Computational morphology.
Day 2. Finite-state transducers.

Alexey Sorokin[1,2]

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

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

Day 2 outline

- Finite transducers.

Day 2 outline

- Finite transducers.
- Finite transducers for linguistic phenomena.

Day 2 outline

- Finite transducers.
- Finite transducers for linguistic phenomena.
- Compiling finite transducers with FOMA.

Finite automata: English plural

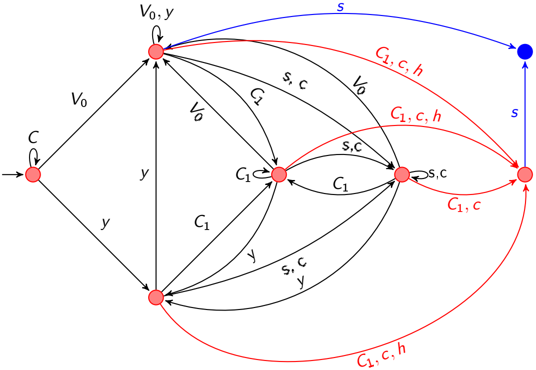
- All plural forms can be decomposed as stem + s, where

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 $\langle q_1, a\rangle \to q_2$ 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.
- 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 $\rightarrow$ expression: difficult, we will not prove it.
- Expression $\rightarrow$ 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: $L_1 = L(M_1), L_2 = L(M_2) \rightarrow L_1 \cdot L_2 = L(M)$

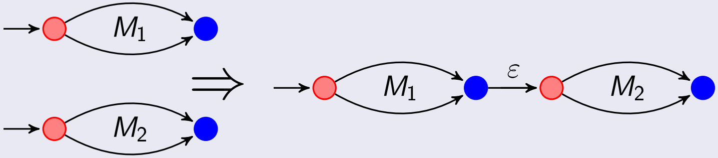
Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

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



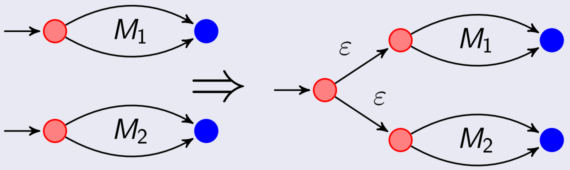
Kleene theorem

Theorem

The classes of automata and regular languages are the same.

Sketch of the proof

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



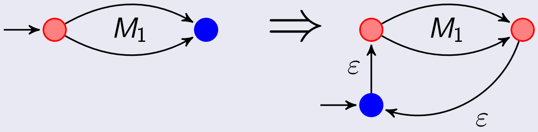
Kleene theorem

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 $\langle q_1, a\rangle \to q'$.
- 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 $\langle q_1, a\rangle \to q_2$.

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).

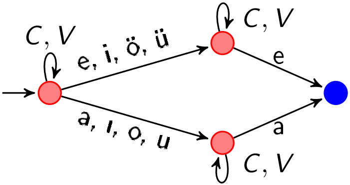
Recursive construction of automata

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, -*An* if the stem ends with *l* and -*Al* otherwise.
- Placeholder *A* equals *ı* after a, *ı*; *u* after *u, o*; *i* after *e, i*; *ü* after *ü, ö*.

Finite transducers: definition

- Finite transducers are automata with output.

Finite transducers: definition

- Finite transducers are automata with output.
- Precisely, now there are two alphabets: Σ (input) and Γ (output).
- Edges have the form $\langle u : v \rangle$, $u \in \Sigma^*$, $v \in \Gamma^*$ and mean "replace u with v".

Finite transducers: definition

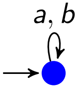
- Finite transducers are automata with output.
- Precisely, now there are two alphabets: Σ (input) and Γ (output).
- Edges have the form $\langle u : v \rangle, u \in \Sigma^*, v \in \Gamma^*$ and mean "replace u with v".
- Summarizing, finite transducers define not sets but relations between inputs and outputs.

Finite transducers: definition

- Finite transducers are automata with output.
- Precisely, now there are two alphabets: Σ (input) and Γ (output).
- Edges have the form $\langle u : v \rangle, u \in \Sigma^*, v \in \Gamma^*$ and mean "replace u with v".
- Summarizing, finite transducers define not sets but relations between inputs and outputs.
- Automata can also be treated as transducers (that output exactly their input for the words accepted by automaton).

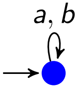
Finite transducers: definition

- Finite transducers are automata with output.
- Precisely, now there are two alphabets: Σ (input) and Γ (output).
- Edges have the form $\langle u : v \rangle$, $u \in \Sigma^*$, $v \in \Gamma^*$ and mean "replace u with v".
- Summarizing, finite transducers define not sets but relations between inputs and outputs.
- Automata can also be treated as transducers (that output exactly their input for the words accepted by automaton).
- Simplest transducer — identity relation (alphabet a, b):



Finite transducers: definition

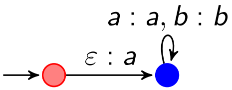
- Finite transducers are automata with output.
- Precisely, now there are two alphabets: Σ (input) and Γ (output).
- Edges have the form $\langle u : v \rangle$, $u \in \Sigma^*$, $v \in \Gamma^*$ and mean "replace u with v".
- Summarizing, finite transducers define not sets but relations between inputs and outputs.
- Automata can also be treated as transducers (that output exactly their input for the words accepted by automaton).
- Simplest transducer — identity relation (alphabet a, b):



- We will formally treat finite transductions as sets of word pairs.

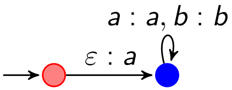
Finite transducers: examples

- Adds *a* to the beginning:

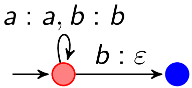


Finite transducers: examples

- Adds a to the beginning:

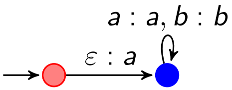


- Removes final b if it is present and rejects other words:

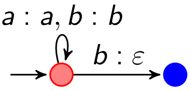


Finite transducers: examples

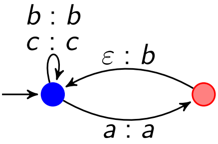
- Adds a to the beginning:



- Removes final b if it is present and rejects other words:

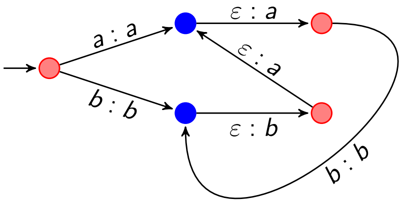


- Adds b after every a:



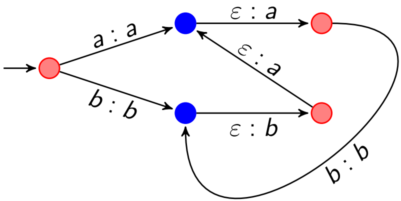
Finite transducers: examples

- Doubles each letter except for the last one:

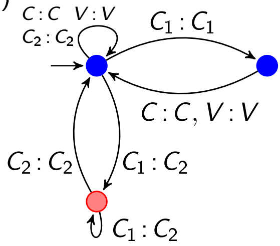


Finite transducers: examples

- Doubles each letter except for the last one:



- Retro-assimilates all C_1 to C_2 (a sequence of C_1-s preceding C_2 is substituted for C_2)



Properties of finite transducers

- Every finite transducer is equivalent to a transducer with labels of the form $a : \varepsilon$, $a \in \Sigma$ and $\varepsilon : b$, $b \in \Gamma$.

Sketch of the proof

- Edges of the form $a_1 \dots a_k : b_1 \dots b_r$ can be decomposed as sequence of edges $a_1 : \varepsilon, \dots, a_k : \varepsilon, \varepsilon : b_1, \dots, \varepsilon : b_r$.

Properties of finite transducers

- Every finite transducer is equivalent to a transducer with labels of the form $a : \varepsilon$, $a \in \Sigma$ and $\varepsilon : b$, $b \in \Gamma$.

Sketch of the proof

- Edges of the form $a_1 \dots a_k : b_1 \dots b_r$ can be decomposed as sequence of edges $a_1 : \varepsilon, \dots, a_k : \varepsilon, \varepsilon : b_1, \dots, \varepsilon : b_r$.
- Edges of the form $\varepsilon : \varepsilon$ are removed as in finite automata.

Properties of finite transducers

- Every finite transducer is equivalent to a transducer with labels of the form $a : \varepsilon$, $a \in \Sigma$ and $\varepsilon : b$, $b \in \Gamma$.

Sketch of the proof

- Edges of the form $a_1 \dots a_k : b_1 \dots b_r$ can be decomposed as sequence of edges $a_1 : \varepsilon, \dots, a_k : \varepsilon, \varepsilon : b_1, \dots, \varepsilon : b_r$.
- Edges of the form $\varepsilon : \varepsilon$ are removed as in finite automata.

- Finite transductions are closed under:
  - Concatenation.
  - Union.
  - Multiplicative iteration($\phi^* = \{u_1 \dots u_k, v_1 \dots v_k \mid \langle u_j, v_j \rangle \in \phi\}$).

Properties of finite transducers

- Every finite transducer is equivalent to a transducer with labels of the form $a : \varepsilon$, $a \in \Sigma$ and $\varepsilon : b$, $b \in \Gamma$.

Sketch of the proof

- Edges of the form $a_1 \dots a_k : b_1 \dots b_r$ can be decomposed as sequence of edges $a_1 : \varepsilon, \; \dots, \; a_k : \varepsilon, \varepsilon : b_1, \; \dots, \; \varepsilon : b_r$.
- Edges of the form $\varepsilon : \varepsilon$ are removed as in finite automata.

- Finite transductions are closed under:
  - Concatenation.
  - Union.
  - Multiplicative iteration($\phi^* = \{u_1 \dots u_k, v_1 \dots v_k \,|\, \langle u_j, v_j \rangle \in \phi\}$).
- Finite transduction domain is an automata language (just keep only input label in the transducer).

Properties of finite transducers

- Every finite transducer is equivalent to a transducer with labels of the form $a : \varepsilon$, $a \in \Sigma$ and $\varepsilon : b$, $b \in \Gamma$.

Sketch of the proof

- Edges of the form $a_1 \dots a_k : b_1 \dots b_r$ can be decomposed as sequence of edges $a_1 : \varepsilon, \dots, a_k : \varepsilon, \varepsilon : b_1, \dots, \varepsilon : b_r$.
- Edges of the form $\varepsilon : \varepsilon$ are removed as in finite automata.

- Finite transductions are closed under:
  - Concatenation.
  - Union.
  - Multiplicative iteration($\phi^* = \{u_1 \dots u_k, v_1 \dots v_k \mid \langle u_j, v_j \rangle \in \phi\}$).
- Finite transduction domain is an automata language (just keep only input label in the transducer).
- Finite transduction range is an automata language.

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).
- Finite transducers are closed under:
  - Reversion: $\phi^{-1} = \{\langle v, u\rangle \mid \langle u, v\rangle \in \phi\}$ (just replace all labels $x : y$ with $y : x$).

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).
- Finite transducers are closed under:
  - Reversion: $\phi^{-1} = \{\langle v, u\rangle \mid \langle u, v\rangle \in \phi\}$ (just replace all labels $x : y$ with $y : x$).
  - Composition: $\phi \circ \psi = \{\langle u, v\rangle \mid \exists w(\langle u, w\rangle \in \phi, \langle w, v\rangle \in \psi)\}$.

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).
- Finite transducers are closed under:
  - Reversion: $\phi^{-1} = \{\langle v, u\rangle \mid \langle u, v\rangle \in \phi\}$ (just replace all labels $x : y$ with $y : x$).
  - Composition: $\phi \circ \psi = \{\langle u, v\rangle \mid \exists w(\langle u, w\rangle \in \phi, \langle w, v\rangle \in \psi)\}$.
  - Priority union:

$$\phi \cup_p \psi = \begin{cases} \phi(x), & \text{if } \phi(x) \text{ is defined,} \\ \psi(x), & \text{otherwise.} \end{cases}$$

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).
- Finite transducers are closed under:
  - Reversion: $\phi^{-1} = \{\langle v, u\rangle \mid \langle u, v\rangle \in \phi\}$ (just replace all labels $x : y$ with $y : x$).
  - Composition: $\phi \circ \psi = \{\langle u, v\rangle \mid \exists w(\langle u, w\rangle \in \phi, \langle w, v\rangle \in \psi)\}$.
  - Priority union:

$$\phi \cup_p \psi = \begin{cases} \phi(x), & \text{if } \phi(x) \text{ is defined,} \\ \psi(x), & \text{otherwise.} \end{cases}$$

- Applications:
  - Reversion: switch between analysis/synthesis.

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).
- Finite transducers are closed under:
  - Reversion: $\phi^{-1} = \{\langle v, u\rangle \mid \langle u, v\rangle \in \phi\}$ (just replace all labels $x : y$ with $y : x$).
  - Composition: $\phi \circ \psi = \{\langle u, v\rangle \mid \exists w(\langle u, w\rangle \in \phi, \langle w, v\rangle \in \psi)\}$.
  - Priority union:
  $$\phi \cup_p \psi = \begin{cases} \phi(x), & \text{if } \phi(x) \text{ is defined,} \\ \psi(x), & \text{otherwise.} \end{cases}$$
- Applications:
  - Reversion: switch between analysis/synthesis.
  - Composition: successive application of operations.

Properties of finite transducers

- Restriction of finite transduction to automata language can be described by finite transducer (trace both the state of the transducer and the state in the automata for the restriction language).
- Finite transducers are closed under:
  - Reversion: $\phi^{-1} = \{\langle v, u\rangle \mid \langle u, v\rangle \in \phi\}$ (just replace all labels $x : y$ with $y : x$).
  - Composition: $\phi \circ \psi = \{\langle u, v\rangle \mid \exists w(\langle u, w\rangle \in \phi, \langle w, v\rangle \in \psi)\}$.
  - Priority union:
$$\phi \cup_p \psi = \begin{cases} \phi(x), & \text{if } \phi(x) \text{ is defined,} \\ \psi(x), & \text{otherwise.} \end{cases}$$
- Applications:
  - Reversion: switch between analysis/synthesis.
  - Composition: successive application of operations.
  - Priority union: separate model for exceptions.

Finite transducers: linguistic examples

English plural

Describe a transducer that transforms a singular form of English noun to plural.

Finite transducers: linguistic examples

English plural

Describe a transducer that transforms a singular form of English noun to plural.

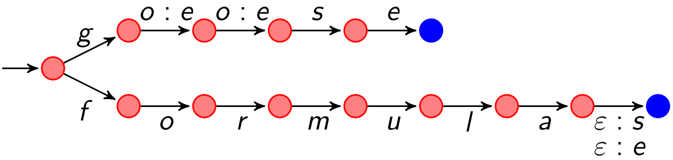
- *torch* $\leftrightarrows$ *torches*
- *monarch*+N+Pl $\leftrightarrows$ *monarchs*
- *ally* $\leftrightarrows$ *allies*
- *play* $\leftrightarrows$ *plays*
- *goose* $\leftrightarrows$ *geese*
- *formula* $\leftrightarrows$ *formulas/formulae*

Finite transducers: linguistic examples

English plural

Describe a transducer that transforms a singular form of English noun to plural.

- Create a separate transducer T_{exc} for exceptions:

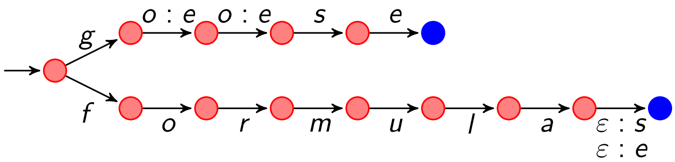


Finite transducers: linguistic examples

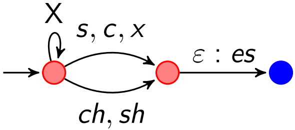
English plural

Describe a transducer that transforms a singular form of English noun to plural.

- Create a separate transducer T_{exc} for exceptions:



- Transducer T_{sib} that adds -*es* after word-final sibilant (X denotes any character):

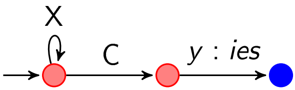


Finite transducers: linguistic examples

English plural

Describe a transducer that transforms a singular form of English noun to plural.

- Transducer T_{exc} for exceptions.
- Transducer T_{sib} that adds *es* after word-final sibilant.
- Transducer T_{Cy} that replaces final y with *-ies* after consonant.

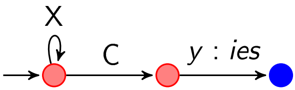


Finite transducers: linguistic examples

English plural

Describe a transducer that transforms a singular form of English noun to plural.

- Transducer T_{exc} for exceptions.
- Transducer T_{sib} that adds *es* after word-final sibilant.
- Transducer T_{Cy} that replaces final *y* with *-ies* after consonant.



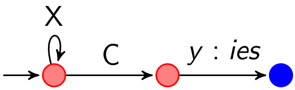
- T_s — transducer that simply appends *s*.
- $T_{exc,sib}$ — transducer that appends *s* to words ending with *-arch* and rejects other words (for *monarchs*, *tetrarchs*, …).

Finite transducers: linguistic examples

English plural

Describe a transducer that transforms a singular form of English noun to plural.

- Transducer T_{exc} for exceptions.
- Transducer T_{sib} that adds *es* after word-final sibilant.
- Transducer T_{Cy} that replaces final *y* with *-ies* after consonant.



- T_s — transducer that simply appends *s*.
- $T_{exc,sib}$ — transducer that appends *s* to words ending with *-arch* and rejects other words (for *monarchs*, *tetrarchs*, ...).
- Final solution:

$$T_{exc} \cup_p T_{exc,sib} \cup_p T_{sib} \cup_p T_{Cy} \cup_p T_s$$

Context replacement

- The most common type of transduction — context replacement:

$$X \to Y \,||\, U\_V$$

"Replace X by Y if left context of X is U and right is V."

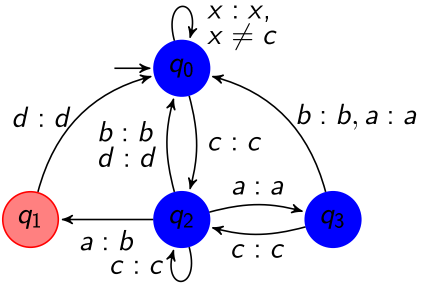
Context replacement

- The most common type of transduction — context replacement:

$$X \to Y \,||\, U\_V$$

"Replace X by Y if left context of X is U and right is V."

- In the simplest case X, Y, U, V are letters.
- Transducer for $a \to b||c\_d$:

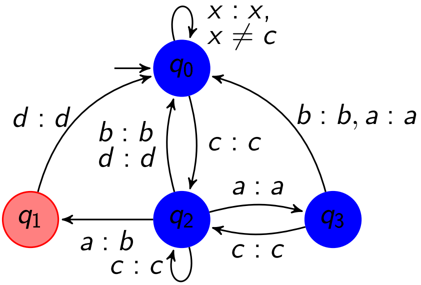


Context replacement

- The most common type of transduction — context replacement:

$$X \to Y \,||\, U\_V$$

  "Replace X by Y if left context of X is U and right is V."
- In the simplest case X, Y, U, V are letters.
- Transducer for $a \to b||c\_d$:



- X, Y, U, V can be arbitrary regular expressions.

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es*, *-ies*, *-s*.

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es*, *-ies*, *-s*.
- There are only ending *-s* and phonotactic alterations. How to model this?

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es*, *-ies*, *-s*.
- There are only ending *-s* and phonotactic alterations. How to model this?
- Apply phonotactic rules in cascade.
- Rules are formulated with context replacements:
  - T_s: append !*s* to the end of the word (! is the placeholder) $\varepsilon \rightarrow !s \,||\, \_\,\$$ (\$ marks the end of the word).

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es*, *-ies*, *-s*.
- There are only ending *-s* and phonotactic alterations. How to model this?
- Apply phonotactic rules in cascade.
- Rules are formulated with context replacements:
  - T_s: append !*s* to the end of the word (! is the placeholder) $\varepsilon \to !s \,||\, \_\$$ ($ marks the end of the word).
  - T_{sib}: add *e* before ! and after sibilant $\varepsilon \to e \,||\, (s|z|x|sh|ch)\_!$.

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es*, *-ies*, *-s*.
- There are only ending *-s* and phonotactic alterations. How to model this?
- Apply phonotactic rules in cascade.
- Rules are formulated with context replacements:
  - T_s: append !*s* to the end of the word (! is the placeholder) $\varepsilon \rightarrow !s \,||\, \_\$$ ($ marks the end of the word).
  - T_{sib}: add *e* before ! and after sibilant $\varepsilon \rightarrow e \,||\, (s|z|x|sh|ch)\_!$.
  - T_y: replace *y* by *ie* before the marker $y \rightarrow ie \,||\, \_!$.

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es, -ies, -s*.
- There are only ending *-s* and phonotactic alterations. How to model this?
- Apply phonotactic rules in cascade.
- Rules are formulated with context replacements:
  - T_s: append !*s* to the end of the word (! is the placeholder) $\varepsilon \rightarrow !s \,||\, \_\$$ ($ marks the end of the word).
  - T_{sib}: add *e* before ! and after sibilant $\varepsilon \rightarrow e \,||\, (s|z|x|sh|ch)\_!$.
  - T_y: replace *y* by *ie* before the marker $y \rightarrow ie \,||\, \_!$.
  - $T_{exc,sib}$: do nothing with words ending by *arch* following non-empty prefix (actually an automaton).
  - T_c: remove the placeholder $! \rightarrow \varepsilon$.

English plural revisited

- Our model for English plural is inadequate linguistically.
- Actually, there are no separate endings *-es*, *-ies*, *-s*.
- There are only ending *-s* and phonotactic alterations. How to model this?
- Apply phonotactic rules in cascade.
- Rules are formulated with context replacements:
  - T_s: append !*s* to the end of the word (! is the placeholder) $\varepsilon \to !s \,||\, \_\$$ ($ marks the end of the word).
  - T_{sib}: add *e* before ! and after sibilant $\varepsilon \to e \,||\, (s|z|x|sh|ch)\_!$.
  - T_y: replace *y* by *ie* before the marker $y \to ie \,||\, \_!$.
  - $T_{exc,sib}$: do nothing with words ending by *arch* following non-empty prefix (actually an automaton).
  - T_c: remove the placeholder $! \to \varepsilon$.
- Final combination via composition:

$$T_{exc} \cup_p (T_s \circ (T_{exc,sib} \cup_p T_{sib}) \circ T_y \circ T_c)$$

Turkish passive

Turkish passive

Construct a finite transducer, transforming Turkish verb infinitive to its passive infinitive.

- Passive is formed by a suffix inserted before final *-mek*/*-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u*, *o*; *i* after *e*, *i*; *ü* after *ü*, *ö*.
- T_{mark}: insert a marker ! before *-mak*/*-mek*: $\varepsilon \rightarrow ! \mid\mid \_ m(a|e)k\$$.

Turkish passive

Turkish passive

Construct a finite transducer, transforming Turkish verb infinitive to its passive infinitive.

- Passive is formed by a suffix inserted before final *-mek/-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u*, *o*; *i* after *e*, *i*; *ü* after *ü*, *ö*.

- T_{mark}: insert a marker ! before *-mak/-mek*: $\varepsilon \rightarrow ! \mid\mid \_\, m(a|e)k\$$.
- Replace the marker by an appropriate suffix:

Turkish passive

Turkish passive

Construct a finite transducer, transforming Turkish verb infinitive to its passive infinitive.

- Passive is formed by a suffix inserted before final *-mek/-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u*, *o*; *i* after *e*, *i*; *ü* after *ü*, *ö*.

- T_{mark}: insert a marker ! before *-mak/-mek*: $\varepsilon \rightarrow ! \,||\, \_\, m(a|e)k\$$.
- Replace the marker by an appropriate suffix:
  - *-n* after vowel (T_V): $! \rightarrow n \,||\, V\,\_\,\$$,
  - *-In* after *l* (T_l): $! \rightarrow \mathrm{I}n \,||\, l\,\_\,\$$,
  - *-Il* by default (T_{def}): $! \rightarrow \mathrm{I}l \,||\, \_$,

Turkish passive

Turkish passive

Construct a finite transducer, transforming Turkish verb infinitive to its passive infinitive.

- Passive is formed by a suffix inserted before final *-mek/-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u*, *o*; *i* after *e*, *i*; *ü* after *ü*, *ö*.

- T_{mark}: insert a marker ! before *-mak/-mek*: $\varepsilon \rightarrow ! \mid\mid \_\, m(a|e)k\$$.
- Replace the marker by an appropriate suffix:
  - *-n* after vowel (T_V): $! \rightarrow n \mid\mid V \_\, \$$,
  - *-In* after *l* (T_l): $! \rightarrow \mathrm{I}n \mid\mid l \_\, \$$,
  - *-Il* by default (T_{def}): $! \rightarrow \mathrm{I}l \mid\mid \_$,
  - Combine them all $T_{suf} = T_V \circ T_l \circ T_{def}$.

Turkish passive

Turkish passive infinitive

- Passive is formed by a suffix inserted before final *-mek/-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u*, *o*; *i* after *e*, *i*; *ü* after *ü*, *ö*.

- T_{mark} inserts a marker ! before *-mak/-mek*.
- T_{suf} substitutes the marker for an appropriate suffix.
- T_{fill} fills the placeholder: $T_{fill} = T_{\imath} \circ T_u \circ T_i \circ T_U$, where
  - $T_{\imath}$ checks the condition for ß: $A \to \imath \,||\, (a|\imath)C^* \_$.
  - T_u for u: $A \to u \,||\, (u|o)C^* \_$.
  - T_i for i: $A \to i \,||\, (e|i)C^* \_$.
  - T_U for ü: $A \to ü \,||\, (ü|ö)C^* \_$.

Turkish passive

Turkish passive infinitive

- Passive is formed by a suffix inserted before final *-mek/-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u*, *o*; *i* after *e*, *i*; *ü* after *ü*, *ö*.

- T_{mark} inserts a marker ! before *-mak/-mek*.
- T_{suf} substitutes the marker for an appropriate suffix.
- T_{fill} fills the placeholder: $T_{fill} = T_{ı} \circ T_u \circ T_i \circ T_U$, where
  - $T_{ı}$ checks the condition for ß: $A \to ı \,||\, (a|ı)C^* \_$.
  - T_u for u: $A \to u \,||\, (u|o)C^* \_$.
  - T_i for i: $A \to i \,||\, (e|i)C^* \_$.
  - T_U for ü: $A \to ü \,||\, (ü|ö)C^* \_$.
- Final answer:

$$T_{mark} \circ T_{suf} \circ T_{fill}$$

Nonconcatenative morphology: Yawelmani

stem	gerund	durative
caw “to cry”	caw-inay	cawaa-ʔaa-n
cuum “to destroy”	cum-inay	cumuu-ʔaa-n
hoyoo “to name”	hoy-inay	hoyoo-ʔaa-n
diiyl “to guard”	diyl-inay	diyiil-ʔaa-n
ʔilk “to sing”	ʔilk-inay	ʔiliik-ʔaa-n
hiwiit “to walk”	hiwt-inay	hiwiit-ʔaa-n

Verb forms in Yawelmani (Amerind family)

Nonconcatenative morphology: Yawelmani

stem	gerund	durative
caw "to cry"	caw-inay	cawaa-ʔaa-n
cuum "to destroy"	cum-inay	cumuu-ʔaa-n
hoyoo "to name"	hoy-inay	hoyoo-ʔaa-n
diiyl "to guard"	diyl-inay	diyiil-ʔaa-n
ʔilk "to sing"	ʔilk-inay	ʔiliik-ʔaa-n
hiwiit "to walk"	hiwt-inay	hiwiit-ʔaa-n

Verb forms in Yawelmani (Amerind family)

If the stem was $\alpha_1 V(V)\alpha_2(V)(V)\alpha_3$ where $\alpha_1,\alpha_2 \in C$, $\alpha_3 \in \{C,\varepsilon\}$:

- gerund stem is $\alpha_1 V \alpha_2 \alpha_3$,

Nonconcatenative morphology: Yawelmani

stem	gerund	durative
caw "to cry"	caw-inay	cawaa-ʔaa-n
cuum "to destroy"	cum-inay	cumuu-ʔaa-n
hoyoo "to name"	hoy-inay	hoyoo-ʔaa-n
diiyl "to guard"	diyl-inay	diyiil-ʔaa-n
ʔilk "to sing"	ʔilk-inay	ʔiliik-ʔaa-n
hiwiit "to walk"	hiwt-inay	hiwiit-ʔaa-n

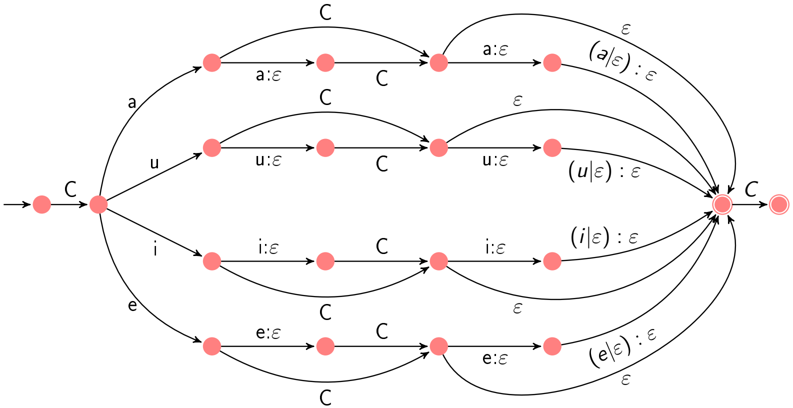
Verb forms in Yawelmani (Amerind family)

If the stem was $\alpha_1 V(V)\alpha_2(V)(V)\alpha_3$ where $\alpha_1, \alpha_2 \in C$, $\alpha_3 \in \{C, \varepsilon\}$:

- gerund stem is $\alpha_1 V \alpha_2 \alpha_3$,
- and durative stem is $\alpha_1 V \alpha_2 VV \alpha_3$.

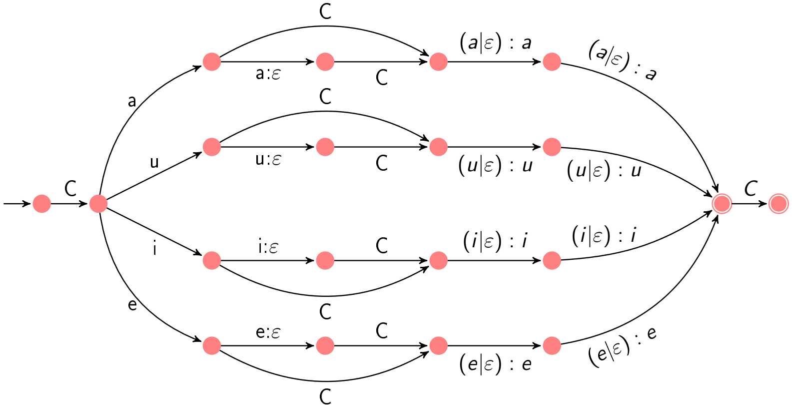
Nonconcatenative morphology: Yawelmani gerund

- Gerund stem:



Nonconcatenative morphology: Yawelmani durative

- Durative stem:



FOMA: a finite-state compiler

- FOMA — a program for compiling finite state transducers.
- Designed by Mans Hulden in 2009–2015, last official version 0.9.18 — June 12th, 2015.

FOMA: a finite-state compiler

- FOMA — a program for compiling finite state transducers.
- Designed by Mans Hulden in 2009–2015, last official version 0.9.18 — June 12th, 2015.
- Release version: `https://code.google.com/archive/p/foma/`,
- development version `https://github.com/mhulden/foma/`.
- Open source program, written in C++, has Python binding (only for basic functionality).

FOMA: a finite-state compiler

- FOMA — a program for compiling finite state transducers.
- Designed by Mans Hulden in 2009–2015, last official version 0.9.18 — June 12th, 2015.
- Release version: `https://code.google.com/archive/p/foma/`,
- development version `https://github.com/mhulden/foma/`.
- Open source program, written in C++, has Python binding (only for basic functionality).
- Main usage: compile context rules to finite-state transducers.

FOMA: a finite-state compiler

- FOMA — a program for compiling finite state transducers.
- Designed by Mans Hulden in 2009–2015, last official version 0.9.18 — June 12th, 2015.
- Release version: `https://code.google.com/archive/p/foma/`,
- development version `https://github.com/mhulden/foma/`.
- Open source program, written in C++, has Python binding (only for basic functionality).
- Main usage: compile context rules to finite-state transducers.
- Also can be used for processing finite automata.

FOMA: a finite-state compiler

- FOMA — a program for compiling finite state transducers.
- Designed by Mans Hulden in 2009–2015, last official version 0.9.18 — June 12th, 2015.
- Release version: `https://code.google.com/archive/p/foma/`,
- development version `https://github.com/mhulden/foma/`.
- Open source program, written in C++, has Python binding (only for basic functionality).
- Main usage: compile context rules to finite-state transducers.
- Also can be used for processing finite automata.
- Flookup utility permits to use foma transducers as binary programs.

FOMA: basic usage

Basic usage: defines a context rule.

```
foma[0]: ##replace all a by b
foma[0]: regex a -> b || _ ;
374 bytes. 1 state, 3 arcs, Cyclic.
foma[1]: net
Sigma: ? @ a b
Size: 2.
Net: E20E6CF
Flags: deterministic pruned minimized epsilon_free
Arity: 2
Sfs0:   <a:b> -> fs0, b -> fs0, @ -> fs0.
foma[1]:
```

FOMA: basic usage

Basic usage: defines a context rule and applies it up and down

```
foma[0]: ##replace all a by b
foma[0]: regex a -> b || _ ;
374 bytes. 1 state, 3 arcs, Cyclic.
foma[1]: down
apply down> bcaba
bcbbb
apply down> bbb
bbb
apply down>
foma[1]: up
apply up> aba
???
apply up> cbdb
cada
cadb
cbda
cbdb
apply up> cdc
cdc
apply up>
foma[1]:
```

FOMA: basic usage

Forming plural for *y*-final nouns:

```
foma[0]: ## filter y-ending words
foma[0]: define yFinal ?* y ;
redefined yFinal: 321 bytes. 2 states, 4 arcs, Cyclic.
foma[0]: ## Vowel+y
foma[0]: define Vowel [ a | e | i | o | u ];
redefined Vowel: 413 bytes. 2 states, 5 arcs, 5 paths.
foma[0]: define yVowel [..] -> s || Vowel y _ .#. ; ## simply append s after Vowel+y
redefined yVowel: 872 bytes. 4 states, 25 arcs, Cyclic.
foma[0]: define yVowel [..] -> s || [ .#. | Vowel ] y _ .#. ; ## simply append s after Vowel+y
redefined yVowel: 872 bytes. 4 states, 25 arcs, Cyclic.
foma[0]: define yCons y -> i e s || \Vowel _ .#. ;
redefined yCons: 920 bytes. 6 states, 28 arcs, Cyclic.
foma[0]: ## combine the variants for vowels and consonants
foma[0]: define yChange yFinal .o. yVowel .o. yCons ;
redefined yChange: 936 bytes. 6 states, 29 arcs, Cyclic.
foma[0]: push yChange
936 bytes. 6 states, 29 arcs, Cyclic.
foma[1]: down
apply down> valley
valleys
apply down> ally
allies
apply down> y
ys
apply down> tray
trays
apply down> granny
grannies
```

FOMA: operations with automata

Operation	Notation
Concatenation of X, Y	XY
Intersection of X, Y	$X\&Y$
Union of X, Y	$X\|Y$
Difference of X, Y	$X - Y$
Iteration of X	X^*
Positive iteration of X	X^+
Negation of X	$\backslash X$
Context restriction	
(X appears only in context Y_Z)	$X \rightarrow Y\_Z$

Operations with automata in FOMA

FOMA: operations with automata

Operation	Notation
Context replacement	$X \rightarrow Y \| \| U\_V$
(Change X to Y in context U_V)	
Composition of X, Y	$X.\mathrm{o}.Y$
Priority union of X, Y	$X.\mathrm{P}.Y$
Cartesian product of X, Y	$X : Y$
Domain (upper part) of X	$X.u$
Range (lower part) of X	$X.l$
Inverse transduction of X	$X.i$
Parallel contexts	$X \rightarrow Y \| \| U1\_V1, U2\_V2$
Parallel replacement	$X1 \rightarrow Y1, X2 \rightarrow Y2 \| \| U\_V$

Operations with transducers in FOMA

FOMA: applying transducers

Operation	Notation
Define a transducer variable	**define** ⟨var_name⟩ ⟨expression⟩
Push defined transducer to the stack	**push** ⟨var_name⟩
Push expression to the stack	**regex** ⟨expression⟩
Apply topmost transducer in stack (downwards)	**down** (apply down)
Apply topmost transducer "reversely" (upwards)	**up** (apply up)
Clear stack	**clear**
Read lexicon file and save to variable	**read** ⟨filename⟩ **define** ⟨var_name⟩
save transducer(s) to binary file	**save stack** ⟨filename⟩

Application of transducers in FOMA

FOMA: external usage and documentation

- Documentation page (concise but useful): `https://code.google.com/archive/p/foma/wikis`.
- Description of available operations: `https://code.google.com/archive/p/foma/wikis/RegularExpressionReference.wiki`.

FOMA: external usage and documentation

- Documentation page (concise but useful): `https://code.google.com/archive/p/foma/wikis`.
- Description of available operations: `https://code.google.com/archive/p/foma/wikis/RegularExpressionReference.wiki`.
- Transducers saved in binary with **save stack** command can be applied from command line by flookup utility.

FOMA: external usage and documentation

- Documentation page (concise but useful): `https://code.google.com/archive/p/foma/wikis`.
- Description of available operations: `https://code.google.com/archive/p/foma/wikis/RegularExpressionReference.wiki`.
- Transducers saved in binary with **save stack** command can be applied from command line by flookup utility.
- Main usage:

  flookup -i -x -w ⟨binary_file⟩ < ⟨input_file⟩ (> ⟨output_file⟩)

- Applies the transducer in binary file to each string in ⟨input_file⟩ and prints the result (or redirects it to ⟨output_file⟩).

FOMA: external usage and documentation

- Documentation page (concise but useful): `https://code.google.com/archive/p/foma/wikis`.
- Description of available operations: `https://code.google.com/archive/p/foma/wikis/RegularExpressionReference.wiki`.
- Transducers saved in binary with **save stack** command can be applied from command line by flookup utility.
- Main usage:

  flookup -i -x -w ⟨binary_file⟩ < ⟨input_file⟩ (> ⟨output_file⟩)

- Applies the transducer in binary file to each string in ⟨input_file⟩ and prints the result (or redirects it to ⟨output_file⟩).
- If -x key is omitted, input word is also printed on the same string as corresponding output.

FOMA: external usage and documentation

- Documentation page (concise but useful): `https://code.google.com/archive/p/foma/wikis`.
- Description of available operations: `https://code.google.com/archive/p/foma/wikis/RegularExpressionReference.wiki`.
- Transducers saved in binary with **save stack** command can be applied from command line by flookup utility.
- Main usage:

  flookup -i -x -w $\langle$binary_file$\rangle$ < $\langle$input_file$\rangle$ (> $\langle$output_file$\rangle$)

- Applies the transducer in binary file to each string in $\langle$input_file$\rangle$ and prints the result (or redirects it to $\langle$output_file$\rangle$).
- If -x key is omitted, input word is also printed on the same string as corresponding output.
- More documentation: `https://code.google.com/archive/p/foma/wikis/FlookupDocumentation.wiki`.

English plural

```
### english.foma ###
read lexc irregular.lexc
define IrregularNounPlural;

define Vowel [ a | i | e | o | u | y ];
define Consonant [ b | c | d | f | g | h | j | k | l | m | n | p | q | r | s | t | v | w | x | z ];
define Letter [Vowel | Consonant];
define Word [ Letter ]+;
define NounMark "+N";
define NounNumber "+Sg" | "+Pl";
define Noun Word NounMark NounNumber;

define NounAffixation "+N" "+Sg" -> "" || _ .#., "+N" "+Pl" -> "!" s || _ .#.;
define Sibilant [ x | s | z | c h | s h ];
define sibException [ Letter ]+ a r c h "!" s ;
define eInsertion [..] -> e || Sibilant _ "!" s .#.;
define checkSibilant [ sibException .P. eInsertion ];
define yReplacement y -> i e || Consonant _ "!" s .#.;
define Cleanup "!" -> "" || _ ;
define RegularNoun [ NounAffixation .o. yReplacement .o. checkSibilant .o. Cleanup ] ;
define Grammar Noun .o. [ IrregularNounPlural .P. RegularNoun ];
push Grammar
```

Turkish passive

Turkish passive

Construct a finite transducer, transforming Turkish verb infinitive to its passive infinitive.

- Passive is formed by a suffix inserted before final *-mek/-mak*.
- Passive suffix: *-n* after vowel, *-In* after *l* and *-Il* otherwise.
- Placeholder I: *ı* after a, *ı*; *u* after *u, o*; *i* after *e, i*; *ü* after *ü, ö*.

```
# symbol classes
define HardStraightVowel a | I ;
define HardRoundVowel o | u ;
define SoftStraightVowel e | i ;
define SoftRoundVowel O | U ;
define HardVowel HardStraightVowel | HardRoundVowel ;
define SoftVowel SoftStraightVowel | SoftRoundVowel ;
define Vowel HardVowel | SoftVowel ;
define Consonant b | c | C | d | f | g | G | h | j | k | l | m | n | p | r | s | S | t | v | y | z ;
define Letter Consonant | Vowel ;
```

Turkish passive

```
# contexts for stem
define LastVowelHard HardVowel Consonant* ;
define LastVowelSoft SoftVowel Consonant* ;
define LastVowelHardRound HardRoundVowel Consonant* ;
define LastVowelHardStraight HardStraightVowel Consonant* ;
define LastVowelSoftRound SoftRoundVowel Consonant* ;
define LastVowelSoftStraight SoftStraightVowel Consonant* ;
# infinitive vowel check
define Stem Letter* Vowel Letter* ;
define InfinitiveSuffixInsertion [..] -> E || _ .#. ;
define InfinitiveSuffix [ E -> m a k || HardVowel Consonant* _ .#. ] .o. [ E -> m e k || SoftVowel
    Consonant* _ .#. ] ;
define SuffixTransform Stem .o. InfinitiveSuffixInsertion .o. InfinitiveSuffix ;
define Infinitive SuffixTransform.l ;
define Input Infinitive "+Pass";

# suffix insertion
define MarkerInsertion [..] -> "!" || _ m [ a | e ] k "+Pass" .#. ;
define MarkerAfterVowel "!" -> l || Vowel _ ;
define MarkerAfterL "!" -> A n || l _ ;
define MarkerAfterAll "!" -> A l || _ ;
define MarkerReplacement MarkerAfterVowel .o. MarkerAfterL .o. MarkerAfterAll ;

# combining all
define VowelFill [ A -> I || LastVowelHardStraight _ ] .o. [ A -> e || LastVowelSoftStraight _ ] .o. [
    A -> u || LastVowelHardRound _ ].o. [ A -> U || LastVowelSoftRound _ ] ;
define Cleanup "+Pass" -> "" ;
define Grammar Input .o. MarkerInsertion .o. MarkerReplacement .o. VowelFill ;

push Grammar
```

Yawelmani verb forms

stem	gerund	durative
caw "to cry"	caw-inay	cawaa-ʔaa-n
cuum "to destroy"	cum-inay	cumuu-ʔaa-n
hoyoo "to name"	hoy-inay	hoyoo-ʔaa-n
diiyl "to guard"	diyl-inay	diyiil-ʔaa-n
ʔilk "to sing"	ʔilk-inay	ʔiliik-ʔaa-n
hiwiit "to walk"	hiwt-inay	hiwiit-ʔaa-n

Verb forms in Yawelmani (Amerind family)

If the stem was $\alpha_1 V(V)\alpha_2(V)(V)\alpha_3$, where $\alpha_1, \alpha_2 \in C$, $\alpha_3 \in \{C, \varepsilon\}$:

- gerund stem is $\alpha_1 V \alpha_2 \alpha_3$,
- and durative stem is $\alpha_1 V \alpha_2 V V \alpha_3$.

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?
- First step: express alternations as context rules.

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?
- First step: express alternations as context rules.
- Gerund: remove all vowels except for the leftmost.
- Left context for such vowels: C^*VC^*.

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?
- First step: express alternations as context rules.
- Gerund: remove all vowels except for the leftmost.
- Left context for such vowels: C^*VC^*.
- Durative:
  - Remove the second vowel in the first syllable (left context$\hat{}C^+V$).

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?
- First step: express alternations as context rules.
- Gerund: remove all vowels except for the leftmost.
- Left context for such vowels: C^*VC^*.
- Durative:
    - Remove the second vowel in the first syllable (left context $\hat{}C^+V$).
    - Insert to the second syllable twice the same vowel as in the first.

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?
- First step: express alternations as context rules.
- Gerund: remove all vowels except for the leftmost.
- Left context for such vowels: C^*VC^*.
- Durative:
  - Remove the second vowel in the first syllable (left context $\hat{\ }C^+V$).
  - Insert to the second syllable twice the same vowel as in the first.
- Checking the equality of vowels for durative:
  - Try to insert all pairs of identical vowels (*aa*, *ee*, *oo*, *uu*).

Yawelmani verb forms

- We constructed the transducer for Yawelmani verbs manually.
- Can we do it with FOMA?
- First step: express alternations as context rules.
- Gerund: remove all vowels except for the leftmost.
- Left context for such vowels: C^*VC^*.
- Durative:
  - Remove the second vowel in the first syllable (left context $\hat{}C^+V$).
  - Insert to the second syllable twice the same vowel as in the first.
- Checking the equality of vowels for durative:
  - Try to insert all pairs of identical vowels (*aa*, *ee*, *oo*, *uu*).
  - Check vowel harmony between syllables by enumerating all variants of the type $C^+xC^+xC^*$ where x is an arbitrary vowel.

Yawelmani verb forms

```
### youlumne.foma ###
define Vowel [a | i | o | u];
define Consonant [ c | w | m | h | y | d | l | g | k | t ];
define Letter [Consonant | Vowel];
define Stem Consonant Vowel (Vowel) Consonant (Vowel) (Vowel) (Consonant) ;
define VerbMark "+V";
define Mood "+Ger" "+Dur";
define Mark [ VerbMark Mood ];
# vowel harmony
define VowelPattern [ [Consonant | a]+ | [Consonant | i]+ | [Consonant | o]+ | [Consonant | u]+ ];
define Word Stem & VowelPattern;
# left context for not a leftmost vowel
define LeftContext1 Consonant Vowel [ Letter ]* ;
define VowelRemoval Vowel -> [] || LeftContext1 _ ;
# left context for second syllable vowels
define LeftContext2 Consonant Vowel Consonant+ ;
# durative vowel insertion
define DurativeVowelInsertion [..] -> [ a a | i i | o o | u u ] || LeftContext2 _ (Consonant) .#. ;
define GerundSuffixInsertion ["+V" "+Ger"] : [ i n a y ] ;
define DurativeSuffixInsertion ["+V" "+Dur"] : [ g a a n ] ;
define GerundStem Word .o. VowelRemoval ;
# check that word possesses vowel harmony after vowel insertion
define DurativeStem GerundStem .o. DurativeVowelInsertion .o. VowelPattern ;
define Gerund [ GerundStem GerundSuffixInsertion ] ;
define Durative [ DurativeStem DurativeSuffixInsertion ] ;
define Grammar [ Gerund | Durative ] ;
```