Computational morphology. Day 1. Theory of formal languages.

Alexey Sorokin[1,2]

[1]Moscow State University, [2]Moscow Institute of Physics and Technology

European Summer School
in Logic, Language and Information,
Toulouse, 24-28 July, 2017

Outline of the course

- Day 1: What is computational morphology? Theory of formal languages: regular expressions and finite automata.

Outline of the course

- Day 1: What is computational morphology? Theory of formal languages: regular expressions and finite automata.
- Day 2: Finite transducers. Their application to natural languages.

Outline of the course

- Day 1: What is computational morphology? Theory of formal languages: regular expressions and finite automata.
- Day 2: Finite transducers. Their application to natural languages.
- Day 3: Context-based morphology. Hidden Markov models.

Outline of the course

- Day 1: What is computational morphology? Theory of formal languages: regular expressions and finite automata.
- Day 2: Finite transducers. Their application to natural languages.
- Day 3: Context-based morphology. Hidden Markov models.
- Day 4: Applying hidden Markov models to morphological analysis.

Outline of the course

- Day 1: What is computational morphology? Theory of formal languages: regular expressions and finite automata.
- Day 2: Finite transducers. Their application to natural languages.
- Day 3: Context-based morphology. Hidden Markov models.
- Day 4: Applying hidden Markov models to morphological analysis.
- Day 5: Other methods and models for morphological analysis.

Day 1 outline

- What is computational morphology?

Day 1 outline

- What is computational morphology?
- Regular expressions.

Day 1 outline

- What is computational morphology?
- Regular expressions.
- Finite automata.

Day 1 outline

- What is computational morphology?
- Regular expressions.
- Finite automata.
- Finite automata for linguistic phenomena.

What is morphology?

"Morphology is the study of the forms of words, and the ways in which words are related to other words of the same language." (R. Andersen).

"Morphology is the part of linguistics which studies the word in all its relevant aspects." (I. A. Melchuk).

What is morphology?

"Morphology is the study of the forms of words, and the ways in which words are related to other words of the same language." (R. Andersen).

"Morphology is the part of linguistics which studies the word in all its relevant aspects." (I. A. Melchuk).

Informally, morphology studies:

- How the word changes in different contexts (word inflection).
- What factors determine these changes (morphological categories).
- What parts of the word reflect these changes (morpheme analysis).

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons ("(we will) read") $\mapsto \mathrm{lire+Fut+Pl+1}$

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons ("(we will) read") $\mapsto$ lire+Fut+Pl+1

- Morphological synthesis:

  lire+Fut+Pl+1 $\mapsto$ *lirons*

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons ("(we will) read") $\mapsto$ $\mathrm{lire{+}Fut{+}Pl{+}1}$

- Morphological synthesis:

  $\mathrm{lire{+}Fut{+}Pl{+}1}$ $\mapsto$ *lirons*

- Lemmatization:

  parent $\mapsto$ parent "parent", parer "(to) block"

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons (“(we will) read”) $\mapsto$ $\mathrm{lire{+}Fut{+}Pl{+}1}$

- Morphological synthesis:

  $\mathrm{lire{+}Fut{+}Pl{+}1}$ $\mapsto$ *lirons*

- Lemmatization:

  parent $\mapsto$ parent “parent”, parer “(to) block”

- Morpheme segmentation:

  overcomed $\mapsto$ over + com(e) + ed

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons ("(we will) read") $\mapsto$ $\mathrm{lire{+}Fut{+}Pl{+}1}$

- Morphological synthesis:

  $\mathrm{lire{+}Fut{+}Pl{+}1}$ $\mapsto$ *lirons*

- Lemmatization:

  parent $\mapsto$ parent "parent", parer "(to) block"

- Morpheme segmentation:

  overcomed $\mapsto$ over + com(e) + ed

- Paradigm detection:

  parler $\mapsto$ parl-er, parl-e, parl-es, parl-e, parl-ons, parl-ez, parl-ent

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons (“(we will) read”) $\mapsto \mathrm{lire{+}Fut{+}Pl{+}1}$

- Morphological synthesis:

  $\mathrm{lire{+}Fut{+}Pl{+}1} \mapsto$ *lirons*

- Lemmatization:

  parent $\mapsto$ parent “parent”, parer “(to) block”

- Morpheme segmentation:

  overcomed $\mapsto$ over + com(e) + ed

- Paradigm detection:

$$\begin{array}{lcl} \textit{parler} & \mapsto & \text{parl-er, parl-e, parl-es, parl-e,} \\ & & \text{parl-ons, parl-ez, parl-ent} \\ \textit{parler} & \mapsto & 1{+}\text{er},\ 1{+}\text{e},\ 1{+}\text{es},\ 1{+}\text{e},\ 1{+}\text{ons},\ 1{+}\text{ez},\ 1{+}\text{ent} \end{array}$$

Tasks of computational morphology

Basic tasks of computational morphology:

- Morphological analysis (tagging):

  lirons ("(we will) read") $\mapsto$ $\mathrm{lire{+}Fut{+}Pl{+}1}$

- Morphological synthesis:

  $\mathrm{lire{+}Fut{+}Pl{+}1}$ $\mapsto$ *lirons*

- Lemmatization:

  parent $\mapsto$ parent "parent", parer "(to) block"

- Morpheme segmentation:

  overcomed $\mapsto$ over + com(e) + ed

- Paradigm detection:

$$\begin{array}{rcl} \textit{parler} & \mapsto & \text{parl-er, parl-e, parl-es, parl-e,} \\ & & \text{parl-ons, parl-ez, parl-ent} \\ \textit{parler} & \mapsto & 1{+}\text{er, } 1{+}\text{e, } 1{+}\text{es, } 1{+}\text{e, } 1{+}\text{ons, } 1{+}\text{ez, } 1{+}\text{ent} \\ \textit{trouver} & \mapsto & 1{+}\text{er, } 1{+}\text{e, } 1{+}\text{es, } 1{+}\text{e, } 1{+}\text{ons, } 1{+}\text{ez, } 1{+}\text{ent} \end{array}$$

Context-dependent morphology

- Morphological synthesis and paradigm detection do not depend on context.
- But lemmatization and analysis DO!

Context-dependent morphology

- Morphological synthesis and paradigm detection do not depend on context.
- But lemmatization and analysis DO!
- *parent* $\mapsto$ parent+NOUN+Masc+Sg:

Mon	*parent*	*es*	*grand*
"My	parent	is	tall"

Context-dependent morphology

- Morphological synthesis and paradigm detection do not depend on context.
- But lemmatization and analysis DO!
- *parent* $\mapsto$ parent+NOUN+Masc+Sg:

Mon	*parent*	*es*	*grand*
"My	parent	is	tall"

- *parent* $\mapsto$ parer+VERB+Pres+Pl+3:

Les défenseurs	*parent*	*tous*	*les tirs*
"The defenders	block	all	the shots"

Context-dependent morphology

- Morphological synthesis and paradigm detection do not depend on context.
- But lemmatization and analysis DO!
- *parent* $\mapsto$ parent+NOUN+Masc+Sg:

Mon	*parent*	*es*	*grand*
"My	parent	is	tall"

- *parent* $\mapsto$ parer+VERB+Pres+Pl+3:

Les défenseurs	*parent*	*tous*	*les tirs*
"The defenders	block	all	the shots"

- The effect of context is far more strong in highly inflective languages (Russian, Czech etc.).

Applications

- Machine translation:

$$\textit{Pete bought a book} \mapsto \text{Petya kupil knigu}$$

$$\begin{array}{c} \textsf{bought} \\ \downarrow \\ \text{buy+Past} \ \textsf{(with single masculine object)} \\ \downarrow \\ \text{kupit'+Past+Sg+3+Masc} \\ \downarrow \\ \text{kupil} \end{array}$$

Applications

- Machine translation:

  Pete bought a book $\mapsto$ Petya kupil knigu

  bought

  $\downarrow$

  buy+Past (with single masculine object)

  $\downarrow$

  kupit'+Past+Sg+3+Masc

  $\downarrow$

  kupil

- Information retrieval.

Applications

- Machine translation:

$$\textit{Pete bought a book} \mapsto \text{Petya kupil knigu}$$

$$\begin{array}{c} \text{bought} \\ \downarrow \\ \text{buy+Past (with single masculine object)} \\ \downarrow \\ \text{kupit'+Past+Sg+3+Masc} \\ \downarrow \\ \text{kupil} \end{array}$$

- Information retrieval.
- Language modelling: making a probability model more sparse.

Applications

- Machine translation:

$$\textit{Pete bought a book} \mapsto \text{Petya kupil knigu}$$

$$\begin{array}{c} \text{bought} \\ \downarrow \\ \mathrm{buy{+}Past}\ \text{(with single masculine object)} \\ \downarrow \\ \mathrm{kupit'{+}Past{+}Sg{+}3{+}Masc} \\ \downarrow \\ \mathrm{kupil} \end{array}$$

- Information retrieval.
- Language modelling: making a probability model more sparse.
- Actually, morphological tagging is a preprocessing step for almost all NLP tasks.

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).
- A syllable can be described as:
  - Arbitrary number of consonants (possibly zero).

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).
- A syllable can be described as:
  - Arbitrary number of consonants (possibly zero).
  - Followed by one vowel.

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).
- A syllable can be described as:
  - Arbitrary number of consonants (possibly zero).
  - Followed by one vowel.
  - Followed by arbitrary number of consonants (possibly zero).

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).
- A syllable can be described as:
  - Arbitrary number of consonants (possibly zero).
  - Followed by one vowel.
  - Followed by arbitrary number of consonants (possibly zero).
- Formally, a syllable is C^*VC^* where $*$ stands for an arbitrary number of symbols.

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).
- A syllable can be described as:
  - Arbitrary number of consonants (possibly zero).
  - Followed by one vowel.
  - Followed by arbitrary number of consonants (possibly zero).
- Formally, a syllable is C^*VC^* where * stands for an arbitrary number of symbols.
- Now let us describe a word...
- A word includes at least one vowel and arbitrary number of consonants.

Regular languages: first example

- How to describe phonological conditions formally?
- A syllable is a sequence of letters containing one vowel (V) and arbitrary number of consonants (C).
- A syllable can be described as:
  - Arbitrary number of consonants (possibly zero).
  - Followed by one vowel.
  - Followed by arbitrary number of consonants (possibly zero).
- Formally, a syllable is C^*VC^* where $*$ stands for an arbitrary number of symbols.
- Now let us describe a word...
- A word includes at least one vowel and arbitrary number of consonants.
- Answer: $(C|V)^*V(C|V)^*$ where $|$ stands for OR.

More complex examples

- We wish to describe the syllable structure of the word more carefully.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens $(-)$.
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one.
  - All unstressed syllables are followed by a hyphen. That is $C^* VC^* -$ (V stands for unstressed).

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one.
  - All unstressed syllables are followed by a hyphen. That is $C^* VC^* -$ (V stands for unstressed).
  - We have an arbitrary number of such groups $(C^* VC^* -)^*$ followed by a stressed syllable $C^* V_0 C^*$.
  - Concatenating, we obtain $(C^* VC^* -)^* C^* V_0 C^*$.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one. $(C^* V_0 C^* -)^* C^* V_0 C^*$
- **Second case**: stressed syllable is **not the last** one.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one. $(C^* V_0 C^* -)^* C^* V_0 C^*$
- **Second case**: stressed syllable is **not the last** one.
  - Arbitrary number of hyphenated unstressed syllables, followed by a hyphenated stressed syllable,
  - followed by arbitrary number of hyphenated unstressed syllables, followed by an unstressed syllable.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one. $(C^* V_0 C^* -)^* C^* V_0 C^*$
- **Second case**: stressed syllable is **not the last** one.
  - Arbitrary number of hyphenated unstressed syllables, followed by a hyphenated stressed syllable,
  - followed by arbitrary number of hyphenated unstressed syllables, followed by an unstressed syllable.
  - Together, $(C^* VC^* -)^* C^* V_0 C^* - (C^* VC^* -)^* C^* VC^*$.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens $(-)$.
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one. $(C^* V_0 C^* -)^* C^* V_0 C^*$
- **Second case**: stressed syllable is **not the last** one. $(C^* VC^* -)^* C^* V_0 C^* - (C^* VC^* -)^* C^* VC^*$
- The answer is $((C^* VC^* -)^* C^* V_0 C^*)|((C^* VC^* -)^* C^* V_0 C^* - (C^* VC^* -)^* C^* VC^*)$.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^* V_0 C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one. $(C^* V_0 C^* -)^* C^* V_0 C^*$
- **Second case**: stressed syllable is **not the last** one. $(C^* VC^* -)^* C^* V_0 C^* - (C^* VC^* -)^* C^* VC^*$
- The answer is $((C^* VC^* -)^* C^* V_0 C^*) | ((C^* VC^* -)^* C^* V_0 C^* - (C^* VC^* -)^* C^* VC^*)$.
- Regrouping (? is "can be present or not"): $((C^* VC^* -)^* C^* V_0 C^*)((-(C^* VC^* -)^* C^* VC^*)?)$.

More complex examples

- We wish to describe the syllable structure of the word more carefully.
- We add the condition that exactly one syllable is stressed V_0 and the syllables are separated by hyphens ($-$).
- Then a stressed syllable is $C^*V_0C^*$.
- Let us separate two cases. **First case**: stressed syllable is **the last** one. $(C^*V_0C^*-)^*C^*V_0C^*$
- **Second case**: stressed syllable is **not the last** one. $(C^*VC^*-)^*C^*V_0C^* - (C^*VC^*-)^*C^*VC^*$
- The answer is $((C^*VC^*-)^*C^*V_0C^*)|((C^*VC^*-)^*C^*V_0C^* - (C^*VC^*-)^*C^*VC^*)$.
- Regrouping (? is “can be present or not”): $((C^*VC^*-)^*C^*V_0C^*)((-(C^*VC^*-)^*C^*VC^*)?)$.
- Another variant: $(C^*VC^*-)^*C^*V_0C^*(-C^*VC^*)^*$.

Examples for morphology

- Spanish verb infinitive ends with *-ar,-ir,-er* which is followed by *-se* in case of reflexive verbs.

Examples for morphology

- Spanish verb infinitive ends with *-ar*,*-ir*,*-er* which is followed by *-se* in case of reflexive verbs.
- It is simple: $(C|V)^*(a|i|e)r(se)?$.
- C is an arbitrary consonant (just join all consonants with |) and V is a vowel.

Examples for morphology

- More complex example: the plural form of English nouns:

Examples for morphology

- More complex example: the plural form of English nouns:
  - -*es* follows a sibilant (*s*, *x*, *z*, *ch*, *sh*).
  - -*s* cannot appear after e preceded by a consonant (*sky* $\mapsto$ *skies*).

Examples for morphology

- More complex example: the plural form of English nouns:
  - -*es* follows a sibilant (*s*, *x*, *z*, *ch*, *sh*).
  - -*s* cannot appear after e preceded by a consonant (*sky* $\mapsto$ *skies*).
- For this task it is easier to parse *witches* as *witche+s*, not to deal with -*es*.

Examples for morphology

- More complex example: the plural form of English nouns:
  - -*es* follows a sibilant (*s*, *x*, *z*, *ch*, *sh*).
  - -*s* cannot appear after e preceded by a consonant (*sky* $\mapsto$ *skies*).
- For this task it is easier to parse *witches* as *witche*+*s*, not to deal with -*es*.
- But -*s* must be avoided after *s*, *x*, *z*, *ch*, *sh*, **C***y*, where **C** is arbitrary consonant.

Examples for morphology

- More complex example: the plural form of English nouns:
  - -*es* follows a sibilant (*s*, *x*, *z*, *ch*, *sh*).
  - -*s* cannot appear after e preceded by a consonant (*sky* $\mapsto$ *skies*).
- For this task it is easier to parse *witches* as *witche*+*s*, not to deal with -*es*.
- But -*s* must be avoided after *s*, *x*, *z*, *ch*, *sh*, **C***y*, where **C** is arbitrary consonant.
- But regular expression cannot express negative patterns.

Examples for morphology

- More complex example: the plural form of English nouns:
    - -*es* follows a sibilant (*s*, *x*, *z*, *ch*, *sh*).
    - -*s* cannot appear after e preceded by a consonant (*sky* $\mapsto$ *skies*).
- For this task it is easier to parse *witches* as *witche*+*s*, not to deal with -*es*.
- But -*s* must be avoided after *s*, *x*, *z*, *ch*, *sh*, **C***y*, where **C** is arbitrary consonant.
- But regular expression cannot express negative patterns.
- Solution: list all that is allowed.

Examples for morphology

- A plural form is a stem followed by *-s*, where a stem can be anything that:

Examples for morphology

- A plural form is a stem followed by -*s*, where a stem can be anything that:
  - Ends with vowel not equal to y: $(C|V)^*(a|e|i|o|u)$.

Examples for morphology

- A plural form is a stem followed by -*s*, where a stem can be anything that:
  - Ends with vowel not equal to *y*: $(C|V)^*(a|e|i|o|u)$.
  - Ends with vowel+y: $(C|V)^*Vy$.

Examples for morphology

- A plural form is a stem followed by -*s*, where a stem can be anything that:
  - Ends with vowel not equal to y: $(C|V)^*(a|e|i|o|u)$.
  - Ends with vowel+y: $(C|V)^*Vy$.
  - Contains a vowel and ends with a consonant not equal to s, x, z, h (let $\mathbf{C}'$ denote their complete list): $(C|V)^*V(C|V)^*C'$

Examples for morphology

- A plural form is a stem followed by -*s*, where a stem can be anything that:
  - Ends with vowel not equal to y: $(C|V)^*(a|e|i|o|u)$.
  - Ends with vowel+y: $(C|V)^*Vy$.
  - Contains a vowel and ends with a consonant not equal to s, x, z, h (let $\mathbf{C}'$ denote their complete list): $(C|V)^*V(C|V)^*C'$
  - Contains a vowel and ends with h or $\mathbf{C}''h$, where $\mathbf{C}''$ stands for all consonants except s, c: $(C|V)^*V(C|V)^*C''?h$

Examples for morphology

- A plural form is a stem followed by -*s*, where a stem can be anything that:
    - Ends with vowel not equal to y: $(C|V)^*(a|e|i|o|u)$.
    - Ends with vowel+y: $(C|V)^*Vy$.
    - Contains a vowel and ends with a consonant not equal to s, x, z, h (let $\mathbf{C}'$ denote their complete list): $(C|V)^*V(C|V)^*C'$
    - Contains a vowel and ends with h or $\mathbf{C}''h$, where $\mathbf{C}''$ stands for all consonants except s, c: $(C|V)^*V(C|V)^*C''?h$
- Grouping all together: $(C|V)^*((a|e|i|o|u|Vy)|V(C|V)^*(C'|C''?h))s$.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.
- Languages — sets of words: $L \subseteq \Sigma^*$.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.
- Languages — sets of words: $L \subseteq \Sigma^*$.
- Operations on languages:
  - Boolean operations: $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$(complement).

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.
- Languages — sets of words: $L \subseteq \Sigma^*$.
- Operations on languages:
  - Boolean operations: $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$(complement).
  - Concatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1, w_2 \in L_2\}$.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.
- Languages — sets of words: $L \subseteq \Sigma^*$.
- Operations on languages:
  - Boolean operations: $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$(complement).
  - Concatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1, w_2 \in L_2\}$.
  - Power $L^k = \underbrace{L \cdot \ldots \cdot L}_{k \text{ times}}$. $L^0 = \{\varepsilon\}$, $L^1 = L$.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.
- Languages — sets of words: $L \subseteq \Sigma^*$.
- Operations on languages:
  - Boolean operations: $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$(complement).
  - Concatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1, w_2 \in L_2\}$.
  - Power $L^k = \underbrace{L \cdot \ldots \cdot L}_{k \text{ times}}$. $L^0 = \{\varepsilon\}$, $L^1 = L$.
  - Iteration (Kleene star): $L^* = \bigcup\limits_{k=0}^{\infty} L^k$.

Formal definitions

- Alphabet — arbitrary finite set Σ, its elements — letters.
- Words — finite sequences of letters, the set of words — Σ^*.
- ε — empty word.
- $\cdot$ — concatenation of words, $ad \cdot bc = adbc$.
- Languages — sets of words: $L \subseteq \Sigma^*$.
- Operations on languages:
  - Boolean operations: $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$(complement).
  - Concatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1, w_2 \in L_2\}$.
  - Power $L^k = \underbrace{L \cdot \ldots \cdot L}_{k \text{ times}}$. $L^0 = \{\varepsilon\}$, $L^1 = L$.
  - Iteration (Kleene star): $L^* = \bigcup\limits_{k=0}^{\infty} L^k$.
  - $\{a, b\}^* = \{a, b\}^0 \cup \{a, b\}^1 \cup \{a, b\}^2 \cup \ldots = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions $(\mathrm{Reg}(\Sigma))$:
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions $(\mathrm{Reg}(\Sigma))$:
    - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
    - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions $(\mathrm{Reg}(\Sigma))$:
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
  - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha|\beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha|\beta)) = L(\alpha) \cup L(\beta)$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions ($\mathrm{Reg}(\Sigma)$):
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
  - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha|\beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha|\beta)) = L(\alpha) \cup L(\beta)$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha \cdot \beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha \cdot \beta)) = L(\alpha) \cdot L(\beta)$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions $(\mathrm{Reg}(\Sigma))$:
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
  - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha|\beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha|\beta)) = L(\alpha) \cup L(\beta)$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha \cdot \beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha \cdot \beta)) = L(\alpha) \cdot L(\beta)$.
  - If $\alpha \in \mathrm{Reg}(\Sigma)$, then $\alpha^* \in \mathrm{Reg}(\Sigma)$, $L(\alpha^*) = L(\alpha)^*$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions ($\mathrm{Reg}(\Sigma)$):
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
  - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha|\beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha|\beta)) = L(\alpha) \cup L(\beta)$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha \cdot \beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha \cdot \beta)) = L(\alpha) \cdot L(\beta)$.
  - If $\alpha \in \mathrm{Reg}(\Sigma)$, then $\alpha^* \in \mathrm{Reg}(\Sigma)$, $L(\alpha^*) = L(\alpha)^*$.
- Priority of operations: *, $\cdot$, $|$, so $\alpha^*\beta|\gamma = ((\alpha^*) \cdot \beta)|\gamma$.

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions ($\mathrm{Reg}(\Sigma)$):
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
  - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha|\beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha|\beta)) = L(\alpha) \cup L(\beta)$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha \cdot \beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha \cdot \beta)) = L(\alpha) \cdot L(\beta)$.
  - If $\alpha \in \mathrm{Reg}(\Sigma)$, then $\alpha^* \in \mathrm{Reg}(\Sigma)$, $L(\alpha^*) = L(\alpha)^*$.
- Priority of operations: *, $\cdot$, $|$, so $\alpha^*\beta|\gamma = ((\alpha^*) \cdot \beta)|\gamma$.
- Common conventions: $\alpha^+ = \alpha\alpha^*$ (positive iteration), $\alpha? = (\alpha|1)$ (optionality).

Regular expressions: what is it formally

- We distinguish regular expression α and its language $L(\alpha)$.
- For example, if $\alpha = (a|b)(a|c)$, then $L(\alpha) = \{aa, ac, ba, bc\}$.
- Let some alphabet Σ be fixed.
- Regular expressions ($\mathrm{Reg}(\Sigma)$):
  - Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$.
  - $0, 1$ are regular expressions, $L(0) = \varnothing$, $L(1) = \{\varepsilon\}$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha|\beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha|\beta)) = L(\alpha) \cup L(\beta)$.
  - For all $\alpha, \beta \in \mathrm{Reg}(\Sigma)$ also $(\alpha \cdot \beta) \in \mathrm{Reg}(\Sigma)$, $L((\alpha \cdot \beta)) = L(\alpha) \cdot L(\beta)$.
  - If $\alpha \in \mathrm{Reg}(\Sigma)$, then $\alpha^* \in \mathrm{Reg}(\Sigma)$, $L(\alpha^*) = L(\alpha)^*$.
- Priority of operations: *, $\cdot$, $|$, so $\alpha^*\beta|\gamma = ((\alpha^*) \cdot \beta)|\gamma$.
- Common conventions: $\alpha^+ = \alpha\alpha^*$ (positive iteration), $\alpha? = (\alpha|1)$ (optionality).
- Regular languages: languages that can be expressed by regular expressions.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.
- Left to a b occurs closer than c: Exercise.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.
- Left to a b occurs closer than c: Exercise.
- No repeating letters (alphabet a, b): .

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.
- Left to a b occurs closer than c: Exercise.
- No repeating letters (alphabet a, b): $b?(ab)^*a?$.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.
- Left to a b occurs closer than c: Exercise.
- No repeating letters (alphabet a, b): $b?(ab)^*a?$.
- Non-empty word with no repetitions: $H = a(ba)^*b?|b(ab)^*a?$.

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.
- Left to a b occurs closer than c: Exercise.
- No repeating letters (alphabet a, b): $b?(ab)^*a?$.
- Non-empty word with no repetitions: $H = a(ba)^*b?|b(ab)^*a?$.
- No repeating letters (alphabet a, b, c): .

Examples of regular languages

- Words with exactly two a-s (alphabet a, b): $(a|b)^*a(a|b)^*a(a|b)^*$.
- Words with even number of a-s (alphabet a, b): $((a|b)^*a(a|b)^*a)^*(a|b)^*$.
- Words with odd number of a-s (alphabet a, b): Exercise.
- a is immediately followed by b (alphabet a, b, c): $(b|c|ab)^*$.
- a is immediately preceded by b: Exercise.
- After every a b occurs earlier than c (alphabet a, b, c, d): $(ad^*b|b|c|d)^*$.
- Left to a b occurs closer than c: Exercise.
- No repeating letters (alphabet a, b): $b?(ab)^*a?$.
- Non-empty word with no repetitions: $H = a(ba)^*b?|b(ab)^*a?$.
- No repeating letters (alphabet a, b, c): $H?(cH)^*c?$.

Exercise: vowel harmony

- Words that have at least one letter among V, V_1, V_2, but not V_1 and V_2 together.

Exercise: vowel harmony

- Words that have at least one letter among V, V_1, V_2, but not V_1 and V_2 together.
- Explanation: V_1 and V_2 are disharmonic types of vowels (say, soft and round). V are neutral vowels, C are consonants.

Exercise: vowel harmony

- Words that have at least one letter among V, V_1, V_2, but not V_1 and V_2 together.
- Explanation: V_1 and V_2 are disharmonic types of vowels (say, soft and round). V are neutral vowels, C are consonants. $C^*(V|V_1)(C|V|V_1)^*|C^*(V|V_2)(C|V|V_2)^*$.

Exercise: vowel harmony

- Words that have at least one letter among V, V_1, V_2, but not V_1 and V_2 together.
- Explanation: V_1 and V_2 are disharmonic types of vowels (say, soft and round). V are neutral vowels, C are consonants.
  $C^*(V|V_1)(C|V|V_1)^*|C^*(V|V_2)(C|V|V_2)^*$.

Exercise: Turkish infinitives.

In Turkish there are 8 vowels:

	Front	Back
Soft	e i	a ı
Round	ü ö	u o

Infinitive is formed by suffix *-mek/-mak* attached to verb stem, where *e* appears if the last vowel of stem is front and *a* – if it is back. Write a regular expression for Turkish infinitives.

Finite automata

- Regular expressions are convinient to describe patterns.

Finite automata

- Regular expressions are convinient to describe patterns.
- But there is no way to check that a word satisfies to an expression.

Finite automata

- Regular expressions are convinient to describe patterns.
- But there is no way to check that a word satisfies to an expression.
- Example: $a(a|b|c)^*b(a|b|c)$.

Finite automata

- Regular expressions are convinient to describe patterns.
- But there is no way to check that a word satisfies to an expression.
- Example: $a(a|b|c)^*b(a|b|c)$.
- How we can process it:
  - Read the first letter, check that it is a, otherwise reject.

Finite automata

- Regular expressions are convinient to describe patterns.
- But there is no way to check that a word satisfies to an expression.
- Example: $a(a|b|c)^*b(a|b|c)$.
- How we can process it:
  - Read the first letter, check that it is a, otherwise reject.
  - Read the letters until the penultimate letter appears.

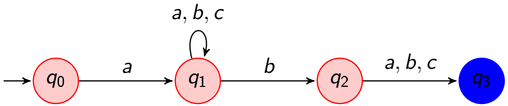
Finite automata

- Regular expressions are convinient to describe patterns.
- But there is no way to check that a word satisfies to an expression.
- Example: $a(a|b|c)^*b(a|b|c)$.
- How we can process it:
  - Read the first letter, check that it is a, otherwise reject.
  - Read the letters until the penultimate letter appears.
  - Check that it is b.
  - Check that exactly one letter remains.

Finite automata

- Regular expressions are convinient to describe patterns.
- But there is no way to check that a word satisfies to an expression.
- Example: $a(a|b|c)^*b(a|b|c)$.
- How we can process it:
  - Read the first letter, check that it is a, otherwise reject.
  - Read the letters until the penultimate letter appears.
  - Check that it is b.
  - Check that exactly one letter remains.

  Schematically:



- That is finite automaton.

Finite automata

- Finite automaton consists of:
  - Final set of states Q.
  - Alphabet Σ.

Finite automata

- Finite automaton consists of:
  - Final set of states Q.
  - Alphabet Σ.
  - Set of transitions (edges) $\Delta \subseteq Q \times \Sigma^* \times Q$:

    $q_1 \xrightarrow{w} q_2 \quad \langle q_1, w \rangle \to q_2$

Finite automata

- Finite automaton consists of:
  - Final set of states Q.
  - Alphabet Σ.
  - Set of transitions (edges) $\Delta \subseteq Q \times \Sigma^* \times Q$:

    $q_1 \xrightarrow{w} q_2 \quad \langle q_1, w \rangle \to q_2$
  - Initial state q_0.

Finite automata

- Finite automaton consists of:
  - Final set of states Q.
  - Alphabet Σ.
  - Set of transitions (edges) $\Delta \subseteq Q \times \Sigma^* \times Q$:

    $q_1 \xrightarrow{w} q_2 \quad \langle q_1, w \rangle \to q_2$
  - Initial state q_0.
  - Set of (possibly multiple) final states $F \subseteq Q$.

Finite automata

- Finite automaton consists of:
  - Final set of states Q.
  - Alphabet Σ.
  - Set of transitions (edges) $\Delta \subseteq Q \times \Sigma^* \times Q$:

    $q_1 \xrightarrow{w} q_2 \quad \langle q_1, w \rangle \to q_2$
  - Initial state q_0.
  - Set of (possibly multiple) final states $F \subseteq Q$.
- Every edge have its label. The label of a path is the concatenation of its edges labels.

Finite automata

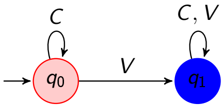
- Finite automaton consists of:
    - Final set of states Q.
    - Alphabet Σ.
    - Set of transitions (edges) $\Delta \subseteq Q \times \Sigma^* \times Q$:

      $q_1 \xrightarrow{w} q_2 \quad \langle q_1, w \rangle \to q_2$

    - Initial state q_0.
    - Set of (possibly multiple) final states $F \subseteq Q$.
- Every edge have its label. The label of a path is the concatenation of its edges labels.
- Automaton $\mathcal{A}$ accepts language $L(\mathcal{A})$ of all words that label paths from initial state to some final.

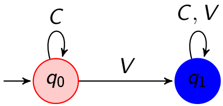
Finite automata: examples

- A syllable: states check vowel presence.

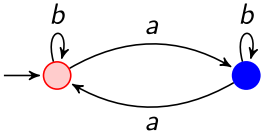


Finite automata: examples

- A syllable: states check vowel presence.

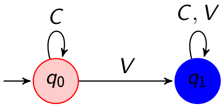


- Even number of a-s, alphabet a, b. States check parity of a-s.

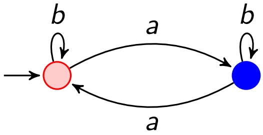


Finite automata: examples

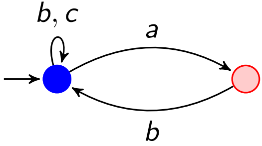
- A syllable: states check vowel presence.



- Even number of a-s, alphabet a, b. States check parity of a-s.

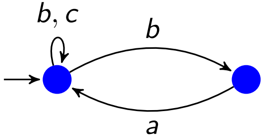


- Every a is immediately followed by b, alphabet a, b, c.



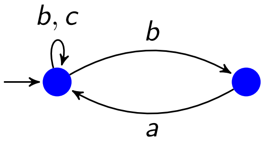
Finite automata: examples

- Every a is immediately preceded by b, alphabet a, b, c.

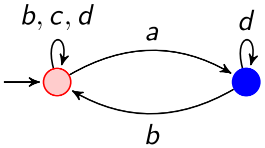


Finite automata: examples

- Every a is immediately preceded by b, alphabet a, b, c.

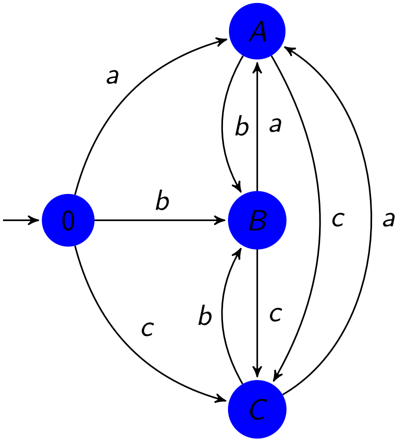


- To the right of every a occurs b with no a, c between them, alphabet a, b, c, d.



Finite automata: examples

No repeating letters, alphabet a, b, c. States correspond to letters:



Finite automata: examples

- Word syllabification: each syllable contains exactly one vowel and exactly one vowel is stressed, syllables are separated by hyphens.

Finite automata: examples

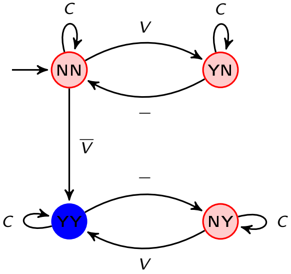
- Word syllabification: each syllable contains exactly one vowel and exactly one vowel is stressed, syllables are separated by hyphens.
- States check two conditions:
  - There was a vowel in current syllable (the first coordinate).

Finite automata: examples

- Word syllabification: each syllable contains exactly one vowel and exactly one vowel is stressed, syllables are separated by hyphens.
- States check two conditions:
  - There was a vowel in current syllable (the first coordinate).
  - There was a stressed vowel (the second coordinate).

Finite automata: examples

- Word syllabification: each syllable contains exactly one vowel and exactly one vowel is stressed, syllables are separated by hyphens.
- States check two conditions:
  - There was a vowel in current syllable (the first coordinate).
  - There was a stressed vowel (the second coordinate).



Finite automata: English plural

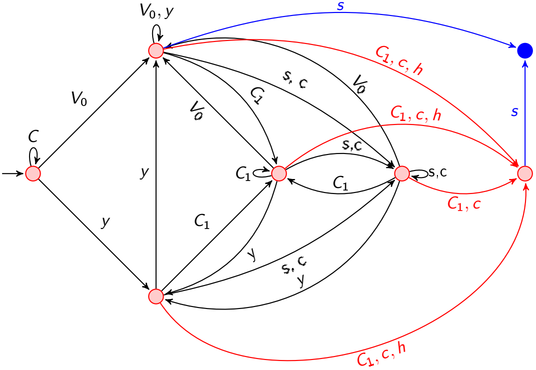
- All plural forms can be decomposed as stem + s, where

Finite automata: English plural

- All plural forms can be decomposed as stem + s, where
- A stem is anything with at least one vowel, but not ending with:
  - *-s, -x, -z, -sh, -ch, -zh* (sibilants).
  - **C***y*.

Finite automata: English plural

- All plural forms can be decomposed as stem + s, where
- A stem is anything with at least one vowel, but not ending with:
  - *-s, -x, -z, -sh, -ch, -zh* (sibilants).
  - **C***y*.
- Automaton for all possible stems
  $(C_0 = C - \{s, x, z, c, h\}, C_1 = C_0 \cup \{s, x, z\})$:



Properties of finite automata

Theorem

Every automata language is recognized by an automaton with single letter labels.

Properties of finite automata

Theorem

Every automata language is recognized by an automaton with single letter labels.

Sketch of the proof

- Split all labels of length $\geqslant 2$ by inserting additional states.
- Now we have only letters and ε as labels.

Properties of finite automata

Theorem

Every automata language is recognized by an automaton with single letter labels.

Sketch of the proof

- Split all labels of length $\geqslant 2$ by inserting additional states.
- Now we have only letters and ε as labels.
- Add an edge $\langle q_1, a\rangle \to q_2$ if there exist states q_3, q_4 such that $(\langle q_3, a\rangle \to q_4) \in \Delta$ and there are ε-paths from q_1 to q_3 and from q_4 to q_2.

Properties of finite automata

Theorem

Every automata language is recognized by an automaton with single letter labels.

Sketch of the proof

- Split all labels of length $\geqslant 2$ by inserting additional states.
- Now we have only letters and ε as labels.
- Add an edge $\langle q_1, a\rangle \rightarrow q_2$ if there exist states q_3, q_4 such that $(\langle q_3, a\rangle \rightarrow q_4) \in \Delta$ and there are ε-paths from q_1 to q_3 and from q_4 to q_2.
- Mark as terminal all states from which terminal states are ε-reachable.
- Now remove all ε-paths.

Properties of finite automata

Properties of finite automata

Definition

An automaton with one-letter labels is deterministic if no state has two outcoming edges with the same label.

Theorem

Every automata language can be recognized by deterministic automata.

Properties of finite automata

Definition

An automaton with one-letter labels is deterministic if no state has two outcoming edges with the same label.

Theorem

Every automata language can be recognized by deterministic automata.

Sketch of the proof

- New automaton states are sets of old states.

Properties of finite automata

Definition

An automaton with one-letter labels is deterministic if no state has two outcoming edges with the same label.

Theorem

Every automata language can be recognized by deterministic automata.

Sketch of the proof

- New automaton states are sets of old states.
- An edge labeled by a leads from set Q_1 to Q_2 if Q_2 contains exactly the states reachable from Q_1 by a.

Properties of finite automata

Definition

An automaton with one-letter labels is deterministic if no state has two outcoming edges with the same label.

Theorem

Every automata language can be recognized by deterministic automata.

Sketch of the proof

- New automaton states are sets of old states.
- An edge labeled by a leads from set Q_1 to Q_2 if Q_2 contains exactly the states reachable from Q_1 by a.
- Start state $Q_0 = \{q_0\}$ (only old start state).

Properties of finite automata

Definition

An automaton with one-letter labels is deterministic if no state has two outcoming edges with the same label.

Theorem

Every automata language can be recognized by deterministic automata.

Sketch of the proof

- New automaton states are sets of old states.
- An edge labeled by a leads from set Q_1 to Q_2 if Q_2 contains exactly the states reachable from Q_1 by a.
- Start state $Q_0 = \{q_0\}$ (only old start state).
- Final states: subsets containing at least one old final state.

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

- We should transform every finite automaton to regular expression and every regular expression to finite automaton.

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

- We should transform every finite automaton to regular expression and every regular expression to finite automaton.
- Automaton $\to$ expression: difficult, we will not prove it.
- Expression $\to$ automaton: simple proof by induction:

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

- We should transform every finite automaton to regular expression and every regular expression to finite automaton.
- Automaton $\rightarrow$ expression: difficult, we will not prove it.
- Expression $\rightarrow$ automaton: simple proof by induction:
- Regular languages are constructed from primitives by means of concatenation, union and iteration.

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

- We should transform every finite automaton to regular expression and every regular expression to finite automaton.
- Automaton $\rightarrow$ expression: difficult, we will not prove it.
- Expression $\rightarrow$ automaton: simple proof by induction:
- Regular languages are constructed from primitives by means of concatenation, union and iteration.
- Primitive regular languages (singletons and empty language) are certainly automata.

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

- We should transform every finite automaton to regular expression and every regular expression to finite automaton.
- Automaton $\rightarrow$ expression: difficult, we will not prove it.
- Expression $\rightarrow$ automaton: simple proof by induction:
- Regular languages are constructed from primitives by means of concatenation, union and iteration.
- Primitive regular languages (singletons and empty language) are certainly automata.
- We should prove that regular operations preserve automata languages.

Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

Concatenation: $L_1 = L(M_1), L_2 = L(M_2) \rightarrow L_1 \cdot L_2 = L(M)$

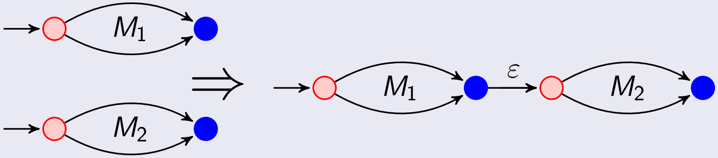
Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

Concatenation: $L_1 = L(M_1), L_2 = L(M_2) \rightarrow L_1 \cdot L_2 = L(M)$



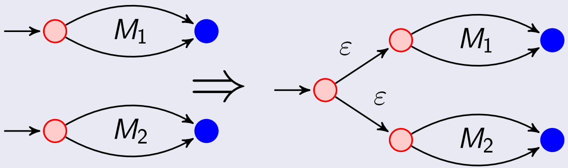
Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

Union: $L_1 = L(M_1), L_2 = L(M_2) \to L_1 \cup L_2 = L(M)$



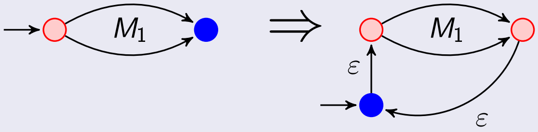
Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

Iteration: $L_1 = L(M_1), L_1^* = L(M)$



Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

- Consider the deterministic automaton for language L.

Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

- Consider the deterministic automaton for language L.
- Complete it: add a new sink state q'.
- If a state q_1 does not have outcoming edge labeled by letter a, add an edge $\langle q_1, a\rangle \to q'$.

Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

- Consider the deterministic automaton for language L.
- Complete it: add a new sink state q'.
- If a state q_1 does not have outcoming edge labeled by letter a, add an edge $\langle q_1, a\rangle \to q'$.
- Add edge $\langle q', a\rangle \to a$ for every letter a.

Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

- Consider the deterministic automaton for language L.
- Complete it: add a new sink state q'.
- If a state q_1 does not have outcoming edge labeled by letter a, add an edge $\langle q_1, a\rangle \rightarrow q'$.
- Add edge $\langle q', a\rangle \rightarrow a$ for every letter a.
- Now for every $q_1 \in Q, a \in \Sigma$ there is an edge of the form $\langle q_1, a\rangle \rightarrow q_2$.

Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

- Consider the deterministic automaton for language L.
- Complete it: add a new sink state q'.
- If a state q_1 does not have outcoming edge labeled by letter a, add an edge $\langle q_1, a\rangle \to q'$.
- Add edge $\langle q', a\rangle \to a$ for every letter a.
- Now for every $q_1 \in Q, a \in \Sigma$ there is an edge of the form $\langle q_1, a\rangle \to q_2$.
- Consequently, every word w leads from q_0 to exactly one state: terminal if $w \in L$ and non-terminal if $w \in \overline{L}$.

Properties of automata languages

Theorem

The class of automata languages is closed under complement.

Sketch of the proof

- Consider the deterministic automaton for language L.
- Complete it: add a new sink state q'.
- If a state q_1 does not have outcoming edge labeled by letter a, add an edge $\langle q_1, a\rangle \to q'$.
- Add edge $\langle q', a\rangle \to a$ for every letter a.
- Now for every $q_1 \in Q, a \in \Sigma$ there is an edge of the form $\langle q_1, a\rangle \to q_2$.
- Consequently, every word w leads from q_0 to exactly one state: terminal if $w \in L$ and non-terminal if $w \in \overline{L}$.
- Switching non-terminal and terminal states yields automaton for the complement.

Properties of automata languages

Theorem

The class of automata languages is closed under intersection.

Sketch of the proof

Properties of automata languages

Theorem

The class of automata languages is closed under intersection.

Sketch of the proof

- Easy variant: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

Properties of automata languages

Theorem

The class of automata languages is closed under intersection.

Sketch of the proof

- Easy variant: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.
- Complex (but effective) variant: consider complete deterministic automata M_1 for L_1 and M_2 for L_2.
- Let Q_1, Q_2 be their sets of states, q_{01}, q_{02} be initial states and F_1, F_2 be sets of final states.

Properties of automata languages

Theorem

The class of automata languages is closed under intersection.

Sketch of the proof

- Easy variant: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.
- Complex (but effective) variant: consider complete deterministic automata M_1 for L_1 and M_2 for L_2.
- Let Q_1, Q_2 be their sets of states, q_{01}, q_{02} be initial states and F_1, F_2 be sets of final states.
- Consider a new automaton whose states are pairs $\langle q_1, q_2 \rangle$, $q_1 \in Q_1, q_2 \in Q_2$.

Properties of automata languages

Theorem

The class of automata languages is closed under intersection.

Sketch of the proof

- Easy variant: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.
- Complex (but effective) variant: consider complete deterministic automata M_1 for L_1 and M_2 for L_2.
- Let Q_1, Q_2 be their sets of states, q_{01}, q_{02} be initial states and F_1, F_2 be sets of final states.
- Consider a new automaton whose states are pairs $\langle q_1, q_2 \rangle$, $q_1 \in Q_1, q_2 \in Q_2$.
- Its start state is $\langle q_{01}, q_{02} \rangle$.
- On the first coordinate it operates like M_1, on the second like M_2.

Properties of automata languages

Theorem

The class of automata languages is closed under intersection.

Sketch of the proof

- Easy variant: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.
- Complex (but effective) variant: consider complete deterministic automata M_1 for L_1 and M_2 for L_2.
- Let Q_1, Q_2 be their sets of states, q_{01}, q_{02} be initial states and F_1, F_2 be sets of final states.
- Consider a new automaton whose states are pairs $\langle q_1, q_2 \rangle$, $q_1 \in Q_1, q_2 \in Q_2$.
- Its start state is $\langle q_{01}, q_{02} \rangle$.
- On the first coordinate it operates like M_1, on the second like M_2.
- Finite states are pairs of final states (the automaton accepts iff it accepts for both coordinates).

Recursive construction of automata

- Finite automata are closed under a couple of operations.

Recursive construction of automata

- Finite automata are closed under a couple of operations.
- Moreover, this closure is effective: corresponding automata are built algorithmically.

Recursive construction of automata

- Finite automata are closed under a couple of operations.
- Moreover, this closure is effective: corresponding automata are built algorithmically.
- Therefore we may combine automata just as regular expressions, but with more operations.

Recursive construction of automata

- Finite automata are closed under a couple of operations.
- Moreover, this closure is effective: corresponding automata are built algorithmically.
- Therefore we may combine automata just as regular expressions, but with more operations.
- For example, the automata for English plural can be expressed as:

$$(L_{sib} \cdot es) \cup (((\overline{L_{sib}} \cap L_C) \cup L_{Cy} \cup L_V) \cdot s),$$

  where

  - L_{sib} — words ending with sibilant.
  - L_C — words ending with consonant.
  - L_{Cy} — words ending with consonant+y.
  - L_V — words ending with vowel (not y).

Recursive construction of automata

- Finite automata are closed under a couple of operations.
- Moreover, this closure is effective: corresponding automata are built algorithmically.
- Therefore we may combine automata just as regular expressions, but with more operations.
- For example, the automata for English plural can be expressed as:

$$(L_{sib} \cdot es) \cup (((\overline{L_{sib}} \cap L_C) \cup L_{Cy} \cup L_V) \cdot s),$$

  where
    - L_{sib} — words ending with sibilant.
    - L_C — words ending with consonant.
    - L_{Cy} — words ending with consonant+y.
    - L_V — words ending with vowel (not y).
- The basic languages are the automata ones; the automaton for the whole expression could be constructed recursively.

Recursive construction of automata

Turkish infinitive

Construct a finite automaton for Turkish infinitive

- Infinitive has the form stem+*mEk*.
- Placeholder *E* is filled by *e* if the stem ends with *e*, *i*, *ö*, *ü* and *a* if it ends with *a*, *ı*, *o*, *u*.

Recursive construction of automata

Turkish infinitive

Construct a finite automaton for Turkish infinitive

- Infinitive has the form stem+*mEk*.
- Placeholder *E* is filled by *e* if the stem ends with *e*, *i*, *ö*, *ü* and *a* if it ends with *a*, *ı*, *o*, *u*.

- M_1 is the automaton for expression $C^*V(C|V)^*m(a|e)k$ (it is easy to construct it).

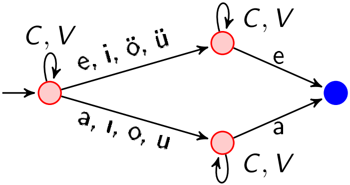
Recursive construction of automata

Turkish infinitive

Construct a finite automaton for Turkish infinitive

- Infinitive has the form stem+*mEk*.
- Placeholder *E* is filled by *e* if the stem ends with *e, i, ö, ü* and *a* if it ends with *a, ı, o, u*.

- M_1 is the automaton for expression $C^*V(C|V)^*m(a|e)k$ (it is easy to construct it).
- M_2 checks the condition for vowels:



- $M_1 \cap M_2$ is the required automaton.

Recursive construction of automata

Turkish infinitive

Construct a finite automaton for turkish passive infinitive

- Infinitive has the form stem+X+*mEk*.
- Placeholder *E* is filled by *e* if the stem ends with *e, i, ö, ü* and *a* if it ends with *a, ı, o, u*.
- Suffix *X* is -*n* if the stem ends with vowel, -*In* if the stem ends with *l* and -*Il* otherwise.
- Placeholder I equals *ı* after a, *ı*; *u* after *u, o*; *i* after *e, i*; *ü* after *ü, ö*.

Where to get presentations

- https://www.irit.fr/esslli2017/courses/33.
- http://tipl.philol.msu.ru/~otipl/index.php/department/faculty/AAS/esslli

For the next day:
Install (simply download and unpack) finite-state compiler FOMA from https://code.google.com/archive/p/foma/.