Natural Language Processing

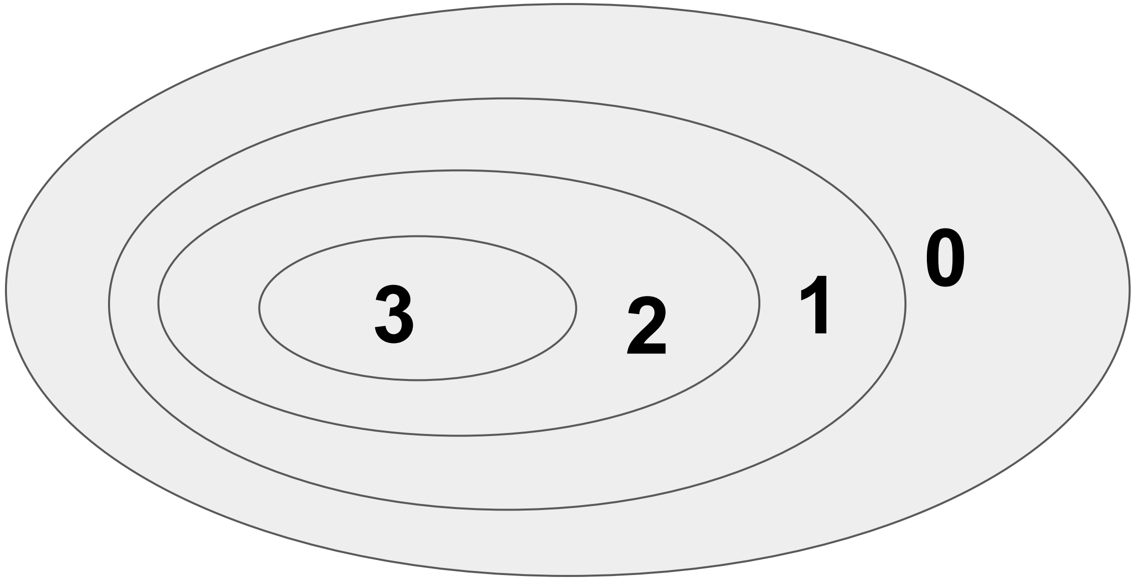
Lecture 4

Formal grammars

Formal grammars

1. **Listing grammar**: if the language *L* consists of some chains then the language *L* can be described as a list of all chains;
2. **Generating grammar** specifies rules by which you can create any chain (“sentence”) of this language;
3. **Recognizing grammar** (analytical grammar) allows to define if a word is contained in the language or not.

Generating grammars



0 - Unrestricted;
1 - Context-sensitive;
2 - Context-free;
3 - Regular
© N.Chomsky

Generating grammars

A grammar *G* is formally defined as a tuple <*N*, *Σ*, *P*, *S*>, where

- *N* is a finite set of nonterminal symbols;
- *Σ* is a finite set of terminal symbols that is disjoint from *N*;
- *P* is a finite set of production rules $\alpha \rightarrow \beta$;
- *S* is a distinguished symbol $S \in N$ that is the start symbol (sentence symbol).

Generating grammars

Alphabet is non-empty finite set, its elements are called symbols.

Chain (word, string) of symbols in alphabet Σ is a sequence of symbols from alphabet Σ. The number of symbols in chain α is called **length** and denoted as $|\alpha|$. Empty chain doesn't contain any symbols and denoted as e.

If α and β are chains then the chain $\omega = \alpha\beta$ is called **concatenation** of α and β. For any symbol a and integer k ($k \geq 0$) concatenation is denoted as a^k: $a^0 = e$, $a^1 = a$ and $a^{k+1} = a^k a$ for $k \geq 1$.

For any chains α, β and γ the chain β is **prefix** of chain $\beta\alpha$, **suffix** of chain $\alpha\beta$ and **subchain** of chain $\alpha\beta\gamma$.

Reversal of chain α (α^R) is a chain α, written in the reverse order, $e^R = e$.

Generating grammars

Example:

Binary **alphabet** is a set {0, 1}. The sequence α = 000110 is a chain with **length** 6. α is a **concatenation** of chains γ = 0001 and β = 10, i.e. $\alpha = \gamma\beta$.

Chains 0011, 000110, 1 are **subchains** of α

1^0 is **empty chain**

$\alpha^R = 011000$

$1^3 = 111$

Generating grammars

Let Σ be an alphabet. Σ^* is a set of all chains in alphabet Σ including empty chain e, Σ^+ is a set $\Sigma^* \setminus e$.

Formally a set Σ^* is defined by the following rules:

1. $e \in \Sigma^*$;
2. if $\alpha \in \Sigma^*$ and $a \in \Sigma$ then $\alpha a \in \Sigma^*$;
3. $\alpha \in \Sigma^*$ iff α is a chain in alphabet Σ due to 1 and 2.

Set Σ^* is called **Kleene closure**.

For a set of symbols we get a set of chains:

$\{‘a’, ‘b’, ‘c’\}^* = \{e, ‘a’, ‘b’, ‘c’, ‘aa’, ‘ab’, ‘ac’, ‘ba’, ‘bb’, ‘bc’, ‘ca’, ‘cb’, ‘cc’, ‘aaa’...\}$.

Generating grammars

An arbitrary set L of chains in Σ, i.e. $L \subseteq \Sigma^*$, is called a **language** in the alphabet Σ.

Let L_1 be a language in the alphabet Σ_1 and L_2 be a language in the alphabet Σ_2. Language L_1L_2 is called the **concatenation** of the languages L_1 and L_2 if it defined by a set $\{\alpha\beta : \alpha \in L_1, \beta \in L_2\}$.

The **iteration** of a language L (denoted as L^*) is defined by the following rules:

1. $L^0 = \{e\}$;
2. $L^n = LL^{n-1}$ for $n \geq 1$;
3. $L^* = \cup L^n$ (from $n \geq 0$).

Positive iteration of a language L (denoted as L^+) is $\cup L^n$ (from $n \geq 1$).

Generating grammars

Example:

The grammar *G* is a tuple <{*A*, *S*}, {0, 1}, *P*, *S*>, where *P* consists of the following rules:

- $S \rightarrow 0A1$;
- $0A \rightarrow 00A1$;
- $A \rightarrow e$.

A, *S* - nonterminal symbols;

0, 1 - terminal symbols;

S - start symbol.

Generating grammars

Let G = <N, Σ, P, S> be a grammar. The relation of a **direct deducibility** $\Rightarrow_G$ on a set $(N \cup \Sigma)^*$ is defined as follows: if $\alpha\beta\gamma$ is a chain from $(N \cup \Sigma)^*$ and $\beta \rightarrow \sigma$ is a rule from P then $\alpha\beta\gamma \Rightarrow_G \alpha\sigma\gamma$.

Transitive closure of the relation of a direct deducibility is denoted as $\varphi \Rightarrow^+_G \psi$.

Reflective and transitive closure of the relation $\Rightarrow_G$ is denoted as $\varphi \Rightarrow^*_G \psi$. **Reflexivity**: $\varphi \Rightarrow^*_G \varphi$; **transitivity**: $\varphi \Rightarrow^*_G \psi$, $\psi \Rightarrow^*_G \theta$ then $\varphi \Rightarrow^*_G \theta$.

The chain α is called **deduced chain** in grammar G if $S \Rightarrow^*_G \alpha$.

The language generated by grammar G (denoted as $L(G)$) is a set of terminal deduced chains of grammar G, i.e. $L(G) = \{\alpha : \alpha \in \Sigma^*, S \Rightarrow^*_G \alpha\}$.

Regular grammar

G = <{A, S}, {0, 1}, P, S> and the deduction $S \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 0011$ by the rules.

Then $S \Rightarrow^3 0011$, $S \Rightarrow^+ 0011$, $S \Rightarrow^* 0011$ and the chain 0011 is belonged to a language $L(G)$.

One can show that the grammar generates the language $L(G) = \{0^n1^n : n \geq 1\}$.

Context-free grammar

G = <{E, T, F}, {a,+,*,(,)}, P, E>, where P consists of the following rules:

- $E \rightarrow E + T \mid T$;
- $T \rightarrow T * F \mid F$;
- $F \rightarrow (E) \mid a$.

We can obtained the following deduction: $E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$.

Context-sensitive grammar

Let grammar P is defined by the following rules:

- $S \to aSBC$;
- $S \to abC$;
- $CB \to BC$;
- $bB \to bb$;
- $bC \to bc$;
- $cC \to cc$.

Possible deduction: $S \Rightarrow aSBC \Rightarrow aabCBC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$.

This grammar generates a language $L(G) = \{a^n b^n c^n : n \geq 1\}$.

Regular sets

Regular operations: union, concatenation, iteration. Regular sets are come out from elementary languages as a result of applying a finite number of regular operations.

Let Σ be a finite alphabet. **Regular set** in alphabet Σ is defined by the following recursive features:

1. Ø is a regular set;
2. $\{e\}$ is a regular set;
3. $\{a\}$ is a regular set $\forall\ a \in \Sigma$;
4. if P, Q are regular sets then sets $P \cup Q$, PQ, P^* are also regular sets;
5. there isn't any others regular sets in alphabet Σ.

Regular sets are sets of symbols chains on a given alphabet, made in specified way (with using of regular operations).

Regular expressions

Regular expressions (RE) in alphabet *Σ* is defined recursively in the following way:

1. Ø is a RE denoting a regular set Ø;
2. *e* is a RE denoting a regular set {*e*};
3. if *a* ∈ *Σ* then a is RE denoting a regular set {*a*};
4. if *p* and *q* are REs denoting regular sets *P* and *Q* respectively then: union (*p*+*q*) is RE denoting a regular set *P* ∪ *Q*; concatenation (*pq*) is RE denoting a regular set *PQ*; iteration $(p)^*$ is RE denoting a regular set P^*;
5. there isn't any others REs.

Regular expressions

Example 1:

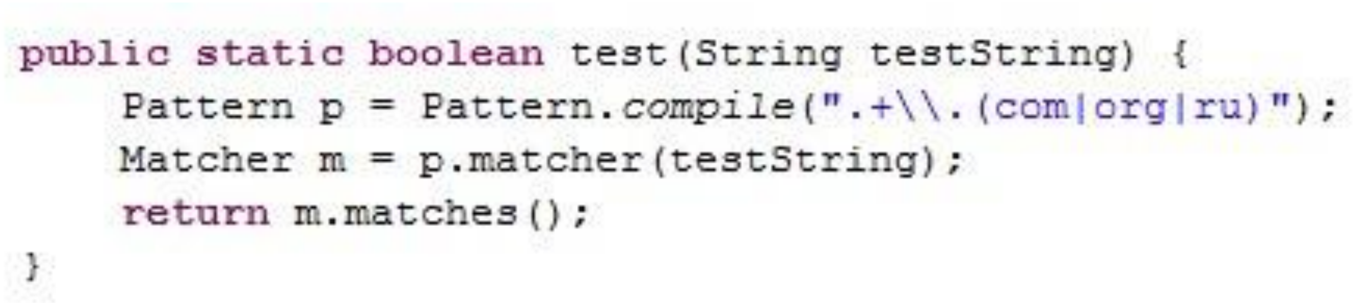
The set of all chains composed from 0 and 1 and ended with a chain 001 can be described with RE $(0+1)^{*}011$.

Example 2:

RE $(a+b)(a+b+0+1)^*$ denotes the set of all chains from $\{0, 1, a, b\}^{*}$ started with *a* or *b*.

Regular expressions

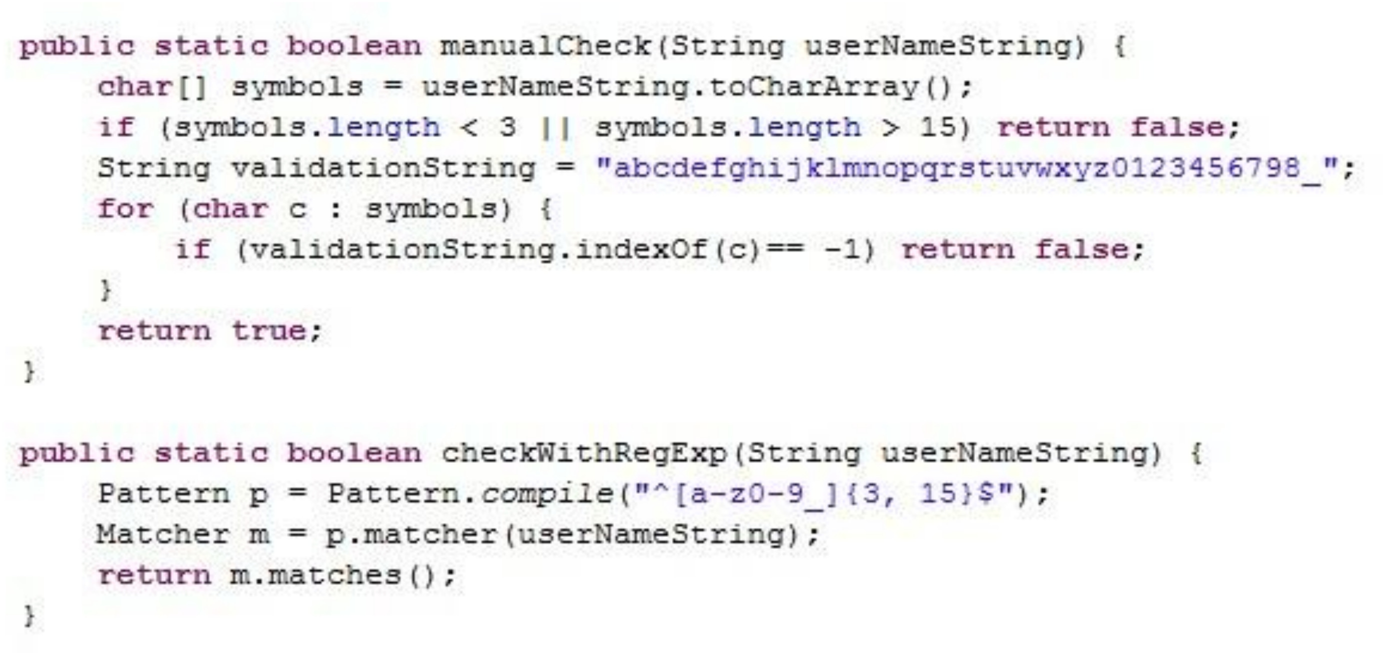
Check if a string is ended with ‘*.com*’ or ‘*.org*’ or ‘*.ru*’



```java
public static boolean test(String testString) {
    Pattern p = Pattern.compile(".+\\.(com|org|ru)");
    Matcher m = p.matcher(testString);
    return m.matches();
}
```

Regular expressions

Check if users names is valid ("*@_BEST*" and "*miha_10*"): the first one is not valid, the second one is valid.

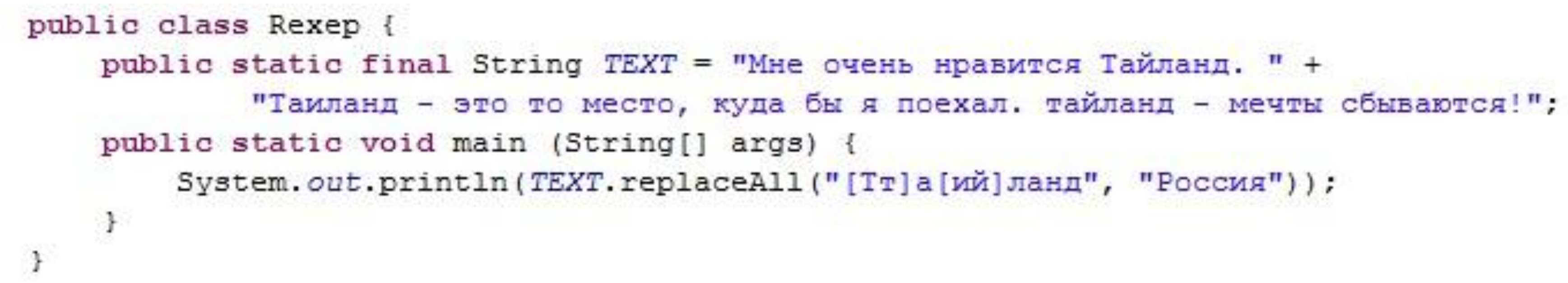


```java
public static boolean manualCheck(String userNameString) {
    char[] symbols = userNameString.toCharArray();
    if (symbols.length < 3 || symbols.length > 15) return false;
    String validationString = "abcdefghijklmnopqrstuvwxyz0123456798_";
    for (char c : symbols) {
        if (validationString.indexOf(c)== -1) return false;
    }
    return true;
}

public static boolean checkWithRegExp(String userNameString) {
    Pattern p = Pattern.compile("^[a-z0-9_]{3, 15}$");
    Matcher m = p.matcher(userNameString);
    return m.matches();
}
```

Regular expressions

Replace all variants of “*Тайланд*” with “*Россия*”.



```
public class Rexep {
    public static final String TEXT = "Мне очень нравится Тайланд. " +
            "Таиланд - это то место, куда бы я поехал. тайланд - мечты сбываются!";
    public static void main (String[] args) {
        System.out.println(TEXT.replaceAll("[Тт]а[ий]ланд", "Россия"));
    }
}
```

Relationships between regular sets, regular grammars and finite-state machines

Ways to determine the regular languages:

- regular grammars;
- finite-state machines (FSM);
- regular sets

Regular expressions and Regular grammars

- For any regular language, determined by a regular expression, one may build a regular grammar determining the same language;
- For any regular language, determined by a regular grammar, one may get a regular expression determining the same language

Regular expressions and FSMs

- For any regular language, determined by a regular expression, one may build a FSM determining the same language;
- For any regular language, determined by a FSM, one may get a regular expression determining the same language

Regular grammars and FSMs

- Based on a regular grammar one may build the equivalent FSM;
- For determined FSM one may build the equivalent regular grammar

Example:

Let a rightmost regular grammar G = ({A, S}, {a, b, c}, P, S), where P consists of the following rules:

- $S \rightarrow aS$;
- $S \rightarrow bA$;
- $A \rightarrow e$;
- $A \rightarrow cA$.

This grammar describes the same language $L(G) = \{a^n bc^n : n \geq 1\}$ as the regular expression $a^* bc^*$.

Recognizing grammars

Finite-state machine (FSM) is a transducer which allows to compare an input with the corresponding output, moreover this output can depend on not only the current input but also on previous FSM performance.

Non-determined finite-state machine (NFSM) is a tuple $M = <Q, \Sigma, \sigma, q_0, F>$, where:

- Q - finite non-empty set of states;
- Σ - input alphabet (non-empty set of symbols);
- σ - mapping a set $Q*\Sigma$ on set Q, which is called **state-transition function**;
- q_0 - initial state, the element of Q;
- F - the set of final states.

Recognizing grammars

If $M = <Q, \Sigma, \sigma, q_0, F>$ is FSM then a pair $(q, \omega) \in Q * \Sigma^*$ is called the **machine M configuration**. (q_0, ω) is called **start configuration** and (q, e) where $q \in F$ is called **final configuration**.

The step of machine *M* performance is a binary relation $\vdash_M$, determined on the configurations. If $\sigma(q, a)$ contains p then $(q, a\omega) \vdash_M (p, \omega)$ for each $\omega \in \Sigma^*$.

The relation $\vdash_M^+$ is a transitive closure of the relation $\vdash_M$. The relation $\vdash_M^*$ is reflexive and transitive closure of the relation $\vdash_M$.

FMS M **accepts** the chain $\omega \in \Sigma^*$ if $(q_0, \omega) \vdash_M^* (q, e)$ for some $q \in F$. **The language determined (recognized, accepted)** by FSM M (denoted as $L(M)$) is a set of input chains, accepted by FMS M:

$L(M) = \{\omega : \omega \in \Sigma^* \text{ and } (q_0, \omega) \vdash_M^* (q, e) \text{ for some } q \in F\}$.

Recognizing grammars

FMS begins to work in state q_0, reading symbols from the input chain x one by one. Read symbol transits the FSM into another state according to the state-transition function. In the end FSM is in a state q'. If this state is final then FMS accepts the chain x.

Determined finite-state machine (DFSM) is a finite-state machine $M = <Q, \Sigma, \sigma, q_0, F>$ where a set of state-transition functions $\sigma(q, a)$ contains no more than one state for any states $q \in Q$ and for input symbols $a \in \Sigma$.

Recognizing grammars

Example:

Build DFSM *M* which recognizes two nulls which are standing together.

This DFSM is a tuple $M = <\{p, q, r\}, \{0, 1\}, \sigma, p, \{r\}>$ where σ is defined by the following table:

State	**Input**	
	0	1
p	{*q*}	{*p*}
q	{*r*}	{*p*}
r	{*r*}	{*r*}

Recognizing grammars

Example:

Build NFSM accepting chains in the alphabet {1, 2, 3} for which the last symbol is already contained in the chain. For example: 121 is accepted, 31312 is not accepted.

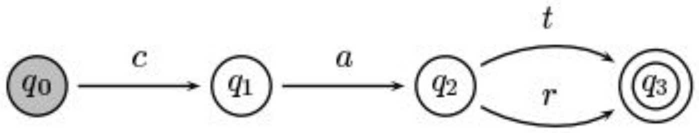
NFSM is a tuple $M = \langle\{q_0, q_1, q_2, q_3, q_f\}, \{1, 2, 3\}, \sigma, q_0, \{q_f\}\rangle$, σ is defined by the following table:

State	**Input**		
	1	2	3
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
q_1	$\{q_1, q_f\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_f\}$	$\{q_2\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
q_f	$\varnothing$	$\varnothing$	$\varnothing$

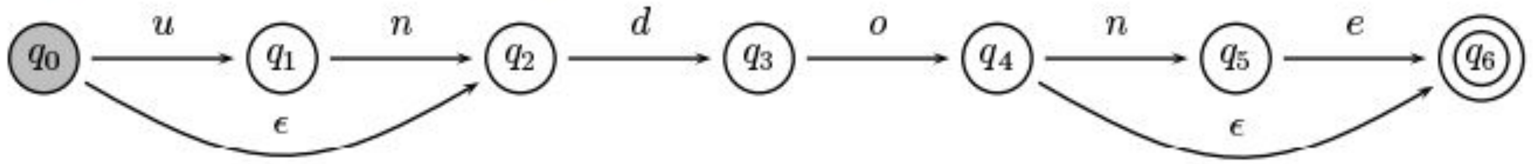
Recognizing grammars

The diagram of states (**the graph of transitions**) is annotated oriented graph, the heads correspond to the states, arcs correspond to the transitions from one state to another.

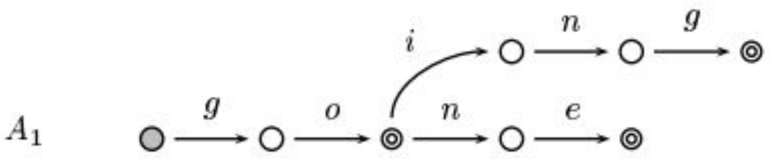
Let M = <Q, Σ, σ, q_0, F> be the FSM. If symbol $a \in \Sigma : q \in \sigma(p, a)$ exists then in the graph of transitions there is an arc (p, q).

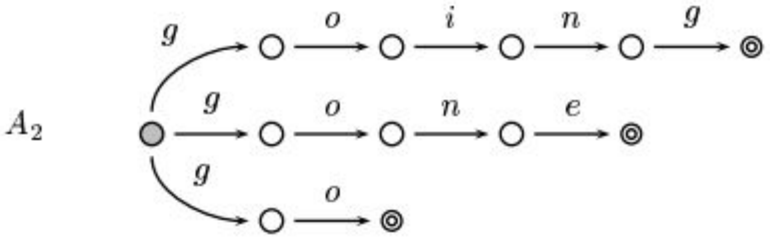


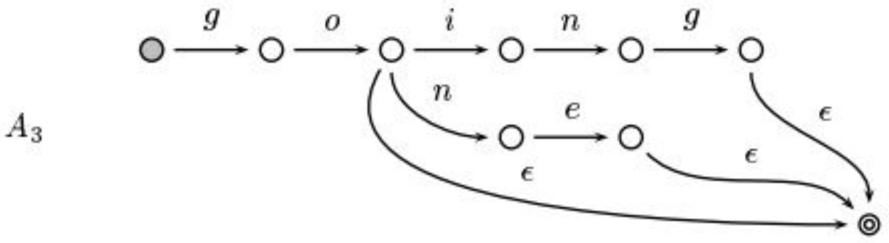
The language accepted by the following automaton is $\{$*do, undo, done, undone*$\}$:



The following three finite-state automata are equivalent: they all accept the set {*go, gone, going*}.

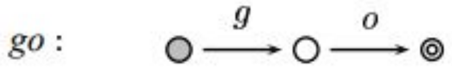




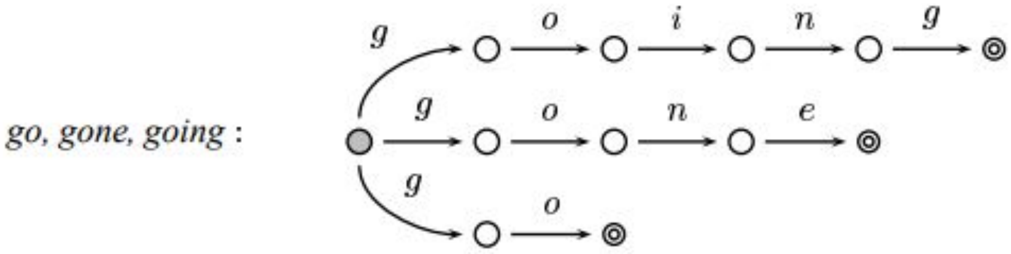


Note that A_1 is deterministic: for any state and alphabet symbol there is at most one possible transition. A_2 is not deterministic: the initial state has three outgoing arcs all labeled by g. The third automaton, A_3, has ϵ-arcs and hence is not deterministic. While A_2 might be the most readable, A_1 is the most compact as it has the fewest nodes.

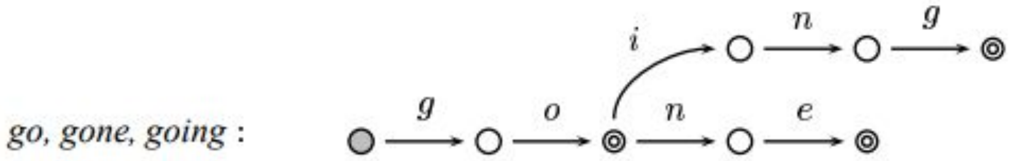
FSM in NLP: dictionaries



To represent more than one word, we can simply add paths to our "lexicon", one path for each additional word. Thus, after adding the words *gone* and *going*, we might have:



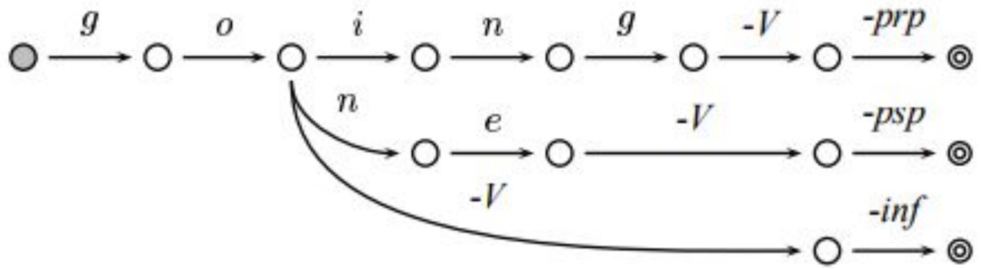
This automaton can then be determinized and minimized:



FSM in NLP: morphology

$$\Sigma = \{a, b, c, \ldots, y, z, \text{-}N, \text{-}V, \text{-}sg, \text{-}pl, \text{-}inf, \text{-}prp, \text{-}psp\}$$

With the extended alphabet, we might construct the following automaton:



The language generated by the above automaton is no longer a set of words in English. Rather, it is a set of (simplisticly) "analyzed" strings, namely {*go-V-inf, gone-V-psp, going-V-prp*}.

Recognizing grammars

Theorem (equivalency of NFSM and DFSM): any NFSM can be transformed into DFSM and their languages will overlap. Such machines are called equivalent.

Thompson algorithm for building equivalent DFSM from NFSM:

- take all parallel arcs NFSM;
- take all states in these arcs, in which NFSM can be located concurrently;
- join them into the new state of DFSM.

Recognizing grammars

Kleene’s theorem: the classes of regular sets and automatic languages are overlapped.

Algorithm for building NFSM from a regular expression:

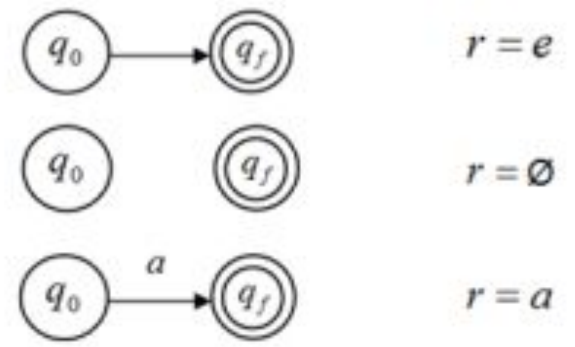
input: RE r in alphabet Σ

output: NFSM M: $L(M) = L(r)$

method: the machine for expression is built with the composition of machines, corresponding to subexpressions. On each step of building the machine has only one final state. In the start state there isn’t any transitions from other states, and there isn’t any transitions from the final state to others.

Recognizing grammars

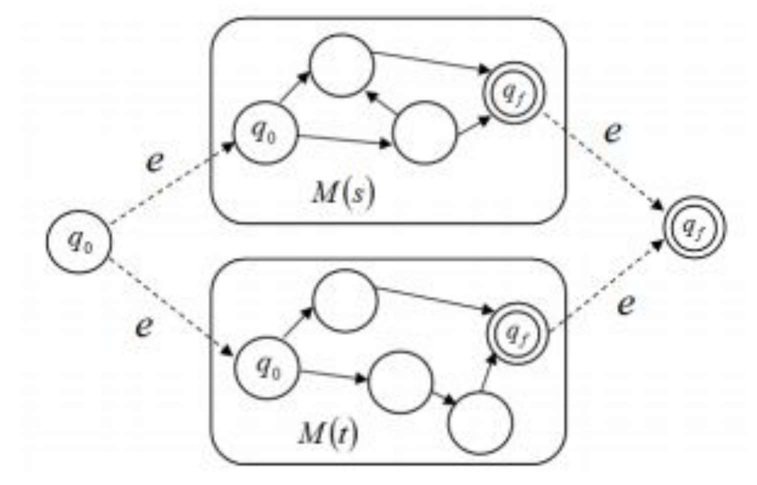
According to the definition, a set of chains in the alphabet is called **regular** iff it is one of the sets Ø, {*e*}, {*a*} ∀ *a* ∈ Σ or it can be deduced from these sets by applying the operations of joining, concatenation and iteration.



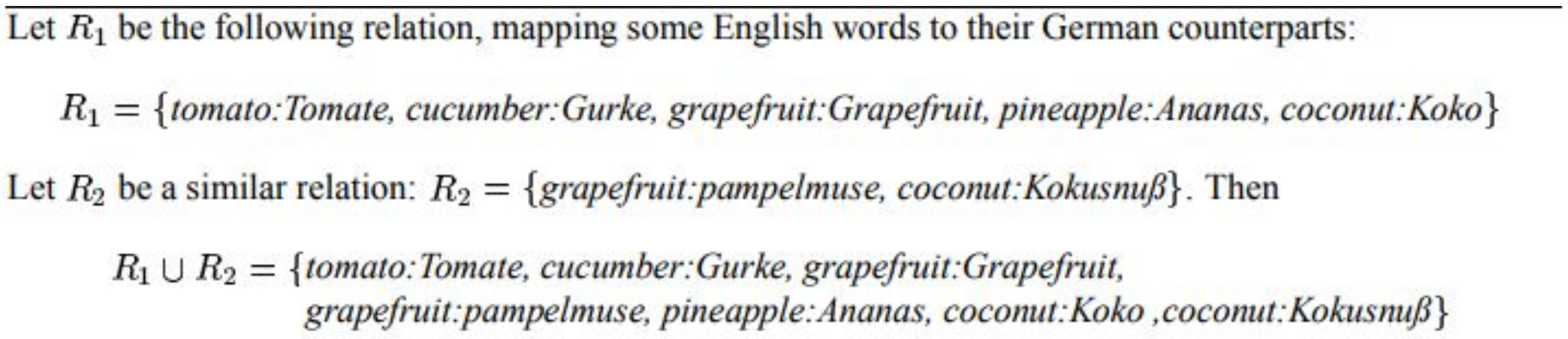
Building FSM M (s+t)

q_0 (out of the frame) is new initial state;

q_f (out of the frame) is new final state



Building FSM M (s+t)



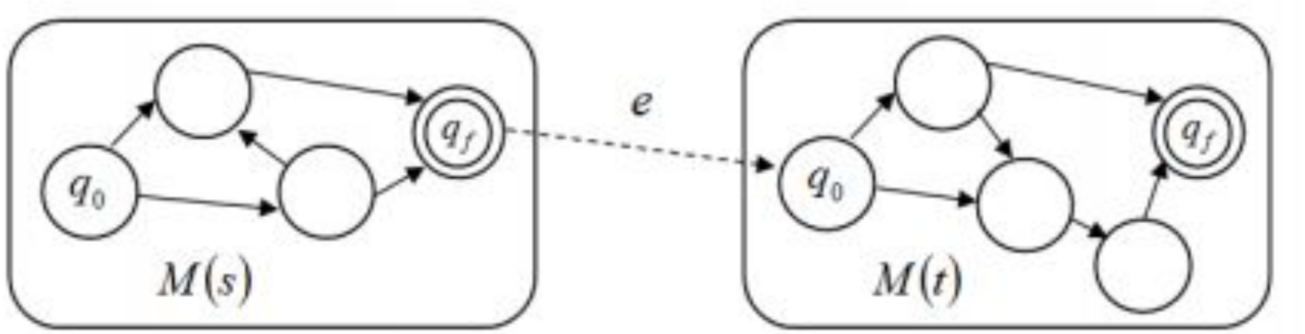
Let R_1 be the following relation, mapping some English words to their German counterparts:

$$R_1 = \{\textit{tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, pineapple:Ananas, coconut:Koko}\}$$

Let R_2 be a similar relation: $R_2 = \{\textit{grapefruit:pampelmuse, coconut:Kokusnuß}\}$. Then

$$\begin{aligned} R_1 \cup R_2 = \{&\textit{tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit,}\\ &\textit{grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko ,coconut:Kokusnuß}\} \end{aligned}$$

Building FSM M (st)



Building FSM M (st)

Let R_1 be the following relation, mapping some English words to their German counterparts:

$$R_1 = \{\textit{tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit,} \\ \textit{grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko ,coconut:Kokusnuß}\}$$

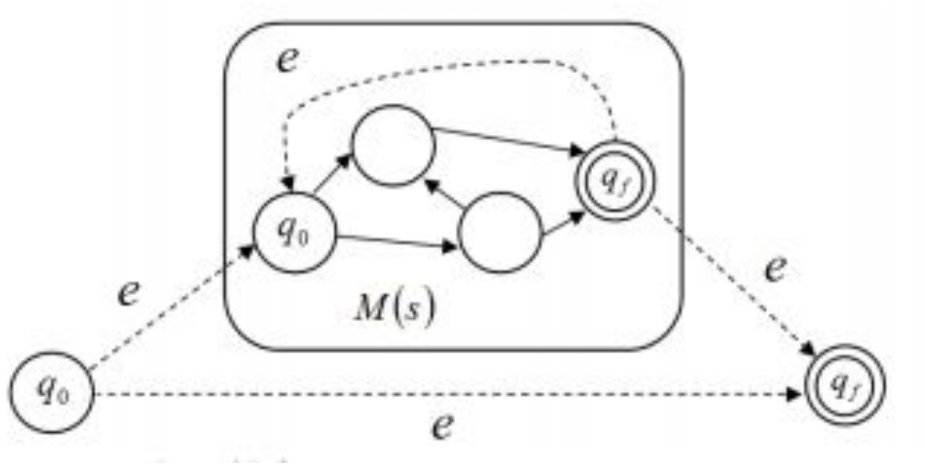
Let R_2 be a similar relation, mapping French words to their English translations:

$$R_2 = \{\textit{tomate:tomato, ananas:pineapple, pampelmousse:grapefruit,} \\ \textit{concombre:cucumber, cornichon:cucumber, noix-de-coco:coconut}\}$$

Then $R_2 \circ R_1$ is a relation mapping French words to their German translations (the English translations are used to compute the mapping, but are not part of the final relation):

$$R_2 \circ R_1 = \{\textit{tomate:Tomate, ananas:Ananas, pampelmousse:Grapefruit, pampelmousse:Pampelmuse,} \\ \textit{concombre:Gurke, cornichon:Gurke, noix-de-coco:Koko, noix-de-coco:Kokusnuße}\}$$

Building FSM M (s^*)



Thank you for your attention!