings of 0 and 1 that contain the substring 01.
- To prove that this is the language accepted by the machine we would need to make an inductive argument based on the length of the string w.

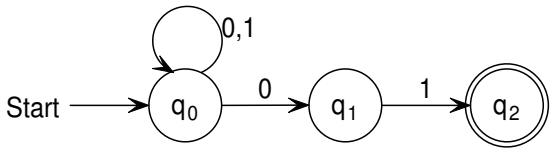
Non-deterministic finite automata (NFA's)

Non-deterministic FA (NFA)

- An NFA is like a DFA, except that it can be in several states at once.
- This can be seen as the ability to guess something about the input.
- Useful for searching texts.

NFA: example

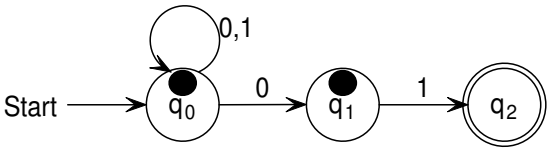
An NFA accepting all strings that end in 01:



It is non-deterministic because input 0 in state q_0 can lead to both q_0 and q_1.

Using the NFA

Last time we saw how to use the NFA. We saw that when in state q_0 given input 0 the resulting states are q_0 or q_1:



We also saw that when in state q_2 any input with cause the token to die and that the same occurs when we have input 0 to state q_1.

Formal definition of NFA

Definition. A **non-deterministic finite automaton** (**NFA**) consists of

- a finite set of **states**, often denoted Q,
- a finite set Σ of **input symbols**,
- a **transition function** $\delta : Q \times \Sigma \to P(Q)$,
- a **start state** $q_0 \in Q$, and
- a set $F \subseteq Q$ of **final** or **accepting states**.

Difference between NFA and DFA

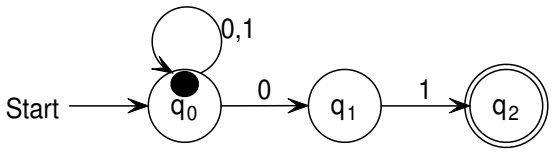
Suppose that q is a state and a is an input symbol.

- In a DFA, we have $\delta(q, a) \in Q$, that is, $\delta(q, a)$ is a state.
- In a NFA, we have $\delta(q, a) \in P(Q)$, that is, $\delta(q, a)$ is a **set of states**; it can be seen as the possible states that can result from input a in state q.

Formal approach to accepted strings

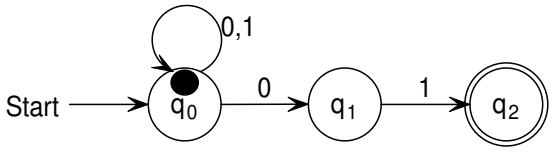
- We are aiming to describe the language $L(A)$ accepted by a NFA A.
- This description is similar to the DFA case, but a bit more sophisticated.
- As in the DFA case, we first define the **extended transition function**: $\hat{\delta} : Q \times \Sigma \to P(Q)$.
- That function $\hat{\delta}$ will be used to define $L(A)$.

Example of $\hat{\delta}$ (input string 100101)



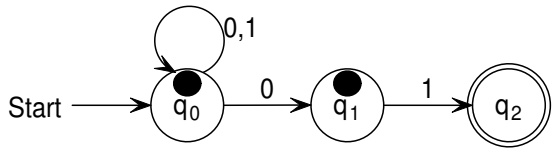
Before reading any symbols, the set of possible states is $\hat{\delta}(q_0, \epsilon) = \{q_0\}$.

Example of $\hat{\delta}$ (input string: 100101)



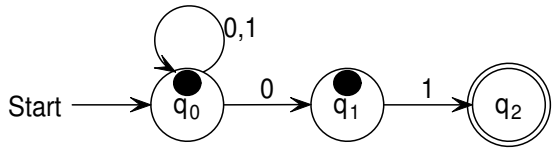
We have $\hat{\delta}(q_0, 1) = \{q_0\}$.

Example of $\hat{\delta}$ (input string: 100101)



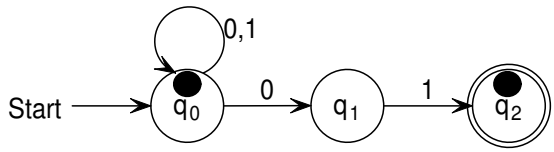
We have $\hat{\delta}(q_0, 10) = \{q_0, q_1\}$.

Example of $\hat{\delta}$ (input string: 100101)



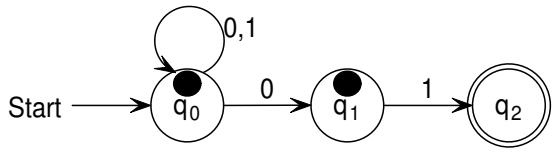
We have $\hat{\delta}(q_0, 100) = \{q_0, q_1\}$.

Example of $\hat{\delta}$ (input string: 100101)



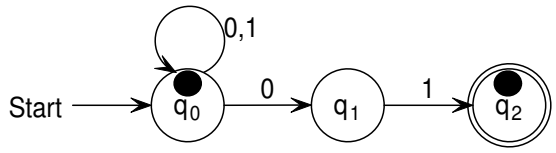
We have $\hat{\delta}(q_0, 1001) = \{q_0, q_2\}$.

Example of $\hat{\delta}$ (input string: 100101)



We have $\hat{\delta}(q_0, 10010) = \{q_0, q_1\}$.

Example of $\hat{\delta}$ (input string: 100101)



We have $\hat{\delta}(q_0, 100101) = \{q_0, q_2\}$. Because $\{q_0, q_2\} \cap F = \{q_0, q_2\} \cap \{q_2\} = \{q_2\} \neq \emptyset$, the NFA accepts.

Formal definition of $\hat{\delta}$

Definition. The **extended transition function** $\hat{\delta} : Q \times \Sigma \to P(Q)$ of an NFA is defined inductively as follows:

- Induction basis (length 0):

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

- Induction step (from length l to length $l + 1$):

$$\hat{\delta}(q, va) = \bigcup_{q' \in \hat{\delta}(q,v)} \delta(q', a).$$

The language of an NFA

- Intuitively, the language of a NFA A is the set of strings w that lead from the start state to an accepting possible state.
- Formally, the language $L(A)$ accepted by the FA A is defined as follows:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

Exercise

Give NFA to accept the following languages.

1. The set of strings over an alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has appeared before.
2. The set of strings over an alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has **not** appeared before.
3. The set of strings of 0's and 1's such that there are two 0's separated by a number of positions that is a multiple of 4.

DFA's and NFA's

- Evidently, DFA's are precisely those NFA's for which the set of states $\delta(q, a)$ has exactly one element for all q and a.
- So, trivially, every language accepted by a DFA is also accepted by some NFA.
- Is every language accepted by an NFA also accepted by some DFA?
- Surprisingly, the answer is "yes"!

Simulation of an NFA by a DFA

Let $N = (Q_N, \Sigma, \delta_N, q_0^N, F_N)$ be a NFA. The equivalent DFA D is obtained from the so-called **powerset construction** (also called "subset construction".) We define

$$D = (Q_D, \Sigma, \delta_D, q_0^D, F_D),$$

where...

Simulation of an NFA by a DFA

- The alphabet of D is that of N.
- The states of D are sets of states of N:

$$Q_D = P(Q_N)$$

- The initial state q_0^D of D is $\{q_0^N\}$.

Simulation of an NFA by a DFA

- The final states of D are those sets that contain the final state of N:

$$F_D = \{S \in P(Q_N) \mid S \cap F_N \neq \emptyset\}$$

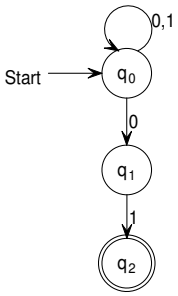
- The transition function of D arises from the transition function of N as follows:

$$\delta_D(S, a) = \bigcup_{q' \in S} \delta_N(q', a)$$

That is, $\delta_D(S, a)$ is the set of all states of N that are reachable from some state q via a.

Example of powerset construction: table

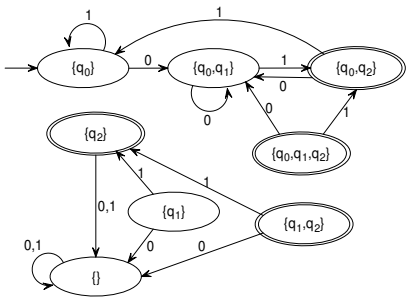
NFA



DFA

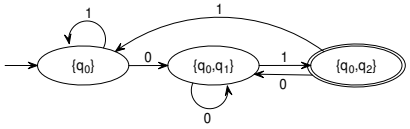
	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Example of powerset construction: graph



Transition graph of the resulting DFA.

Example of powerset construction: graph



Optionally, we can remove the unreachable states of the DFA.

Proposition about the simulation

Proposition. For every NFA N, there is a DFA D such that $L(D) = L(N)$.

Proof of the proposition (part 1/3)

First, we show that for every string w we have

$$\widehat{\delta_D}(\{q_0\}, w) = \widehat{\delta_N}(q_0, w) \quad (1)$$

We proceed by induction on the length l of w.

- Base case ($l = 0$): in this case, w is the empty string, ϵ. We have

$$\begin{aligned} \widehat{\delta_D}(\{q_0\}, \epsilon) &= \{q_0\} && \text{(by defn. of } \widehat{\delta_D}) \\ &= \widehat{\delta_N}(q_0, \epsilon) && \text{(by defn. of } \widehat{\delta_N}). \end{aligned}$$

Proof of the proposition (part 2/3)

- Induction step (from l to $l + 1$): in this case, w, which is of length $l + 1$, is of the form va, where v is a string of length l and a is a symbol. We have

$$\begin{aligned} \widehat{\delta_D}(\{q_0\}, va) &= \delta_D(\widehat{\delta_D}(\{q_0\}, v), a) && \text{(by defn. of } \widehat{\delta_D}) \\ &= \delta_D(\widehat{\delta_N}(q_0, v), a) && \text{(by indn. hypoth.)} \\ &= \bigcup_{q' \in \widehat{\delta_N}(q_0, v)} \delta_N(q', a) && \text{(by defn. of } \delta_D) \\ &= \widehat{\delta_N}(q_0, va) && \text{(by defn. of } \widehat{\delta_N}). \end{aligned}$$

Proof of the proposition (part 3/3)

Finally, we use Equation (1), which we just proved, to prove that the languages of D and N are equal:

$$\begin{aligned} w \in L(D) &\iff \widehat{\delta_D}(\{q_0\}, w) \in F_D && \text{(by defn. of } L(D)) \\ &\iff \widehat{\delta_N}(q_0, w) \in F_D && \text{(by Equation (1))} \\ &\iff \widehat{\delta_N}(q_0, w) \cap F_N \neq \emptyset && \text{(by defn. of } F_D) \\ &\iff w \in L(N) && \text{(by defn. of } L(N)). \end{aligned}$$

Languages accepted by DFAs and NFAs

The proposition implies:

Corollary. A language L is accepted by some DFA if and only if L is accepted by some NFA.

Proof. $\Rightarrow$: this is the powerset construction we have just seen.

$\Leftarrow$: this is true because every DFA is a special case of an NFA, as observed earlier.

Warning

- Let N be an NFA, and let D be the DFA that arises from the powerset construction.
- As we have seen, we have $Q_D = P(Q_N)$.
- So, if Q_N has size k, then the size of Q_D is 2^k.
- This exponential growth of the number of states makes the powerset construction unusable in practice.
- It can be shown that removing unreachable states does not prevent this exponential growth.

Non-Deterministic Finite Automaton (NFA).

Exercise

Convert the following NFA to a DFA:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\{\}$
$*s$	$\{s\}$	$\{s\}$.

Formal definition of NFA

Definition. A **non-deterministic finite automaton** (**NFA**) consists of

- a finite set of **states**, often denoted Q,
- a finite set Σ of **input symbols**,
- a **transition function** $\delta : Q \times \Sigma \rightarrow P(Q)$,
- a **start state** $q_0 \in Q$, and
- a set $F \subseteq Q$ of **final** or **accepting states**.

Exercise

Convert the following NFA to a DFA:

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\{\}$	$\{p\}$.

Formal definition of $\hat{\delta}$

Definition. The **extended transition function** $\hat{\delta} : Q \times \Sigma \rightarrow P(Q)$ of an NFA is defined inductively as follows:

- Induction basis (length 0):

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

- Induction step (from length l to length $l + 1$):

$$\hat{\delta}(q, va) = \bigcup_{q' \in \hat{\delta}(q,v)} \delta(q', a).$$

Exercise

Convert the following NFA to a DFA:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r, s\}$	$\{t\}$
r	$\{p, r\}$	$\{t\}$
$*s$	$\{\}$	$\{\}$
$*t$	$\{\}$	$\{\}$

Describe informally the language accepted by this NFA accept? (Don't worry if you need tutor's help for this.)

The language of an NFA

- Intuitively, the language of a NFA A is the set of strings w that lead from the start state to an accepting possible state.
- Formally, the language $L(A)$ accepted by the FA A is defined as follows:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

Simulation of an NFA by a DFA

Let $N = (Q_N, \Sigma, \delta_N, q_0^N, F_N)$ be a NFA. The equivalent DFA D is obtained from the so-called **powerset construction** (also called "subset construction".) We define

$$D = (Q_D, \Sigma, \delta_D, q_0^D, F_D),$$

where...

Simulation of an NFA by a DFA

- The alphabet of D is that of N.
- The states of D are sets of states of N:

$$Q_D = P(Q_N)$$

- The initial state q_0^D of D is $\{q_0^N\}$.

Simulation of an NFA by a DFA

- The final states of D are those sets that contain the final state of N:

$$F_D = \{S \in P(Q_N) \mid S \cap F_N \neq \emptyset\}$$

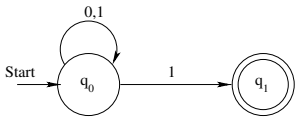
- The transition function of D arises from the transition function of N as follows:

$$\delta_D(S, a) = \bigcup_{q' \in S} \delta_N(q', a)$$

That is, $\delta_D(S, a)$ is the set of all states of N that are reachable from some state q via a.

Example of powerset construction: table

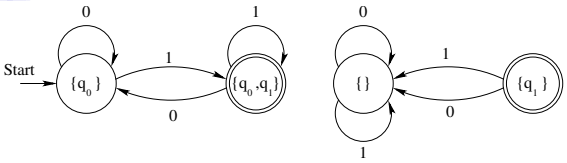
NFA:



DFA:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$*\{q_1\}$	$\emptyset$	$\emptyset$
$*\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_1\}$

Resulting DFA:



We cannot reach the states $\emptyset$ and $\{q_1\}$ from the start state so we can remove that part of the machine. You should be able to see that this resulting machine accepts the same language and how it is forced to work in a slightly more subtle way.

Proposition about the simulation

Proposition. For every NFA N, there is a DFA D such that $L(D) = L(N)$.

We used the powerset construction to show this.

We have an NFA N with start state q_0 and extended transition function $\widehat{\delta_N}$ and the DFA D obtained using the powerset construction which has start state $\{q_0\}$ and extended transition function $\widehat{\delta_D}$.

Summary of proof

- Firstly we assume that for every string w we have

$$\widehat{\delta_D}(\{q_0\}, w) = \widehat{\delta_N}(q_0, w). \quad (1)$$

We use this to show that the languages accepted by N and D are the same that is, $L(N) = L(D)$.

- We then need to show that (1) is true and we do this by induction on the length of the input string.

Languages accepted by DFAs and NFAs

The proposition implies:

Corollary. A language L is accepted by some DFA if and only if L is accepted by some NFA.

Proof. $\Rightarrow$: this is the powerset construction summarised above.

$\Leftarrow$: this is true because every DFA is a special case of an NFA, as observed earlier.

Regular expressions

Regular expressions: motivation

- Useful for describing text patterns (with wildcards etc.).
- Used e.g. for text search in the text editor "Emacs" and in the Unix search command "grep".
- Used in compilers for recognizing **tokens** of programming languages, e.g. identifiers, floating-point-numbers, and so on. (See compilers lecture.)

First example

The regular expression

$$01^* + 10^*$$

denotes the language consisting of all strings that are either a single 0 followed by any number of 1's, or a single 1 followed by any number of 0's.

Operations on languages

Before describing the regular-expression notation, we need to define the operations on languages that the operators of regular expressions represent.

Concatenation

- The **concatenation** $L \cdot L'$ (or just LL') of languages L and L' is defined to be the set of strings ww' where $w \in L$ and $w \in L'$.
- For example, if $L = \{001, 10, 111\}$ and $L' = \{\varepsilon, 001\}$, then $LL' = \{001, 10, 111, 001001, 10001, 111001\}$.

Self-concatenation

- For a language L, we write L^n for

$$\underbrace{L \cdot L \cdot \cdots \cdot L}_{n \text{ times}}$$

- That is, L^n is the language that consists of strings $w_1 w_2 \ldots w_n$, where each w_i is in L.
- For example, if $L = \{\varepsilon, 001\}$, then $L^3 = \{\varepsilon, 001, 001001, 001001001\}$.
- Note that $L^1 = L$. The language L^0 is defined to be $\{\varepsilon\}$.

Closure (Kleene-star)

- The **closure** (or **star** or **Kleene closure**) L^* of a language L is defined to be

$$L^* = \bigcup_{n \geq 0} L^n$$

- That is, L^* is the language that consists of strings $w_1 w_2 \ldots w_k$, where k is **any** non-negative integer and each w_i is in L.
- E.g. if $L = \{0, 11\}$, then L^* consists of all strings such that the 1's come in pairs, e.g. 011, 11110, and ε, but not 01011 or 101.

Regular expressions: definition

Definition. The **regular expressions** over an alphabet Σ are defines as follows:

- Every symbol $a \in \Sigma$ is a regular expression.
- If E and E' are regular expressions, then so is $E + E'$ and $E \cdot E'$. (We shall abbreviate the latter by EE'.)
- If E is a regular expressions, then so is E^*.
- The symbol ε is a regular expression.
- The symbol $\emptyset$ is a regular expression.

Semantics of regular expressions

(Remark: "semantics" is the technical term for "meaning".)

Regular expression E	denoted language $L(E)$
$a \in \Sigma$	$\{a\}$
$E + E'$	$L(E) \cup L(E')$
$E \cdot E'$	$L(E) \cdot L(E')$
E^*	$(L(E))^*$
ε	$\{\varepsilon\}$
$\emptyset$	the empty language, $\emptyset$

Example

- Suppose you want to search some messy text file for the street parts of addresses, e.g. "Milsom Street" or "Wells Road".
- Let $[A - Z]$ stand for $A + B + \cdots + Z$.
- Let $[a - z]$ stand for $a + b + \cdots + z$.
- You may want to use a regular expression like

  $[A{-}Z][a{-}z]^* \ \ (Street{+}St.{+}Road{+}Rd.{+}Lane)$
- Expressions like this are accepted e.g. by the UNIX command `grep`, the EMACS text editor, and various other tools.

Exercises

Write regular expressions for the following languages:

1. The set of strings over alphabet $\{a, b, c\}$ with at least one a and at least one b.
2. The set of strings of 0's and 1's whose tenth symbol from the right end is 1.
3. The set of strings of 0's and 1's with at most on pair of consecutive 1's.

Exercises

Write regular expressions for the following languages:

1. The set of all strings of 0's and 1's such that every pair of adjacent 0's appears before any pair of adjacent 1's.
2. The set of strings of 0's and 1's whose number of 0's is divisible by five.

Exercise

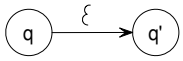
For any alphabet Σ, which are the subsets S of Σ^* such that the set S^* is finite?

Regular expressions and FA's: overview

- As we shall see, for every regular expression E, there is an NFA (and therefore also a DFA) that accepts the language defined by E.
- Tools that scan text for regular expressions work in this way.
- Also, for every DFA (and therefore for every NFA) A, there is a regular expression that denotes the language accepted by A.
- So finite automata and regular expressions are equivalent with respect to the definable languages.

NFA's with ε-transitions

- For simulating regular expressions, it is helpful to introduce **NFA's with ε-transitions**, or **ε-NFA's in short**.
- The only difference between ε-NFA's and NFA's is that the former can make spontaneous transitions, i.e. transitions that use up no input—technically speaking, the empty string ε.



NFA's with ε-transitions

- More formally, an ε-NFA differs from an NFA only in that its transition function also accepts ε as an argument:

$$\delta : Q \times (\Sigma \cup \varepsilon) \to P(Q)$$

- It can be shown by some **modified powerset construction** that for every ε-NFA there is a DFA accepting the same language (see Hopcroft/Motwani/Ullman).

From regular expressions to FA's

Formally, we shall prove:

Proposition. For every regular expression E, there is an ε-NFA N_E such that $L(N_E) = L(E)$.

We shall see how this works in the next lecture.

So, we need to prove that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w).$$

We prove this by induction on the length l of w.

Base Case: $l = 0$

If $l = 0$ then $w = \varepsilon$ the empty string.

$$\hat{\delta}_D(\{q_0\}, \varepsilon) = \{q_0\} = \hat{\delta}_N(q_0, w)$$

(No transition made; $\hat{\delta}_N$ gives us sets of states)

Inductive Step:

- Assume that $\hat{\delta}_D(\{q_0\}, r) = \hat{\delta}_N(q_0, r)$ where r is any string of length l.
- If w has length $l+1$ we can write this $w = ra$ where r has length l and a is a symbol.

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D\left(\hat{\delta}_D(\{q_0\}, r), a\right)$$

$$= \delta_D\left(\hat{\delta}_N(q_0, r), a\right)$$

$$= \bigcup_{q \in \hat{\delta}_N(q_0, r)} \delta_N(q, a)$$

$$= \hat{\delta}_N(q_0, ra)$$

$$= \hat{\delta}_N(q_0, w)$$

Proposition:

For every NFA N there is a DFA D such that $L(N) = L(D)$.

Proof:

If we can show that

$$* \quad \hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

(start both machines in their start states with the same input string w and they end up in the same set of states.)

then we can argue:

$w \in L(D)$ (w is in the language accepted by D)

$\iff \hat{\delta}_D(\{q_0\}, w) \in F_D$ (start D in $\{q_0\}$ with input w and end in an accepting state)

$\iff \hat{\delta}_N(q_0, w) \in F_D$

($*$ says $\hat{\delta}_N(q_0, w) = \hat{\delta}_D(\{q_0\}, w)$)

$\iff \hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$

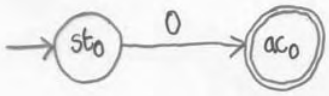
$\iff w \in L(N)$

(w is in the language accepted by N.)

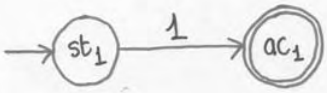
($F_D = \{S \in P(Q_N) \mid S \cap F_N \neq \emptyset\}$ and $\hat{\delta}_N(q_0, w)$ is a member of F_D... so $\hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$ which means $\hat{\delta}_N(q_0, w)$ is an accepting state of N.)

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

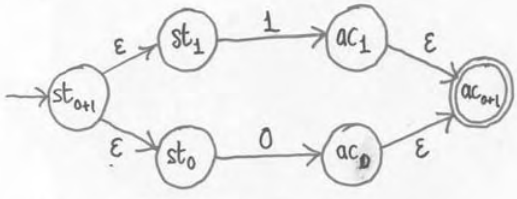
0:



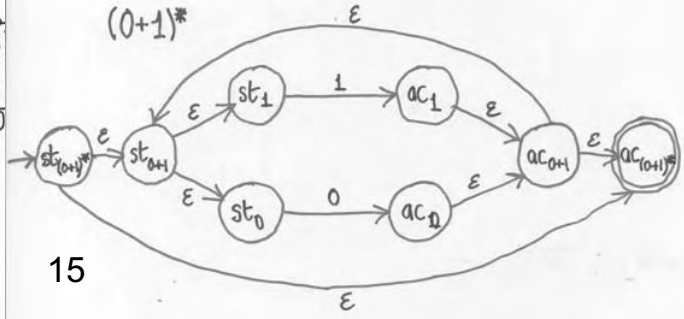
1:

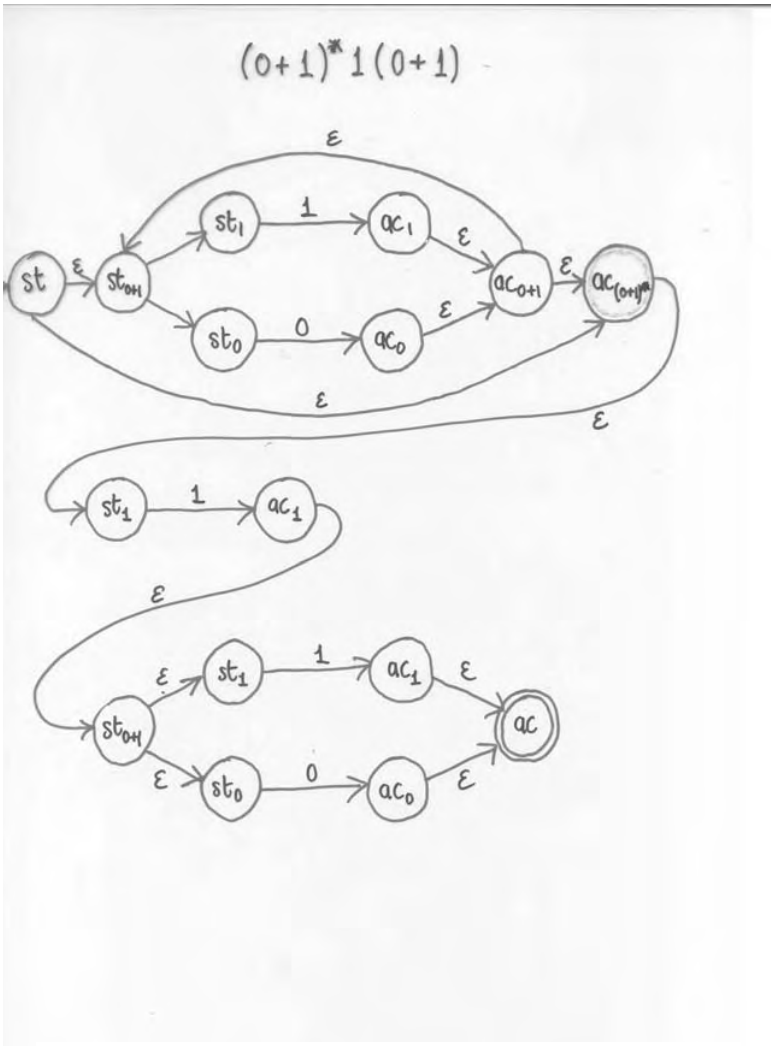


0+1:



$(0+1)^*$





Regular expressions ctd.

Operations on languages

For languages L and L'

- The **Concatenation** $L \cdot L'$ (or just LL') is defined to be the set of strings ww' where $w \in L$ and $w \in L'$.
- We write L^n for $L \cdot L \cdot \cdots \cdot L$ (n times). L^n is the language that consists of strings $w_1 w_2 \ldots w_n$, where each w_i is in L.
- The **closure** (or **star** or **Kleene closure**) L^* is defined to be $L^* = \bigcup_{n \geq 0} L^n$. L^* is the language that consists of strings $w_1 w_2 \ldots w_k$, where $k \in \mathbb{Z}$, $k \geq 0$ and each w_i is in L.

Regular expressions: definition

Definition. The **regular expressions** over an alphabet Σ are defines as follows:

- Every symbol $a \in \Sigma$ is a regular expression.
- If E and E' are regular expressions, then so is $E + E'$ and $E \cdot E'$. (We shall abbreviate the latter by EE'.)
- If E is a regular expressions, then so is E^*.
- The symbol ε is a regular expression.
- The symbol $\emptyset$ is a regular expression.

Semantics of regular expressions

(Remark: “semantics” is the technical term for “meaning”.)

Regular expression E	denoted language $L(E)$
$a \in \Sigma$	$\{a\}$
$E + E'$	$L(E) \cup L(E')$
$E \cdot E'$	$L(E) \cdot L(E')$
E^*	$(L(E))^*$
ε	$\{\varepsilon\}$
$\emptyset$	the empty language, $\emptyset$

Exercises 5.1 (last lecture)

Write regular expressions for the following languages:

1. The set of strings over alphabet $\{a, b, c\}$ with at least one a and at least one b.
2. The set of strings of 0’s and 1’s whose tenth symbol from the right end is 1.
3. The set of strings of 0’s and 1’s with at most on pair of consecutive 1’s.

Exercises 5.2 (last lecture)

Write regular expressions for the following languages:

1. The set of all strings of 0’s and 1’s such that every pair of adjacent 0’s appears before any pair of adjacent 1’s.
2. The set of strings of 0’s and 1’s whose number of 0’s is divisible by five.

Exercise 5.3 (last lecture)

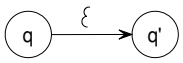
For any alphabet Σ, which are the subsets S of Σ^* such that the set S^* is finite?

Regular expressions and FA's: overview

- As we shall see, for every regular expression E, there is an NFA (and therefore also a DFA) that accepts the language defined by E.
- Tools that scan text for regular expressions work in this way.
- Also, for every DFA (and therefore for every NFA) A, there is a regular expression that denotes the language accepted by A.
- So finite automata and regular expressions are equivalent with respect to the definable languages.

NFA's with ε-transitions

- For simulating regular expressions, it is helpful to introduce **NFA's with ε-transitions**, or **ε-NFA's in short**.
- The only difference between ε-NFA's and NFA's is that the former can make spontaneous transitions, i.e. transitions that use up no input—technically speaking, the empty string ε.



NFA's with ε-transitions

- More formally, an ε-NFA differs from an NFA only in that its transition function also accepts ε as an argument:

$$\delta : Q \times (\Sigma \cup \varepsilon) \to P(Q)$$

- It can be shown by some **modified powerset construction** that for every ε-NFA there is a DFA accepting the same language (see Hopcroft/Motwani/Ullman).

From regular expressions to FA's

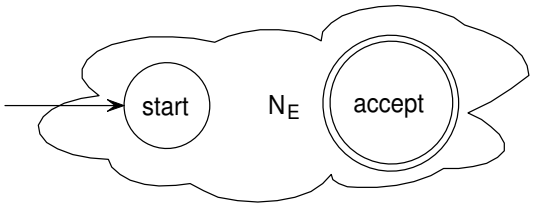
Formally, we shall prove:

Proposition. For every regular expression E, there is an ε-NFA N_E such that $L(N_E) = L(E)$.

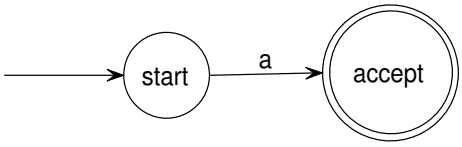
We shall see how this works on the next few slides.

The ε-NFA N_E of a regular expression E

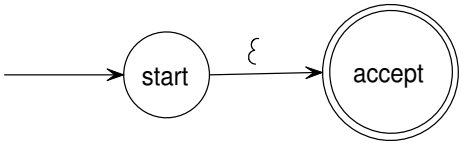
For every regular expression E, we shall build an ε-NFA N_E with exactly one accepting state, from which no further transitions are possible:



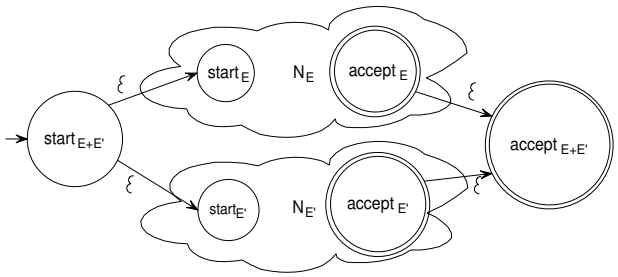
N_E for $E = a \in \Sigma$



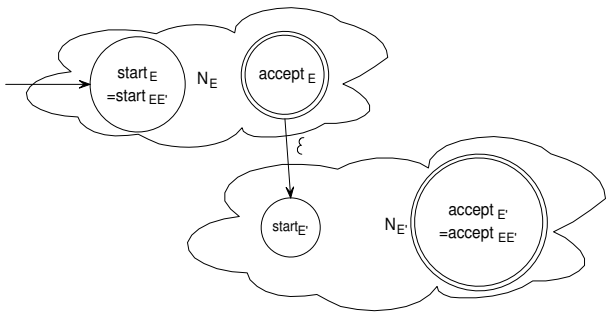
N_E for $E = \varepsilon$



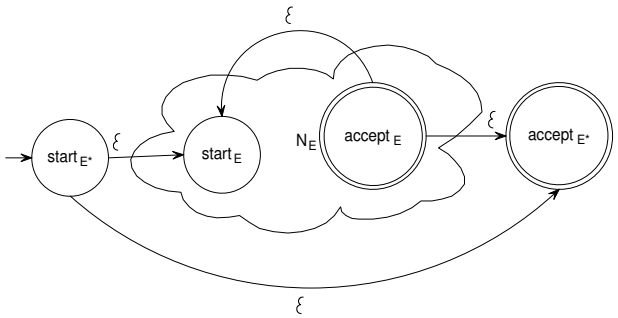
$N_{E+E'}$



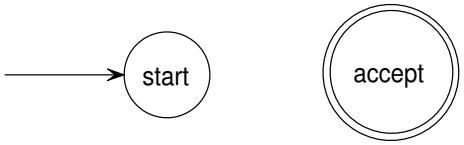
$N_{E \cdot E'}$



N_{E^*}



N_E for $E = \emptyset$



There is no way to get from the start state to the accepting state.

NFA for $(0+1)^*1(0+1)$

See overhead.

Exercises 6.1

Convert each of the following regular expressions to an ϵ-NFA:

1. 01^*
2. $(0+1)01$
3. $00(0+1)^*$

The big picture (part 1/2)

- Trivially, every DFA is an NFA, and every NFA is an ϵ-NFA.
- One goal is to show that all three types of automata accept the same languages.
- To show this, it suffices to show that for every ϵ-NFA there is a DFA that accepts the same language.
- To that end, we shall use a **modified powerset construction.**

The big picture (part 2/2)

- We have also seen that every regular expression is accepted by an ϵ-NFA.
- We shall see later that for every FA there is a regular expression describing the same language.
- So all **four** formalisms (DFA's, NFA's, ϵ-NFA's, and regular expressions) describe the same languages.

From ϵ-NFA to DFA

- Suppose that N is an ϵ-NFA. We shall now study the **modified powerset construction**, which produces a DFA D that accepts the same language as N.
- To that end, we need one auxiliary definition: given a set S of states of N, the **ϵ-closure** $cl(S)$ of S is the set of states that are reachable from S by any number of ϵ-transitions.

From ϵ-NFA to DFA

The construction of D from N looks as the powerset construction, except that we use cl:

- The alphabet of D is that of N.
- The states of D are sets of the form $cl(S)$, where $S \in P(N)$.
- The initial state q_0^D of D is $cl\{q_0^N\}$.
- The final states of D are those sets of the form $cl(S)$ that contain a final state of N:

$$F_D = \{cl(S) \mid S \cap F_N \neq \emptyset\}$$

From ϵ-NFA to DFA

- The transition function of D arises from the transition function of N as follows:

$$\delta_D(S, a) = \bigcup_{q \in S} cl(\delta_N(q, a))$$

That is, $\delta_D(S, a)$ is the set of all states of N that are reachable from some state $q \in S$ via a, followed by any number of ϵ-transitions.

The simulation proposition

Proposition. For every ϵ-NFA N, the DFA D resulting from the modified powerset construction accepts the same language.

Proof. One shows that every string w that

$$w \in L(D) \iff w \in L(N).$$

The proof works by induction on the length of w, and is only slighty more complicated than the proof we have seen for the (ordinary) powerset construction.

Exercise 6.2

Consider the following ϵ-NFA.

	ϵ	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

1. Compute the ϵ-closure of each state.
2. Give all strings of length three or less accepted by this automaton.
3. Convert the automaton to a DFA.

Exercise 6.3

Repeat the previous exercise for the following ϵ-NFA.

	ϵ	a	b	c
$\rightarrow p$	$\{q, r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

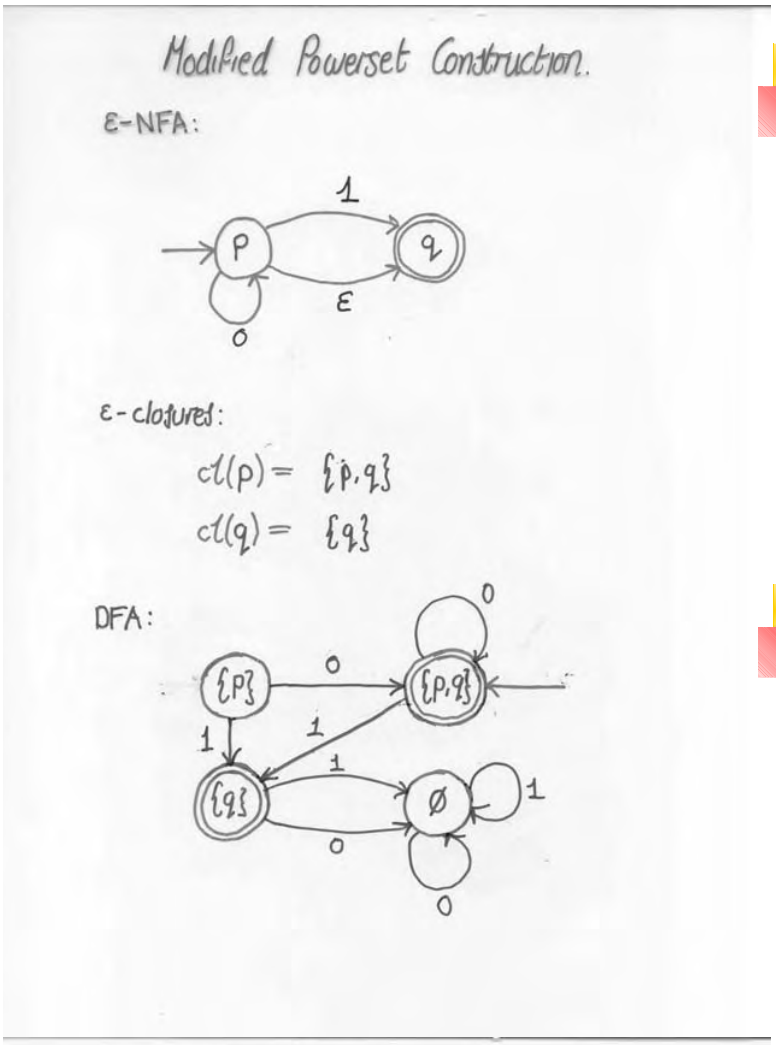
Exercise 6.4

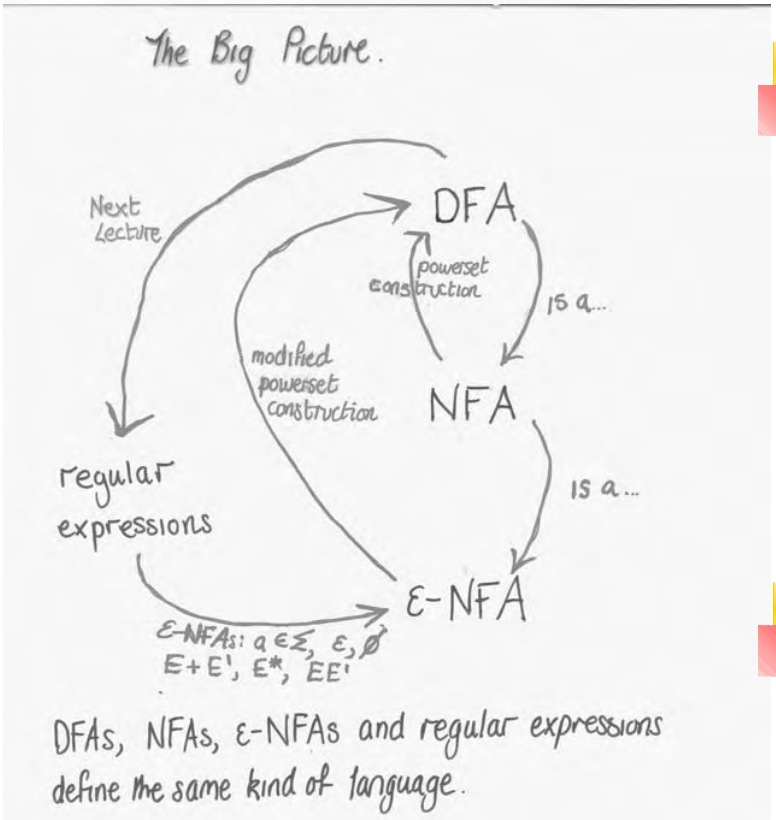
In an earlier exercise, we converted the regular expressions 01^*, $(0 + 1)01$, and $00(0 + 1)^*$ into ϵ-NFA's. Convert each of those ϵ-NFA's into a DFA.

Formal languages

Next time we will start on formal languages.

- A **formal language** (or simpy "language") is a set L of strings over some finite alphabet Σ. That is, a subset $L \subseteq \Sigma^*$.
- Finite automata and regular expressions describe certain formal languages but many important languages are not regular e.g. programming languages.
- To describe formal languages we will use formal grammars which we will introduce next lecture.





Formal languages

Formal languages: overview

- A **formal language** (or simpy "language") is a set L of strings over some finite alphabet Σ. That is, a subset $L \subseteq \Sigma^*$.
- Finite automata and regular expressions describe certain formal languages.
- But many important formal languages, e.g. programming languages, are not regular.
- To describe formal languages, we shall use **formal grammars**.

Formal grammars: overview

- Important for describing programming languages.
- Different kinds of formal grammars are described by the the **Chomsky hierarchy**.
- Among the simplest grammars in the Chomsky hierarchy are the **regular grammars**, which—as we shall see later—describe the same languages as regular expressions.

Formal grammars: basic idea

To generate strings by beginning with a **start symbol** S and then apply rules that indicate how certain combinations of symbols may be replaced with other combinations of symbols.

Formal grammar: definition

Definition. A formal grammar $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \to v$$

where u and v are strings in $(\Sigma \cup N)^*$ and u contains at least one non-terminal symbol;

- a **start symbol** S in N.

Example

The grammar G with non-terminal symbols $N = \{S, B\}$, terminal symbols $\Sigma = \{a, b, c\}$, and productions

$$S \to abc$$
$$S \to aSBc$$
$$cB \to Bc$$
$$bB \to bb$$

Following a common practice, we use capital letters for non-terminal symbols and small letters for terminal symbols.

Language of a formal grammar

Definition. The language of a formal grammar $G = (N, \Sigma, P, S)$, denoted as $L(G)$, is defined as all those strings over Σ that can be generated by starting with the start symbol S and then applying the production rules in P until no more non-terminal symbols are present.

The Chomsky hierarchy

There are 4 different types of formal grammars that we will meet. Some we will spend more time on than others.

- Type 3: regular grammars.
- Type 2: context free grammars.
- Type 1: context sensitive grammars.
- Type 0: unrestricted or free grammars.

To define each type we need only restrict the type of production rules permitted.

Context-free grammars and languages

Motivation

- It turns out that many important languages, e.g. programming languages, cannot be described by finite automata/regular expressions.
- To describe such languages, we shall introduce **context-free grammars** (CFG's).

Example

The grammar G with non-terminal symbols $N = \{S\}$, terminal symbols $\Sigma = \{a, b\}$, and productions

$$S \to aSb$$
$$S \to ab.$$

Following a common practice, we use capital letters for non-terminal symbols and small letters for terminal symbols.

Context-free grammar: definition

Definition. A **context-free grammar** $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \to v$$

where u is in N and v is a string in $(\Sigma \cup N)^*$;

- a **start symbol** S in N.

Example: integer expressions

A simple form of integer expressions.

- The non-terminal symbols are

symbol	intended meaning
E	expressions, e.g. $(x + y) * 3$
I	identifiers, e.g. x, y
C	constants (here: natural numbers).

- The alphabet Σ of terminal symbols is $\{x, y, 0, 1, \ldots, 9, *, +, (,)\}$.

Productions for integer expressions

$$E \to C \qquad C \to 0 \qquad C \to C0$$
$$E \to I \qquad C \to 1 \qquad C \to C1$$
$$E \to E + E \qquad \vdots \qquad \vdots$$
$$E \to E * E \qquad C \to 9 \qquad C \to C9$$
$$E \to (E)$$

$$I \to x$$
$$I \to y$$

Compact notation

More compact notation for productions:

$$E \to I \,|\, C \,|\, E + E \,|\, E * E \,|\, (E)$$
$$C \to 0 \,|\, C0 \,|\, \ldots \,|\, 9 \,|\, C9$$
$$I \to x \,|\, y$$

Language of a formal grammar

Definition. The language of a formal grammar $G = (N, \Sigma, P, S)$, denoted as $L(G)$, is defined as all those strings over Σ that can be generated by beginning with the start symbol S and then applying the productions P.

Parse trees

- Every string w in a context-free language has a **parse tree**.
- The root of a parse tree is the start symbol S.
- The leaves of a parse tree are the terminal symbols that make up the string w.
- The branches of the parse tree describe how the productions are applied.

Exercise

Consider the grammar

$$S \to A1B$$
$$A \to 0A \,|\, \epsilon$$
$$B \to 0B \,|\, 1B \,|\, \epsilon.$$

Give parse trees for the strings

1. 00101
2. 1001
3. 00011

Exercise

For the CFG G defined by the productions

$$S \to aS \,|\, Sb \,|\, a \,|\, b,$$

prove by induction on the size of the parse tree that the no string in language $L(G)$ has ba as a substring.

Exercises

Give a context-free grammar for the language of all palindromes over the alphabet $\{a, b, c\}$. (A **palindrome** is a word that is equal to the reversed version of itself, e.g. “abba”, “bab”.)

Parsing

- Finding a parse tree for a string is called **parsing**. Software tools that do this are called **parsers**.
- A compiler must parse every program before producing the executable code.
- There are tools called “parser generators” that turn CFG’s into parsers. One of them is the famous YACC (“Yet Another Compiler Compiler”).
- Parsing is a science unto itself, and described in detail in courses about compilers.

Ambiguity

- A context-free grammar with more than one parse tree for some expression is called **ambiguous**.
- Ambiguity is dangerous, because it can affect the meaning of expressions; e.g. in the expression $b * a + b$, it matters whether $*$ has precedence over $+$ or vice versa.
- To address this problem, parser generators (like YACC) allow the language designer to specify the operator precedence.

Exercise

Consider the grammar

$$S \to aS \,|\, aSbS \,|\, \epsilon.$$

Show that this grammar is ambiguous.

Exercise

Consider the grammar

$$S \to A1B$$
$$A \to 0A \,|\, \epsilon$$
$$B \to 0B \,|\, 1B \,|\, \epsilon.$$

Show that this grammar is unambiguous.

Regular languages

Regular grammars

- Next, we shall see certain CFG’s called **regular grammars** and show that they define the same languages as finite automata and regular expressions.
- These are the regular languages and are type 3 on the Chomsky hierarchy.
- They form a subset of the context free languages, which are type 2, defined by the context free grammars we have been talking about.

Regular grammars: definition

Definition. A CFG is called **regular** if every production has one of the three forms below

$$A \to aB$$
$$A \to a$$
$$A \to \varepsilon$$

where A and B are terminal symbols and a is some non-terminal symbol.

Example

The grammar G with $N = \{A, B, C, D\}$, $\Sigma = \{0, 1\}$, $S = A$, and the productions below is regular.

$$A \to 0C \,|\, 1B \,|\, \varepsilon$$
$$B \to 0D \,|\, 1A$$
$$C \to 0A \,|\, 1D$$
$$D \to 0B \,|\, 1C$$

Non-example

The grammar below is not regular because of the b on the right of S.

$$S \to aSb$$
$$S \to \epsilon.$$

Lecture 8: Regular languages

Formal grammar: definition

Definition. A formal grammar $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \to v$$

  where u and v are strings in $(\Sigma \cup N)^*$ and u contains at least one non-terminal symbol;
- a **start symbol** S in N.

The Chomsky hierarchy

There are 4 different types of formal grammars that we will meet. Some we will spend more time on than others.

- Type 3: regular grammars.
- Type 2: context free grammars.
- Type 1: context sensitive grammars.
- Type 0: unrestricted or free grammars.

To define each type we need only restrict the type of production rules permitted.

Context-free grammar: definition

Definition. A **context-free grammar** $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \to v$$

  where u is in N and v is a string in $(\Sigma \cup N)^*$;
- a **start symbol** S in N.

Regular grammars

- Next, we shall see certain CFG's called **regular grammars** and show that they define the same languages as finite automata and regular expressions.
- These are the regular languages and are type 3 on the Chomsky hierarchy.
- They form a subset of the context free languages, which are type 2, defined by the context free grammars we have been talking about.

Regular grammars: definition

Definition. A CFG is called **regular** if every production has one of the three forms below

$$A \to aB$$
$$A \to a$$
$$A \to \varepsilon$$

where A and B are non-terminal symbols and a is some terminal symbol.

Example

The grammar G with $N = \{S, A\}$, $\Sigma = \{0, 1\}$, S the start state and the productions below is regular.

$$S \to 0S$$
$$S \to 1B$$
$$B \to \varepsilon$$

What language does this grammar describe? Can you find a DFA and a regular expression that define the same language?

Non-example

The grammar below is not regular because of the b on the right of S.

$$S \to aSb$$
$$S \to \varepsilon.$$

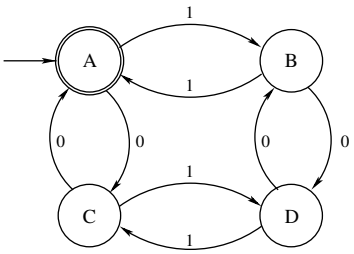
This is a context free grammar that describes the language $\{a^n b^n | n \geq 1\}$. You cannot write a regular grammar that describes this language. Why not?

From NFA to regular grammar

For an NFA over alphabet Σ, we construct a regular grammar G over alphabet Σ as follows:

1. The non-terminal symbols are the states of the NFA.
2. The start symbol is the initial state of the NFA.
3. For every transition $q \xrightarrow{a} q'$ in the NFA, introduce a production $q \to aq'$.
4. For every final state q, add a production $q \to \varepsilon$.

Parity Checker



$Q = \{A, B, C, D\}$, $\Sigma = \{0, 1\}$, $F = \{A\}$.

Parity checker

The grammar G with $N = \{A, B, C, D\}$, $\Sigma = \{0, 1\}$, $S = A$, and the productions below is regular.

$$A \to 0C \,|\, 1B \,|\, \varepsilon$$
$$B \to 0D \,|\, 1A$$
$$C \to 0A \,|\, 1D$$
$$D \to 0B \,|\, 1C$$

8.1 Exercise

Turn some of the NFA's in this lecture course into regular grammars.

From reg. grammar to reg. expression

We shall explain this in a number of steps.

First, we write the grammar in its compact notation, e.g.

$$X \to aY \,|\, bZ \,|\, \varepsilon \qquad (1)$$
$$Y \to cX \,|\, dZ \qquad (2)$$
$$Z \to eZ \,|\, fY \qquad (3).$$

From reg. grammar to reg. expression

Next, we replace $\to$ by $=$ and $|$ by $+$, to make the grammar look like an equation system:

$$X = aY + bZ + \varepsilon \qquad (1)$$
$$Y = cX + dZ \qquad (2)$$
$$Z = eZ + fY \qquad (3).$$

The trick is now to get a solution for the start symbol X **that does not rely on** Y **and** Z.

From reg. grammar to reg. expression

$$X = aY + bZ + \varepsilon \qquad (1)$$
$$Y = cX + dZ \qquad (2)$$
$$Z = eZ + fY \qquad (3).$$

We shall reduce the three equations above to two equations as follows:

1. Find a solution for Z in terms of only X and Y.
2. Replace that solution for Z in equations (1) and (2). (That yields equations for X and Y that don't rely on Z anymore.)

From reg. grammar to reg. expression

How do we find a solution for

$$Z = eZ + fY \qquad (3)$$

that doesn't rely on Z anymore? The idea is

"Z can produce an e and become Z again for a number of times, but finally Z must become fY."

Formally, the solution of Equation (3) is

$$Z = e^* fY.$$

From reg. grammar to reg. expression

Generally, the solution of any equation of the form

$$A = cA + B$$

is

$$A = c^* B.$$

From reg. grammar to reg. expression

Next, we replace the solution

$$Z = e^* fY$$

in equations (1) and (2)

$$X = aY + bZ + \varepsilon \qquad (1)$$
$$Y = cX + dZ \qquad (2).$$

This yields

$$X = aY + be^* fY + \varepsilon \qquad (1')$$
$$Y = cX + de^* fY \qquad (2').$$

From reg. grammar to reg. expression

Simplifying (1') by using the **distributivity law** yields

$$X = (a + be^* f)Y + \varepsilon \qquad (1')$$

Now instead of three equations over X, Y, Z, we have only two equations over X, Y:

$$X = (a + be^* f)Y + \varepsilon \qquad (1')$$
$$Y = cX + de^* fY \qquad (2').$$

From reg. grammar to reg. expression

$$X = (a + be^* f)Y + \varepsilon \qquad (1')$$
$$Y = cX + de^* fY \qquad (2').$$

Next, we repeat our game of eliminating non-terminals and find a solution for Y:

$$Y = (de^* f)^* cX.$$

From reg. grammar to reg. expression

Using $Y = (de^* f)^* cX$ in (1') yields

$$X = (a + be^* f)(de^* f)^* cX + \varepsilon \qquad (1'')$$

The solution of (1") is

$$X = ((a + be^* f)(de^* f)^* c)^* \varepsilon = ((a + be^* f)(de^* f)^* c)^*$$

So we found a regular expression for the language generated by X.

8.2 Exercise

Consider the NFA given by the transition table below.

	0	1
$\rightarrow X$	$\{X\}$	$\{X, Y\}$
$*Y$	$\{Y\}$	$\{Y\}$

1. Give a regular grammar for it.
2. Calculate a regular expression that describes the language.

8.3 Exercise

Repeat the exercise for the NFA below.

	0	1
$\rightarrow *X$	$\{X\}$	$\{Y\}$
Y	$\{Y\}$	$\{Z\}$
Z	$\{Z\}$	$\{X\}$

Also, describe the accepted language in English.

The big picture: final version

See lecture for diagram.

Definition. Languages that are definable by regular expressions or regular grammars or finite automata are called **regular languages**.

A non-regular language

Proposition. The language

$$L = \{a^n b^n \mid n \geq 1\}$$

(for which we gave a CFG earlier) is not regular.

Proof. By contradiction: Assume that L is regular.

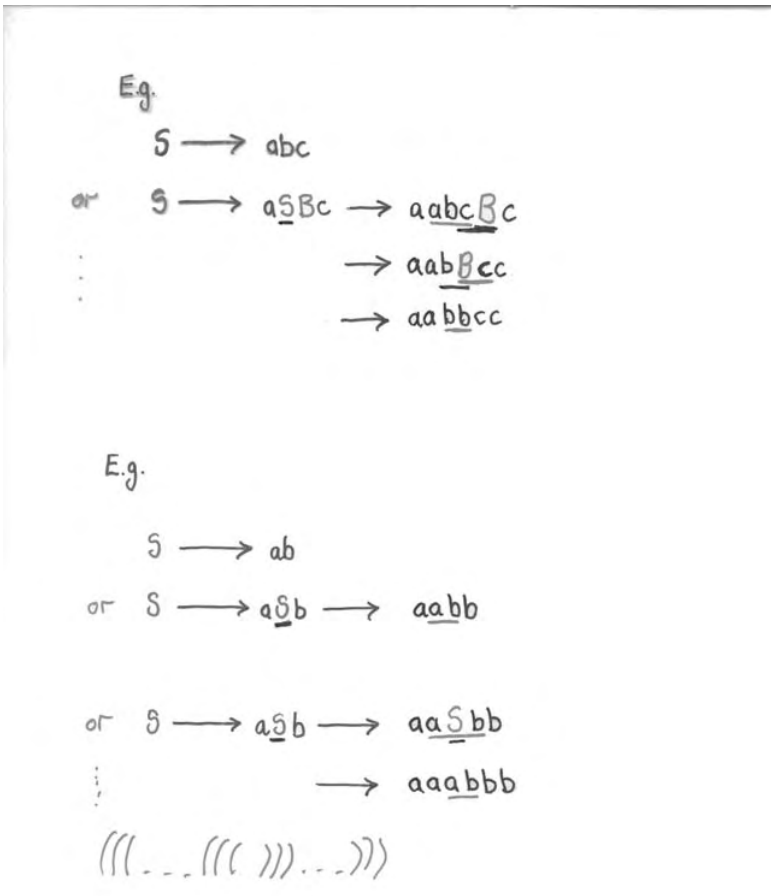
- Then there is a DFA A that accepts L.
- Let n be the number of states of A.
- We read in the string a^n. We make n state transitions so must "visit" $n+1$ states hence we must "visit" one state twice, lets call that state q.

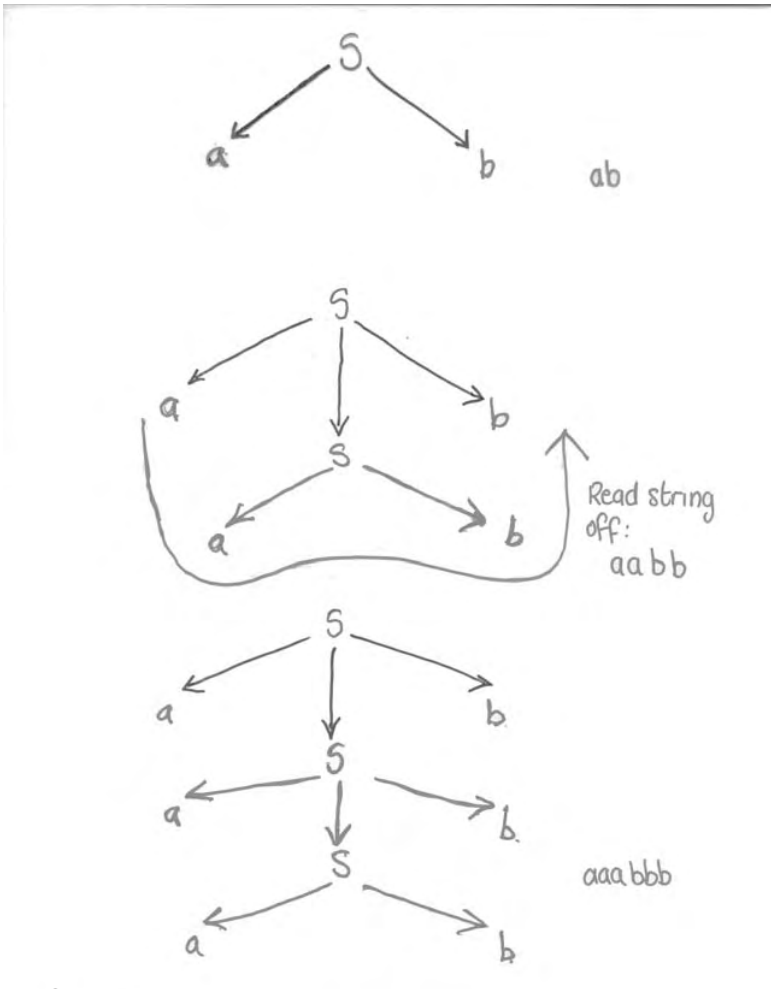
A non-regular language

- That means there are two substrings of a^n, say a^m and a^k with $m < k$ such that $q = \hat{\delta}(q_0, a^m) = \hat{\delta}(q_0, a^k)$.
- Now, the machine accepts $a^m b^m$ so the machine reads in a^m and is in state q and starts from q, reads in b^m and then accepts the string.
- If we input $a^k b^m$ the machine reads in a^k and is in state q. We know that from state q with input b^m the machine ends up in an accepting state so the machine accepts $a^k b^m$ which implies $a^k b^m \in L$, $k < m$. Contradiction!
- The machine "can't remember" whether the input was a^m or a^k.

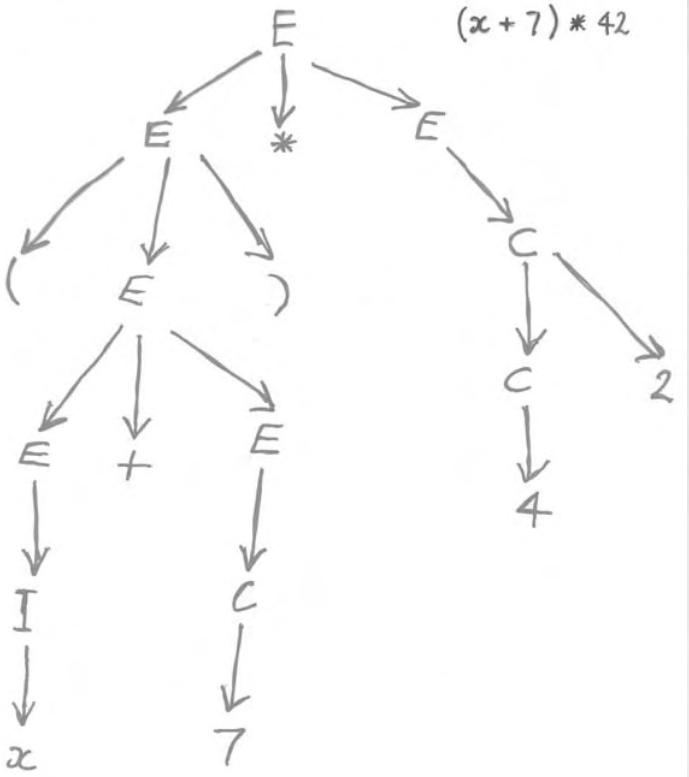
Significance of the example

- That L is not regular has great practical significance:
- Recall that a string $a^n b^n$ can be seen as n opening brackets followed by n closing brackets.
- Realistic programming languages contain balanced brackets.
- So it can be shown that realistic programming languages are are not regular!

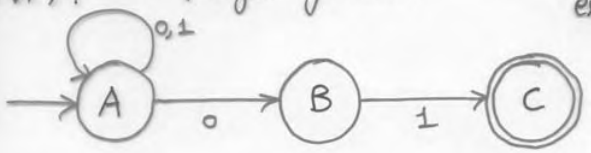




$L(G) = \{ a^n b^n \mid n \geq 1 \}$



NFA ⟶ regular grammar ⟶ regular expression.



Regular grammar $N = \{A, B, C\}$

$S = A \quad \Sigma = \{0, 1\}$

$A \to 0A \mid 0B \mid 1A$

$B \to 1C \qquad C \to \varepsilon$

Regular expression

$A = 0A + 0B + 1A$

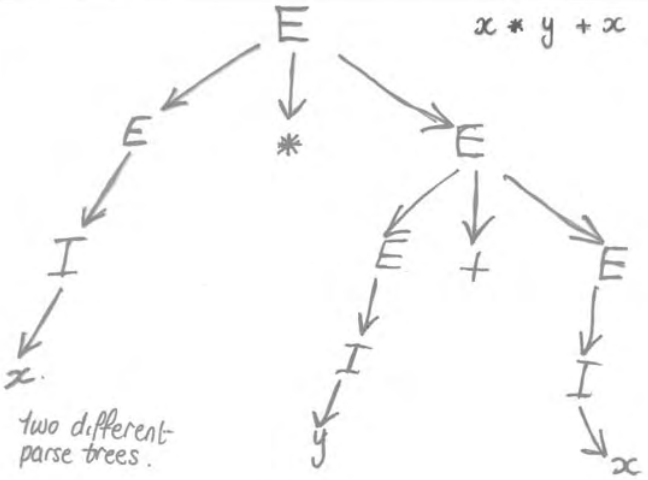
$B = 1C$

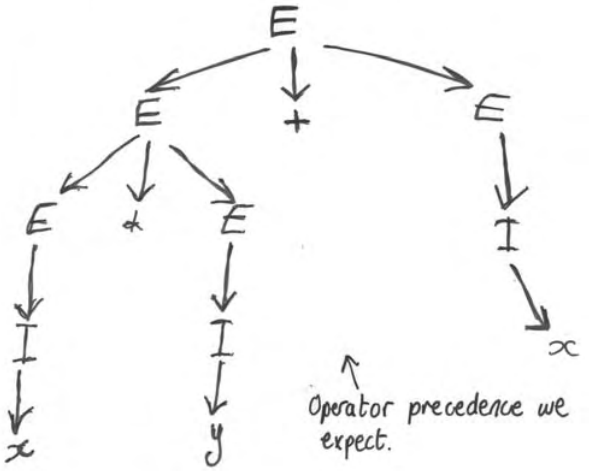
$C = \varepsilon \Rightarrow B = 1\varepsilon \Rightarrow B = 1$

$\Rightarrow A = 0A + 01 + 1A$

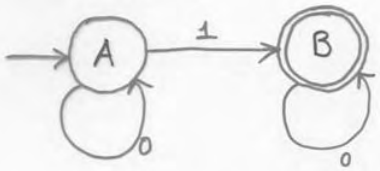
$A = (0 + 1)A + 01$

$A = (0 + 1)^* 01$





NFA ⟶ regular grammar ⟶ regular expression.



Regular grammar: $\Sigma = \{0, 1\}$. $N = \{A, B\}$.

$S = A$

$A \to 0A \mid 1B$

$B \to 0B \mid \varepsilon$

Regular expression.

$A = 0A + 1B$

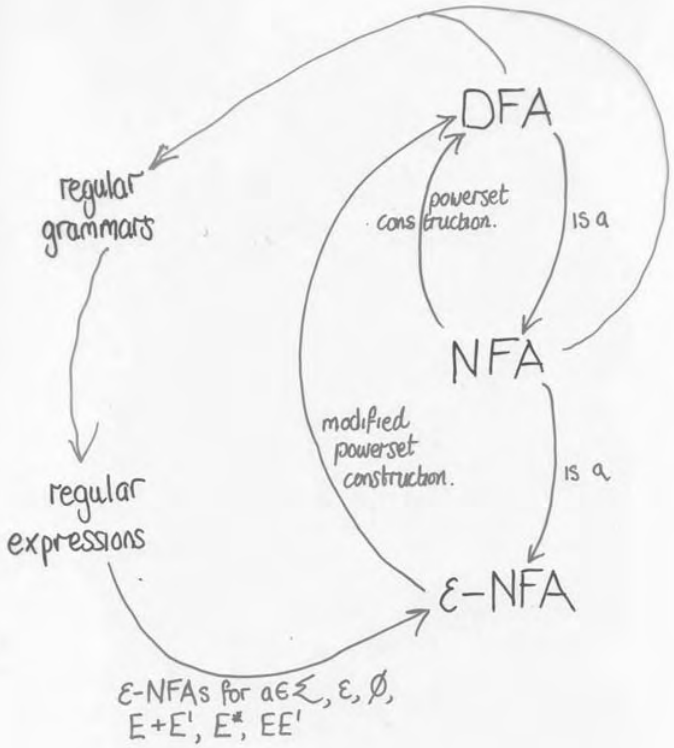
$B = 0B + \varepsilon$

$\Rightarrow B = 0^*\varepsilon = 0^*$

$\Rightarrow A = 0A + 10^*$

$\Rightarrow A = 0^*10^*$

The complete big picture.



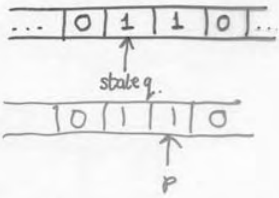
Turing machine - example.

Start with two 1's on tape, scanning leftmost:

$\rightarrow 01_q10$

011_p0

01_r10



Start with no 1's on tape:

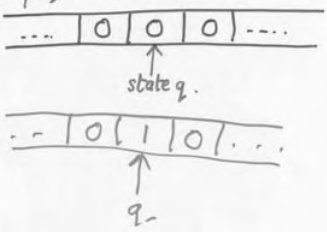
$\rightarrow 00_q0$

01_q0

010_p0

011_p0

01_r10



Pushdown automaton - $L = \{a^n b^n \mid n \geq 1\}$.

- Suppose input is $aabb = a^2b^2$.
- Start in start state q_0.
- Stack is the empty string ε.

input: aabb stack: ε

$\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, \$)\}$ \$ - marks bottom of stack.

input: aabb stack: \$. q_1

$\delta(q_1, a, \varepsilon) = \{(q_1, a)\}$

input: abb stack: \$a

$\delta(q_1, a, \varepsilon) = \{(q_1, a)\}$

input: bb stack: \$aa

$\delta(q_1, b, a) = \{(q_2, \varepsilon)\}$

input: b stack: \$a

$\delta(q_2, b, a) = \{(q_2, \varepsilon)\}$

input: ε stack: \$

$\delta(q_2, \varepsilon, \$) = \{(q_3, \varepsilon)\}$

input: ε stack: ε.

Lecture 9: The Chomsky hierarchy

The Chomsky hierarchy

There are 4 different types of formal grammars that we will meet. Some we will spend more time on than others.

- Type 3: regular grammars.
- Type 2: context free grammars.
- Type 1: context sensitive grammars.
- Type 0: unrestricted or free grammars.

To define each type we need only restrict the type of production rules permitted.

Formal grammar: definition

Definition. A formal grammar $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \to v$$

where u and v are strings in $(\Sigma \cup N)^*$ and u contains at least one non-terminal symbol;

- a **start symbol** S in N.

Context-free grammar: definition

Definition. A **context-free grammar** $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \to v$$

where u is in N and v is a string in $(\Sigma \cup N)^*$;

- a **start symbol** S in N.

Example

The grammar G with non-terminal symbols $N = \{S\}$, terminal symbols $\Sigma = \{a, b\}$, and productions

$$S \to aSb$$
$$S \to ab.$$

- This grammar describes the language $L = \{a^n b^n | n \geq 1\}$.
- We cannot build an FA that accepts this language.
- What type of machines accept context free languages?

Pushdown automata

Pushdown automata

- Informally a pushdown automata (PDA) is an ϵ-NFA with one additional capability.
- We introduce a stack on which we can store a string of symbols.
- This means a pushdown automata can "remember" an arbitary amount of information.
- However we can only access the information on the stack in a last-in, first-out way.

The stack

- The PDA works like an FA does except for the stack.
- The PDA can write symbols on to the top of the stack, "pushing down" all the other symbols on the stack.
- The PDA can read and remove a symbol from the top of the stack and the symbols below move back up.
- Writing a symbol is often referred to as "pushing"; reading and removing a symbol is referred to as "popping".

Formal definition of PDA

Definition. Let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. A (non deterministic) **pushdown automaton** (**PDA**) consists of

- a finite set of **states**, often denoted Q,
- a finite set Σ of **input symbols**,
- a finite set Γ of **stack symbols**,
- a **transition function** $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma^*)$,
- a **start state** $q_0 \in Q$,
- a set $F \subseteq Q$ of **final** or **accepting states**.

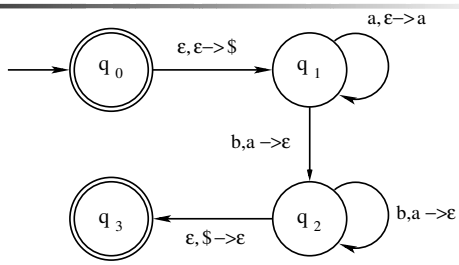
The transition function

- The transition function δ has as input the current state $q \in Q$, the next input symbol $a \in \Sigma_\varepsilon$ and the top symbol of the stack $s \in \Gamma_\varepsilon$.
- The PDA is non deterministic so the transition function produces a **set** of pairs. Each pair is a new state q' and a **string** of stack symbols $\gamma \in \Gamma^*$.
- If $a = \varepsilon$ the PDA changes state without reading an input symbol.
- If $s = \varepsilon$ the PDA changes state without reading a stack symbol.

Transition diagrams and tables

- We can still draw transition diagrams and transition tables but they need to include the stack information as well.
- We label a transition arrow from q to q' with $a, s \to \gamma$ to signify that on input a we move to state q' and pop s off the stack and push the string γ on to the stack.
- The columns of the transition table are split into subcolumns for each stack symbol.

Transition graph: example



- The language accepted by the machine is $L = \{a^n b^n | n \geq 1\}$.
- How does it "remember" the number of a's?

Transition table: example

$Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, \$\}$, $F = \{q_0, q_3\}$ and δ is defined by the table:

input	a			b			ε		
stack	a	$\$$	ε	a	$\$$	ε	a	$\$$	ε
q_0	–	–	–	–	–	–	–	–	$\{(q_1, \$)\}$
q_1	–	–	$\{(q_1, a)\}$	$\{(q_2, \varepsilon)\}$	–	–	–	–	–
q_2	–	–	–	$\{(q_2, \varepsilon)\}$	–	–	–	$\{(q_3, \varepsilon)\}$	–
q_3	–	–	–	–	–	–	–	–	–

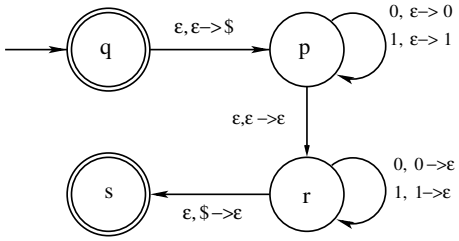
A dash signifies $\emptyset$ (to make it easier to read).

Equivalence of CFG's and PDA's

Theorem A language is context free if and only if some pushdown automaton accepts it.

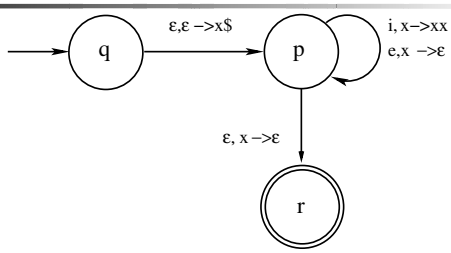
The proof of this theorem is beyond the scope of the course, if you are interested you should see Hopcroft/Motwani/Ullman.

Exercise 9.1



1. Write out the transition table for the above PDA.
2. What is the stack used to store?
3. What language does the PDA accept?

Exercise 9.2



1. What strings of length 1, 2, 3 or 4 does the machine accept?
2. Let i stand for "if" and e stand for "else". What does the PDA do?

A context sensitive grammar

Context-sensitive grammar: definition

Definition. A formal grammar is called **context sensitive** if every production rule is of the form

$$\alpha A \beta \to \alpha \gamma \beta$$

where $\alpha, \beta \in (\Sigma \cup N)^*$, $\gamma \in (\Sigma \cup N) \backslash \{\varepsilon\}$ and A is a non-terminal or of the form

$$S \to \varepsilon$$

if and only if S is not on the right hand side of any rule.

Example

The grammar G with non-terminal symbols $N = \{S, B\}$, terminal symbols $\Sigma = \{a, b, c\}$, and productions

$$S \rightarrow abc$$
$$S \rightarrow aSBc$$
$$cB \rightarrow Bc$$
$$bB \rightarrow bb$$

This grammar describes the language $L = \{a^n b^n c^n | n \geq 1\}$. We cannot build a PDA to accept this language (see Hopcroft/Motwani/Ullman).

Chomsky hierarchy: in summary

- We have seen a lot about the regular languages - see the big picture overhead.
- We have seen a context free language that is not regular - last lecture.
- We have seen machines that accept context free languages (although we have not proved this).
- We are not going to consider the class of context sensitive grammars in any detail but we have seen an example of one.
- What about unrestricted grammars?