Implementing morphology and phonology

- We begin with a simple problem: a lexicon of some natural language is given as a list of words. Suggest a data structure that will provide insertion and retrieval of data. As a first solution, we are looking for time efficiency rather than space efficiency.
- The solution: *trie* (word tree).
- Access time: $O(|w|)$. Space requirement: $O(\sum_w |w|)$.
- A trie can be augmented to store also a morphological dictionary specifying concatenative affixes, especially suffixes. In this case it is better to turn the tree into a graph.
- The obtained model is that of *finite-state automata*.

Finite-state technology

- Finite-state automata are not only a good model for representing the lexicon, they are also perfectly adequate for representing dictionaries (lexicons+additional information), describing morphological processes that involve concatenation etc.
- A natural extension of finite-state automata – finite-state transducers – is a perfect model for most processes known in morphology and phonology, including non-segmental ones.

Formal language theory – definitions

- Formal languages are defined with respect to a given *alphabet*, which is a finite set of symbols, each of which is called a *letter*.
- A finite sequence of letters is called a *string*.

Example: Strings

Let $\Sigma = \{0, 1\}$ be an alphabet. Then all binary numbers are strings over Σ. If $\Sigma = \{a, b, c, d, \ldots, y, z\}$ is an alphabet then *cat*, *incredulous* and *supercalifragilisticexpialidocious* are strings, as are *tac*, *qqq* and *kjshdflkwjehr*.

Formal language theory – definitions

- The *length* of a string w, denoted $|w|$, is the number of letters in w.
- The unique string of length 0 is called the *empty string* and is denoted ϵ.
- If $w_1 = \langle x_1, \ldots, x_n \rangle$ and $w_2 = \langle y_1, \ldots, y_m \rangle$, the *concatenation* of w_1 and w_2, denoted $w_1 \cdot w_2$, is the string $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. $|w_1 \cdot w_2| = |w_1| + |w_2|$.
- For every string w, $w \cdot \epsilon = \epsilon \cdot w = w$.

Formal language theory – definitions

Example: Concatenation

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet.
Then $master \cdot mind = mastermind$, $mind \cdot master = mindmaster$ and $master \cdot master = mastermaster$.
Similarly, $learn \cdot s = learns$, $learn \cdot ed = learned$ and $learn \cdot ing = learning$.

Formal language theory – definitions

- An *exponent* operator over strings is defined in the following way:
  - For every string w, $w^0 = \epsilon$.
  - For $n > 0$, $w^n = w^{n-1} \cdot w$.

Example: Exponent

If $w = go$, then $w^0 = \epsilon$, $w^1 = w = go$, $w^2 = w^1 \cdot w = w \cdot w = gogo$, $w^3 = gogogo$ and so on.

Formal language theory – definitions

- The *reversal* of a string w is denoted w^R and is obtained by writing w in the reverse order. Thus, if $w = \langle x_1, x_2, \ldots, x_n \rangle$, $w^R = \langle x_n, x_{n-1}, \ldots, x_1 \rangle$.
- Given a string w, a *substring* of w is a sequence formed by taking contiguous symbols of w in the order in which they occur in w. If $w = \langle x_1, \ldots, x_n \rangle$ then for any i, j such that $1 \leq i \leq j \leq n$, $\langle x_i, \ldots x_j \rangle$ is a substring of w.
- Two special cases of substrings are *prefix* and *suffix*: if $w = w_l \cdot w_c \cdot w_r$ then w_l is a prefix of w and w_r is a suffix of w.

Formal language theory – definitions

Example: Substrings

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet and $w =$ *indistinguishable* a string over Σ. Then ϵ, *in*, *indis*, *indistinguish* and *indistinguishable* are prefixes of w, while ϵ, *e*, *able*, *distinguishable* and *indistinguishable* are suffixes of w. Substrings that are neither prefixes nor suffixes include *distinguish*, *gui* and *is*.

Formal language theory – definitions

- Given an alphabet Σ, the set of all strings over Σ is denoted by Σ^*.
- A *formal language* over an alphabet Σ is a subset of Σ^*.

Formal language theory – definitions

Example: Languages

Let $\Sigma = \{a, b, c, \ldots, y, z\}$. The following are formal languages:

- Σ^*;
- the set of strings consisting of consonants only;
- the set of strings consisting of vowels only;
- the set of strings each of which contains at least one vowel and at least one consonant;
- the set of palindromes;

Formal language theory – definitions

Example: Languages

Let $\Sigma = \{a, b, c, \ldots, y, z\}$. The following are formal languages:

- the set of strings whose length is less than 17 letters;
- the set of single-letter strings ($= \Sigma$);
- the set $\{i, you, he, she, it, we, they\}$;
- the set of words occurring in Joyce's Ulysses;
- the empty set;

Note that the first five languages are infinite while the last five are finite.

Formal language theory – definitions

- The string operations can be lifted to languages.
- If L is a language then the *reversal* of L, denoted L^R, is the language $\{w \mid w^R \in L\}$.
- If L_1 and L_2 are languages, then $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$.

Example: Language operations

Let $L_1 = \{i, you, he, she, it, we, they\}$, $L_2 = \{smile, sleep\}$.
Then ${L_1}^R = \{i, uoy, eh, ehs, ti, ew, yeht\}$ and $L_1 \cdot L_2 = \{ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep\}$.

Formal language theory – definitions

- If L is a language then $L^0 = \{\epsilon\}$.
- Then, for $i > 0$, $L^i = L \cdot L^{i-1}$.

Example: Language exponentiation

Let L be the set of words $\{bau, haus, hof, frau\}$. Then $L^0 = \{\epsilon\}$, $L^1 = L$ and $L^2 = \{baubau, bauhaus, bauhof, baufrau, hausbau, haushaus, haushof, hausfrau, hofbau, hofhaus, hofhof, hoffrau, fraubau, frauhaus, frauhof, fraufrau\}$.

Formal language theory – definitions

- The *Kleene closure* of L and is denoted L^* and is defined as $\bigcup_{i=0}^{\infty} L^i$.
- $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Example: Kleene closure

Let $L = \{dog, cat\}$. Observe that $L^0 = \{\epsilon\}$, $L^1 = \{dog, cat\}$, $L^2 = \{catcat, catdog, dogcat, dogdog\}$, etc. Thus L^* contains, among its infinite set of strings, the strings ϵ, *cat*, *dog*, *catcat*, *catdog*, *dogcat*, *dogdog*, *catcatcat*, *catdogcat*, *dogcatcat*, *dogdogcat*, etc.

- The notation for Σ^* should now become clear: it is simply a special case of L^*, where $L = \Sigma$.

Regular expressions

- Regular expressions are a formalism for defining (formal) languages.
- Their "syntax" is formally defined and is relatively simple.
- Their "semantics" is sets of strings: the denotation of a regular expression is a set of strings in some formal language.

Regular expressions

Regular expressions are defined recursively as follows:

- $\emptyset$ is a regular expression
- ϵ is a regular expression
- if $a \in \Sigma$ is a letter then a is a regular expression
- if r_1 and r_2 are regular expressions then so are $(r_1 + r_2)$ and $(r_1 \cdot r_2)$
- if r is a regular expression then so is $(r)^*$
- nothing else is a regular expression over Σ.

Regular expressions

Example: Regular expressions

Let Σ be the alphabet $\{a, b, c, \ldots, y, z\}$. Some regular expressions over this alphabet are:

- $\emptyset$
- a
- $((c \cdot a) \cdot t)$
- $(((m \cdot e) \cdot (o)^*) \cdot w)$
- $(a + (e + (i + (o + u))))$
- $((a + (e + (i + (o + u)))))^*$

Regular expressions

For every regular expression r its denotation, $[\![r]\!]$, is a set of strings defined as follows:

- $[\![\emptyset]\!] = \emptyset$
- $[\![\epsilon]\!] = \{\epsilon\}$
- if $a \in \Sigma$ is a letter then $[\![a]\!] = \{a\}$
- if r_1 and r_2 are regular expressions whose denotations are $[\![r_1]\!]$ and $[\![r_2]\!]$, respectively, then $[\![(r_1 + r_2)]\!] = [\![r_1]\!] \cup [\![r_2]\!]$, $[\![(r_1 \cdot r_2)]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$ and $[\![(r_1)^*]\!] = [\![r_1]\!]^*$

Regular expressions

Example: Regular expressions and their denotations

$\emptyset$	$\emptyset$
a	$\{a\}$
$((c \cdot a) \cdot t)$	$\{c \cdot a \cdot t\}$
$(((m \cdot e) \cdot (o)^*) \cdot w)$	$\{mew, meow, meoow, meooow, \ldots\}$
$(a + (e + (i + (o + u))))$	$\{a, e, i, o, u\}$
$((a + (e + (i + (o + u)))))^*$	*all strings of 0 or more vowels*

Regular expressions

Example: Regular expressions

- Given the alphabet of all English letters, $\Sigma = \{a, b, c, \ldots, y, z\}$, the language Σ^* is denoted by the regular expression Σ^*.
- The set of all strings which contain a vowel is denoted by $\Sigma^* \cdot (a + e + i + o + u) \cdot \Sigma^*$.
- The set of all strings that begin in "*un*" is denoted by $(un)\Sigma^*$.
- The set of strings that end in either "*tion*" or "*sion*" is denoted by $\Sigma^* \cdot (s + t) \cdot (ion)$.
- Note that all these languages are infinite.

Regular languages

- A language is **regular** if it is the denotation of some regular expression.
- Not all formal languages are regular.

Properties of regular languages

- *Closure* properties: A class of languages $\mathcal{L}$ is said to be closed under some operation '$\bullet$' if and only if whenever two languages L_1, L_2 are in the class ($L_1, L_2 \in \mathcal{L}$), also the result of performing the operation on the two languages is in this class: $L_1 \bullet L_2 \in \mathcal{L}$.

Properties of regular languages

Regular languages are closed under:

- Union
- Intersection
- Complementation
- Difference
- Concatenation
- Kleene-star
- Substitution and homomorphism

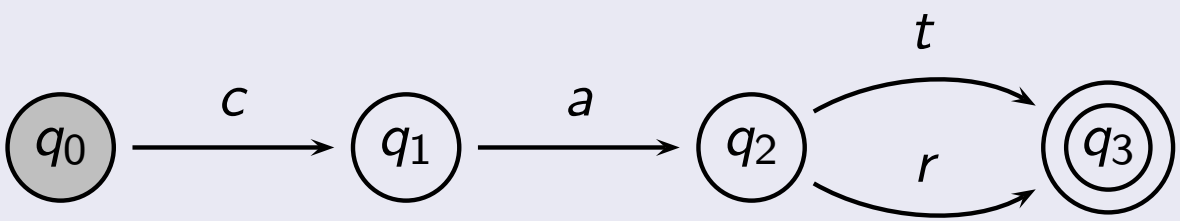
Finite-state automata

- Automata are models of computation: they compute languages.
- A finite-state automaton is a five-tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$, where Σ is a finite set of **alphabet** symbols, Q is a finite set of **states**, $q_0 \in Q$ is the **initial state**, $F \subseteq Q$ is a set of **final** (accepting) states and $\delta : Q \times \Sigma \times Q$ is a relation from states and alphabet symbols to states.

Finite-state automata

Example: Finite-state automaton

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{c, a, t, r\}$
- $F = \{q_3\}$
- $\delta = \{\langle q_0, c, q_1\rangle, \langle q_1, a, q_2\rangle, \langle q_2, t, q_3\rangle, \langle q_2, r, q_3\rangle\}$



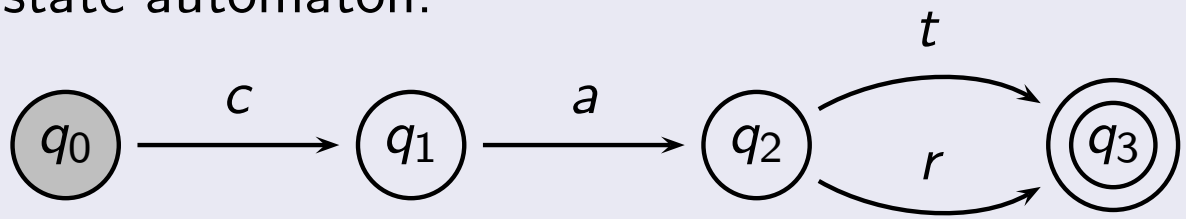
Finite-state automata

- The reflexive transitive extension of the transition relation δ is a new relation, $\hat{\delta}$, defined as follows:
  - for every state $q \in Q$, $(q, \epsilon, q) \in \hat{\delta}$
  - for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $(q, w, q') \in \hat{\delta}$ and $(q', a, q'') \in \delta$ then $(q, w \cdot a, q'') \in \hat{\delta}$.

Finite-state automata

Example: Paths

For the finite-state automaton:



$\hat{\delta}$ is the following set of triples:

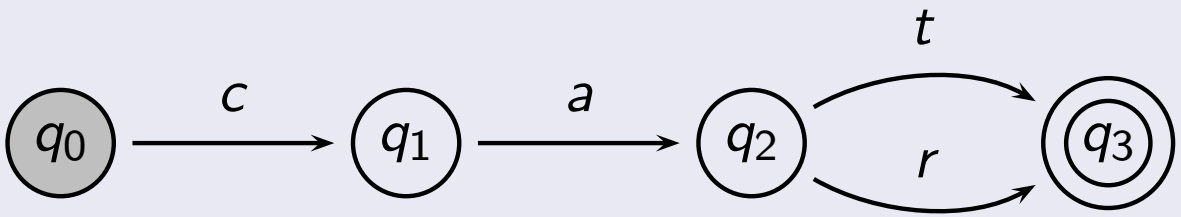
$$\langle q_0, \epsilon, q_0 \rangle, \langle q_1, \epsilon, q_1 \rangle, \langle q_2, \epsilon, q_2 \rangle, \langle q_3, \epsilon, q_3 \rangle,$$
$$\langle q_0, c, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_2, t, q_3 \rangle, \langle q_2, r, q_3 \rangle,$$
$$\langle q_0, ca, q_2 \rangle, \langle q_1, at, q_3 \rangle, \langle q_1, ar, q_3 \rangle,$$
$$\langle q_0, cat, q_3 \rangle, \langle q_0, car, q_3 \rangle$$

Finite-state automata

- A string w is accepted by the automaton $A = \langle Q, q_0, \Sigma, \delta, F \rangle$ if and only if there exists a state $q_f \in F$ such that $(q_0, w, q_f) \in \hat{\delta}$.
- The *language accepted by a finite-state automaton* is the set of all the strings it accepts.

Example: Language

The language of the finite-state automaton:



is $\{cat, car\}$.

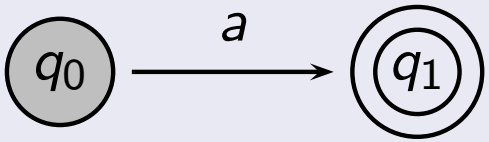
Finite-state automata

Example: Some finite-state automata

$\emptyset$

Finite-state automata

Example: Some finite-state automata



$\{a\}$

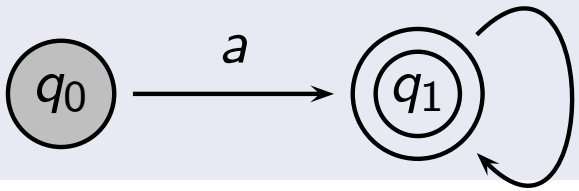
Finite-state automata

Example: Some finite-state automata

$\{\epsilon\}$

Finite-state automata

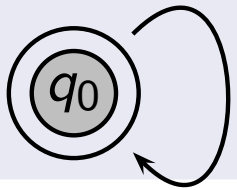
Example: Some finite-state automata



$a \quad \{a, aa, aaa, aaaa, \ldots\}$

Finite-state automata

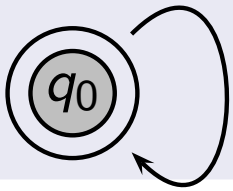
Example: Some finite-state automata



a a^*

Finite-state automata

Example: Some finite-state automata



?

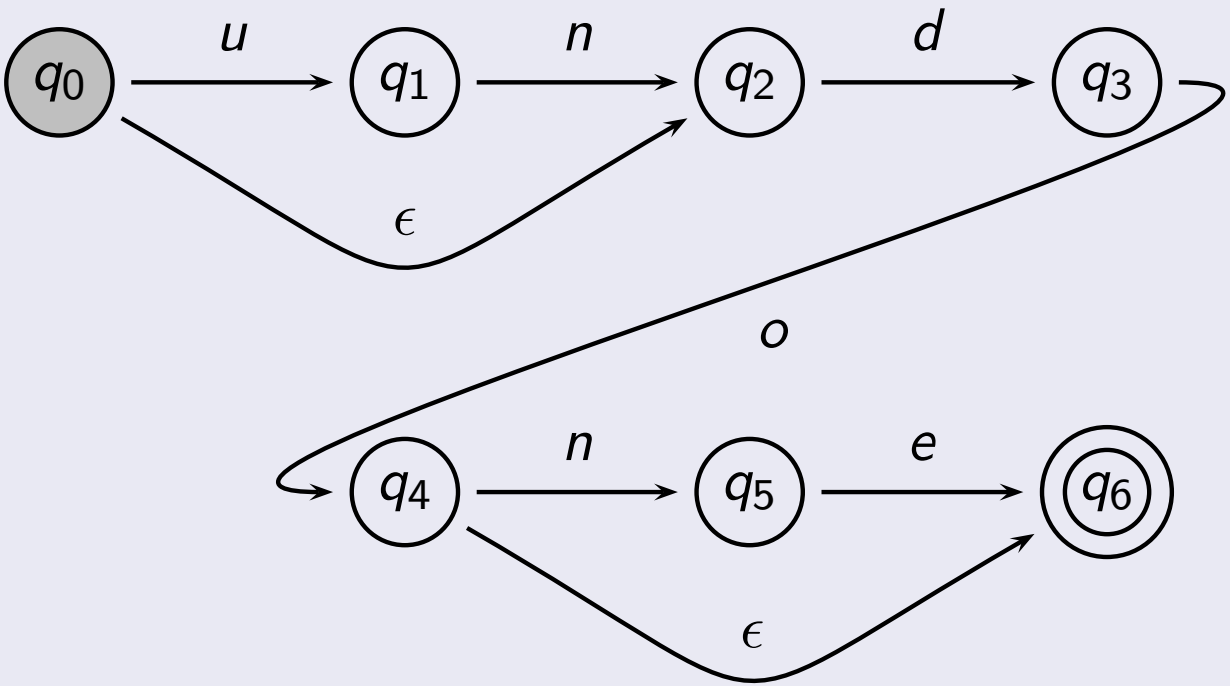
Σ^*

Finite-state automata

- An extension: ϵ-moves.
- The transition relation δ is extended to: $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$

Example: Automata with ϵ-moves

An automaton accepting the language $\{do, undo, done, undone\}$:



Finite-state automata

Theorem (Kleene, 1956)

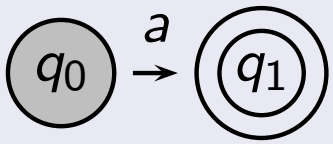
The class of languages recognized by finite-state automata is the class of regular languages.

Finite-state automata

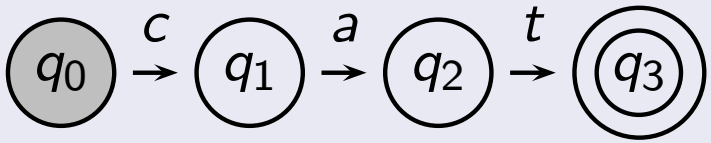
Example: Finite-state automata and regular expressions

$\emptyset$

a



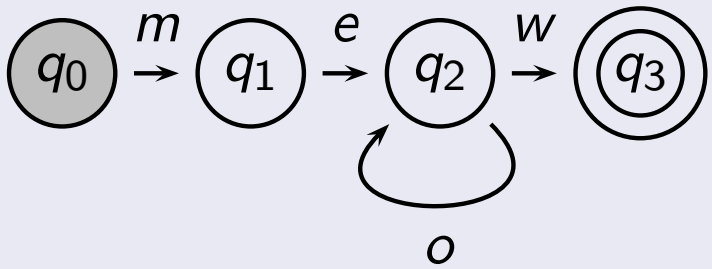
$((c \cdot a) \cdot t)$



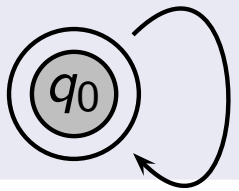
Finite-state automata

Example: Finite-state automata and regular expressions

$$(((m \cdot e) \cdot (o)^*) \cdot w)$$



$$((a + (e + (i + (o + u)))))^*$$



a, e, i, o, u

Operations on finite-state automata

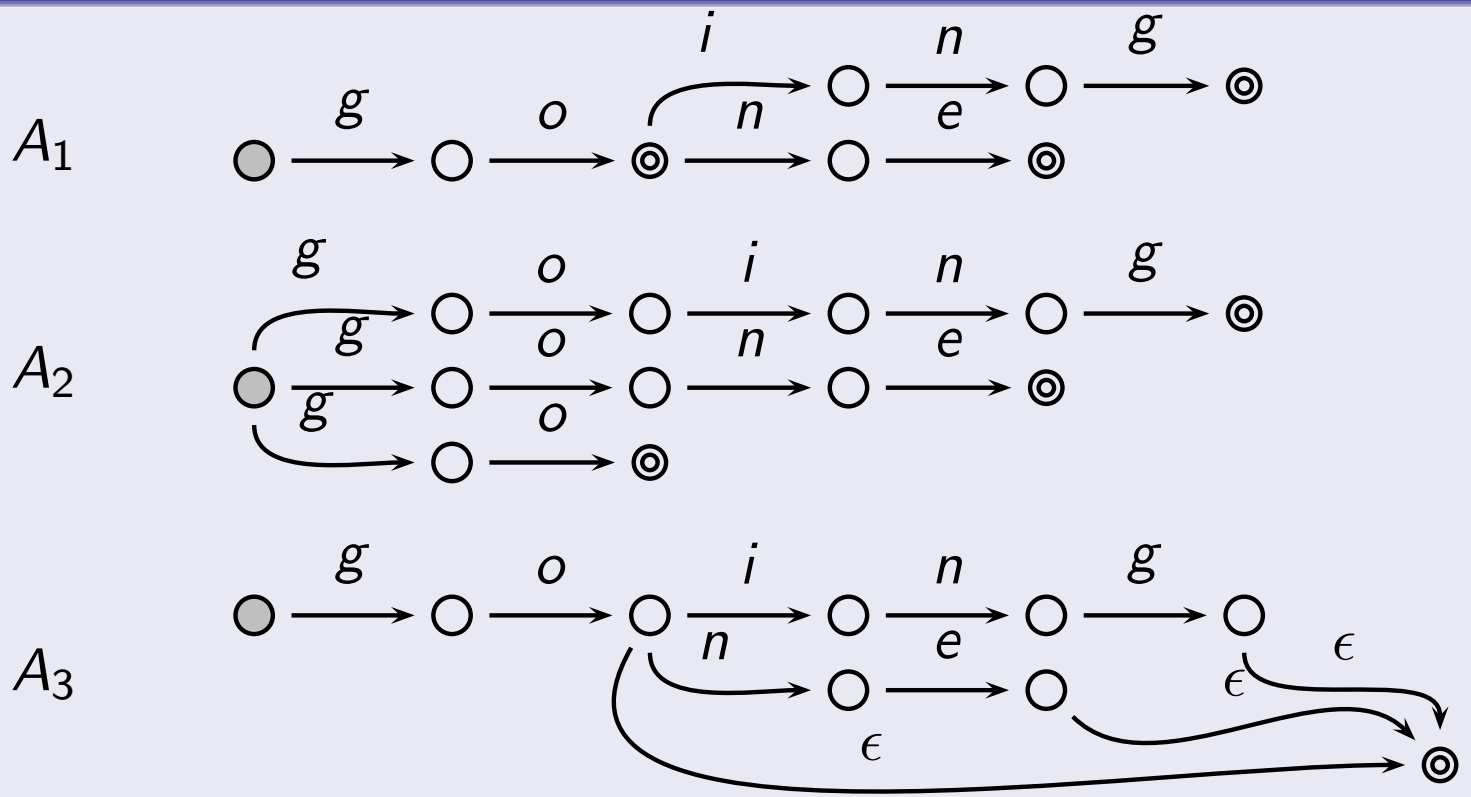
- Concatenation
- Union
- Intersection
- Minimization
- Determinization

Minimization and determinization

- Two automata are equivalent if the accept the same language.
- If L is a regular language then there exists a finite-state automaton A accepting L such that the number of states in A is minimal. A is unique up to isomorphism.
- A finite-state automaton is **deterministic** if its transition relation is a function.
- If L is a regular language then there exists a deterministic, ϵ-free finite-state automaton which accepts it.

Minimization and determinization

Example: Equivalent automata

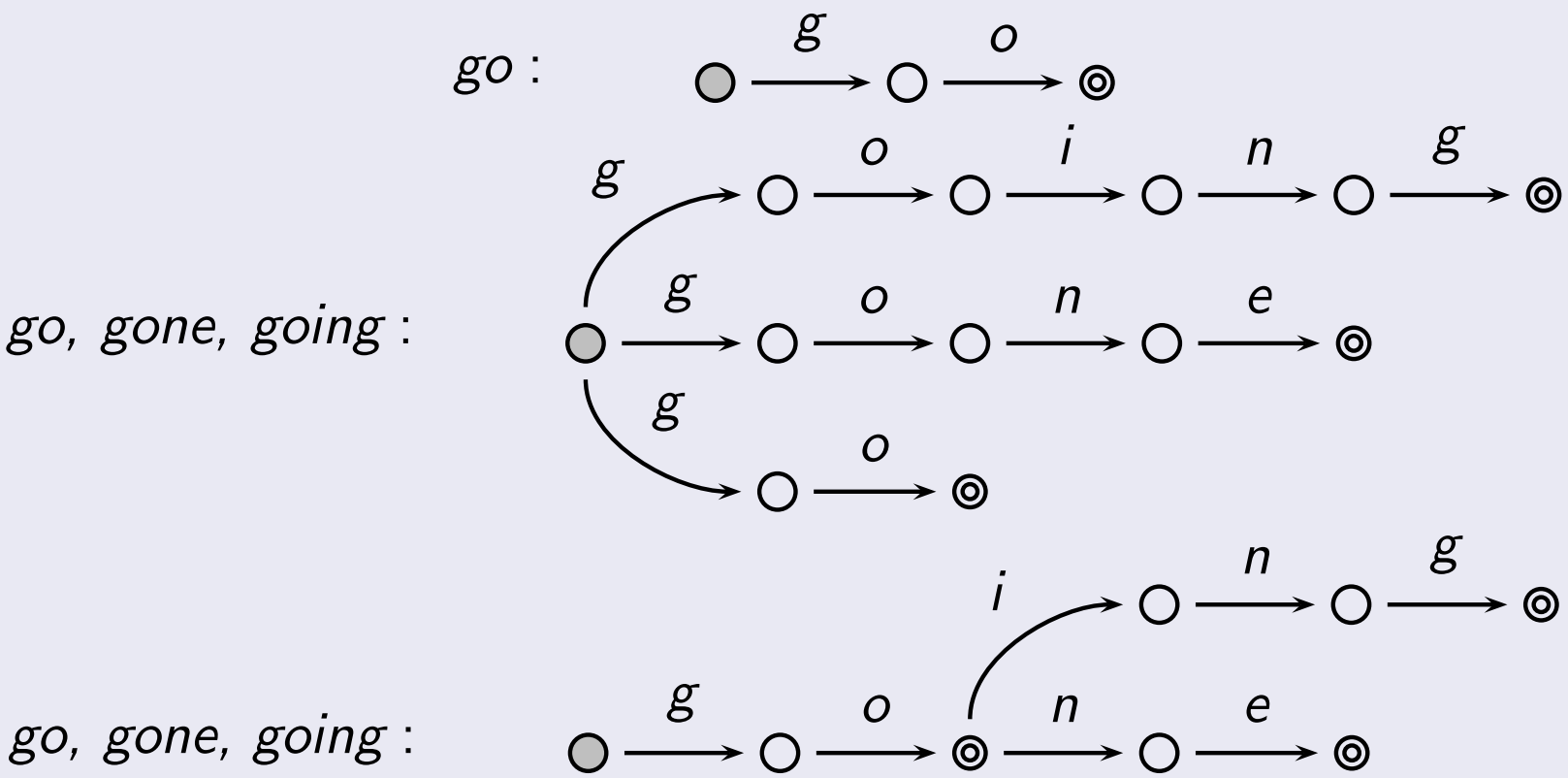


Applications of finite-state automata in NLP

- Finite-state automata are efficient computational devices for generating regular languages.
- An equivalent view would be to regard them as *recognizing* devices: given some automaton A and a word w, applying the automaton to the word yields an answer to the question: Is w a member of $L(A)$, the language accepted by the automaton?
- This reversed view of automata motivates their use for a simple yet necessary application of natural language processing: dictionary lookup.

Applications of finite-state automata in NLP

Example: Dictionaries as finite-state automata

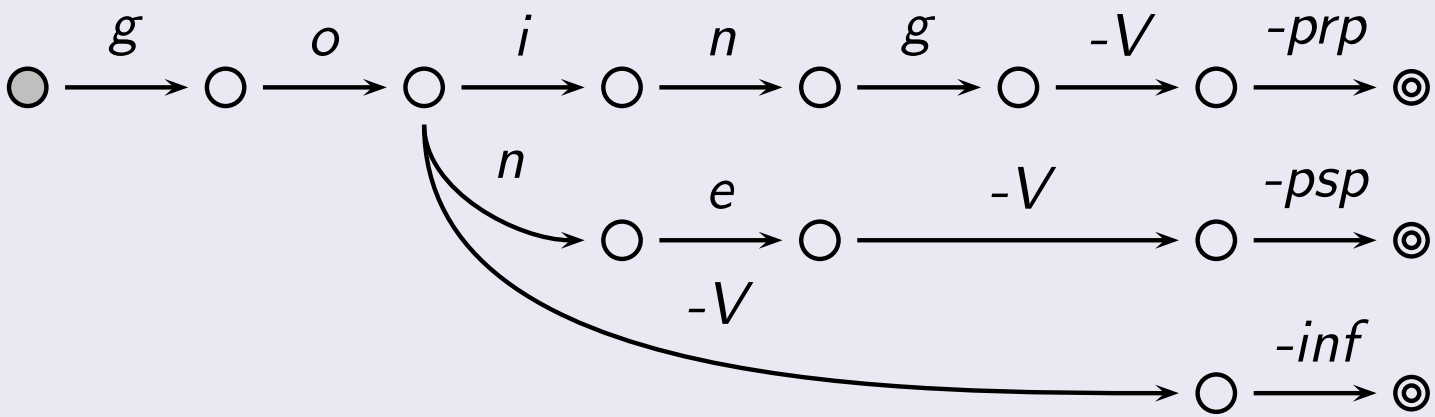


Applications of finite-state automata in NLP

Example: Adding morphological information

Add information about part-of-speech, the number of nouns and the tense of verbs:

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



The appeal of regular languages for NLP

- Most phonological and morphological process of natural languages can be straight-forwardly described using the operations that regular languages are closed under.
- The closure properties of regular languages naturally support modular development of finite-state grammars.
- Most algorithms on finite-state automata are linear. In particular, the recognition problem is linear.
- Finite-state automata are reversible: they can be used both for analysis and for generation.

Regular relations

- While regular expressions are sufficiently expressive for some natural language applications, it is sometimes useful to define relations over two sets of strings.
  Part-of-speech tagging:

Example: Part-of-speech tagging

I	know	some	new	tricks	
PRON	V	DET	ADJ	N	
said	the	Cat	in	the	Hat
V	DET	N	P	DET	N

Regular relations

Example: Morphological analysis

I	know	some	new
I-PRON-1-sg	know-V-pres	some-DET-indef	new-ADJ
tricks	said	the	Cat
trick-N-pl	say-V-past	the-DET-def	cat-N-sg
in	the	Hat	
in-P	the-DET-def	hat-N-sg	

Regular relations

Example: Singular-to-plural mapping

cat	hat	ox	child	mouse	sheep	goose
cats	hats	oxen	children	mice	sheep	geese

Finite-state transducers

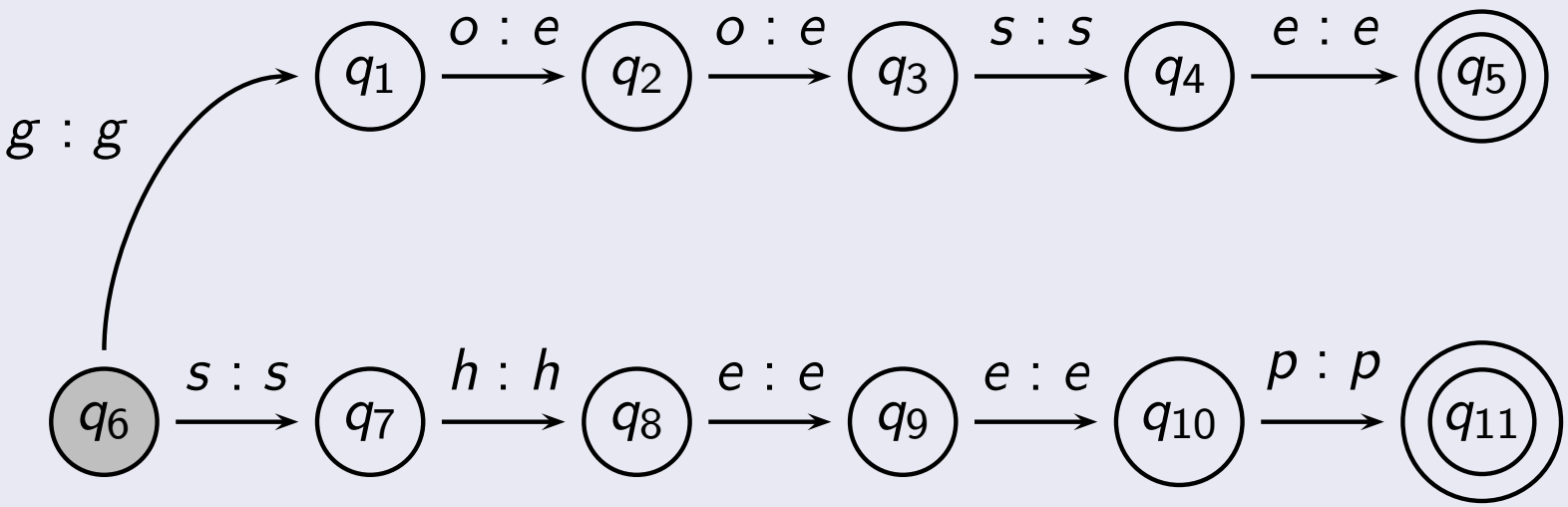
Definition

A finite-state transducer is a six-tuple $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$.

- Q is a finite set of states
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final (or accepting) states,
- Σ_1 and Σ_2 are alphabets: finite sets of symbols, not necessarily disjoint (or different)
- $\delta : Q \times \Sigma_1 \times \Sigma_2 \times Q$ is a relation from states and pairs of alphabet symbols to states.

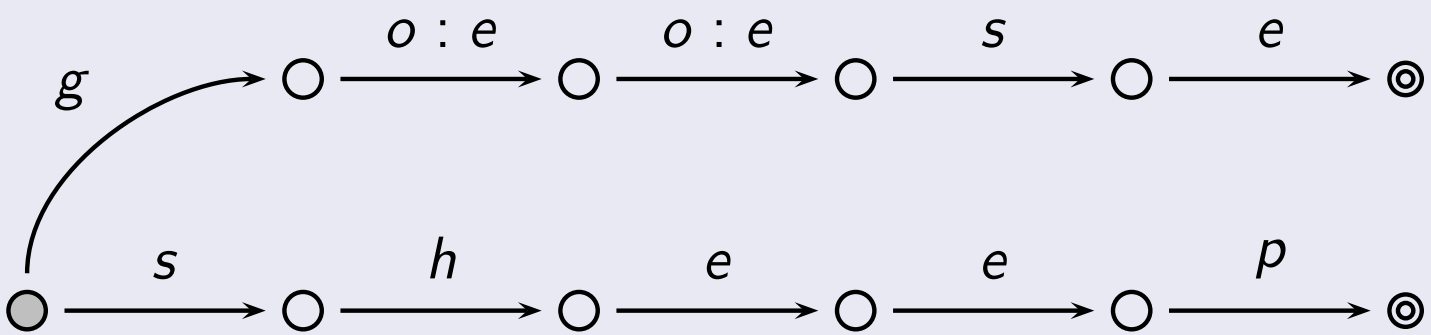
Finite-state transducers

Example:



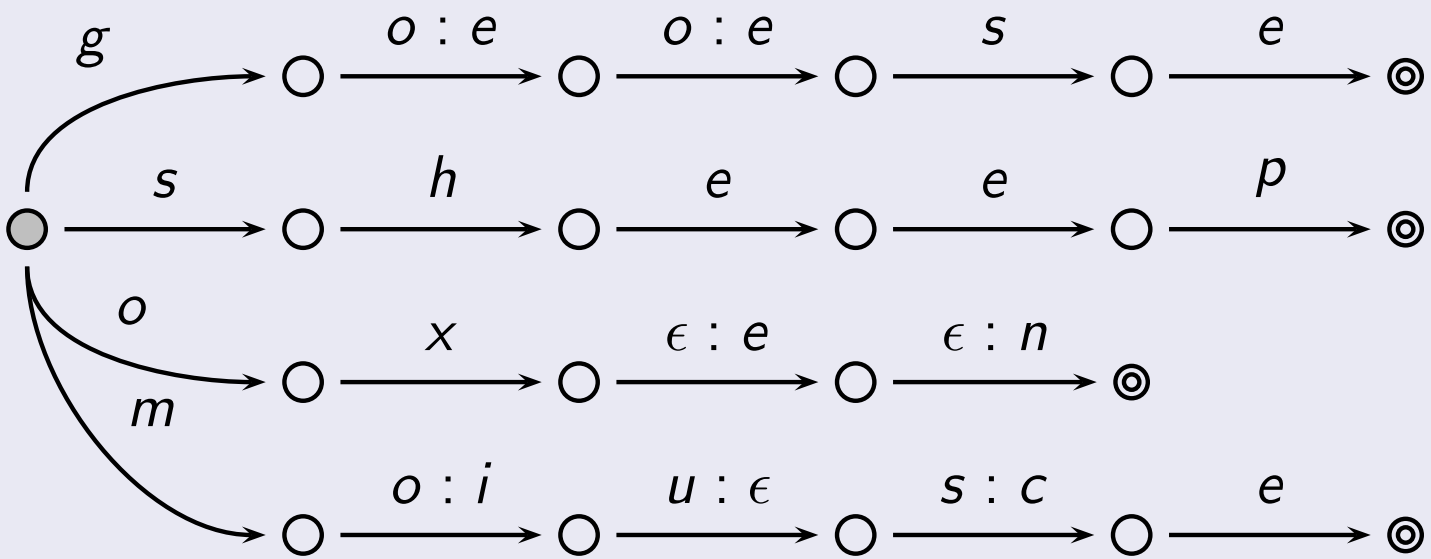
Finite-state transducers

Example: Shorthand notation



Finite-state transducers

Example: Adding ϵ-moves



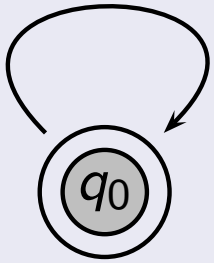
Finite-state transducers

- A finite-state transducer defines a set of pairs: a binary relation over $\Sigma_1^* \times \Sigma_2^*$.
- The relation is defined analogously to how the language of an automaton is defined: A pair $\langle w_1, w_2 \rangle$ is accepted by the transducer $A = \langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$ if and only if there exists a state $q_f \in F$ such that $(q_0, w_1, w_2, q_f) \in \hat{\delta}$.
- The transduction of a word $w \in \Sigma_1^*$ is defined as $T(w) = \{u \mid (q_0, w, u, q_f) \in \hat{\delta} \text{ for some } q_f \in F\}$.

Finite-state transducers

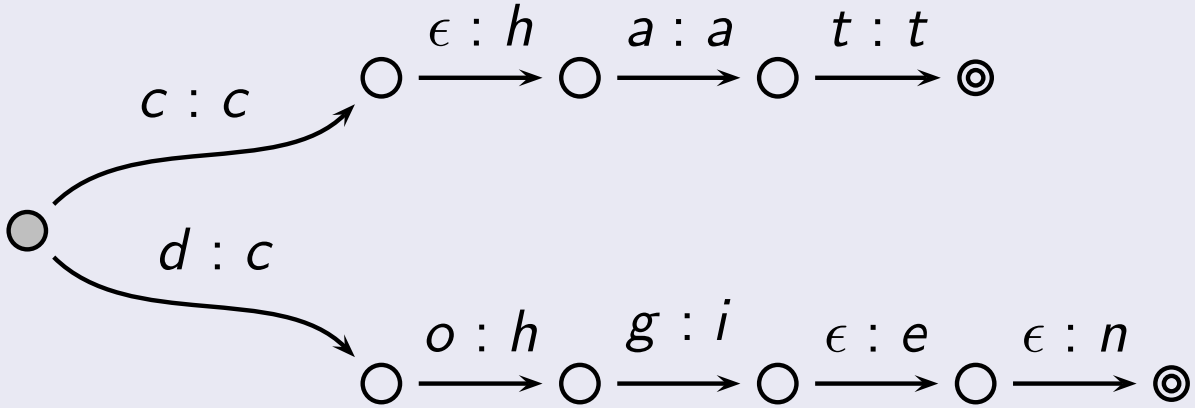
Example: The uppercase transducer

$$a : A, b : B, c : C, \ldots$$



Finite-state transducers

Example: English-to-French



Properties of finite-state transducers

Given a transducer $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$,

- its *underlying automaton* is $\langle Q, q_0, \Sigma_1 \times \Sigma_2, \delta', F \rangle$, where $(q_1, (a, b), q_2) \in \delta'$ iff $(q_1, a, b, q_2) \in \delta$
- its *upper automaton* is $\langle Q, q_0, \Sigma_1, \delta_1, F \rangle$, where $(q_1, a, q_2) \in \delta_1$ iff for some $b \in \Sigma_2$, $(q_1, a, b, q_2) \in \delta$
- its *lower automaton* is $\langle Q, q_0, \Sigma_2, \delta_2, F \rangle$, where $(q_1, b, q_2) \in \delta_2$ iff for some $a \in \Sigma_a$, $(q_1, a, b, q_2) \in \delta$

Properties of finite-state transducers

- A transducer T is *functional* if for every $w \in \Sigma_1^*$, $T(w)$ is either empty or a singleton.
- Transducers are closed under union: if T_1 and T_2 are transducers, there exists a transducer T such that for every $w \in \Sigma_1^*$, $T(w) = T_1(w) \cup T_2(w)$.
- Transducers are closed under inversion: if T is a transducer, there exists a transducer T^{-1} such that for every $w \in \Sigma_1^*$, $T^{-1}(w) = \{u \in \Sigma_2^* \mid w \in T(u)\}$.
- The inverse transducer is $\langle Q, q_0, \Sigma_2, \Sigma_1, \delta^{-1}, F \rangle$, where $(q_1, a, b, q_2) \in \delta^{-1}$ iff $(q_1, b, a, q_2) \in \delta$.

Properties of regular relations

Example: Operations on finite-state relations

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

$$R_2 = \{\textit{grapefruit:pampelmuse, coconut:Kokusnuß}\}$$

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

Properties of finite-state transducers

- Transducers are closed under composition: if T_1 is a transduction from Σ_1^* to Σ_2^* and and T_2 is a transduction from Σ_2^* to Σ_3^*, then there exists a transducer T such that for every $w \in \Sigma_1^*$, $T(w) = T_2(T_1(w))$.
- The number of states in the composition transducer might be $|Q_1 \times Q_2|$.

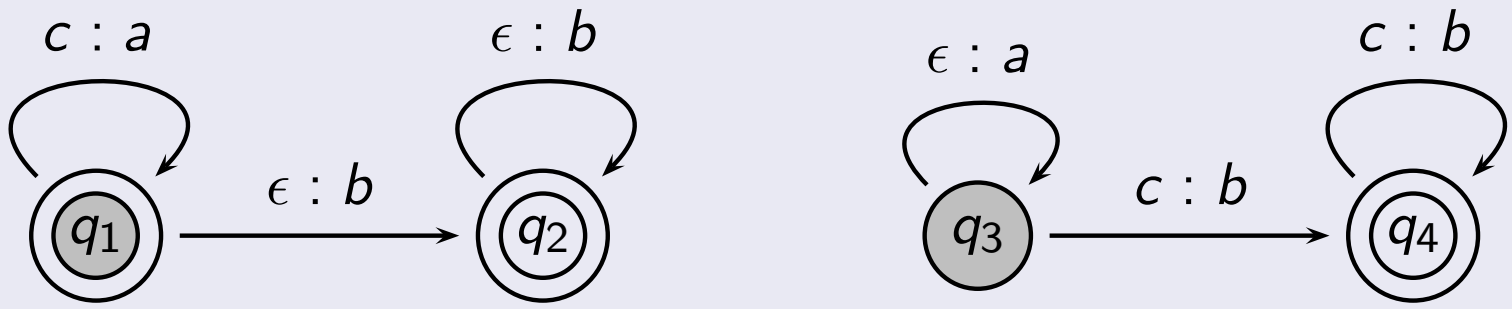
Example: Composition of finite-state relations

$$
\begin{aligned}
R_1 = \{&\textit{tomato:Tomate, cucumber:Gurke,}\\
&\textit{grapefruit:Grapefruit, grapefruit:pampelmuse,}\\
&\textit{pineapple:Ananas,}\\
&\textit{coconut:Koko ,coconut:Kokusnuß}\}\\
R_2 = \{&\textit{tomate:tomato, ananas:pineapple,}\\
&\textit{pampelmousse:grapefruit, concombre:cucumber,}\\
&\textit{cornichon:cucumber, noix-de-coco:coconut}\}\\
R_2 \circ R_1 = \{&\textit{tomate:Tomate, ananas:Ananas,}\\
&\textit{pampelmousse:Grapefruit,}\\
&\textit{pampelmousse:Pampelmuse,}\\
&\textit{concombre:Gurke,cornichon:Gurke,}\\
&\textit{noix-de-coco:Koko, noix-de-coco:Kokusnuße}\}
\end{aligned}
$$

Properties of finite-state transducers

- Transducers are not closed under intersection.

Example:



T_1 T_2

$$T_1(c^n) = \{a^n b^m \mid m \geq 0\}$$
$$T_2(c^n) = \{a^m b^n \mid m \geq 0\} \Rightarrow$$
$$(T_1 \cap T_2)(c^n) = \{a^n b^n\}$$

- Transducers with no ϵ-moves are closed under intersection.

Properties of finite-state transducers

- Computationally efficient
- Denote regular relations
- Closed under concatenation, Kleene-star, union
- Not closed under intersection (and hence complementation)
- Closed under composition
- Weights

Finite-state transducers for phonology and morphology

Example: A motivating example

Consider the following pairs:

accurate	*adequate*	*balanced*	*competent*
inaccurate	*inadequate*	*imbalanced*	*incompetent*
definite	*finite*	*mature*	*nutrition*
indefinite	*infinite*	*immature*	*innutrition*
patience	*possible*	*sane*	*tractable*
impatience	*impossible*	*insane*	*intractable*

The negative forms are constructed by adding the abstract morpheme iN to the positive forms. N is realized as either n or m.

Finite-state transducers for phonology and morphology

- A linguistically accurate description of this phenomenon could be:

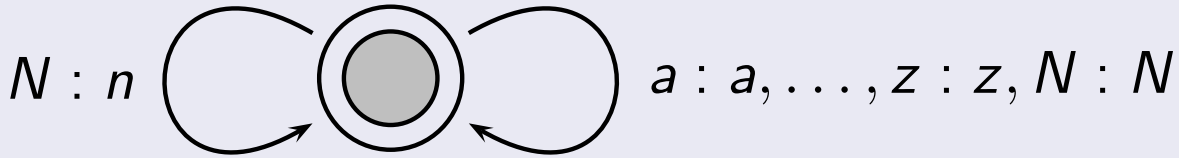
  Rule 1: `N -> m || _ [b|m|p]`

  Rule 2: `N -> n`

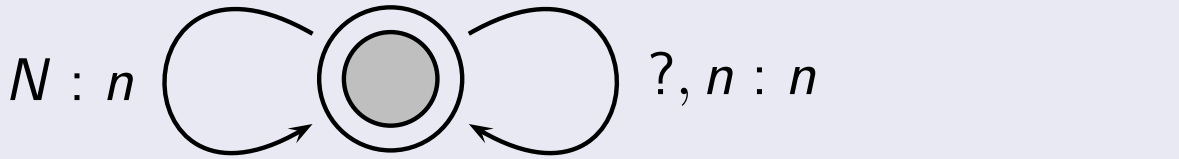
- The rules must be interpreted as *obligatory* and applied in the *order* given.

Finite-state transducers for phonology and morphology

Example: A finite-state transducer for Rule 2 (interpreted as optional)



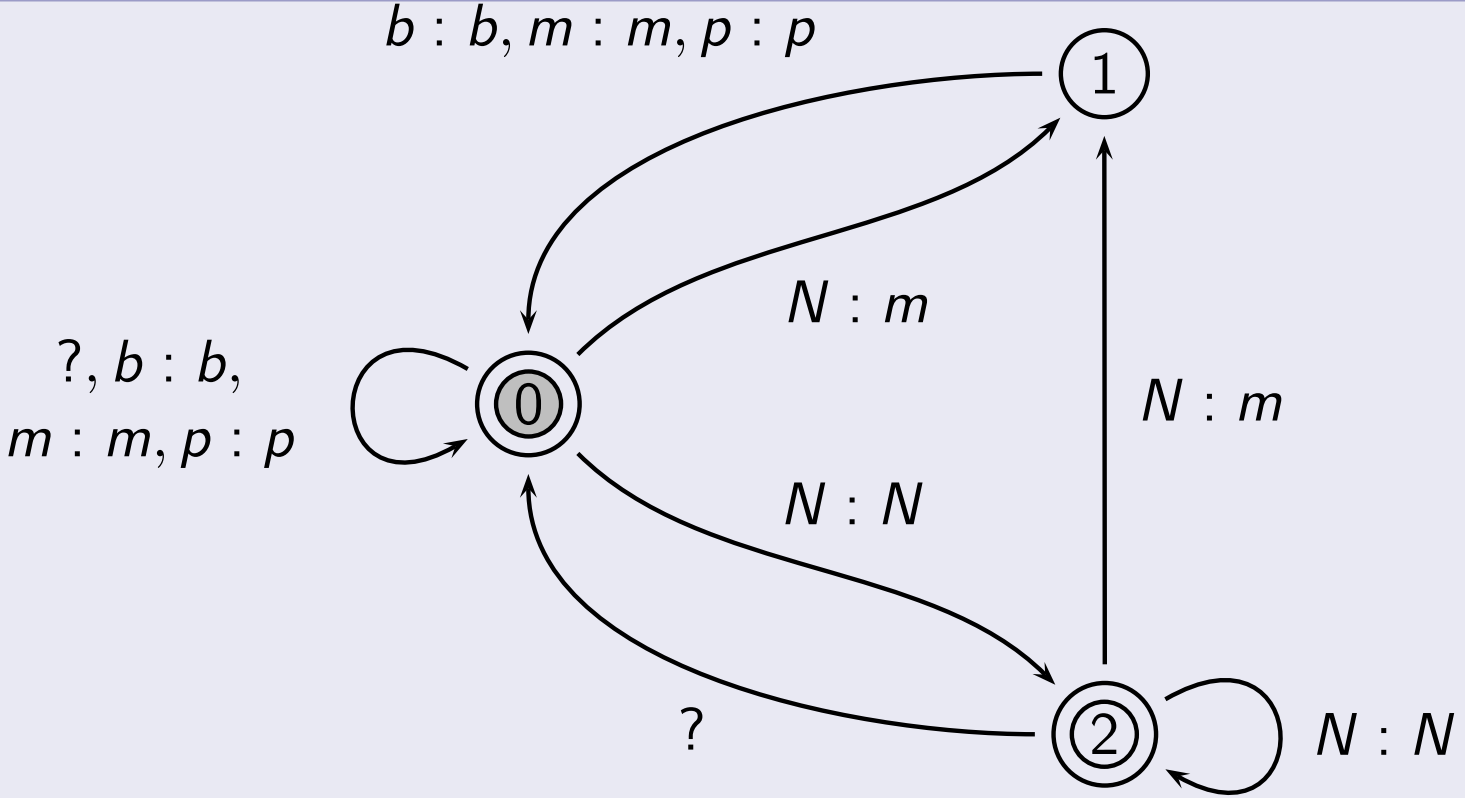
Example: A finite-state transducer for Rule 2 (interpreted as obligatory)



Here, '?' stands for "any symbol that does not occur elsewhere in this rule."

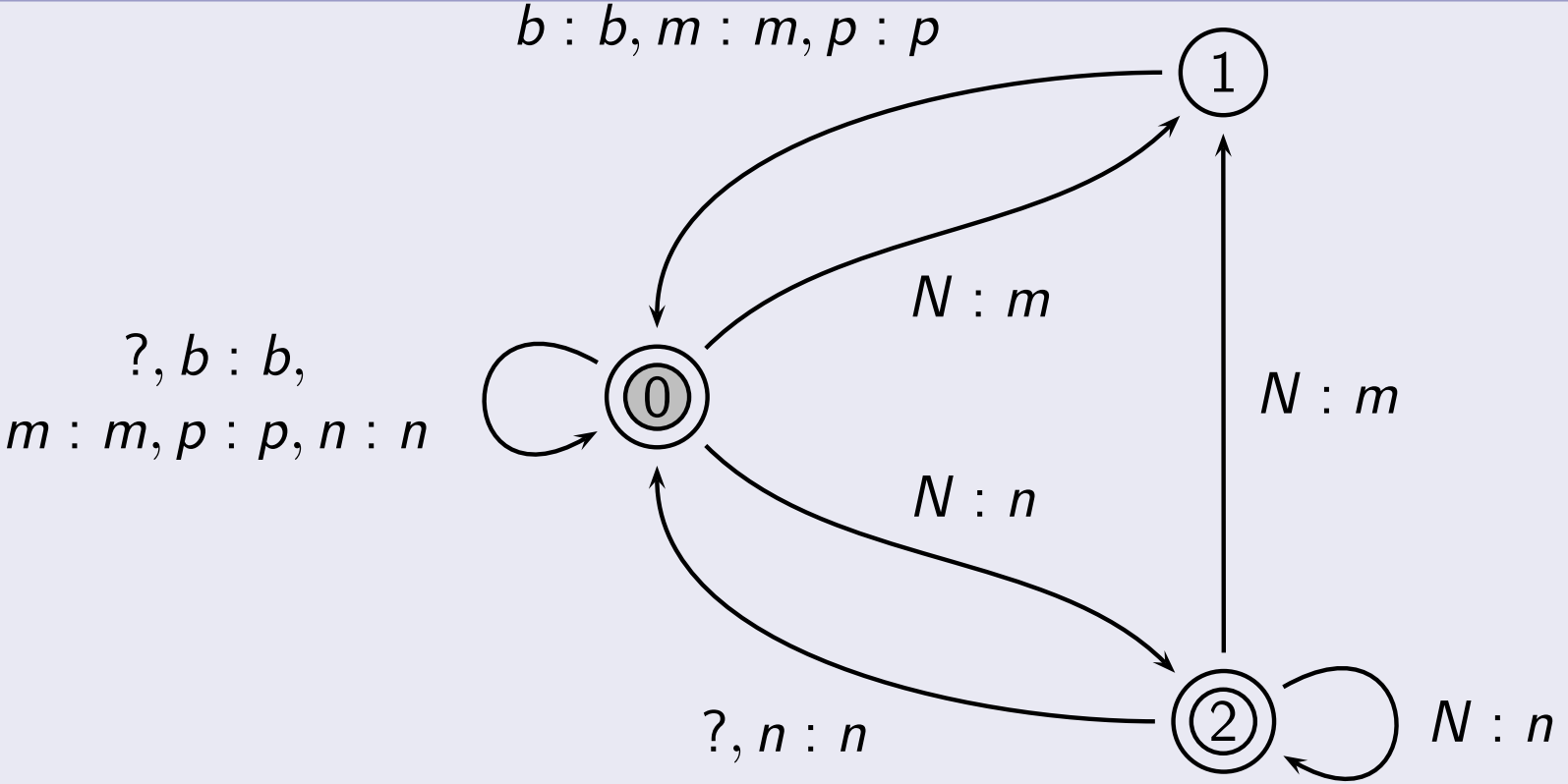
Finite-state transducers for phonology and morphology

Example: A finite-state transducer for Rule 1 (interpreted as obligatory)



Finite-state transducers for phonology and morphology

Example: Composition of obligatory Rules 1 and 2



Implementing composition

- Regular relations are closed under composition.
- Recall that

$$R_1 \circ R_2 = \{\langle x, y\rangle \mid \text{ there exists } z \text{ s. t. } \langle x, z\rangle \in R_1 \text{ and } \langle z, y\rangle \in R_2\}$$

- Let $T_1 = (Q_1, q_1, \Sigma_1, \Sigma_2, \delta_1, F_1)$ and $T_2 = (Q_2, q_2, \Sigma_2, \Sigma_3, \delta_2, F_2)$
- Then

$$T_1 \circ T_2 = T = (Q_1 \times Q_2, \langle q_1, q_2\rangle, \Sigma_1, \Sigma_3, \delta, F_1 \times F_2),$$

where $\delta(\langle s_1, s_2\rangle, a, b) = \{\langle t_1, t_2\rangle \mid \text{ for some } c \in \Sigma_2 \cup \{\epsilon\}, t_1 \in \delta_1(s_1, a, c) \text{ and } t_2 \in \delta_2(s_2, c, b)\}$

- This definition is flawed!

Implementing replace rules

We begin with the simplest rule: unconditional, obligatory replacement: `UPPER -> LOWER`.

Example:

- `a -> b`

```
bcd   abac   aaab
bcd   bbbc   bbbb
```

- `a+ -> b`

```
bcd   abac   aaab   aaab   aaab
bcd   bbbc   bbbb   bbb    bb
```

- `[[a b] | [b c]] -> x`

```
abc   abc
ax    xc
```

Implementing replace rules

Incremental construction:

- `$A`: The language of all strings which contain, as a substring, a string from `A`. Equivalent to `?* A ?*`.
- `noUPPER`: Strings which do not contain strings from `UPPER`.
  Naïvely, this would be defined as `~$UPPER`. However, if `UPPER` happens to include the empty string, then `~$UPPER` is empty.
  A better definition is therefore `~$[UPPER - []]`, which is identical except that if `UPPER` includes the empty string, `noUPPER` includes at least the empty string.

Implementing replace rules

- `UPPER -> LOWER`: Defined as

  `[noUPPER [UPPER .x. LOWER]]* noUPPER`

  A regular relation whose members include any number (possibly zero) of iterations of `[UPPER .x. LOWER]`, interleaved with strings that do not contain `UPPER` which are mapped to themselves.

Implementing replace rules

Example: Special cases

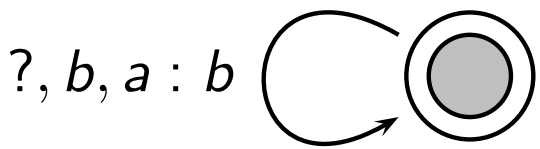
- `[] -> a | b`
  A transducer that freely inserts as and bs in the input string.
- `~$[] -> a | b`
  No replacement; equivalent to `?*`.
- `a | b -> []`
  Removes all as and bs from the input string.
- `a | b -> ~$[]`
  Strings containing as or bs are excluded from the upper language.
  Equivalent to `~$[a | b]`.

Implementing replace rules

Inverse replacement: `UPPER <- LOWER`.

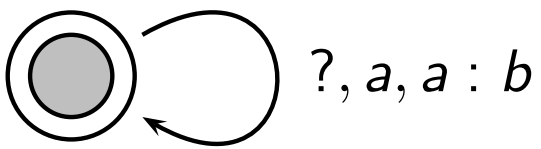
- Meaning: obligatory unconditional inverse replacement.
- Definition and construction: `[LOWER -> UPPER].i`
- (`A.i` is the inverse relation of `A`)
- The difference between `UPPER -> LOWER` and `UPPER <- LOWER`:

$a \rightarrow b$




```
aa bb ab
bb bb bb
```

$a \leftarrow b$




```
aa aa aa
ab aa bb
```

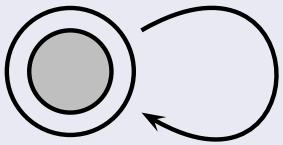
Implementing replace rules

Optional replacement: `UPPER (->) LOWER`.

- Meaning: optionally replace occurrences of strings from `UPPER` by strings from `LOWER`.

Example:

Example: $a\ (\rightarrow)\ b$



$?, a, b, a : b$

```
ab  ab
ab  bb
```

- The tempting (and incorrect) construction: `[UPPER -> LOWER] | [UPPER]`.
- The correct construction: `[?* [UPPER .x. LOWER]]* ?*` or `[ $[UPPER -> LOWER] ]*`.

Conditional replacement

Conditional replacement: `UPPER -> LOWER || L _ R`.

- Meaning: Replace occurrences of strings from `UPPER` by strings from `LOWER` in the context of `L` on the left and `R` on the right.
- Outstanding issue:
  - the interpretation of "between `L` and `R`"
  - interactions between multiple applications of the same rule at different positions in a string: the part that is being replaced may at the same time serve as the context of another adjacent replacement
  - optionality

Conditional replacement

Conditional replacement:

Example:

- Upward oriented: `a b -> x || a b _ a`

  a b a b a b a

  a b x x a

- Downward oriented: `a b -> x \/ a b _ a`

  a b a b a b a a b a b a b a

  a b x a b a a b a b x a

- Right oriented: `a b -> x // a b _ a`

  a b a b a b a

  a b x a b a

- Left oriented: `a b -> x \\ a b _ a`

  a b a b a b a

  a b a b x a

Implementing conditional replacement

We focus on upward oriented, mandatory replacement rules:
`UPPER -> LOWER || L _ R`.

The idea: make the conditional replacement behave exactly like unconditional replacement except that the operation does not take place unless the specified context is present.

The problem: the part being replaced can be at the same time the context of another replacement.

The solution: first, decompose the complex relation into a set of relatively simple components, define them independently of one another, and finally use composition to combine them.

Implementing conditional replacement

Construction:

1. InsertBrackets
2. ConstrainBrackets
3. LeftContext
4. RightContext
5. Replace
6. RemoveBrackets

Implementing conditional replacement

- Two bracket symbols, < and >, are introduced in (1) and (6).
- < indicates the end of a complete left context. > indicates the beginning of a complete right context.
- Their distribution is controlled by (2), (3) and (4). (2) constrains them with respect to each other, whereas (3) and (4) constrain them with respect to the left and right context.
- The replacement expression (5) includes the brackets on both sides of the relation.
- The final result of the composition does not contain any brackets. (1) removes them from the upper side, (6) from the lower side.

Implementing conditional replacement

- Let < and > be two symbols not in Σ.
- InserBrackets eliminates from the upper side language all context markers that appear on the lower side.
- `InsertBrackets = [ ] <- [< | >]`.
- ConstrainBrackets denotes the language consisting of strings that do not contain <> anywhere.
- `ConstrainBrackets = ~$[ < > ]`.

Implementing conditional replacement

- LeftContext denotes the language in which any instance of < is immediately preceded by L and every L is immediately followed by <, ignoring irrelevant brackets.
- [...L] denotes the language of all strings ending in L, ignoring all brackets except for a final <.
- [...L] is defined as [?* L/[<|>]] - [?* <].
- (A/B is A ignore B, the language obtained by splicing in strings from B* anywhere within strings from A)
- Next, [<...] denotes the language of strings beginning with <, ignoring the other bracket: [</> ?*].
- Finally, LeftContext is defined as:

```
~[ ~[...L] [<...] ] & ~[ [...L] ~[<...] ] .
```

Implementing conditional replacement

- RightContext denotes the language in which any instance of > is immediately followed by `R` and any `R` is immediately preceded by >, ignoring irrelevant brackets.
- `[R...]` denotes the language of all strings beginning with `R`, ignoring all brackets except for an initial >.
- `[R...]` is defined as `[ R/[<|>] ?*] - [> ?*]`.
- `[...>]` denotes the language of strings ending with >, ignoring the other bracket.
- `[...>]` is defined as `[?* >/<]`.
- Finally, `RightContext` is defined as

```
~[ [...>] ~[R...] ] & ~[ ~[...>] [R...] ] .
```

Implementing conditional replacement

- `Replace` is the unconditional replacement of `<UPPER>` by `<LOWER>`, ignoring irrelevant brackets.
- `Replace` is defined as:

  ```
  < UPPER/[<|>] > -> < LOWER/[</>] >
  ```

- `RemoveBrackets` denotes the relation that maps the strings of the upper language to the same strings without any context markers.
- `RemoveBrackets=[ <|> ] -> [].`

Implementing conditional replacement

Construction:

```
InsertBrackets
.o.
ConstrainBrackets
.o.
LeftContext
.o.
RightContext
.o.
Replace
.o.
RemoveBrackets
```

Implementing conditional replacement

Special cases:

- $A = \{\epsilon\}$ or $\epsilon \in A$.
- Boundary symbol (.#.): `L _ R` actually means `?* L _ R ?*`.

Introduction to XFST

- XFST is an interface giving access to finite-state operations (algorithms such as union, concatenation, iteration, intersection, composition etc.)
- XFST includes a regular expression compiler
- The interface of XFST includes a lookup operation (*apply up*) and a generation operation (*apply down*)
- The regular expression language employed by XFST is an extended version of standard regular expressions

Introduction to XFST

`a`	a simple symbol
`c a t`	a concatenation of three symbols
`[c a t]`	grouping brackets
`cat`	a single multicharacter symbol
`?`	denotes any single symbol
`%+`	the literal plus-sign symbol
`%*`	the literal asterisk symbol (and similarly for `%?`, `%(`, `%]` etc.)
`` ``+Noun'' ```	single symbol with multicharacter print name
`%+Noun`	single symbol with multicharacter print name

Introduction to XFST

`{cat}`	equivalent to `[c a t]`
`[ ]`	the empty string
`0`	the empty string
`[A]`	bracketing; equivalent to `A`
`A|B`	union
`(A)`	optionality; equivalent to `[A|0]`
`A&B`	intersection
`A B`	concatenation
`A-B`	set difference

Introduction to XFST

`A*`	Kleene-star
`A+`	one or more iterations
`?*`	the universal language
`~A`	the complement of `A`; equivalent to `[?* - A]`
`~[?*]`	the empty language

Introduction to XFST – denoting relations

`A .x. B`	Cartesian product; relates every string in `A` to every string in `B`
`a:b`	shorthand for `[a .x. b]`
`%+Pl:s`	shorthand for `[%+Pl .x. s]`
`%+Past:ed`	shorthand for `[%+Past .x. ed]`
`%+Prog:ing`	shorthand for `[%+Prog .x. ing]`

Introduction to XFST – useful abbreviations

`$A`	the language of all the strings that contain `A`; equivalent to `[?* A ?*]`
`A/B`	the language of all the strings in `A`, ignoring any strings from `B`
`a*/b`	includes strings such as `a`, `aa`, `aaa`, `ba`, `ab`, `aba` etc.
`\A`	any single symbol, minus strings in `A`. Equivalent to `[? - A]`
`\b`	any single symbol, except 'b'. Compare to:
`~A`	the complement of `A`, i.e., `[?* - A]`

Introduction to XFST – user interface

Example: XFST

```
prompt% H:\class\data\shuly\xfst

xfst> help
xfst> help union net
xfst> exit
xfst> read regex [d o g | c a t];
xfst> read regex < myfile.regex
xfst> apply up dog
xfst> apply down dog
xfst> pop stack
xfst> clear stack
xfst> save stack myfile.fsm
```

Introduction to XFST – example

Example: Leave

```
[ [l e a v e %+VBZ .x. l e a v e s] |
  [l e a v e %+VB .x. l e a v e] |
  [l e a v e %+VBG .x. l e a v i n g] |
  [l e a v e %+VBD .x. l e f t] |
  [l e a v e %+NN .x. l e a v e] |
  [l e a v e %+NNS .x. l e a v e s] |
  [l e a f %+NNS .x. l e a v e s ] |
  [l e f t %+JJ .x. l e f t]
]
```

Introduction to XFST – example of lookup and generation

Example: Leave

```
APPLY DOWN> leave+VBD
left
APPLY UP> leaves
leave+NNS
leave+VBZ
leaf+NNS
```

Introduction to XFST – variables

Example: Variables

```
xfst> define Myvar;
xfst> define Myvar2 [d o g | c a t];
xfst> undefine Myvar;


xfst> define var1 [b i r d | f r o g | d o g];
xfst> define var2 [d o g | c a t];
xfst> define var3 var1 | var2;
xfst> define var4 var1 var2;
xfst> define var5 var1 & var2;
xfst> define var6 var1 - var2;
```

Introduction to XFST – variables

Example: Variables

```
xfst> define Root [w a l k | t a l k | w o r k];
xfst> define Prefix [0 | r e];
xfst> define Suffix [0 | s | e d | i n g];
xfst> read regex Prefix Root Suffix;
xfst> words
xfst> apply up walking
```

Introduction to XFST – replace rules

- Replace rules are an extremely powerful extension of the regular expression metalanguage.
- The simplest replace rule is of the form

$$upper \rightarrow lower \parallel leftcontext \_ rightcontext$$

- Its denotation is the relation which maps string to themselves, with the exception that an occurrence of *upper* in the input string, preceded by *leftcontext* and followed by *rightcontext*, is replaced in the output by *lower*.

Introduction to XFST – replace rules

- Word boundaries can be explicitly referred to:

Example:

```
xfst> define Vowel [a|e|i|o|u];
xfst> e -> ’ || [.#.] [c | d | l | s] _ [% Vowel];
```

Introduction to XFST – replace rules

- Contexts can be omitted:

Example:

```
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
```

- This can be used to clear unnecessary symbols introduced for "bookkeeping":

Example:

```
xfst> define Rule1 %^MorphmeBoundary -> 0;
```

Introduction to XFST – replace rules

- The language Bambona has an underspecified nasal morpheme *N* that is realized as a labial *m* or as a dental *n* depending on its environment: *N* is realized as *m* before *p* and as *n* elsewhere.
- The language also has an assimilation rule which changes *p* to *m* when the *p* is followed by *m*.

Example:

```
xfst> clear stack ;
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
xfst> read regex Rule1 .o. Rule2 .o. Rule3 ;
```

Introduction to XFST – replace rules

- Rules can define multiple replacements:

```
[ A -> B, B -> A ]
```

- or multiple replacements that share the same context:

```
[ A -> B, B -> A || L _ R ]
```

- or multiple contexts:

```
[ A -> B || L1 _ R1, L2 _ R2 ]
```

- or multiple replacements and multiple contexts:

```
[ A -> B, B -> A ||  L1 _ R1, L2 _ R2 ]
```

Introduction to XFST – replace rules

- Rules can apply in parallel:

Example:

```
xfst> clear stack
xfst> read regex a -> b .o. b -> a ;
xfst> apply down abba
aaaa
xfst> clear stack
xfst> read regex b -> a .o. a -> b ;
xfst> apply down abba
bbbb
xfst> clear stack
xfst> read regex a -> b , b -> a ;
xfst> apply down abba
baab
```

Introduction to XFST – replace rules

- When rules that have contexts apply in parallel, the rule separator is a double comma:

Example:

```
xfst> clear stack
xfst> read regex
b -> a || .#. s ?* _ ,, a -> b || _ ?* e .#. ;
xfst> apply down sabbae
sbaabe
```

Introduction to XFST – English verb morphology

Example:

```
define Vowel [a|e|i|o|u] ;
define Cons [? - Vowel] ;
define base [{dip} | {print} | {toss} | {bake} | {move}] ;
define suffix [0 | {s} | {ed} | {ing}] ;
define form [base %+ suffix];
define FinalS [s %+ s] -> [s %+ e s] ;
define FinalE e -> 0 || _ %+ [e | i];
define DoubleFinal b -> [b b], d -> [d d], f -> [f f],
       g -> [g g], k -> [k k], l -> [l l], m -> [m m],
       n -> [n n], p -> [p p], r -> [r r], s -> [s s],
       t -> [t t], z -> [z z] || Cons Vowel _ %+ Vowel ;
define RemovePlus %+ -> 0 ;
read regex form .o. DoubleFinal .o. FinalS .o. FinalE
                .o. RemovePlus;
```

Introduction to XFST – English spell checking

Example:

```
clear;
define A [i e] -> [e i] || c _ ,,
         [e i] -> [i e] || [? - c] _ ;

define B [[e i] -> [i e]] .o.
         [[i e] -> [e i] || c _];
read regex B;
```

Introduction to XFST – morphology

Example: Hebrew adjective inflection

```
clear stack ;
define base [{gdwl} | {yph} | {xbrwty} | {rk} | {adwm} |
             {q$h} | {ap$ry}] ;
define suffix [0 | {h} | {ym} | {wt}] ;
define form [base %+ suffix];
define FinalH h -> 0 || _ %+ ? ;
define FinalY h -> t || y %+ _ ;
define RemovePlus %+ -> 0 ;
read regex form .o. FinalH .o. FinalY .o. RemovePlus;
```

Introduction to XFST – marking

- The special symbol "..." in the right-hand side of a replace rule stands for whatever was matched in the left-hand side of the rule.

Example:

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u] -> %[ ... %];
xfst> apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]ly
```

Introduction to XFST – marking

Example:

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u]+ -> %[ ... %];
xfst> apply down feeling
f[e][e]l[i]ng
f[ee]l[i]ng
xfst> apply down poolcleaning
p[o][o]lcl[e][a]n[i]ng
p[oo]lcl[e][a]n[i]ng
p[o][o]lcl[ea]n[i]ng
p[oo]lcl[ea]n[i]ng
xfst> read regex [a|e|i|o|u]+ @-> %[ ... %];
xfst> apply down poolcleaning
p[oo]lcl[ea]n[i]ng
```

Introduction to XFST – shallow parsing

- Assume that text is represented as strings of part-of-speech tags, using 'd' for determiner, 'a' for adjective, 'n' for noun, and 'v' verb, etc. In other words, in this example the regular expression symbols represent whole words rather than single letters in a text.
- Assume that a noun phrase consists of an optional determiner, any number of adjectives, and one or more nouns:

```
[(d) a* n+]
```

- This expression denotes an infinite set of strings, such as "n" (cats), "aan" (discriminating aristocratic cats), "nn" (cat food), "dn" (many cats), "dann" (that expensive cat food) etc.

Introduction to XFST – shallow parsing

- A simple noun phrase parser can be thought of as a transducer that inserts markers, say, a pair of braces { }, around noun phrases in a text.
- The task is not as trivial as it seems at first glance. Consider the expression

```
[(d) a* n+ -> %{ ... %}]
```

- Applied to the input "danvn" (many small cats like milk) this transducer yields three alternative bracketings:

```
xfst> apply down danvn
da{n}v{n}
d{an}v{n}
{dan}v{n}
```

Introduction to XFST – longest match

- For certain applications it may be desirable to produce a unique parse, marking the maximal expansion of each NP: "{dan}v{n}".
- Using the left-to-right, longest-match replace operator `@->` instead of the simple replace operator `->` yields the desired result:

```
[(d) a* n+ @-> %{ ... %}]
```

- ```
  xfst> apply down danvn
  {dan}v{n}
  ```
  ```

Introduction to XFST – the coke machine

- A vending machine dispenses drinks for 65 cents a can.
- It accepts any sequence of the following coins: 5 cents (represented as ‘n’), 10 cents (‘d’) or 25 cents (‘q’).
- Construct a regular expression that compiles into a finite-state automaton that implements the behavior of the soft drink machine, pairing “PLONK” with a legal sequence that amounts to 65 cents.

Introduction to XFST – the coke machine

- The construction `A^n` denotes the concatenation of `A` with itself n times.
- Thus the expression `[n .x. c^5]` expresses the fact that a nickel is worth 5 cents.
- A mapping from all possible sequences of the three symbols to the corresponding value:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
```

- The solution:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
.o.
[c^65 .x. PLONK]
```

Introduction to XFST – the coke machine

Example:

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define BuyCoke
SixtyFiveCents .o. [c^65 .x. PLONK] ;
```

Introduction to XFST – the coke machine

- In order to ensure that extra money is paid back, we need to modify the lower language of BuyCoke to make it a subset of `[PLONK* q* d* n*]`.
- To ensure that the extra change is paid out only once, we need to make sure that quarters get paid before dimes and dimes before nickels.

Example:

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
```

Introduction to XFST – the coke machine

- The next refinement is to ensure that as much money as possible is converted into soft drinks and to remove any ambiguity in how the extra change is to be reimbursed.

Example:

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
define ExactChange ReturnChange .o.
[~$[q q q | [q q | d] d [d | n] | n n]] ;
```

Introduction to XFST – the coke machine

- To make the machine completely foolproof, we need one final improvement.
- Some clients may insert unwanted items into the machine (subway tokens, foreign coins, etc.). These objects should not be accepted; they should passed right back to the client.
- This goal can be achieved easily by wrapping the entire expression inside an ignore operator.

Example:

```
define IgnoreGarbage
[ [ ExactChange ]/[\[q | d | n]]] ;
```

Applications of finite-state technology in NLP

- Phonology; language models for speech recognition
- Representing lexicons and dictionaries
- Morphology; morphological analysis and generation
- Shallow parsing
- Named entity recognition
- Sentence boundary detection; segmentation
- Translation...