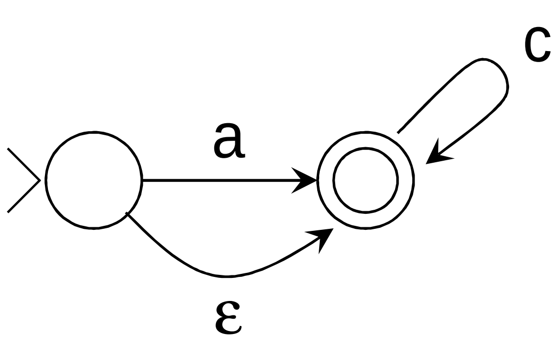
Finite-State Methods

Finite state acceptors (FSAs)



Defines the language a? c*
= {a, ac, acc, accc, …,
ε, c, cc, ccc, …}

- Things you may know about FSAs:
  - Equivalence to regexps
  - Union, Kleene *, concat, intersect, complement, reversal
  - Determinization, minimization
  - Pumping, Myhill-Nerode

n-gram models not good enough

- Want to model grammaticality
- A "training" sentence known to be grammatical:

BOS **mouse traps catch mouse traps** EOS

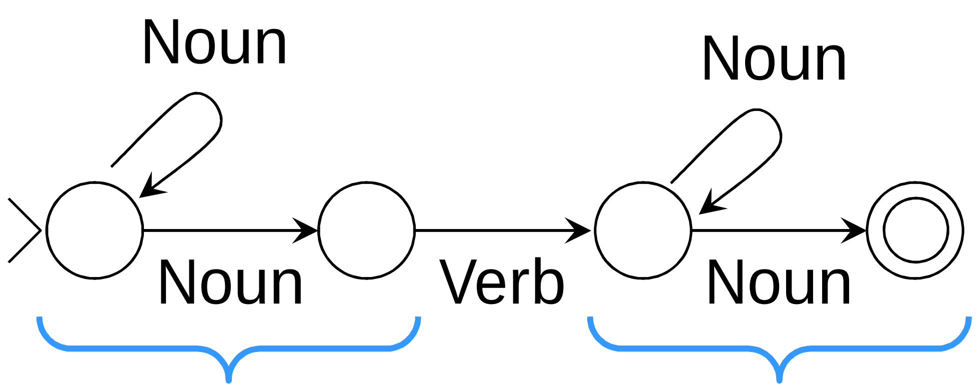
trigram model must allow these trigrams

- Resulting trigram model has to overgeneralize:
  - allows sentences with 0 verbs
    BOS **mouse traps** EOS
  - allows sentences with 2 or more verbs
    BOS **mouse traps catch mouse traps catch mouse traps catch mouse traps** EOS
- Can't remember whether it's in subject or object (i.e., whether it's gotten to the verb yet)

Finite-state models can "get it"

- Want to model grammaticality
  `BOS` **`mouse traps catch mouse traps`** `EOS`
- Finite-state can capture the generalization here:

Noun+ Verb Noun+



preverbal states (still need a verb to reach final state)

postverbal states (verbs no longer allowed)

Allows arbitrarily long NPs (just keep looping around for another Noun modifier).

Still, never forgets whether it's preverbal or postverbal!

(Unlike 50-gram model)

How powerful are regexps / FSAs?

- More powerful than n-gram models
  - The hidden state may “remember” arbitrary past context
  - With k states, can remember which of k “types” of context it’s in
- Equivalent to HMMs
  - In both cases, you observe a sequence and it is “explained” by a hidden path of states. The FSA states are like HMM tags.
- Appropriate for phonology and morphology

Word = Syllable+
= (Onset Nucleus Coda?)+
= (C+ V+ C*)+
= ((b|d|f|...)+ (a|e|i|o|u)+ (b|d|f|...)*)+

How powerful are regexps / FSAs?

- But less powerful than CFGs / pushdown automata
  - Can't do recursive center-embedding
  - Hmm, humans have trouble processing those constructions too ...

- This is the rat that ate the malt.
- This is the malt that the rat ate.

- This is the cat that bit the rat that ate the malt.
- This is the malt that the rat that the cat bit ate.

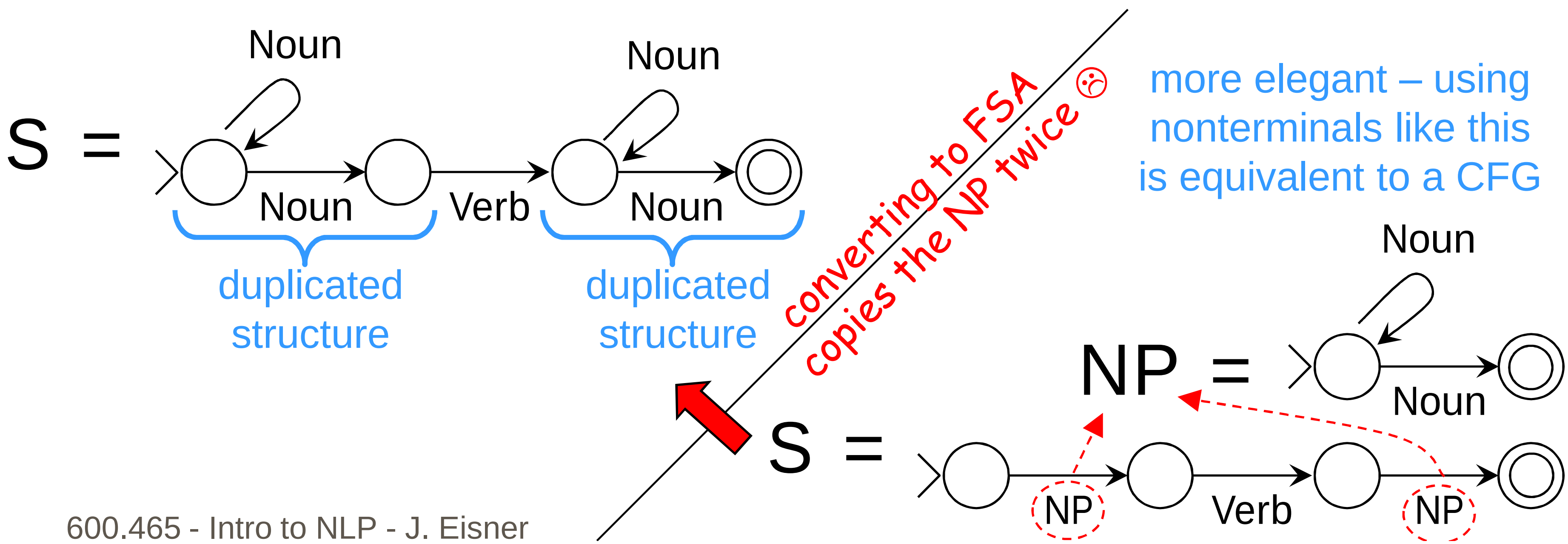
finite-state can handle this pattern (can you write the regexp?)

- This is the dog that chased the cat that bit the rat that ate the malt.
- This is the malt that [the rat that [the cat that [the dog chased] bit] ate].

but not this pattern, which requires a CFG

How powerful are regexps / FSAs?

- But less powerful than CFGs / pushdown automata
- More important: Less explanatory than CFGs
  - An CFG without recursive center-embedding can be converted into an equivalent FSA – but the FSA will usually be far larger
  - Because FSAs can't reuse the same phrase type in different places



We’ve already used FSAs this way ...

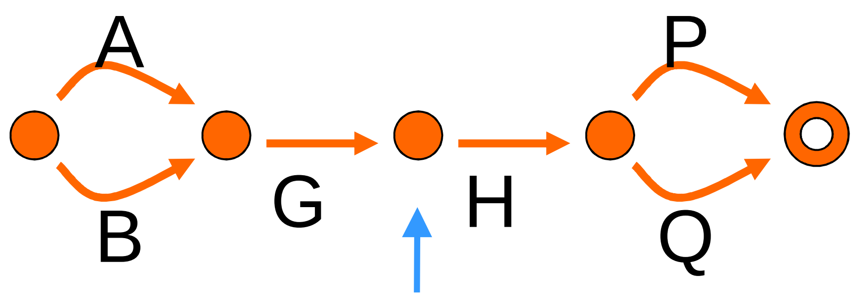
- CFG with regular expression on the right-hand side:

  X ì (A | B) G H (P | Q)

  NP ì (Det | £) Adj* N

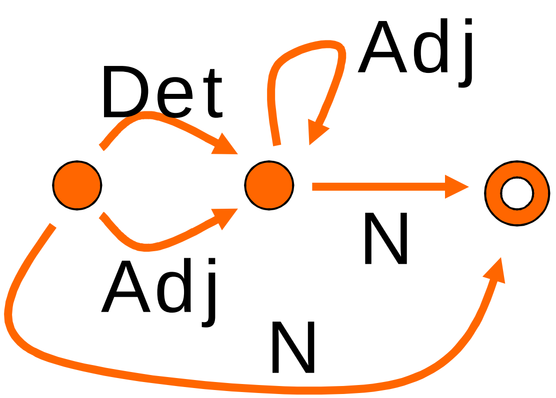
- So each nonterminal has a finite-state automaton, giving a “recursive transition network (RTN)”

X ì



Automaton state replaces dotted rule (X ì A G . H P)

NP ì



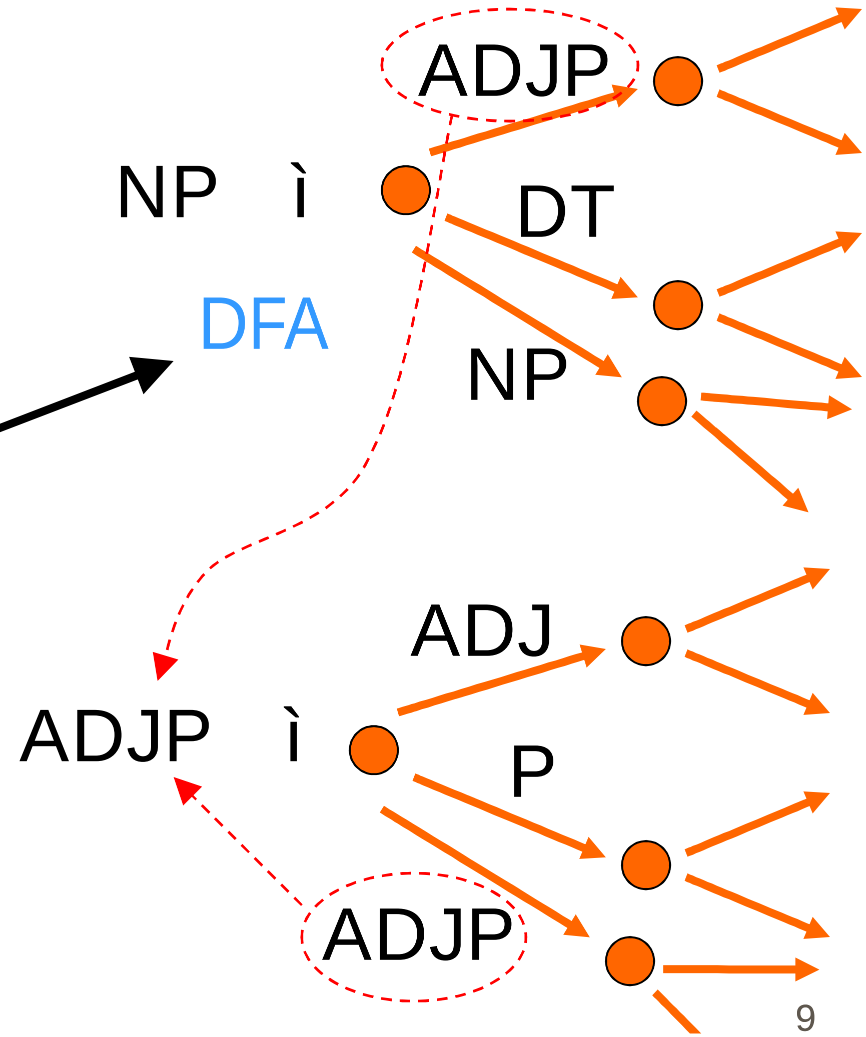
We’ve already used FSAs once ..

NP ì rules from the WSJ grammar become a single DFA

NP ì ADJP ADJP JJ JJ NN NNS
| ADJP DT NN
| ADJP JJ NN
| ADJP JJ NN NNS
| ADJP JJ NNS
| ADJP NN
| ADJP NN NN
| ADJP NN NNS
| ADJP NNS
| ADJP NPR
| ADJP NPRS
| DT
| DT ADJP
| DT ADJP , JJ NN
| DT ADJP ADJP NN
| DT ADJP JJ JJ NN
| DT ADJP JJ NN
| DT ADJP JJ NN NN

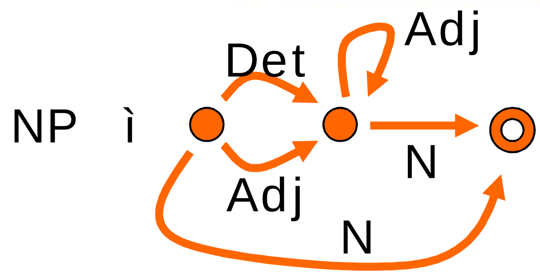
etc.

regular expression

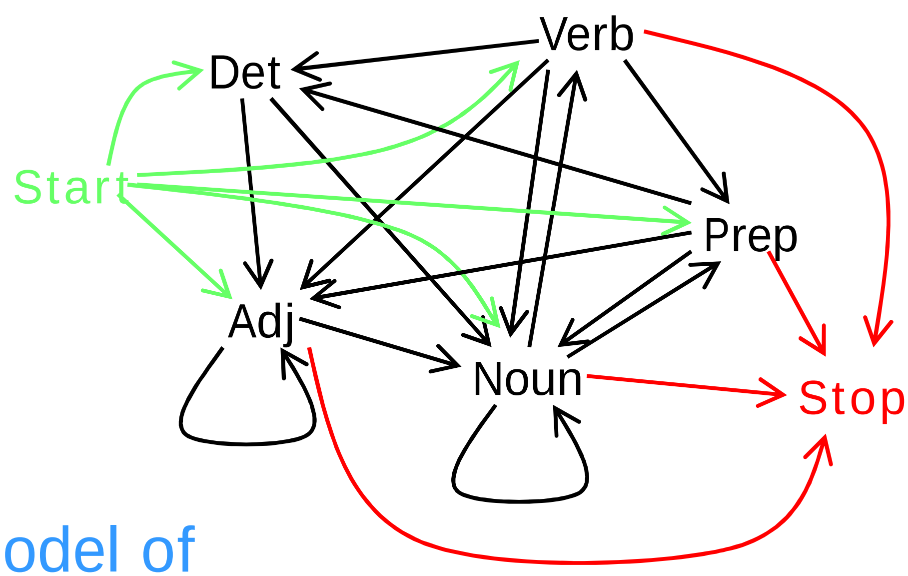


But where can we put our weights?

- CFG / RTN



- bigram model of words or tags (first-order Markov Model)



- Hidden Markov Model of words and tags together??

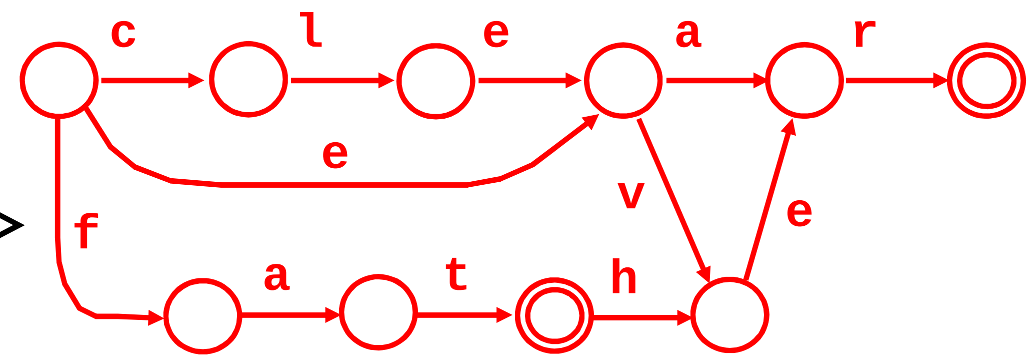
Another useful FSA ...

Wordlist

```
clear
clever
ear
ever
fat
father
```

compile

Network



/usr/dict/words

25K words
206K chars

0.6 sec

FSM

17728 states,
37100 arcs

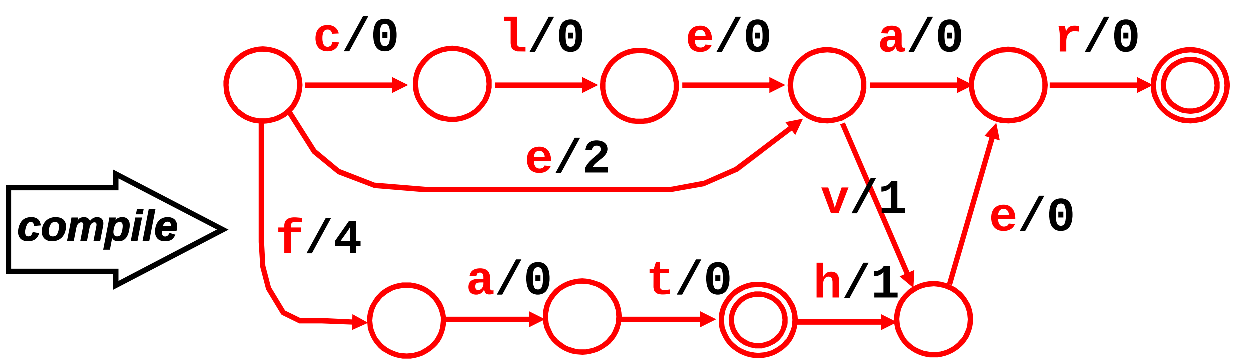
slide courtesy of L. Karttunen (modified)

Weights are useful here too!

Wordlist

clear	0
clever	1
ear	2
ever	3
fat	4
father	5

Network



Computes a perfect hash!

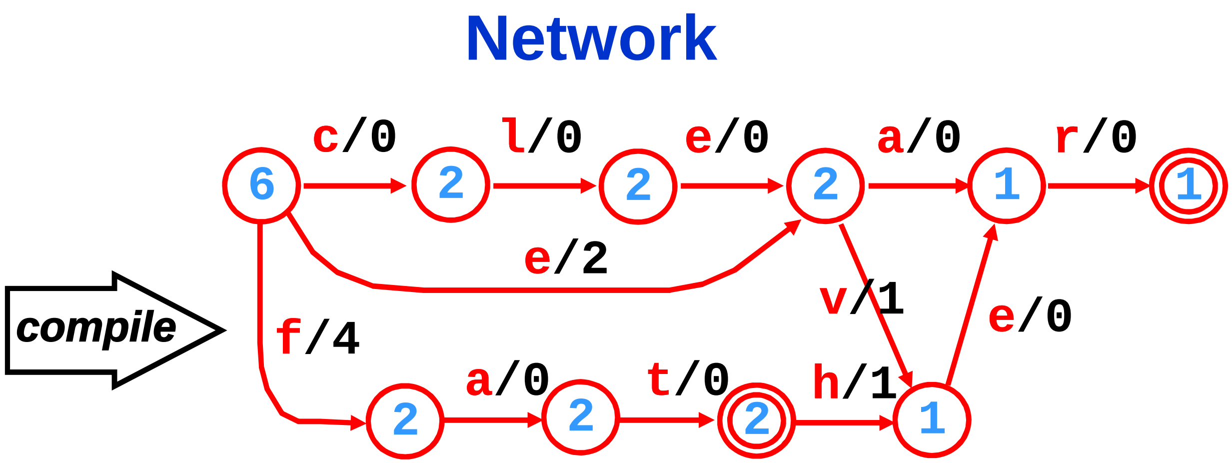
Sum the weights along a word's accepting path.

slide courtesy of L. Karttunen (modified)

Example: Weighted acceptor

Wordlist

clear	0
clever	1
ear	2
ever	3
fat	4
father	5



- Compute number of paths from each state (Q: how?)
  A: recursively, like DFS
- Successor states partition the path set
- Use offsets of successor states as arc weights
- Q: Would this work for an arbitrary numbering of the words?

Example: Unweighted transducer

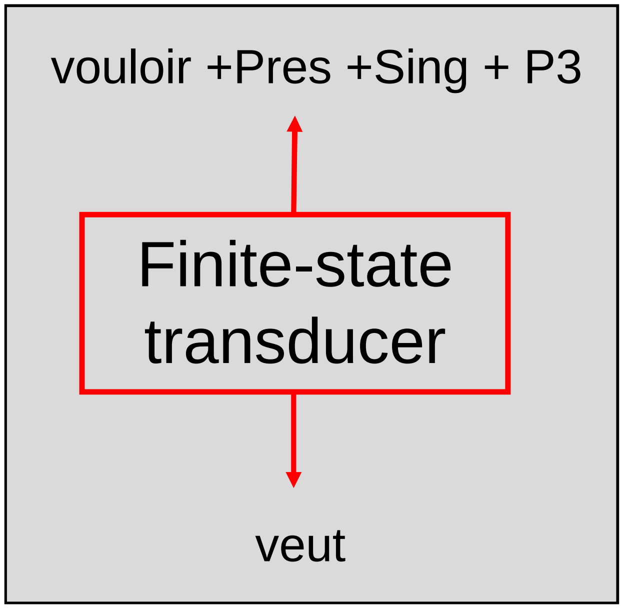
VP [head=vouloir,...]

V[head=vouloir, ...
tense=Present,
num=SG, person=P3]

veut

the problem
of **morphology**
("word shape") -
an area of linguistics

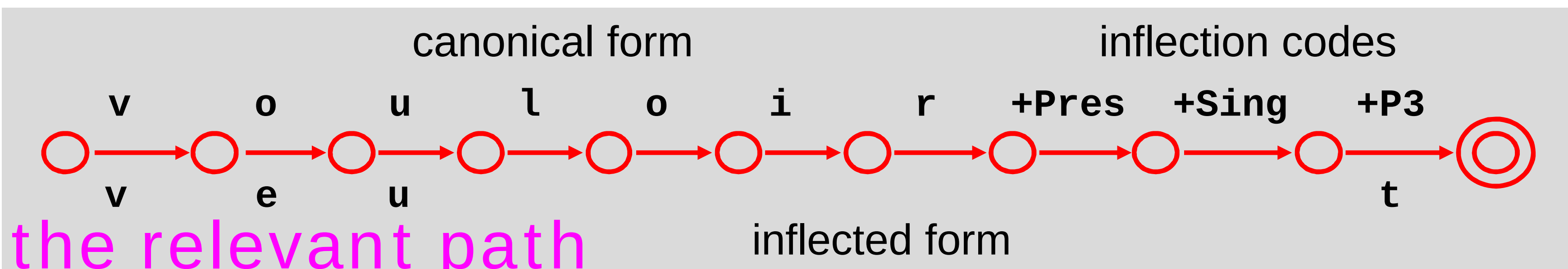
Example: Unweighted transducer



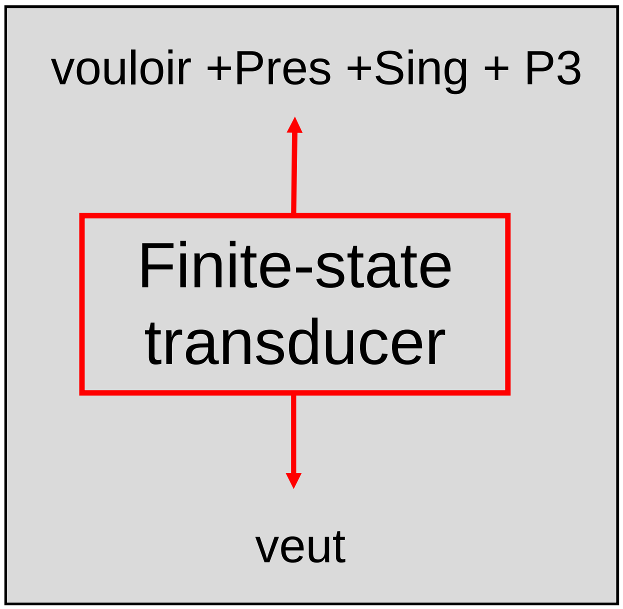
VP [head=vouloir,...]

V[head=vouloir, ...
tense=Present,
num=SG, person=P3]

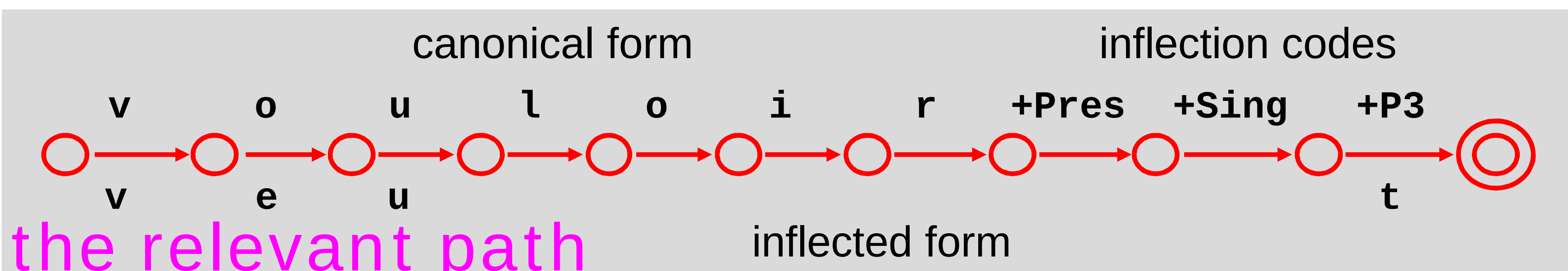
veut



Example: Unweighted transducer



- Bidirectional: generation or analysis
- Compact and fast
- Xerox sells for about 20 languges including English, German, Dutch, French, Italian, Spanish, Portuguese, Finnish, Russian, Turkish, Japanese, ...
- Research systems for many other languages, including Arabic, Malay



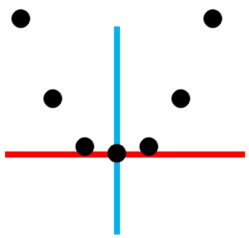
What is a function?

- Formally, a set of <input, output> pairs where each input ∈ domain, output ∈ co-domain.
- Moreover, ∀x ∈ domain, ∃ exactly one y such that <x,y> ∈ the function.

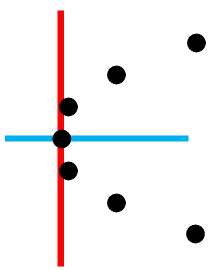
domain co-domain

- square: int → int
  = { <0,0>, <1,1>, <-1,1>, <2,4>, <-2,4>, <3,9>, <-3,9>, ... }
- domain(square) = {0, 1, -1, 2, -2, 3, -3, ...}
- range(square) = {0, 1, 4, 9, ...}

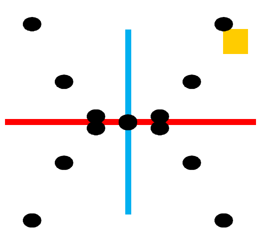
What is a relation?



- square: int → int
  = { <0,0>, <1,1>, <-1,1>, <2,4>, <-2,4>, <3,9>, <-3,9>, ... }



- inverse(square): int → int
  = { <0,0>, <1,1>, <1,-1>, <4,2>, <4,-2>, <9,3>, <9,-3>, ... }
- Is inverse(square) a function?
  - 0 ↦ {0}   9 ↦ {3,-3}   7 ↦ {}   -1 ↦ {}



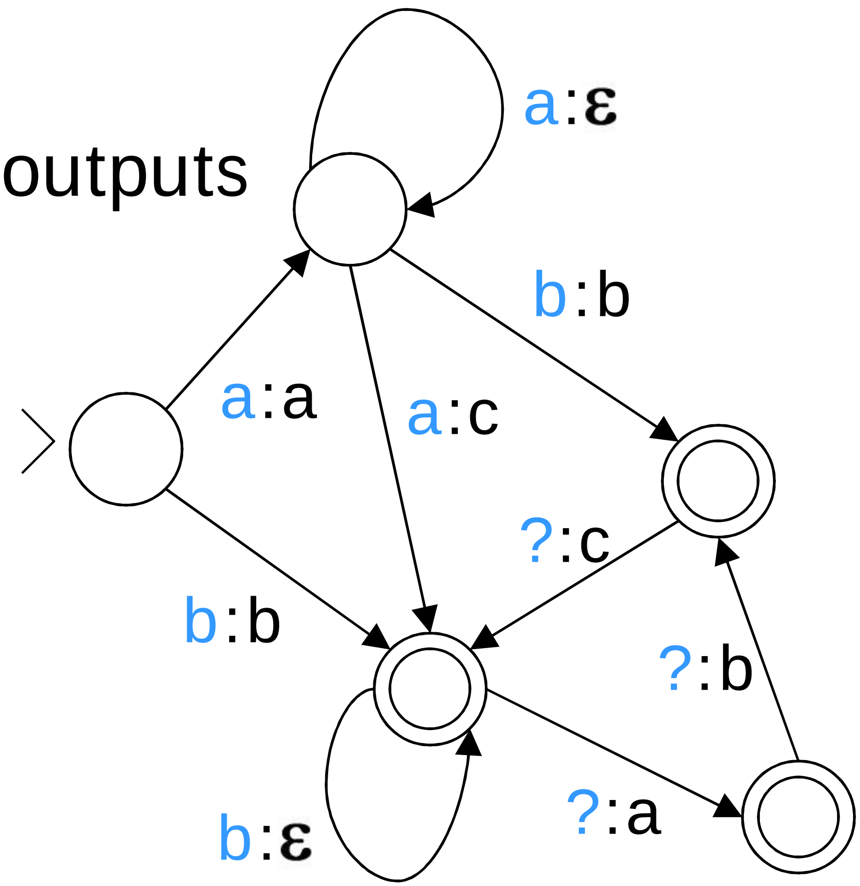
- No - we need a more general notion!
  - A relation is any set of <input, output> pairs

Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs

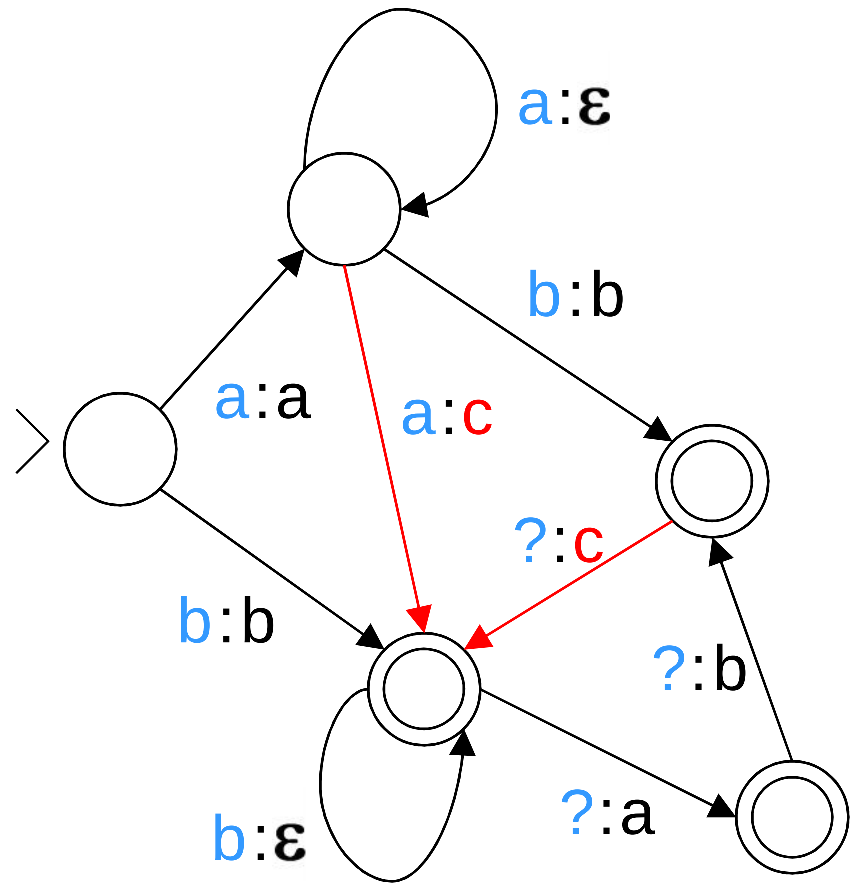
- b → {b}   a → {}
- aaaaa → {ac, aca, acab, acabc}

- Invertible?
- Closed under composition?



Regular Relation (of strings)

- Can weight the arcs: → vs. →
- b → {b}   a → {}
- aaaaa → {ac, aca, acab, acabc}
- How to find best outputs?
  - For aaaaa?
  - For all inputs at once?



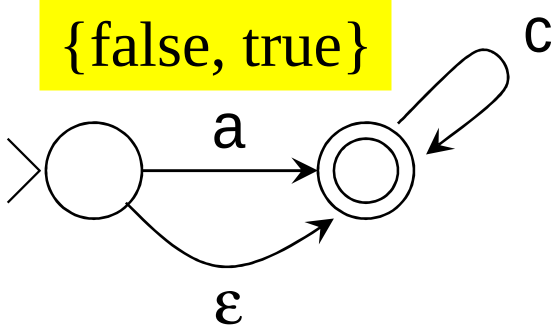
Function from strings to ...

Acceptors (FSAs)

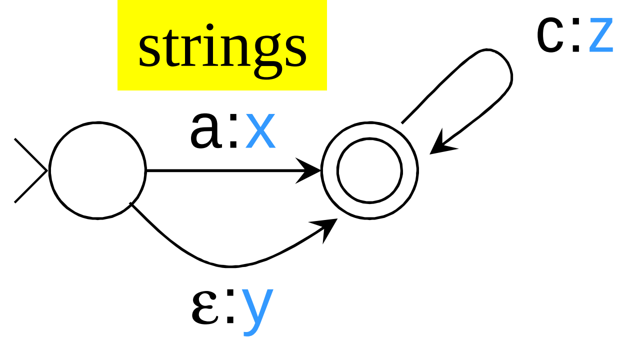
Transducers (FSTs)

Unweighted

{false, true}

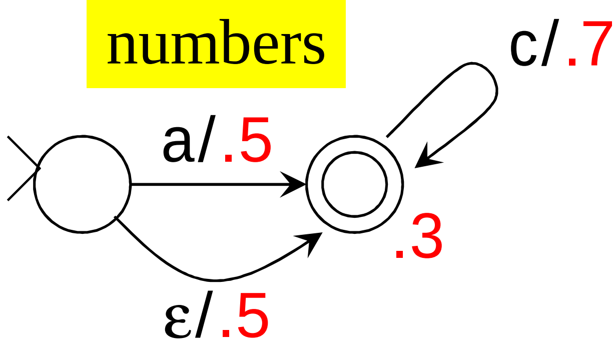


strings

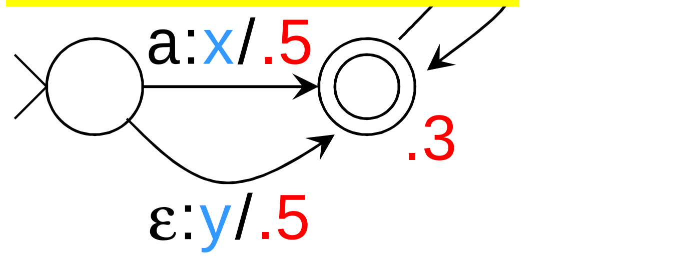


Weighted

numbers



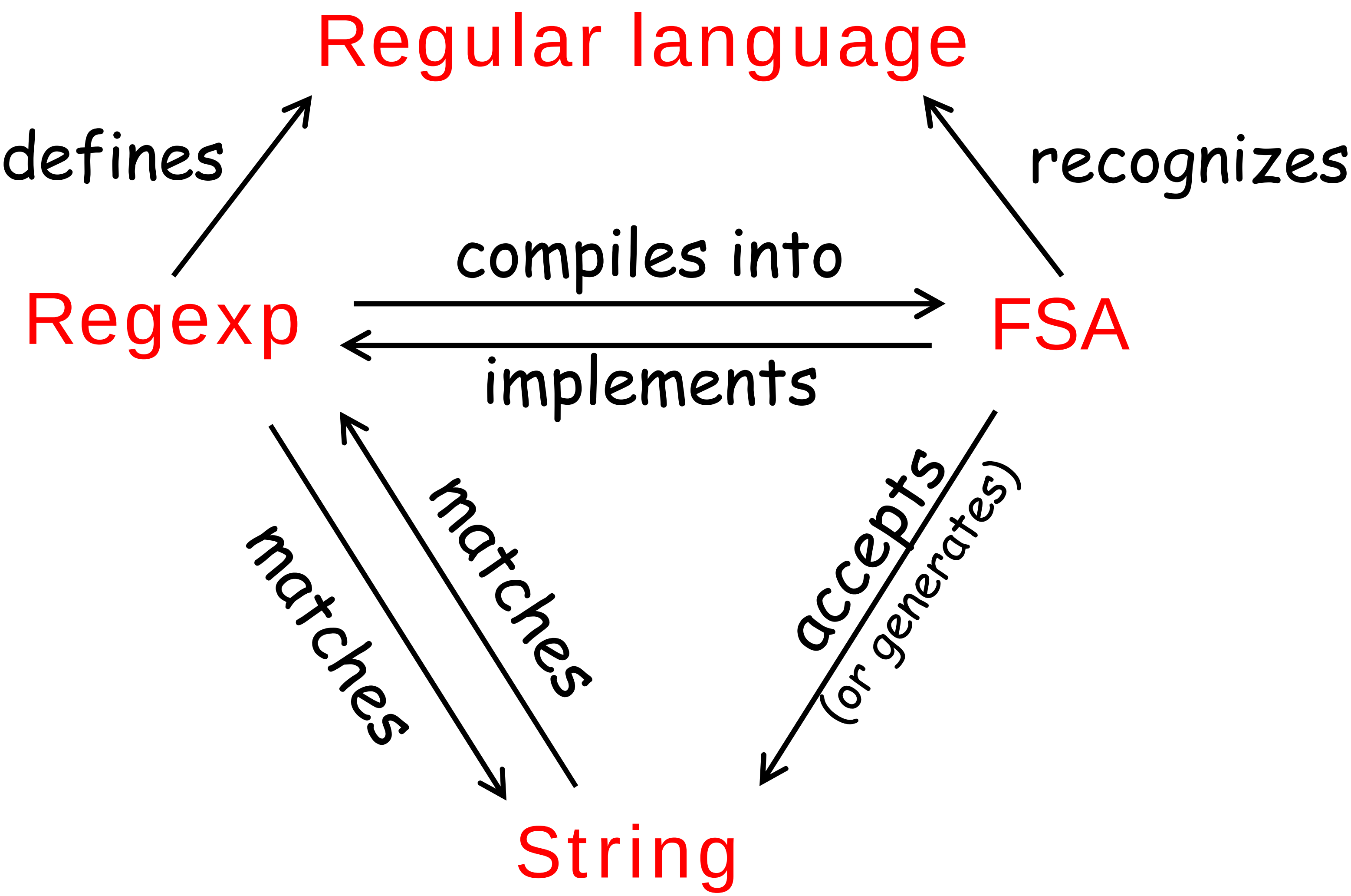
(string, num) pairs



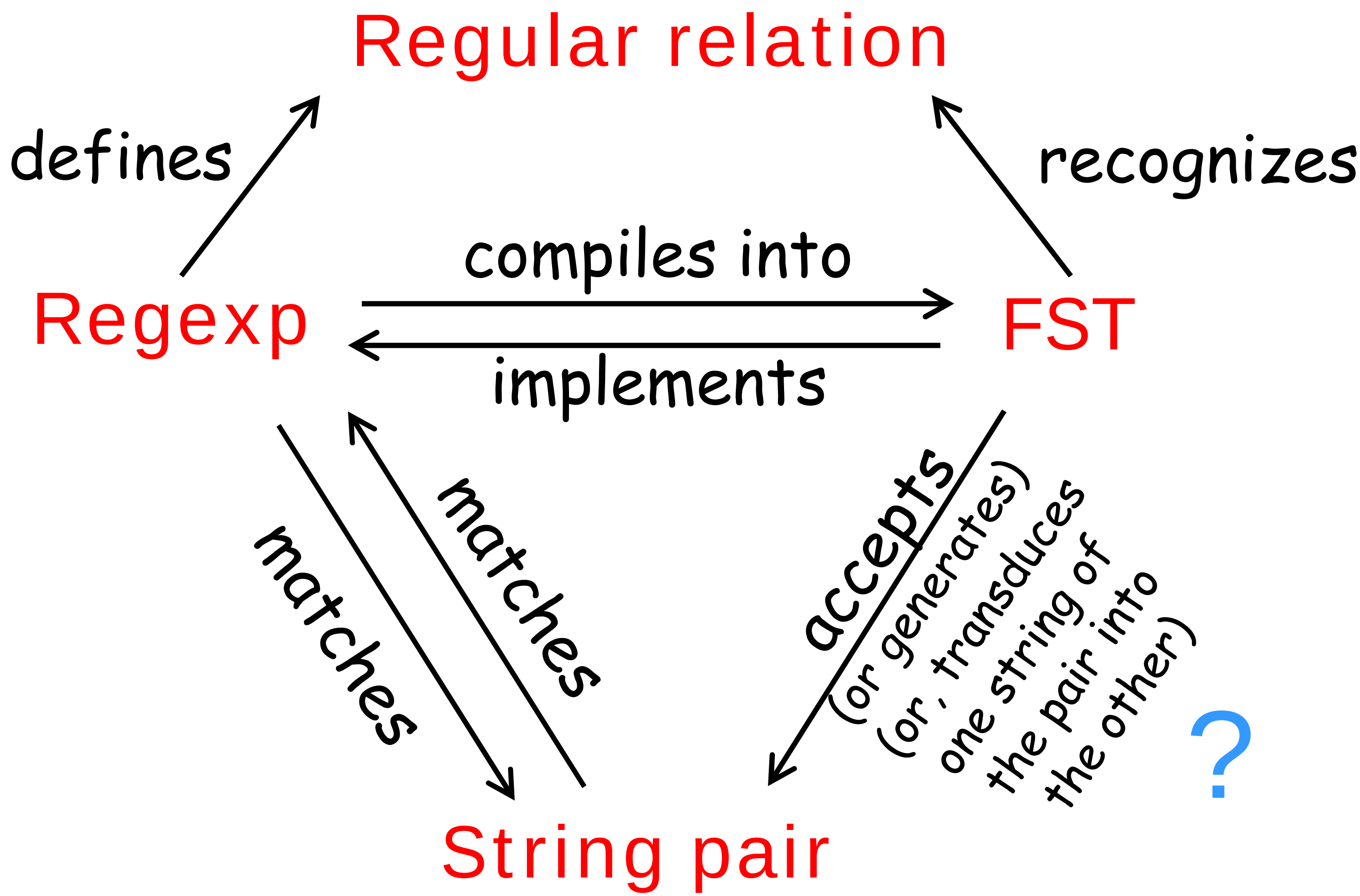
Sample functions

	Acceptors (FSAs)	Transducers (FSTs)
Unweighted	{false, true} Grammatical?	strings Markup Correction Translation
Weighted	numbers How grammatical? Better, how probable?	(string, num) pairs Good markups Good corrections Good translations

Terminology (acceptors)



Terminology (transducers)



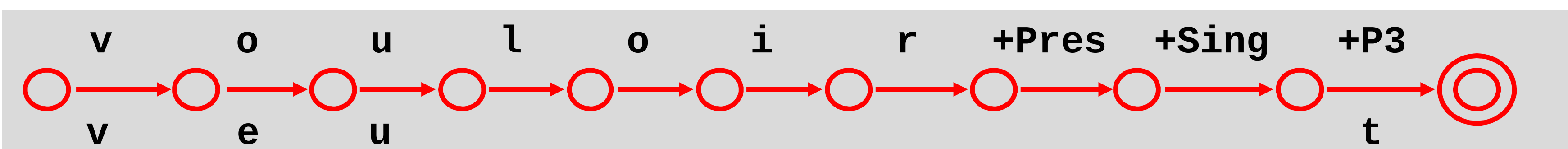
Perspectives on a Transducer

- Remember these CFG perspectives:

3 views of a context-free rule

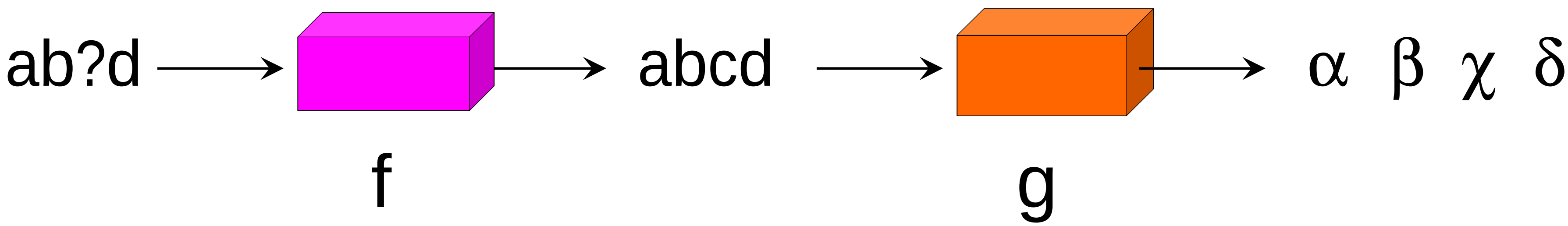
- generation (production): S → NP VP (randsent)
- parsing (comprehension): S ← NP VP (parse)
- verification (checking): S = NP VP

- Similarly, 3 views of a transducer:
  - Given 0 strings, generate a new string pair (by picking a path)
  - Given one string (upper or lower), transduce it to the other kind
  - Given two strings (upper & lower), decide whether to accept the pair

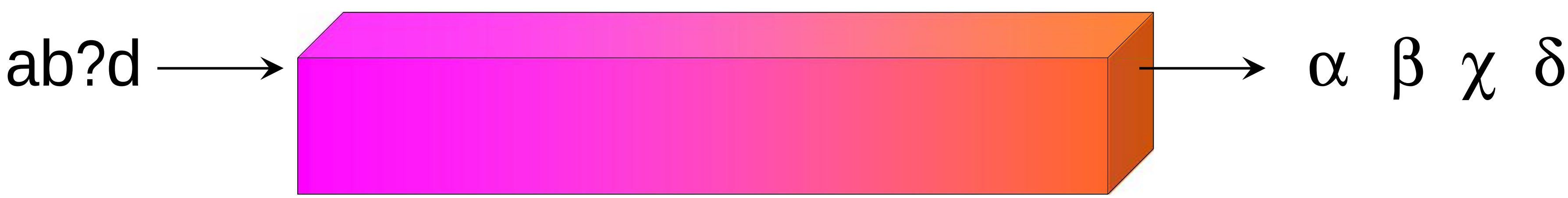


FST just defines the regular relation (mathematical object: set of pairs).
What's "input" and "output" depends on what one asks about the relation.
The 0, 1, or 2 given string(s) constrain which paths you can use.

Functions



Functions



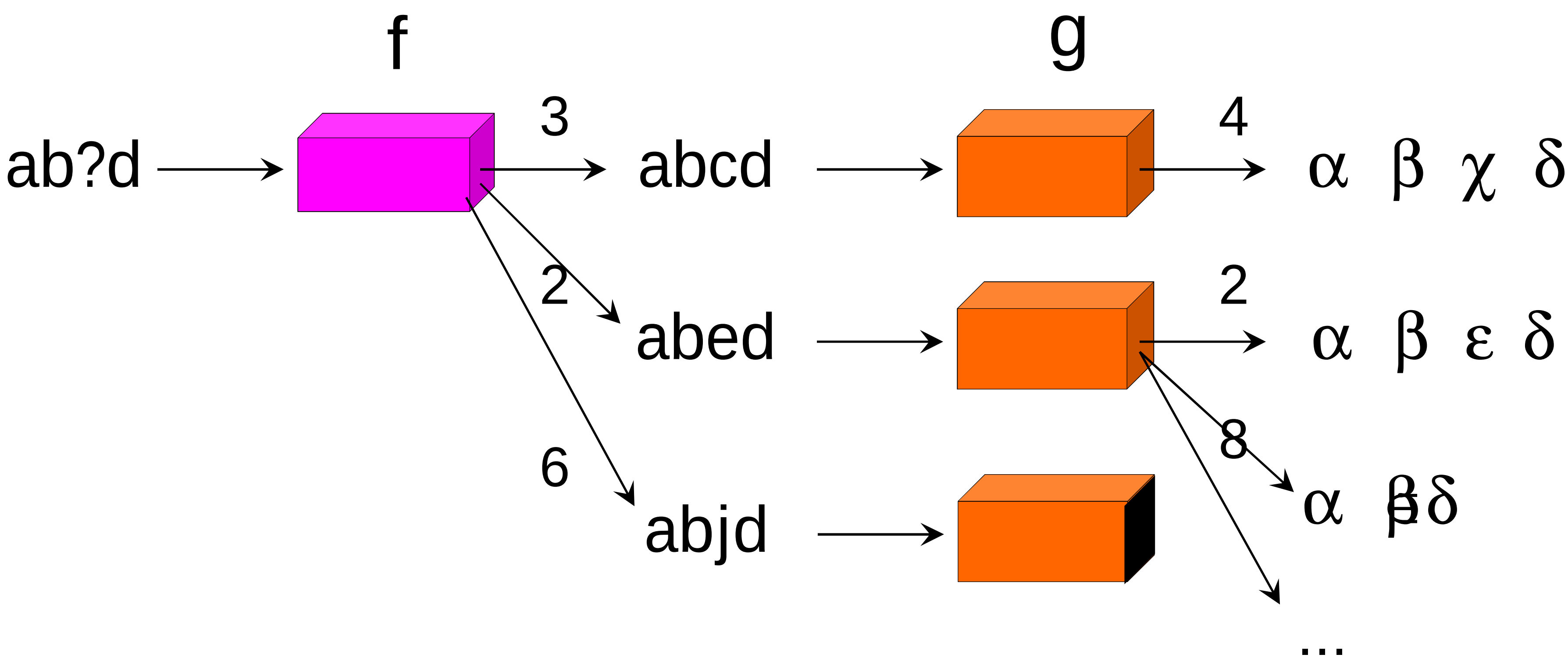
Function composition: f ° g

[first f, then g – intuitive notation, but opposite of the traditional math notation]

Like the Unix pipe: `cat x | f | g > y`

Example: Pass the input through a sequence of ciphers

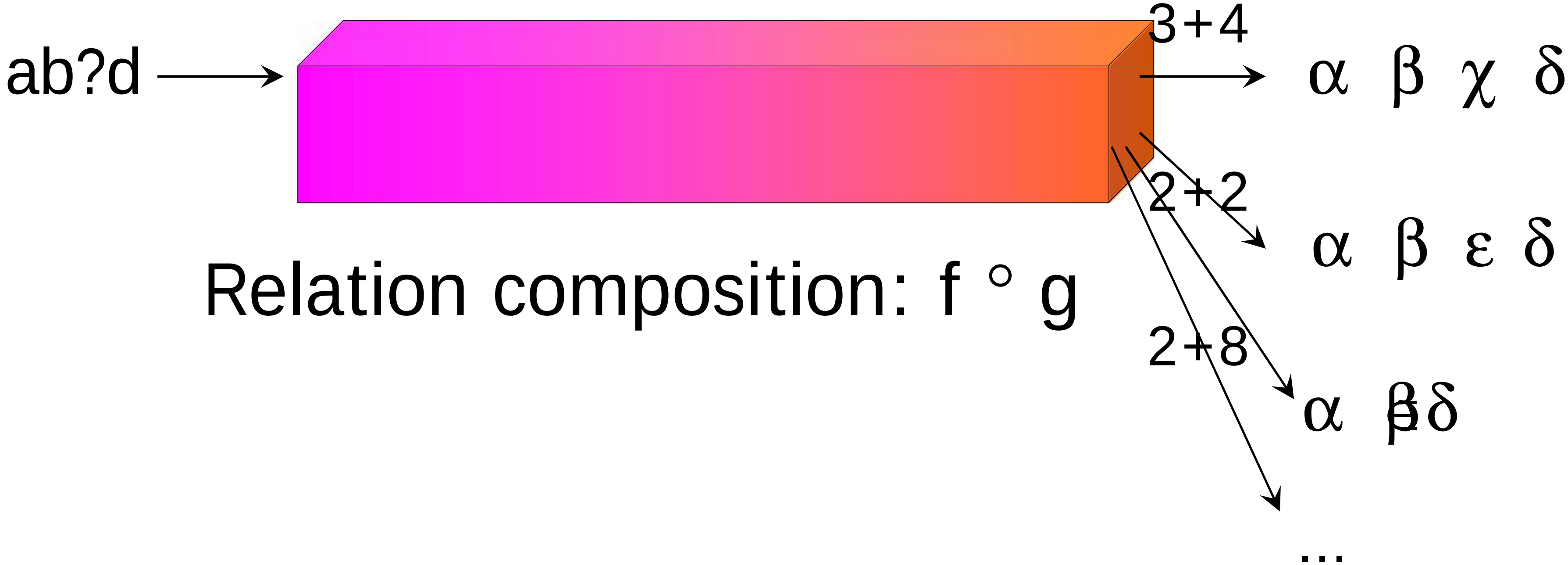
From Functions to Relations



From Functions to Relations

Relation composition: f ° g

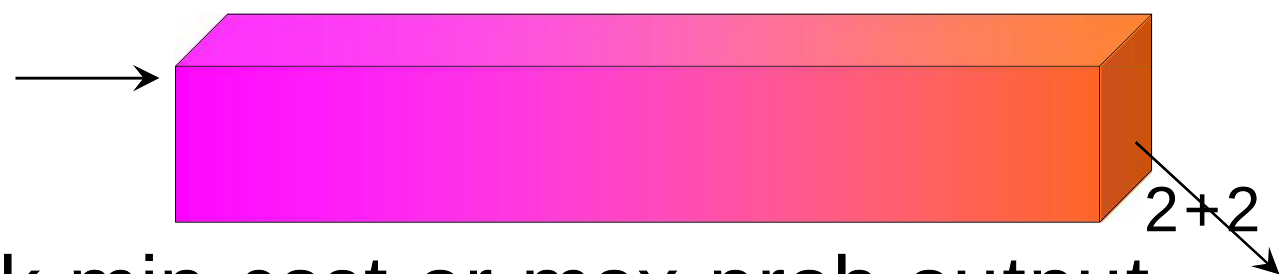
From Functions to Relations



Relation composition: f ° g

From Functions to Relations

ab?d

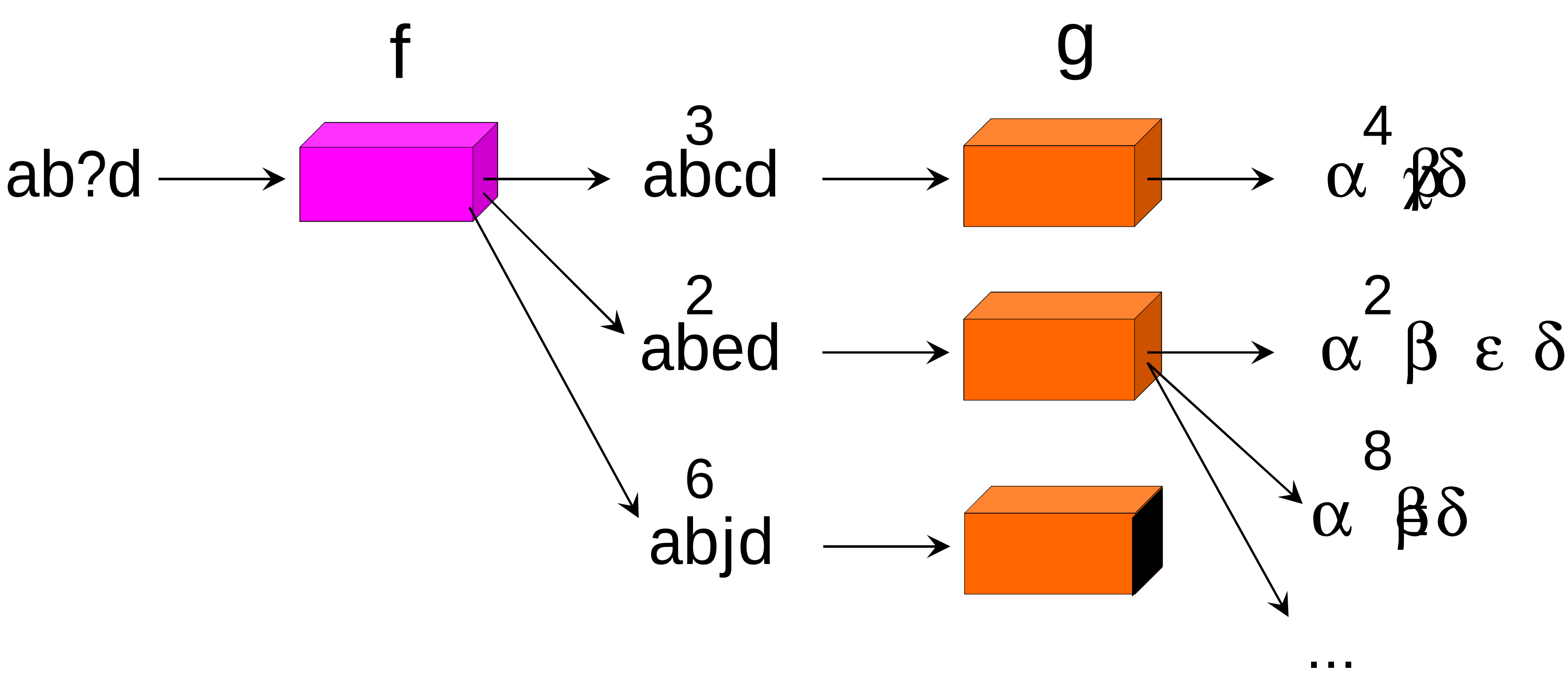


α β ε δ

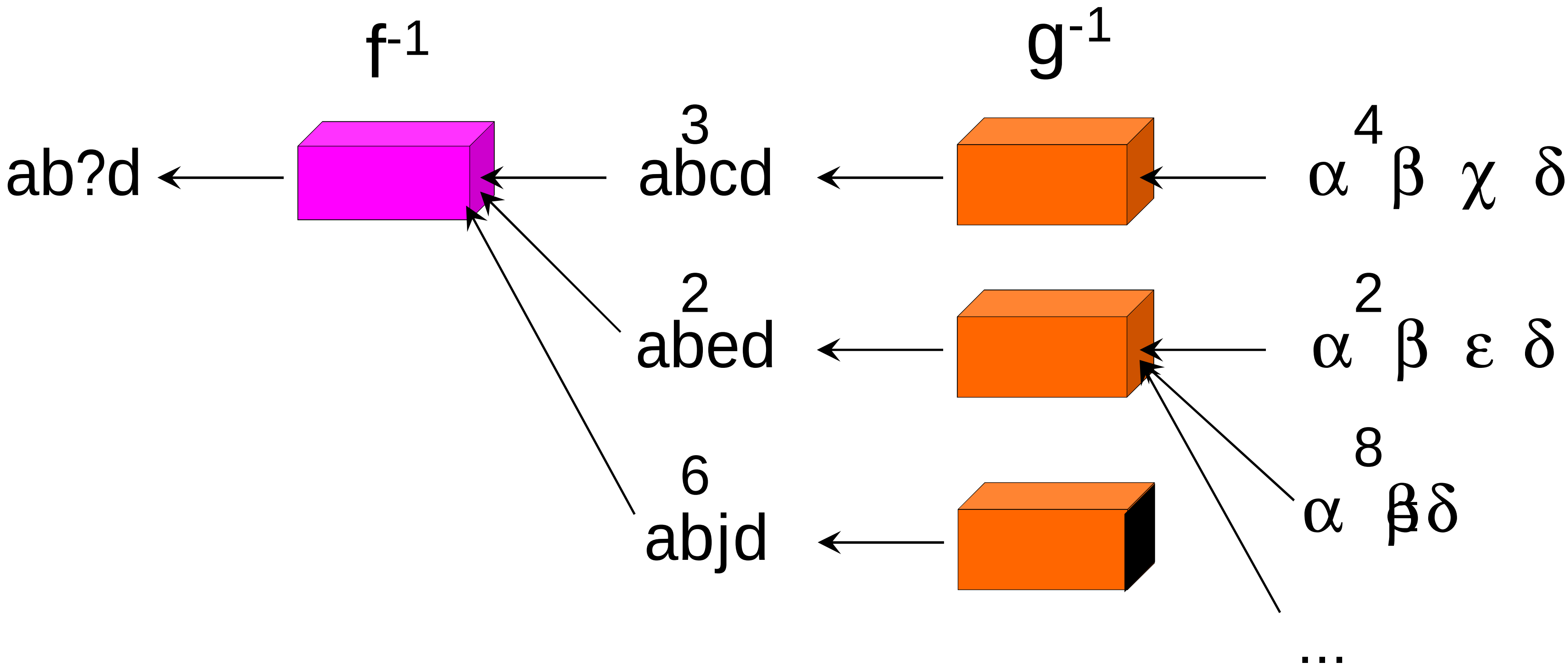
Pick min-cost or max-prob output

Often in NLP, all of the functions or relations involved can be described as finite-state machines, and manipulated using standard algorithms.

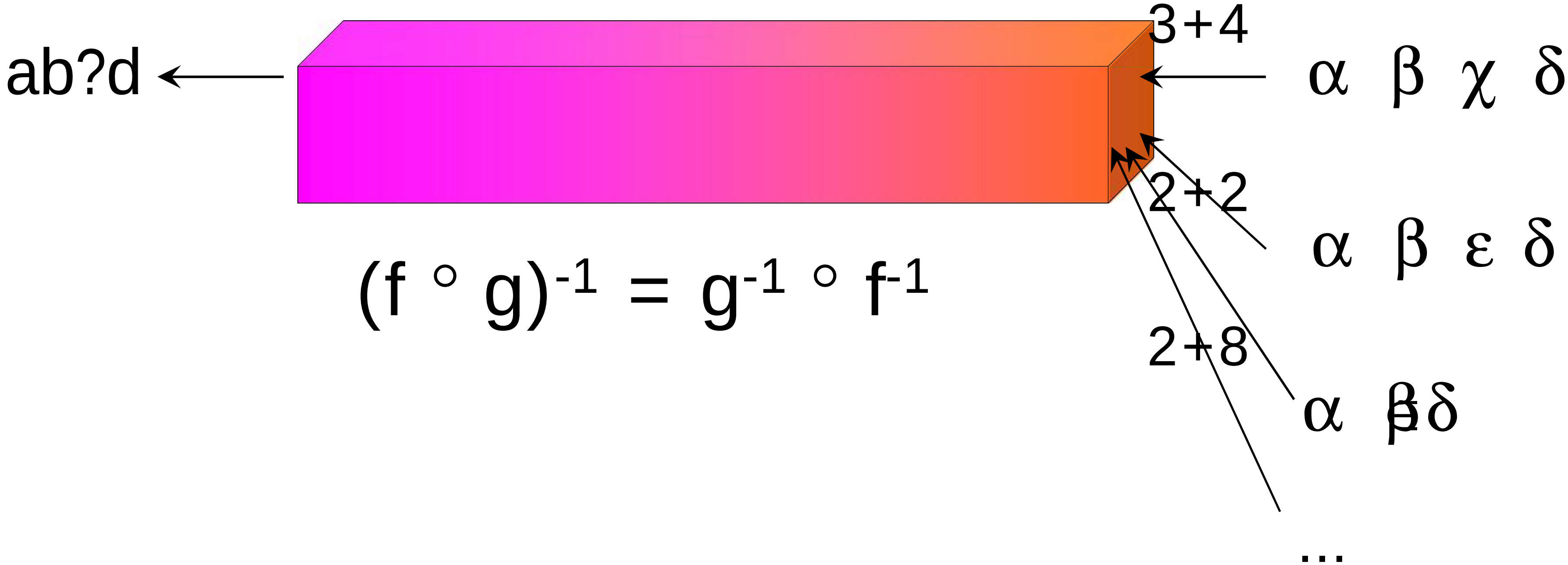
Inverting Relations



Inverting Relations

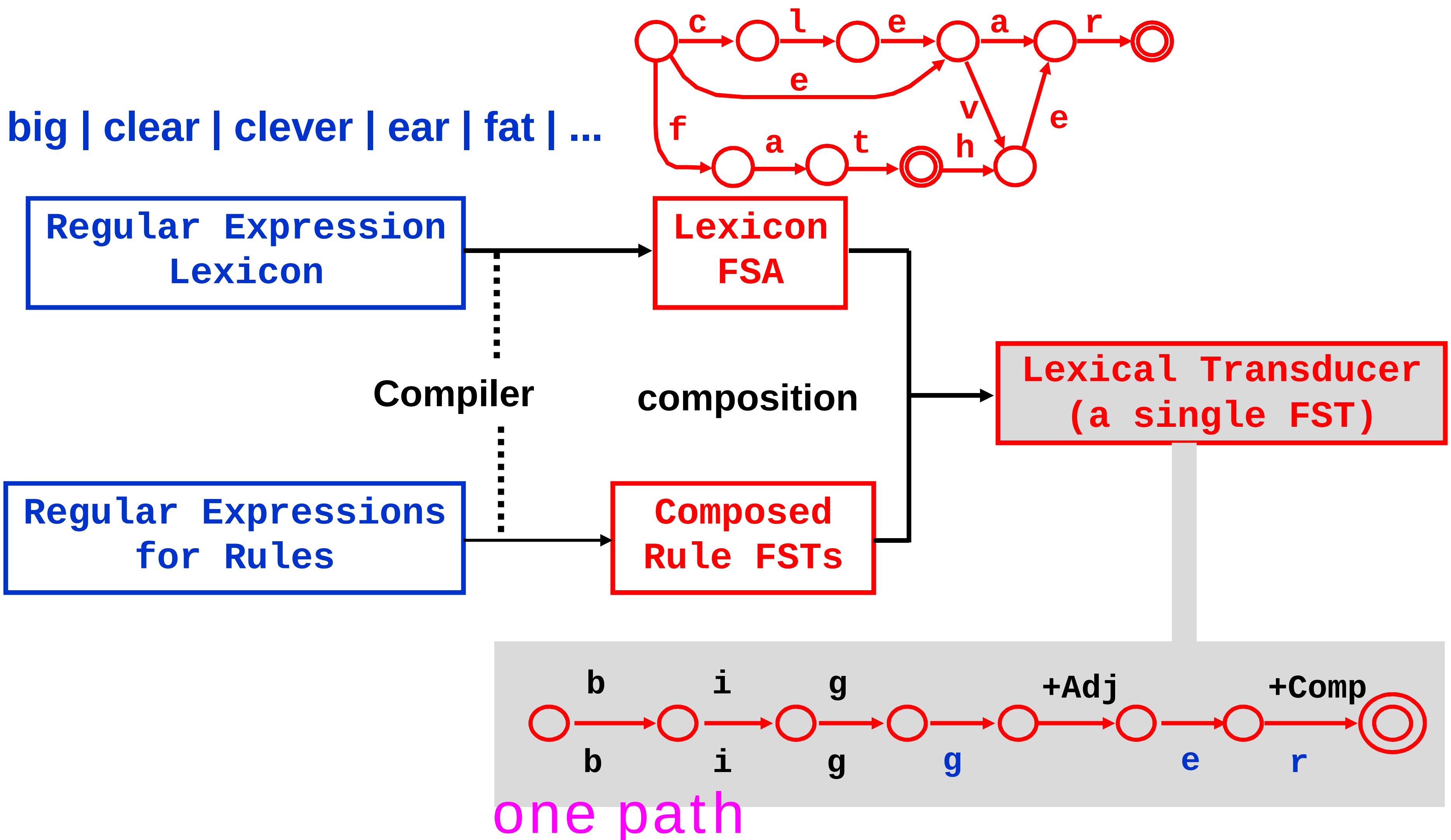


Inverting Relations



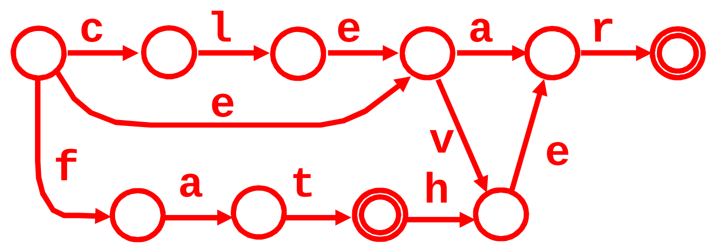
$$(f \circ g)^{-1} = g^{-1} \circ f^{-1}$$

Building a lexical transducer



Building a lexical transducer

big | clear | clever | ear | fat | ...



Regular Expression Lexicon → **Lexicon FSA**

- Actually, the lexicon must contain elements like
  big +Adj +Comp
- So write it as a more complicated expression:
  (big | clear | clever | fat | ...) +Adj (£ | +Comp | +Sup) ← ***adjectives***
  | (ear | father | ...) +Noun (+Sing | +Pl) ← ***nouns***
  | ... ← ***...***
- Q: Why do we need a lexicon at all?

slide courtesy of L. Karttunen (modified)

Weighted version of transducer: Assigns a weight to each string pair