Regular Languages

Natural Language Processing
CS 6120—Spring 2014
Northeastern University

David Smith
with material from Jason Eisner, Andrew McCallum,
and Lari Karttunen

Brief History: 1950s

- Early NLP on machines less powerful than pocket calculators
  - E.g., how to compress a word list into memory
- Foundational work on automata, formal languages, information theory
- First speech systems (Bell Labs)
- Machine translation heavily funded by military, basically just word substitution
- Little formalization of syntax, semantics, pragmatics

Brief History: 1960s

- ALPAC report (Alvey, 1966) ends funding for MT in U.S.
  - Lack of practical results, recommends basic research
- ELIZA and other early AI dialogue systems
  - Risibly easy Turing tests
- Early corpora: Brown Corpus (Kučera & Francis)

Brief History: 1970s

- Winograd's SHRDLU (1971): existence proof of NLP (in tangled Lisp code)
  - Interpreted language about "blocks world"
    - Which cube is sitting on the table?
    - *The large green one which supports the red pyramid.*
    - Is there a large block behind the pyramid?
    - *Yes, three of them. A large red one, a large green cube, and the blue one.*
    - Put a small one onto the green cube which supports a pyramid.
    - *OK.*
- Hidden Markov models for speech recognition

Brief History: 1980s

- Procedural → declarative
  - Grammars, logic programming
  - Separation of processing (parser) from description of linguistic knowledge
- Representations of meaning: procedural semantics (SHRDLU), semantic nets (Schank), logic (starting in 1970s, Montague, Partee)
- Knowledge representation (Lenat: Cyc, still going!)
- MT in limited domains (METEO)
- HMMs for part-of-speech tagging (independently, Church & DeRose)

Brief History: 1990s

- Probabilistic paradigm shift
  - Speech recognition methods take over the world
- IR-style evaluations take over the world
- Finite-state methods in speech and beyond
- Large amounts of monolingual and multilingual text become available, esp. on WWW
- Classification problems and *ambiguity resolution* (in syntax, lexical semantics, translation, etc.)

Brief History: Now

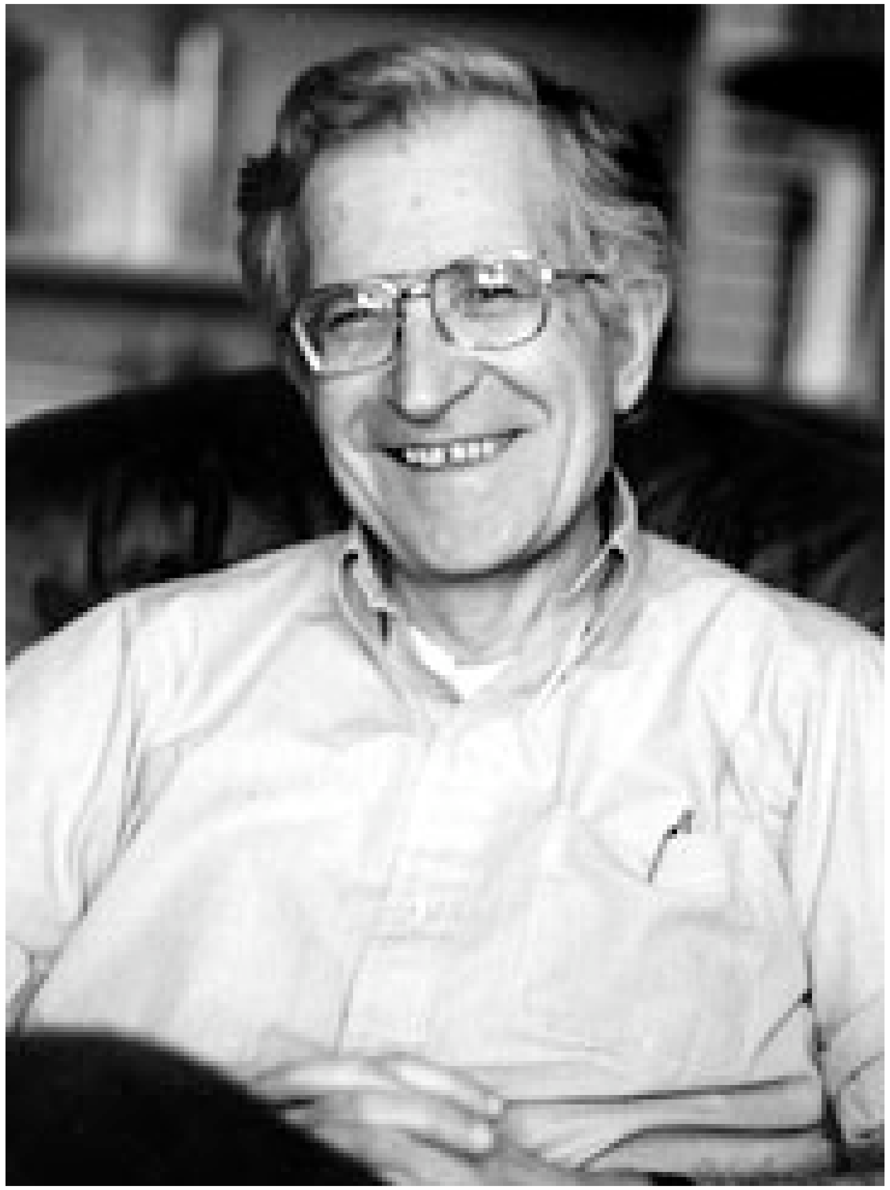
- Even more machine learning
  - Successful unsupervised systems
- Even more data, and tasks
- Widely usable—and used—speech recognition and machine translation
- Widely usable syntactic parsing
- Some usable dialog systems
- Convergence with IR, question answering, probabilistic knowledge representation

Noam Chomsky 1928–

Formal languages
(Chomsky hierarchy)

Generative grammar

Anarcho-Socialist



A Language

- Some sentences in the language
  - ✤ The man took the book.
  - ✤ Colorless green ideas sleep furiously.
  - ✤ This sentence is false.
- Some sentences not in the language
  - ✤ *The girl, the sidewalk, the chalk, drew.
  - ✤ *Backwards is sentence this.
  - ✤ *Je parle anglais.

Languages as Rewriting Systems

- Start with some "non-terminal" symbol **S**
- Expand that symbol, using a **rewrite rule**.
- Keep applying rules until all non-terminals are expanded to terminals.
- The string of terminals is a sentence of the language.

Chomsky Hierarchy

- Let Caps = nonterminals; lower = terminals; Greek = strings of terms/nonterms
- Recursively enumerable (Turing equivalent)
  - Rules: $\alpha \rightarrow \beta$
- Context-sensitive
  - Rules: $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Context-free
  - Rules: $A \rightarrow \alpha$
- Regular (finite-state)
  - Rules: $A \rightarrow aB$; $A \rightarrow a$

Regular Language Example

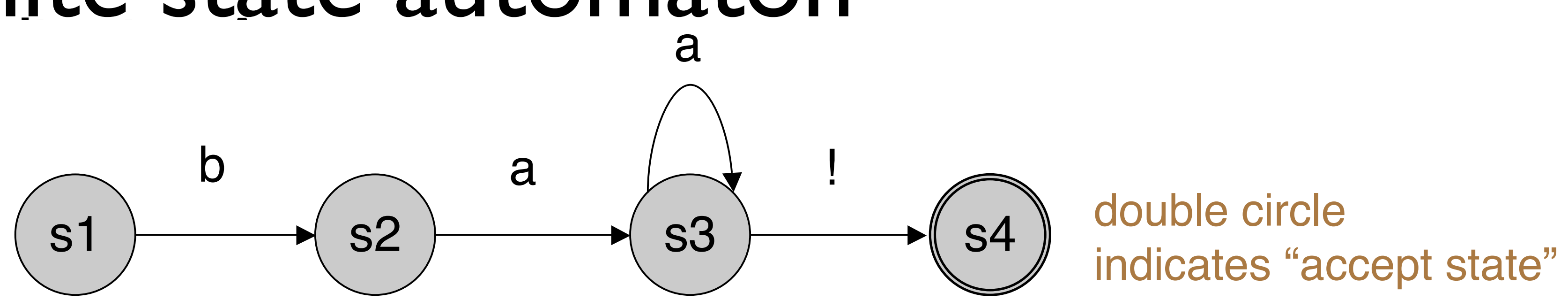
- Nonterminals: S, X
- Terminals: m, o
- Rules:
  - S→mX
  - X→oX
  - X→o
- Start symbol: S

One expansion

S
mX
moX
mooX
mooo

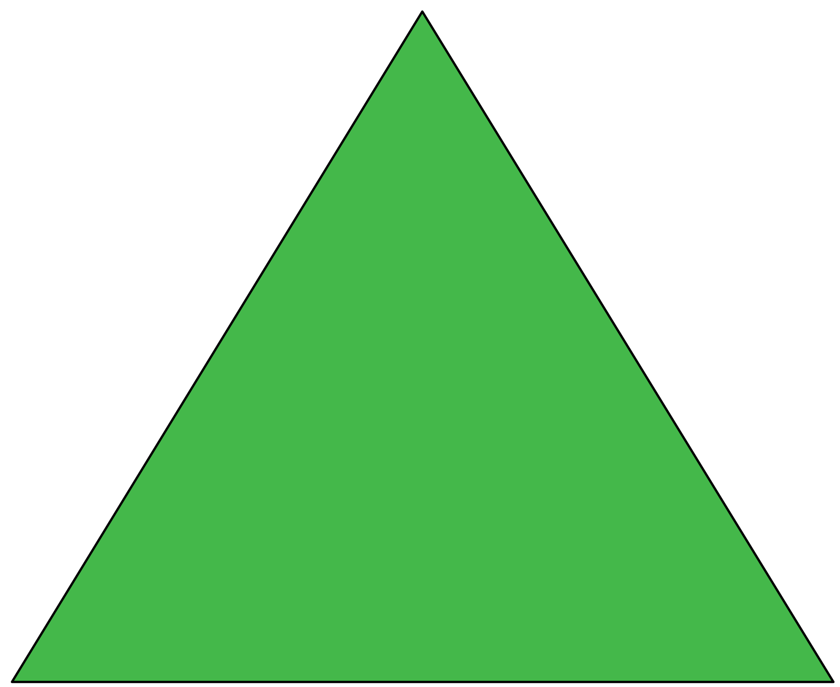
Another Regular Language

- Strings in and not in this language
  - In the language:
    - "ba!", "baa!", "baaaaaaa!"
  - Not in the language:
    - "ba", "b!", "ab!", "bbaaa!", "alibaba!"
- Regular expression: baa*!
- Finite state automaton



Regular Languages

Regular Languages
the accepted strings

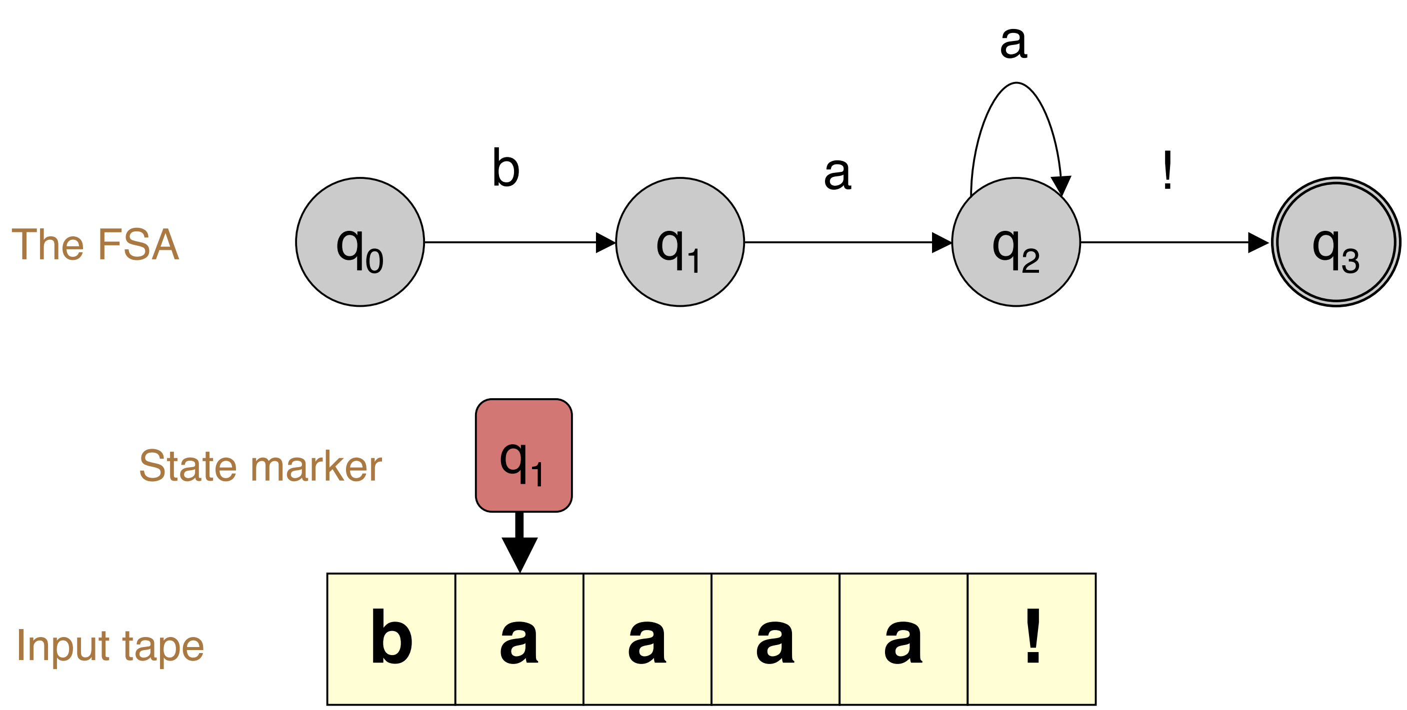


Finite-state Automata
machinery for accepting

Regular Expressions
a way to type the automata

Finite-State Automata

- A (deterministic) finite-state automaton is a 5-tuple $(Q, \Sigma, q_0, F, \delta(q,i))$
  - Q: finite set of states $q_0, q_1, q_2, \ldots, q_N$
  - Σ: finite set of terminals
  - $\delta(q,i)$: transition function (relation if non-deterministic)
  - q_0: start state
  - F: set of final states



Transition Table

State	Input *b*	*a*	*!*
0	1	∅	∅
1	∅	2	∅
2	∅	2	3
3	∅	∅	∅

Regular Expressions

- Two types of characters
- Literal
  - Every "normal" alphanumeric character is an RE, and matches itself
- Meta-characters
  - Special characters that allow you to combine REs in various ways
- Example:
  - **a** matches *a*
  - **a*** matches ε or *a* or *aa* or *aaa* or ...

Regular Expressions

	Pattern	Matches
Concatenation	**`abc`**	*abc*
Disjunction	**`a|b`**	*a* *b*
	`(a|bb)d`	*ad* *bbd*
Kleene star	**`a*`**	ε *a aa aaa ...*
	`c(a|bb)*`	*ca cbba*

The empty string

Regular expressions / FSAs are closed under these operations

Practical Applications

- Word processing find & replace
- Validate fields in database (dates, email, ...)
- Searching for linguistic patterns
- Finite-state machines
    - Language modeling in speech recognition (where things need to be real-time or better)
    - Information extraction
    - Morphology

Syntactic Sugar

	Pattern	Matches
Character Concat	**went**	went
Alternatives	**(go|went)**	go went
	[aeiou]	a o u
disjunc. negation	**[^aeiou]**	b c d f g
wildcard char	**.**	a z &
Loops & skips	**a***	ε a aa aaa ...
one or more	**a+**	a aa aaa
zero or one	**colou?r**	color colour

Syntactic Sugar

- Special characters
  - `\t` tab `\v` vertical tab
  - `\n` newline `\r` carriage return
- Aliases (shorthand)
  - `\d` digits [0-9]
  - `\D` non-digits [^0-9]
  - `\w` alphabetic [a-zA-Z]
  - `\W` non-alphabetic [^a-zA-Z]
  - `\s` whitespace [\t\n\r\f\v]
  - `\w` alphabetic [a-zA-Z]
- Examples
  - `\d+ dollars` 3 dollars, 50 dollars, 982 dollars
  - `\w*oo\w*` food, boo, oodles
- Escape character
  - `\` is the general escape character; e.g. `\.` is not a wildcard, but matches a period `.`
  - if you want to use `\` in a string it has to be escaped `\\`

Syntactic Sugar

- Anchors. AKA, “zero width characters”.
- They match positions in the text.
  - ^ beginning of line
  - **$** end of line
  - **\b** word boundary, i.e. location with **\w** on one side but not on the other.
  - **\B** negated word boundary, i.e. any location that would not match **\b**
- Examples:
  - **\bthe\b** the together

- Counters {1}, {1,2}, {3,}

Syntactic Sugar

- Grouping
  - **`a (good|bad) movie`**
  - **`He said it (again and )*again.`**

- Parens also indicate Registers (saved contents)
  - **`b(\w+)h\1`**
    matches *boohoo* and *baha*, but not *boohaa*
    The digit after the \ indicates which of multiple paren groups, as ordered by when then were opened.

- Grouping without the cost of register saving
  - **`He went (?:this|that) way.`**

Syntactic Sugar

- Non-consuming tests
  - `(?=...)` - Positive lookAHEAD
  - `(?!...)` - Negative lookAHEAD
  - `(?<=...)` - Positive lookBEHIND
  - `(?<!...)` - Negative lookBEHIND
- Example
  - **`(?<=[Hh]e) \w+ed (?=\w+ly)`**

ELIZA (Weizenbaum 1966)

User:	*Men are all alike.*
ELIZA:	IN WHAT WAY?
User:	*They're always bugging us about something or other.*
ELIZA:	CAN YOU THINK OF A SPECIFIC EXAMPLE?
User:	*Well, my boyfriend made me come here.*
ELIZA:	YOUR BOYFRIEND MADE YOU COME HERE.
User:	*He say's I'm depressed much of the time.*
ELIZA:	I'M SORRY TO HEAR THAT YOU ARE DEPRESSED.

Implemented with regular expression substitution!

s/.* I'm (depressed|sad) .*/I AM SORRY TO HEAR THAT YOU ARE \1/

s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE?/

Reading

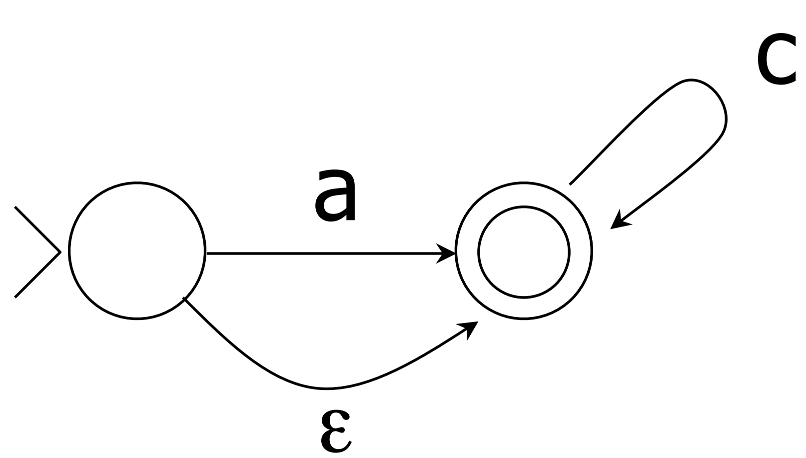
- Karttunen, Chanod, Grefenstette, Schiller. Regular expressions for language engineering. *JNLE*, 1997.

  http://www.stanford.edu/~laurik/publications/jnle-97/rele.pdf

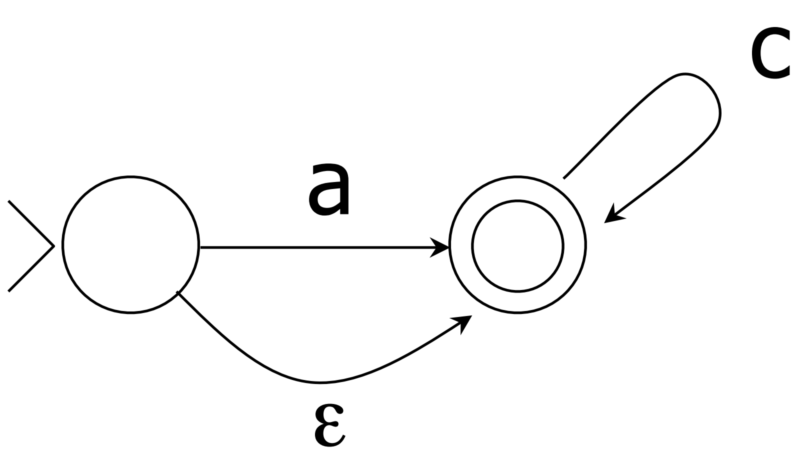
- RE/FSA background: Jurafsky & Martin, c.2

Finite-State Machines: Acceptors and Transducers

Finite state acceptors (FSAs)

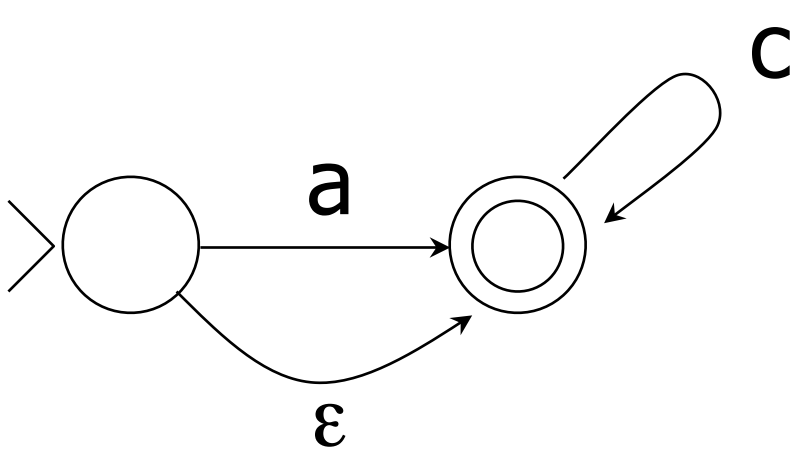


Finite state acceptors (FSAs)



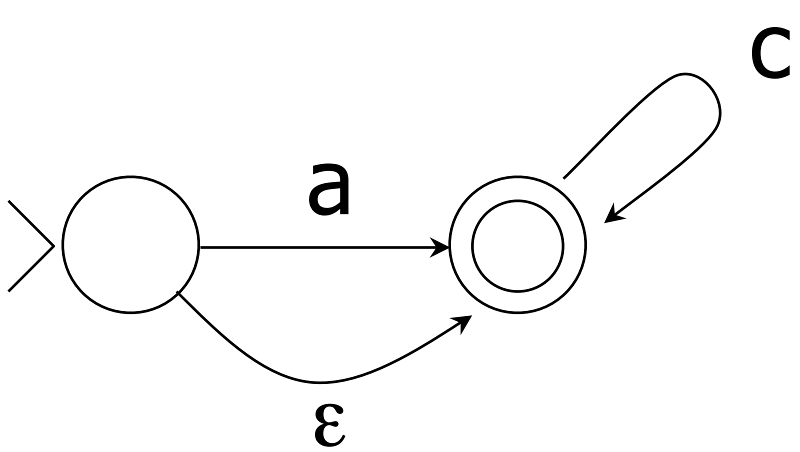
- Regexps

Finite state acceptors (FSAs)



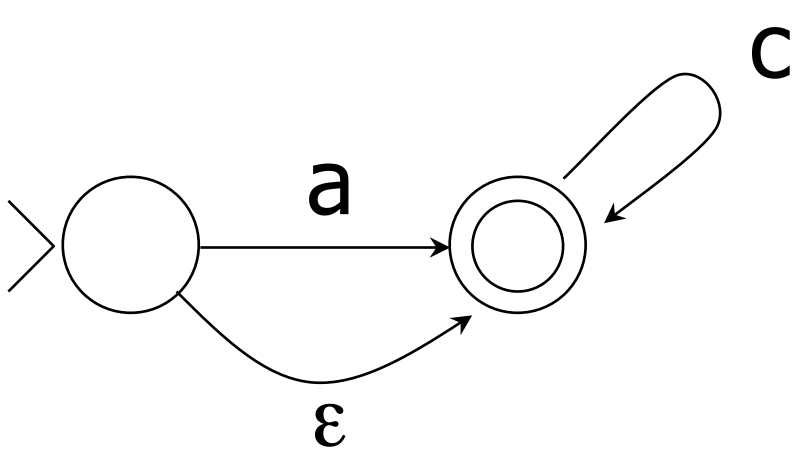
- Regexps
- Union, Kleene *, concat, intersect, complement, reversal

Finite state acceptors (FSAs)



- Regexps
- Union, Kleene *, concat, intersect, complement, reversal
- Determinization, minimization

Finite state acceptors (FSAs)



- Regexps
- Union, Kleene *, concat, intersect, complement, reversal
- Determinization, minimization
- Pumping, Myhill-Nerode

slide courtesy of L. Karttunen (modified)

A useful FSA …

Wordlist

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

`/usr/dict/words`

25K words
206K chars

0.6 sec

FSM

17728 states,
37100 arcs

slide courtesy of L. Karttunen (modified)

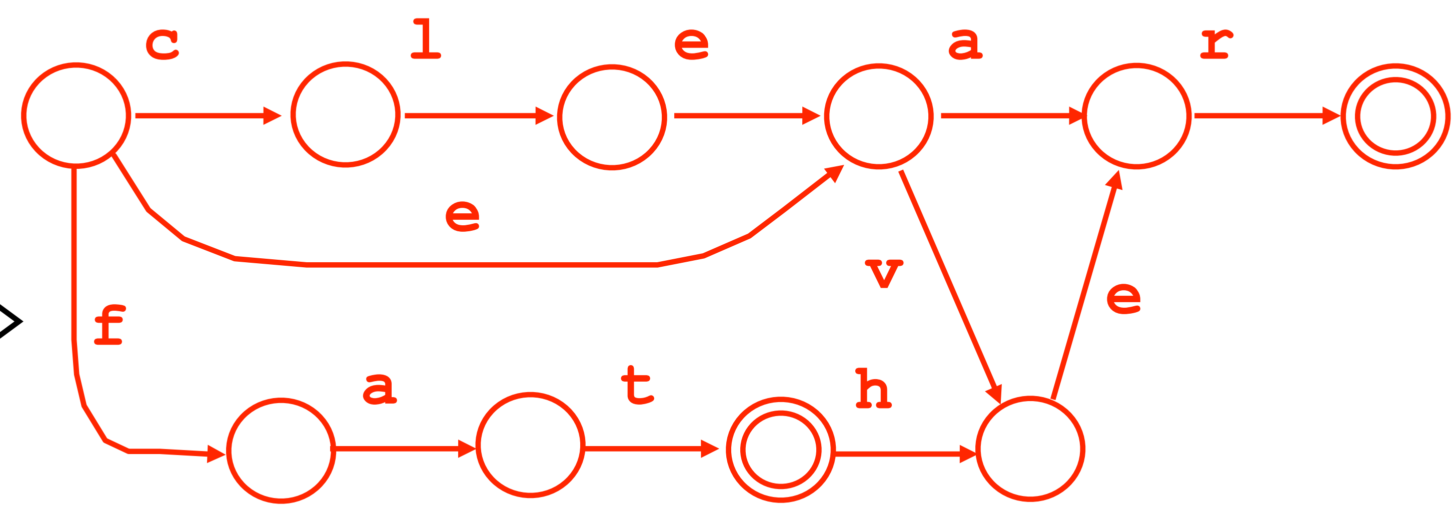
A useful FSA …

Wordlist

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

compile

Network



`/usr/dict/words`

25K words
206K chars

0.6 sec

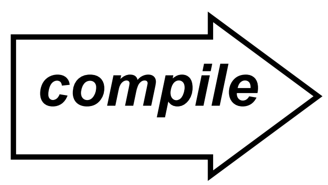
FSM

17728 states,
37100 arcs

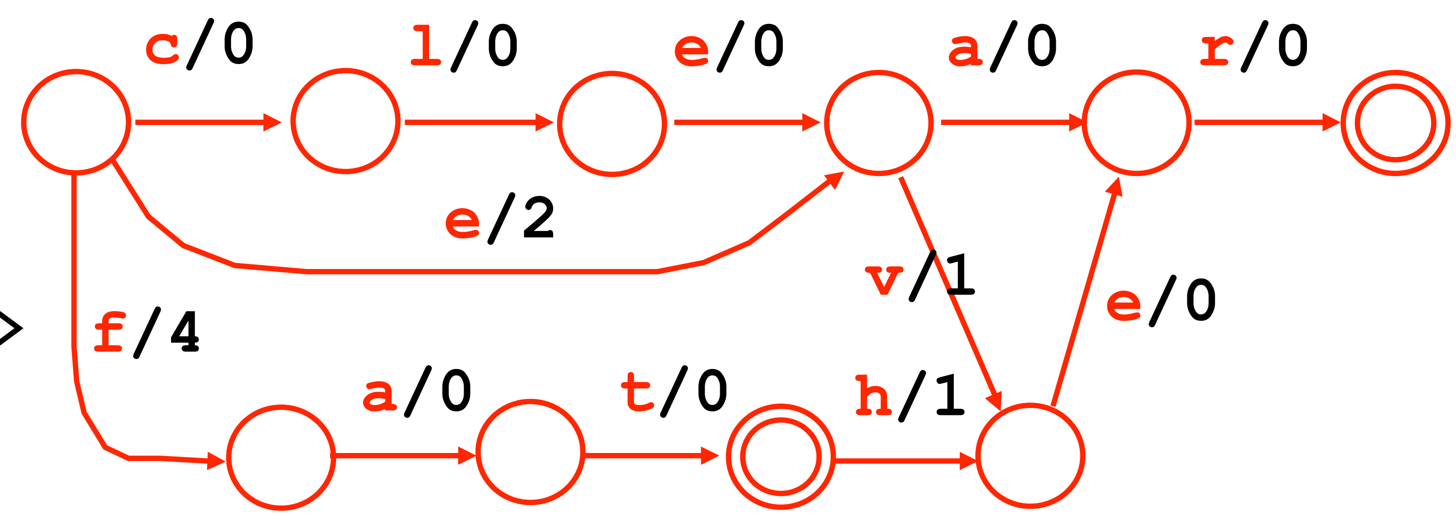
Weights are useful here too!

Wordlist

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



Network



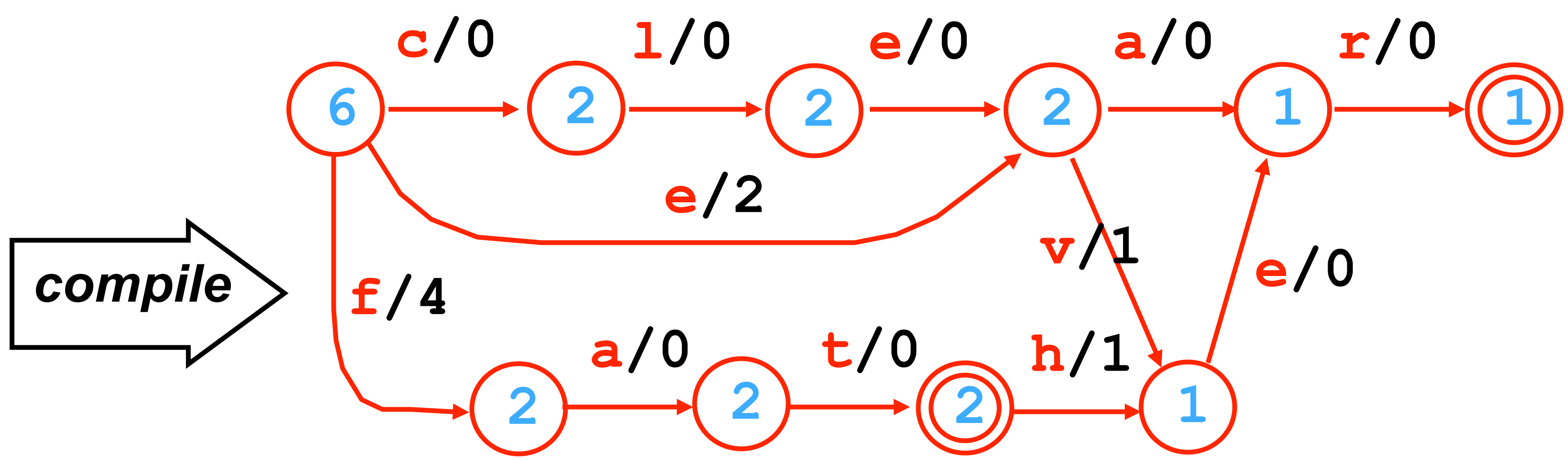
Computes a perfect hash!

Example: Weighted acceptor

Wordlist

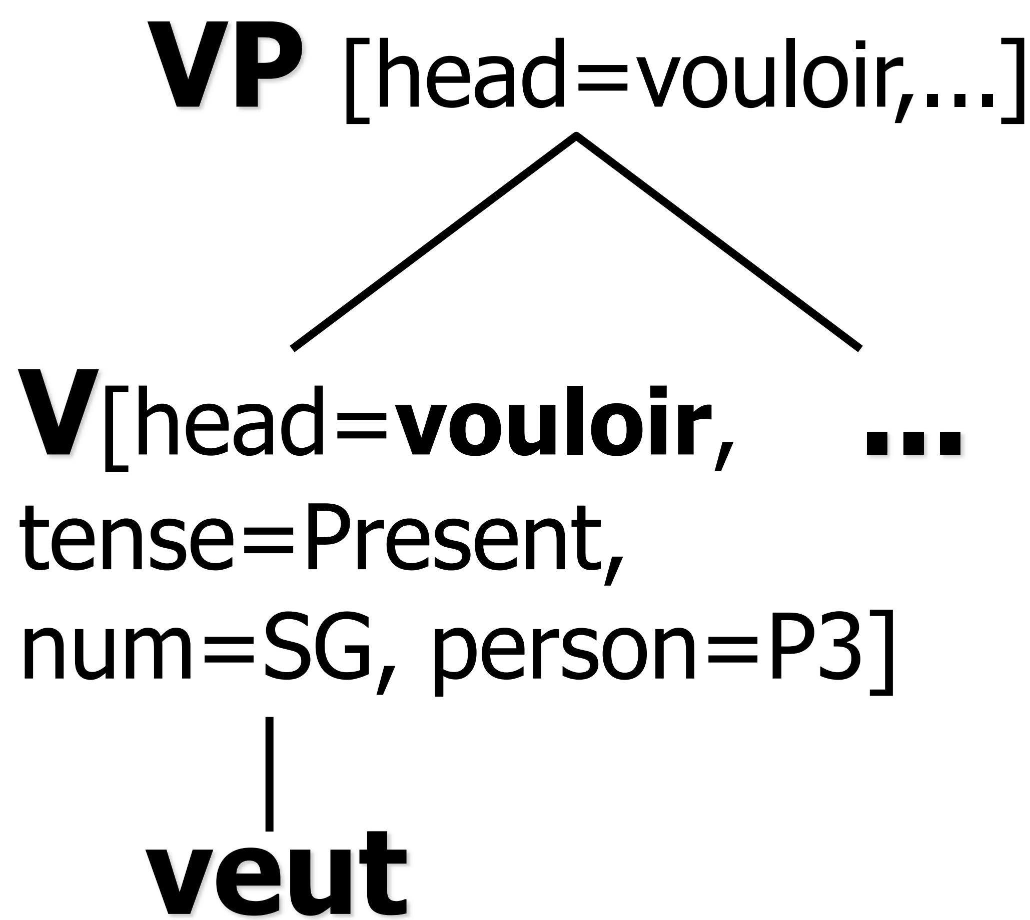
clear	0
clever	1
ear	2
ever	3
fat	4
father	5

Network



- 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



Example: Unweighted transducer

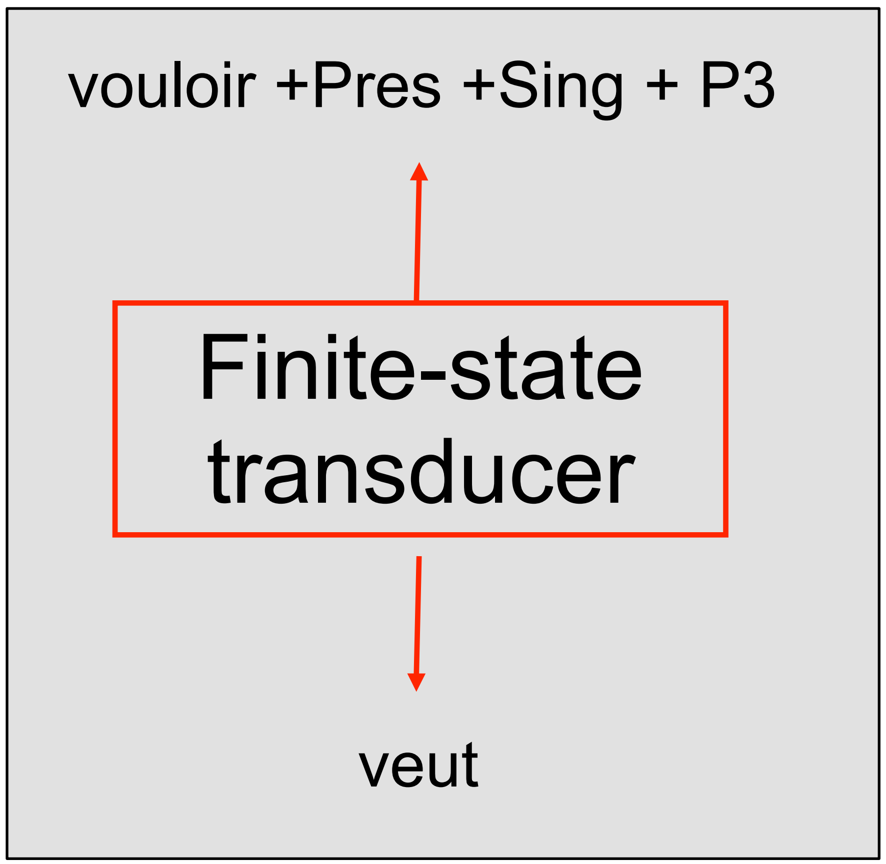
VP [head=vouloir,...]

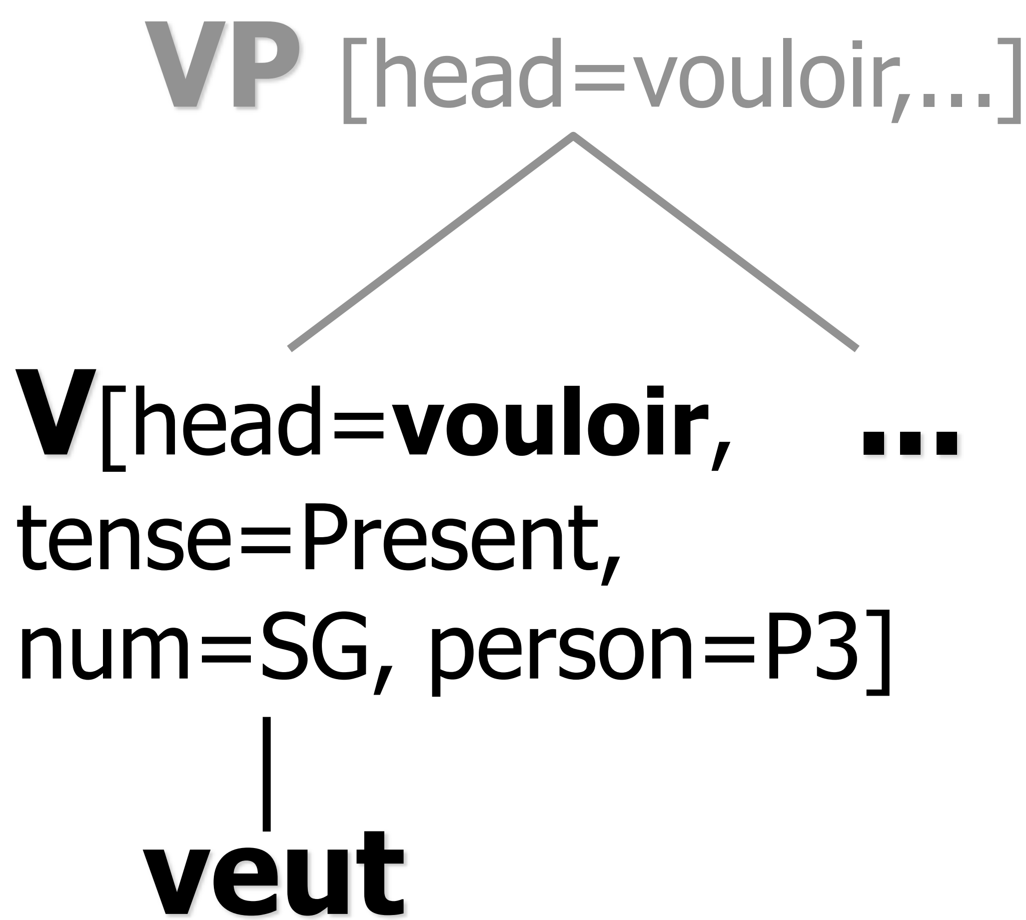
V[head=**vouloir**, tense=Present, num=SG, person=P3] **...**

veut

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

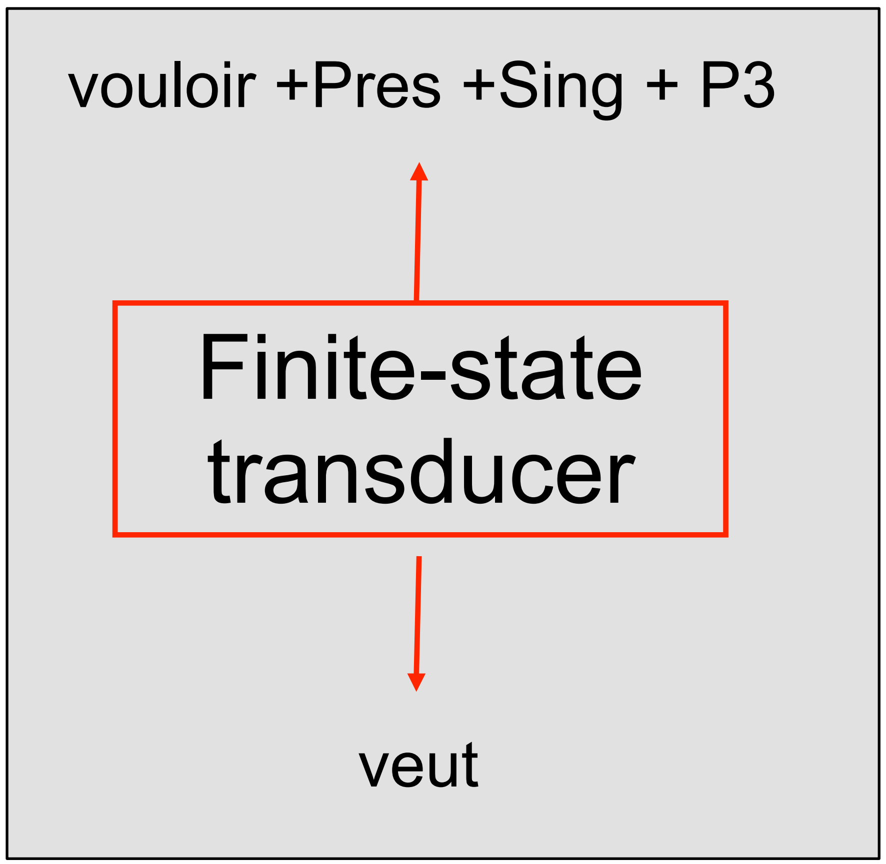
Example: Unweighted transducer

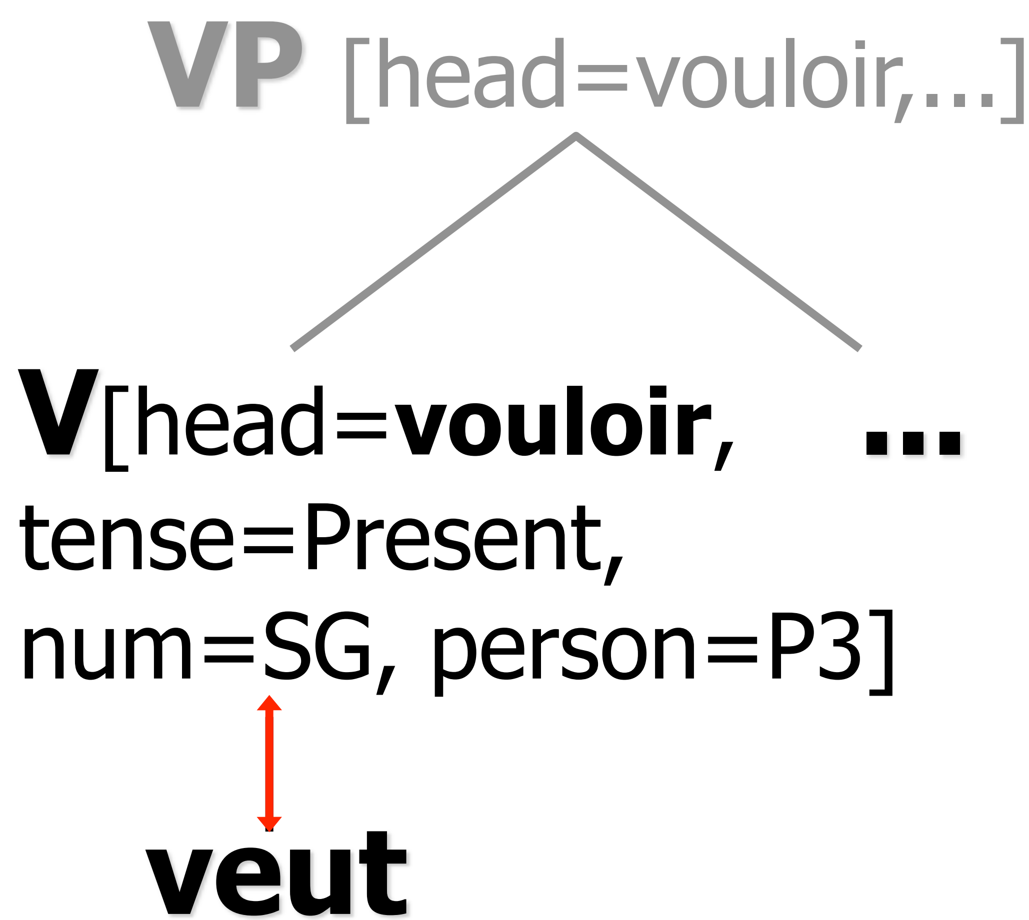




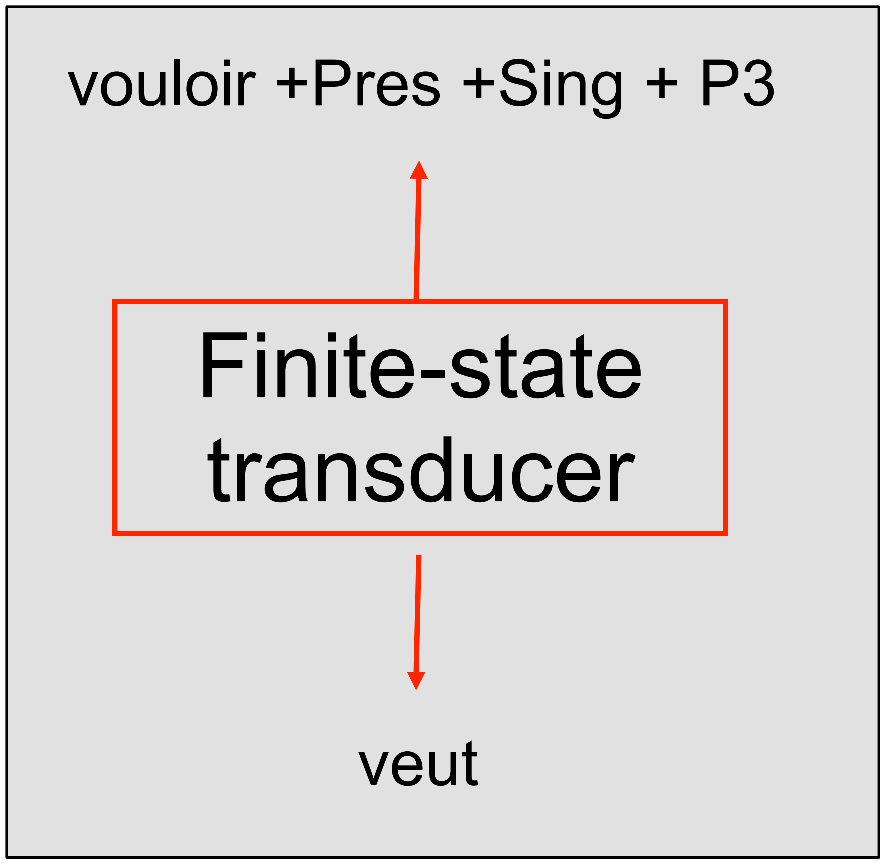
slide courtesy of L. Karttunen (modified)

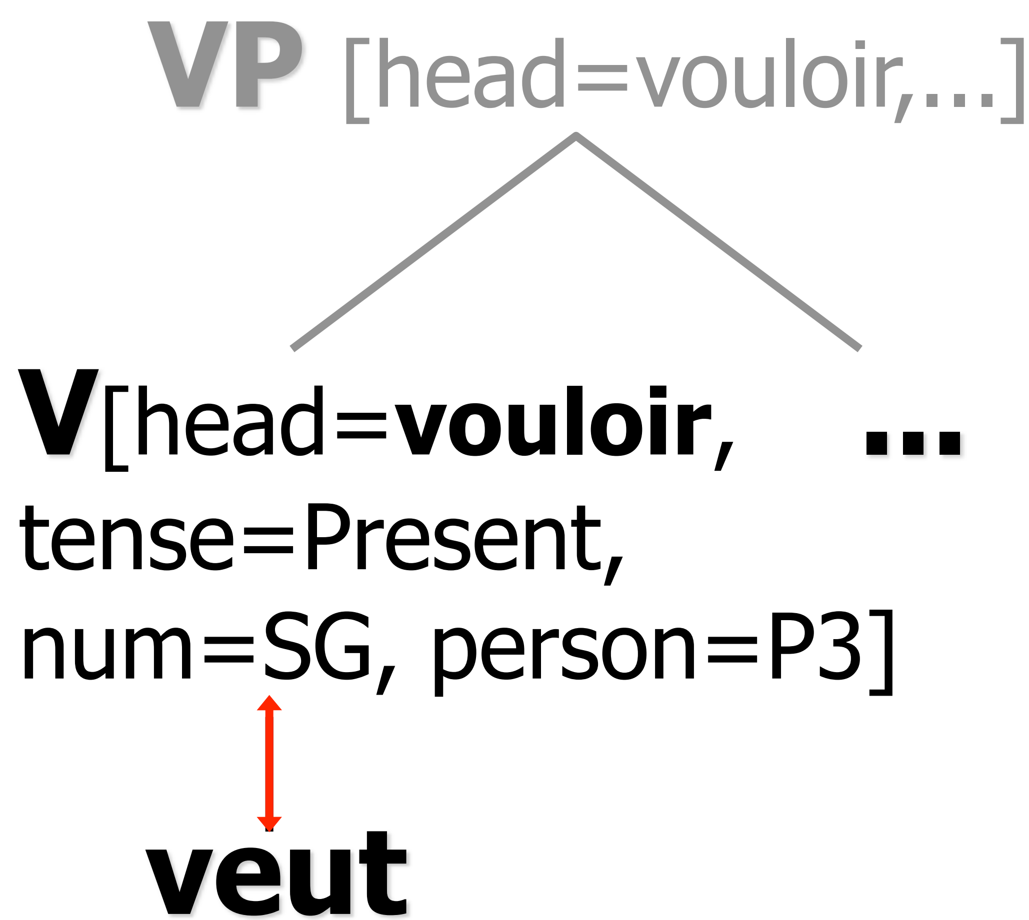
Example: Unweighted transducer

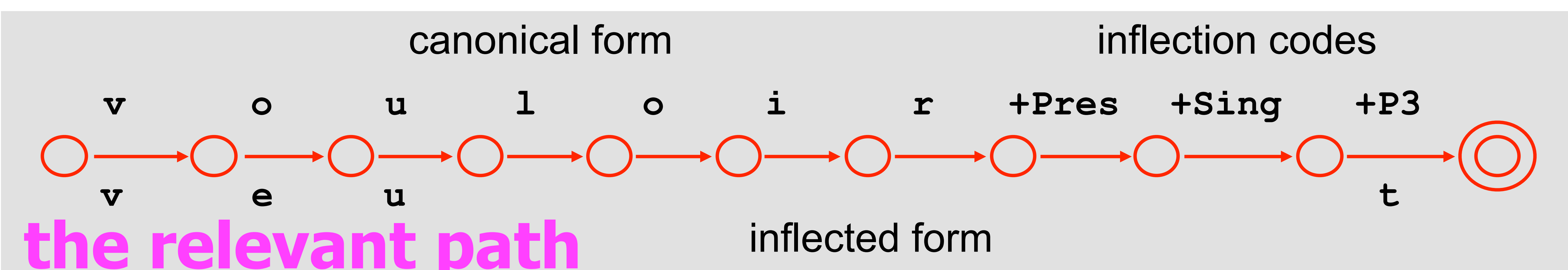




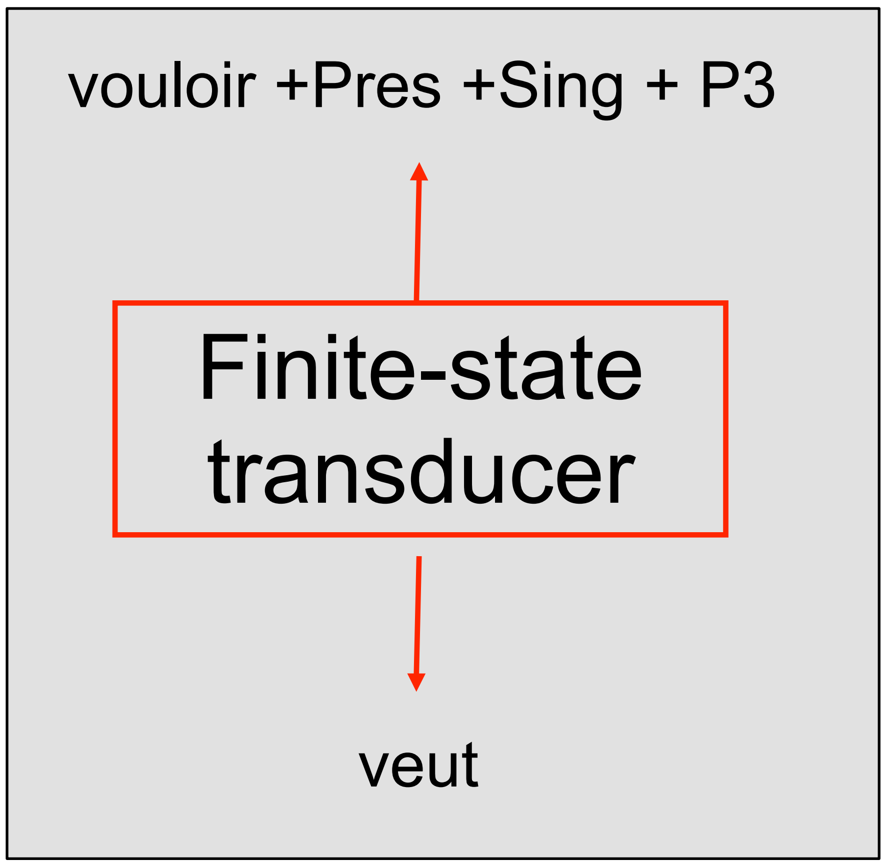
Example: Unweighted transducer



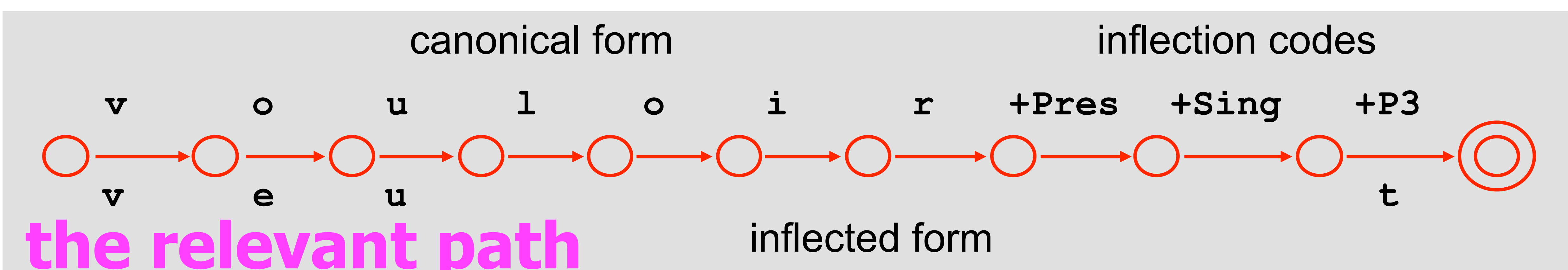




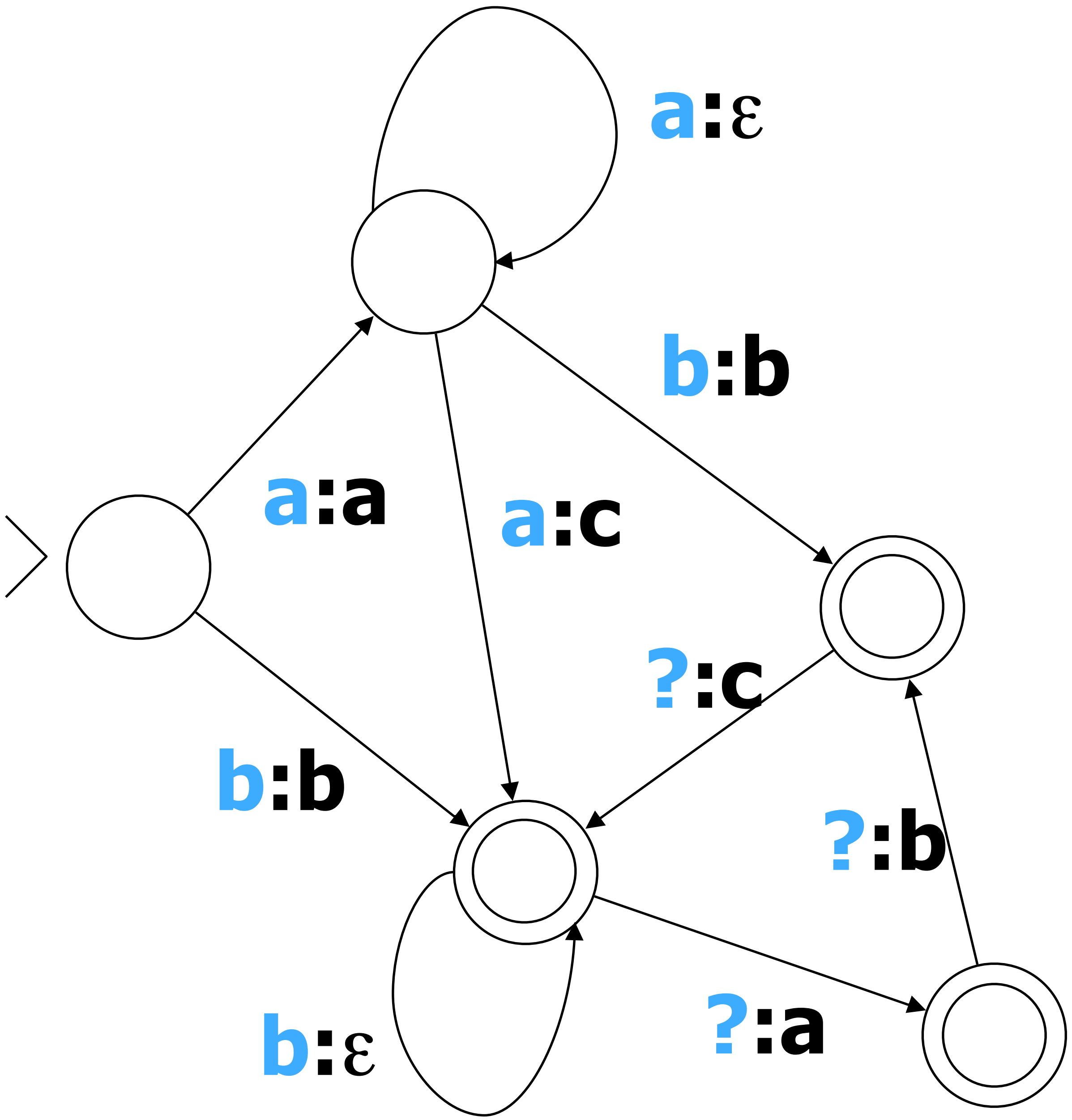
Example: Unweighted transducer



- 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

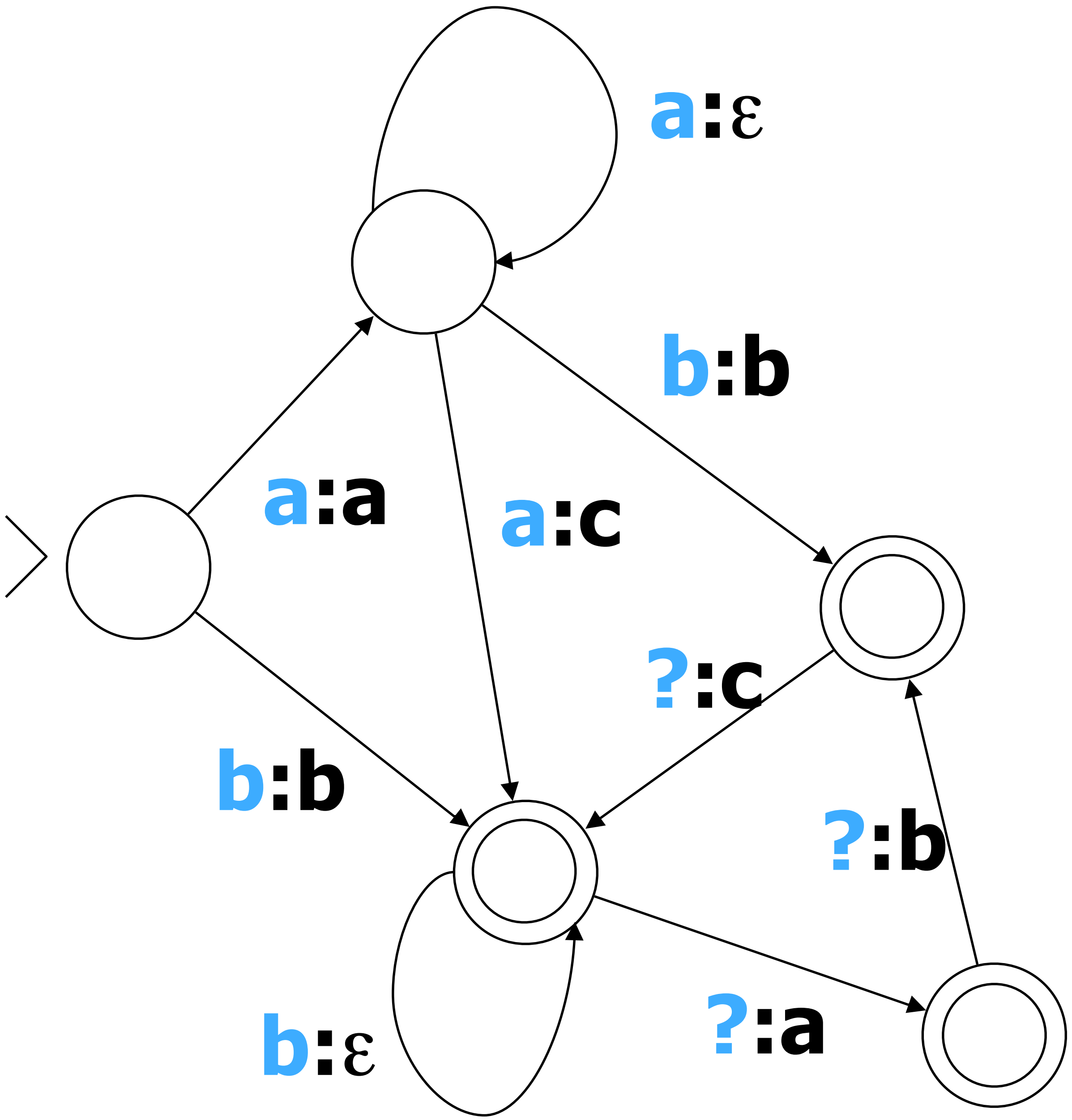


Regular Relation (of strings)



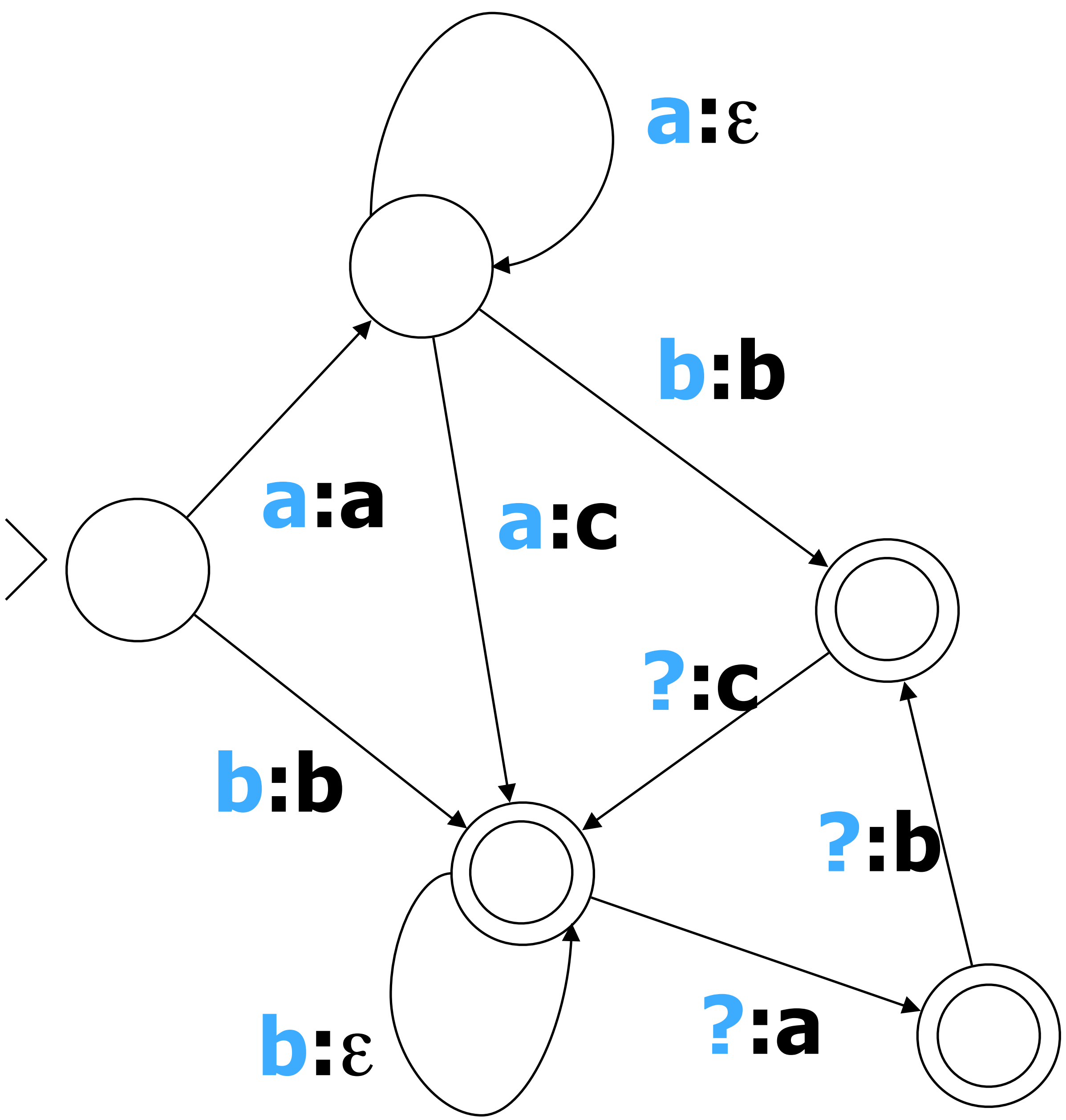
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok



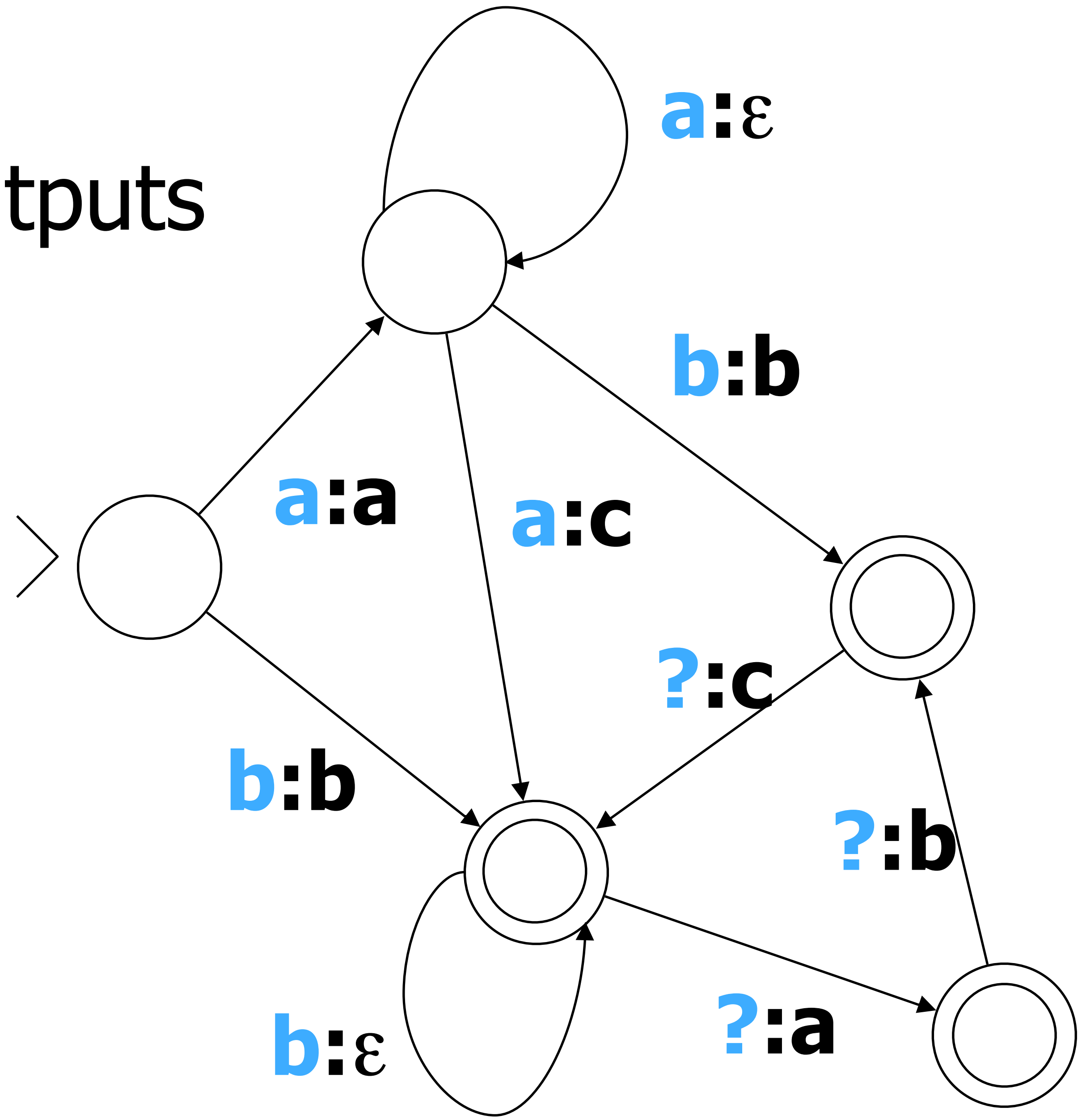
Regular Relation (of strings)

- Relation: like a function, but multiple outputs ok
- Regular: finite-state



Regular Relation (of strings)

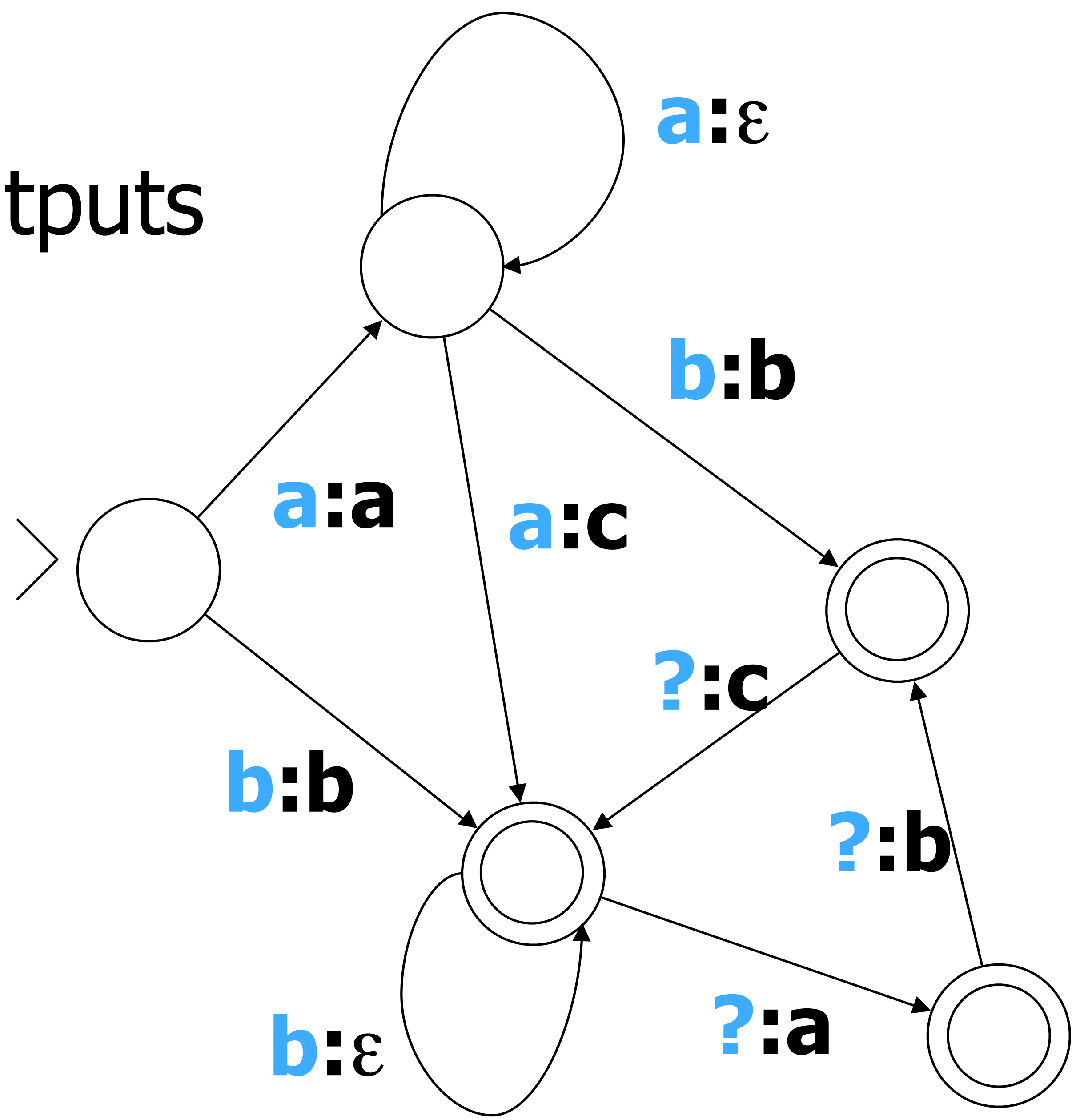
- Relation: like a function, but multiple outputs ok
- Regular: finite-state
- Transducer: automaton w/ outputs



Regular Relation (of strings)

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

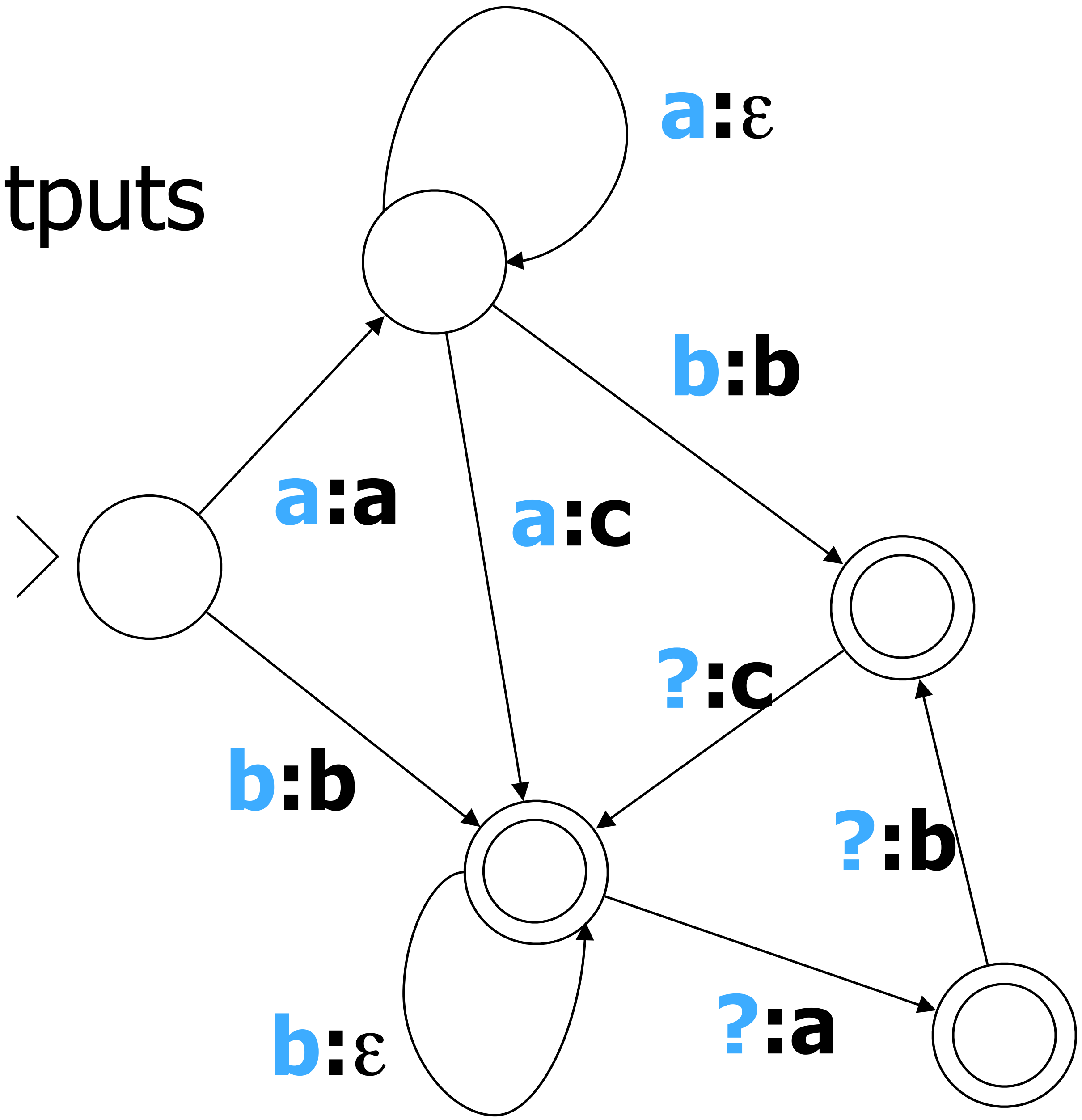
- b → ? a → ?



Regular Relation (of strings)

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

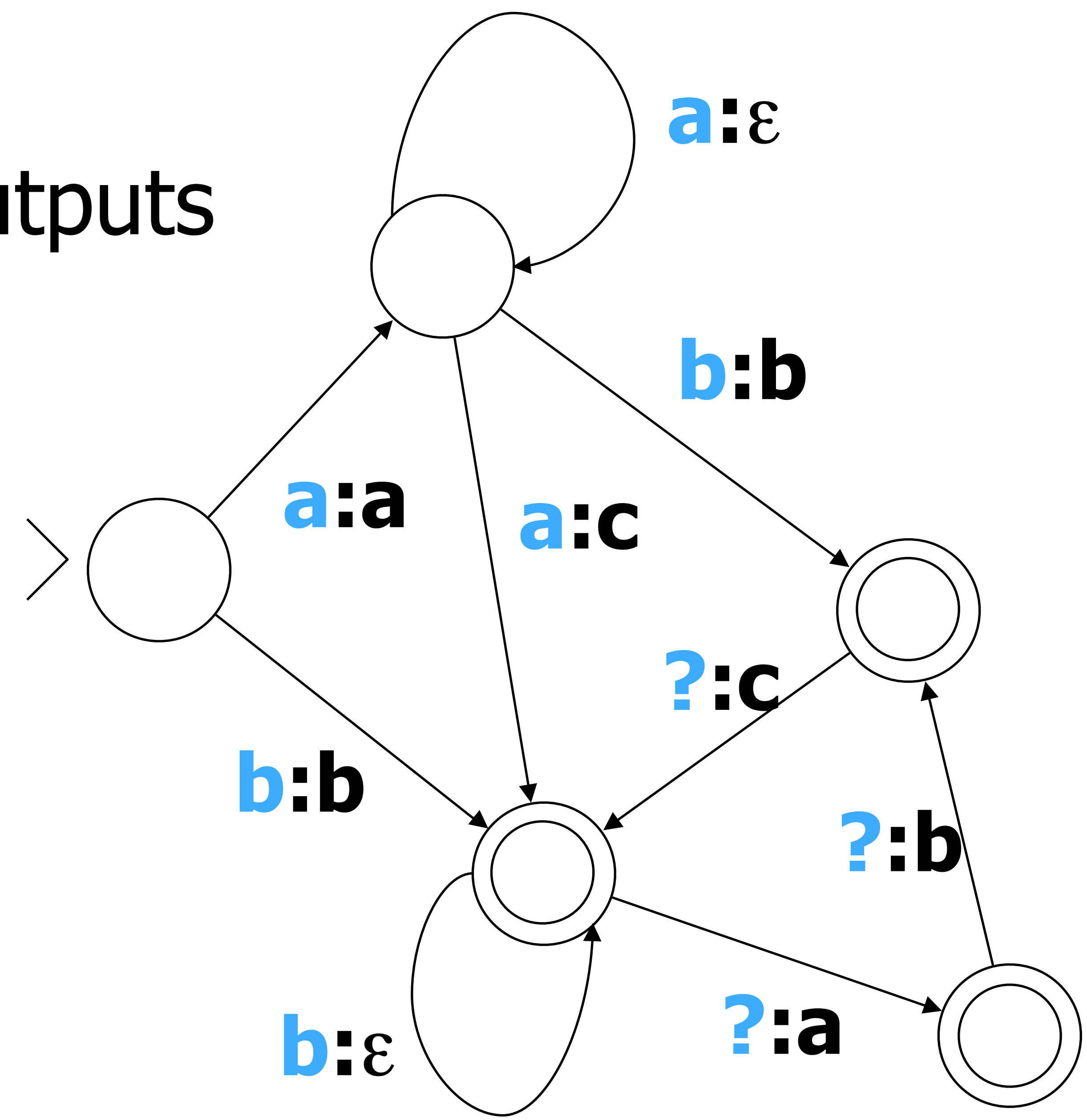
- b → {b} a → ?



Regular Relation (of strings)

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

- b → {b} a → {}

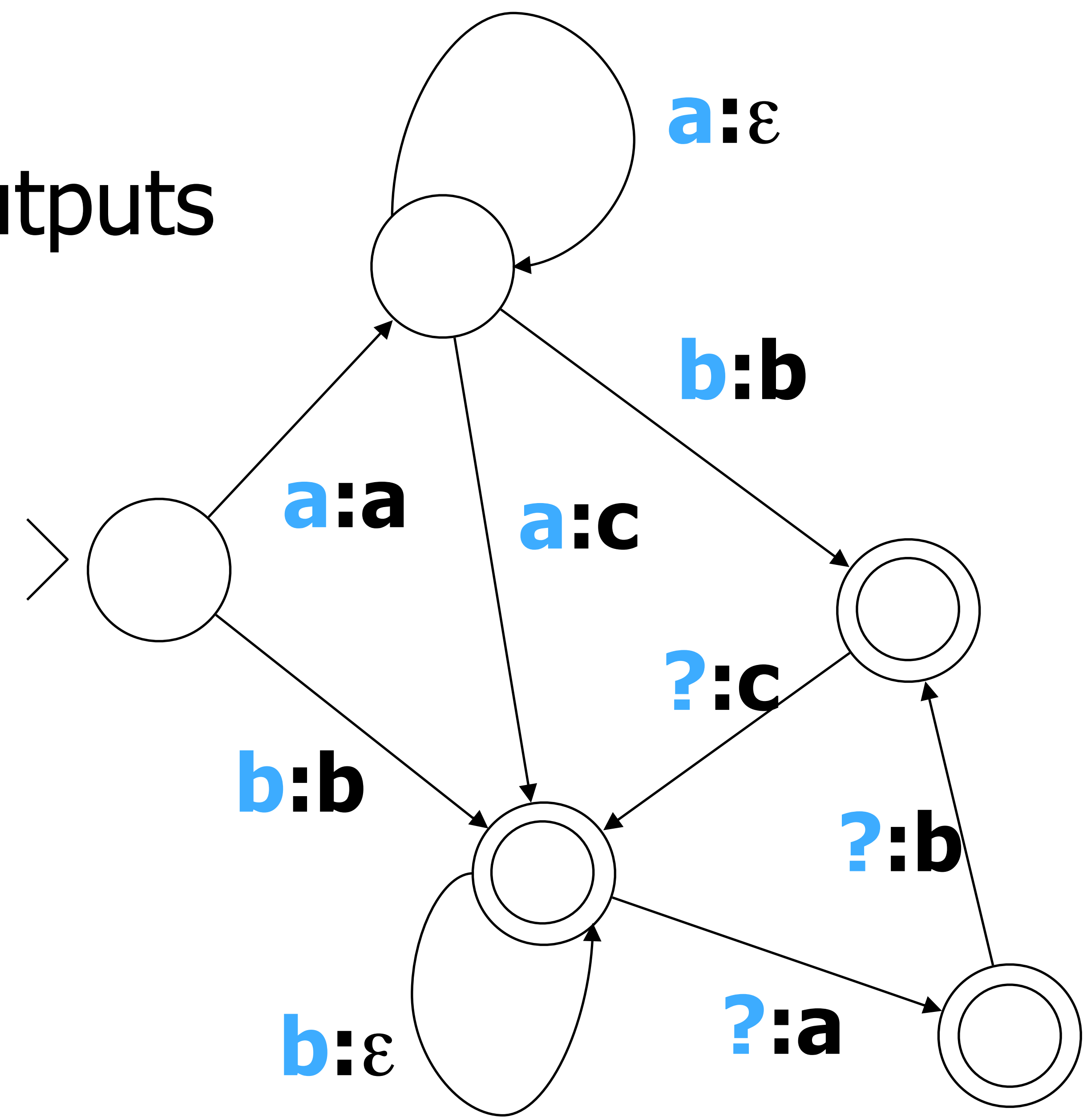


Regular Relation (of strings)

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

- b → {b}   a → {}

{ac, aca, acab, acabc}



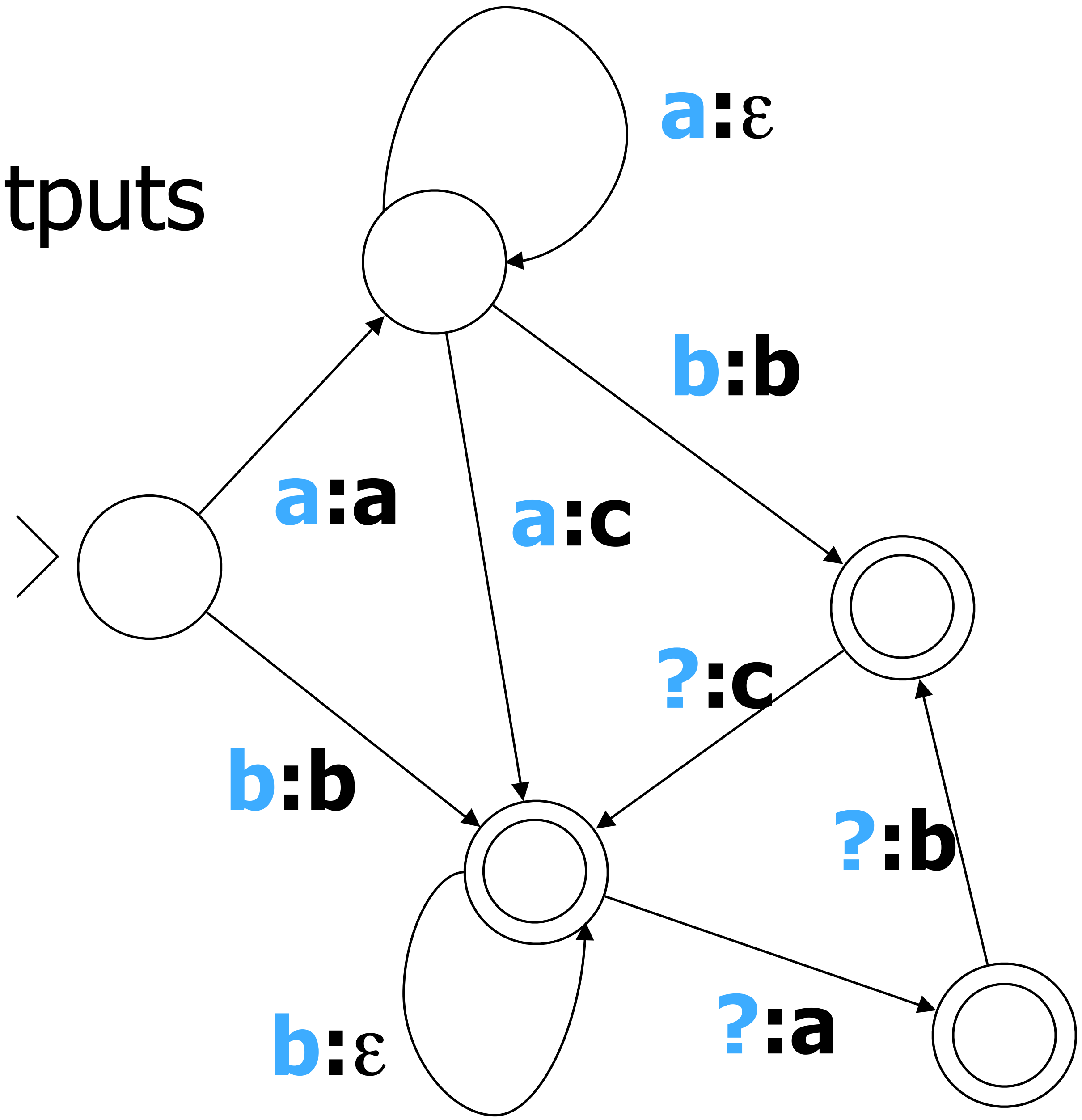
Regular Relation (of strings)

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

- b $\rightarrow$ {b}   a $\rightarrow$ {}

  {ac, aca, acab, acabc}

- Invertible?



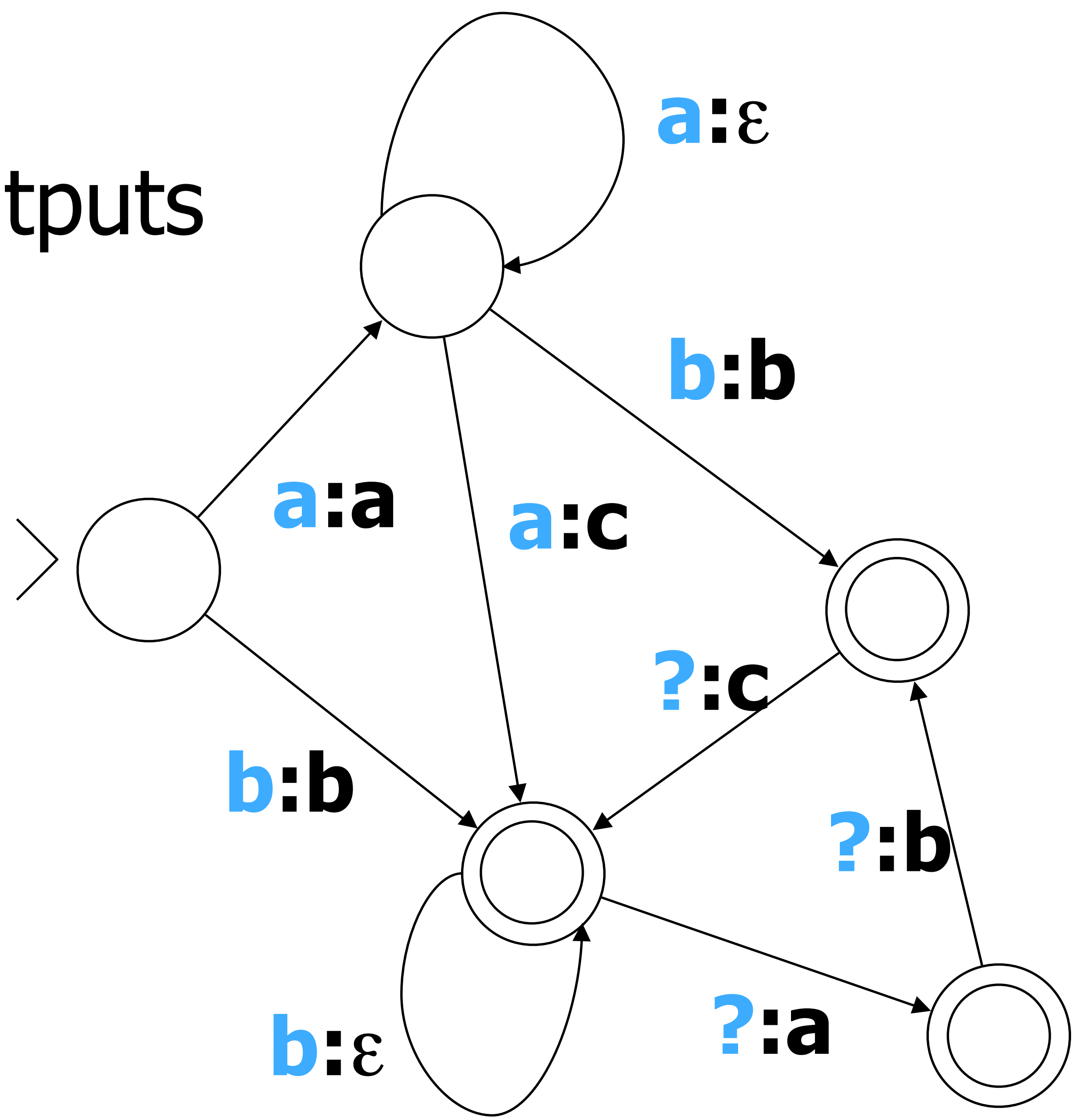
Regular Relation (of strings)

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

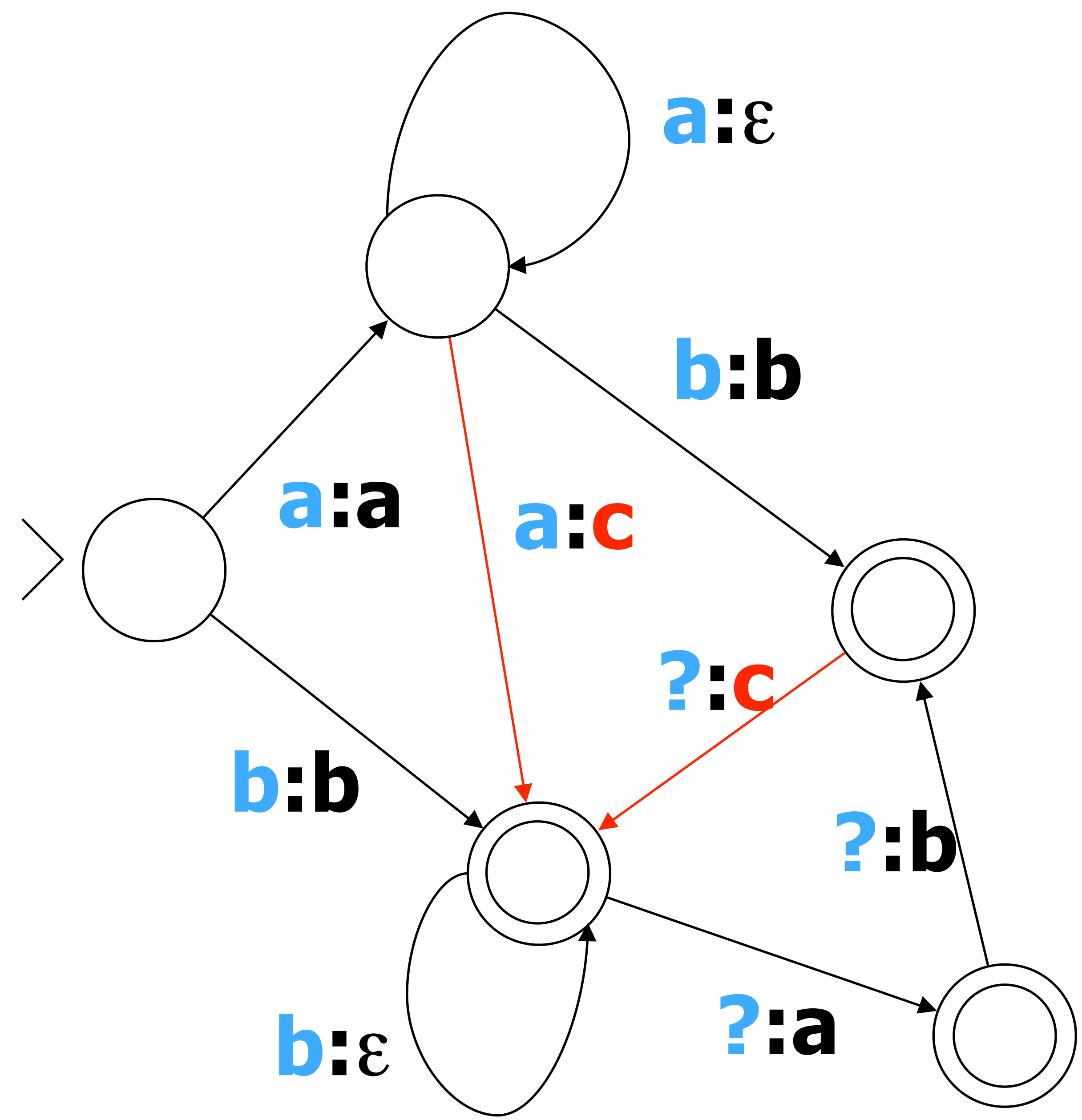
- b → {b}   a → {}

  {ac, aca, acab, acabc}

- Invertible?
- Closed under composition?

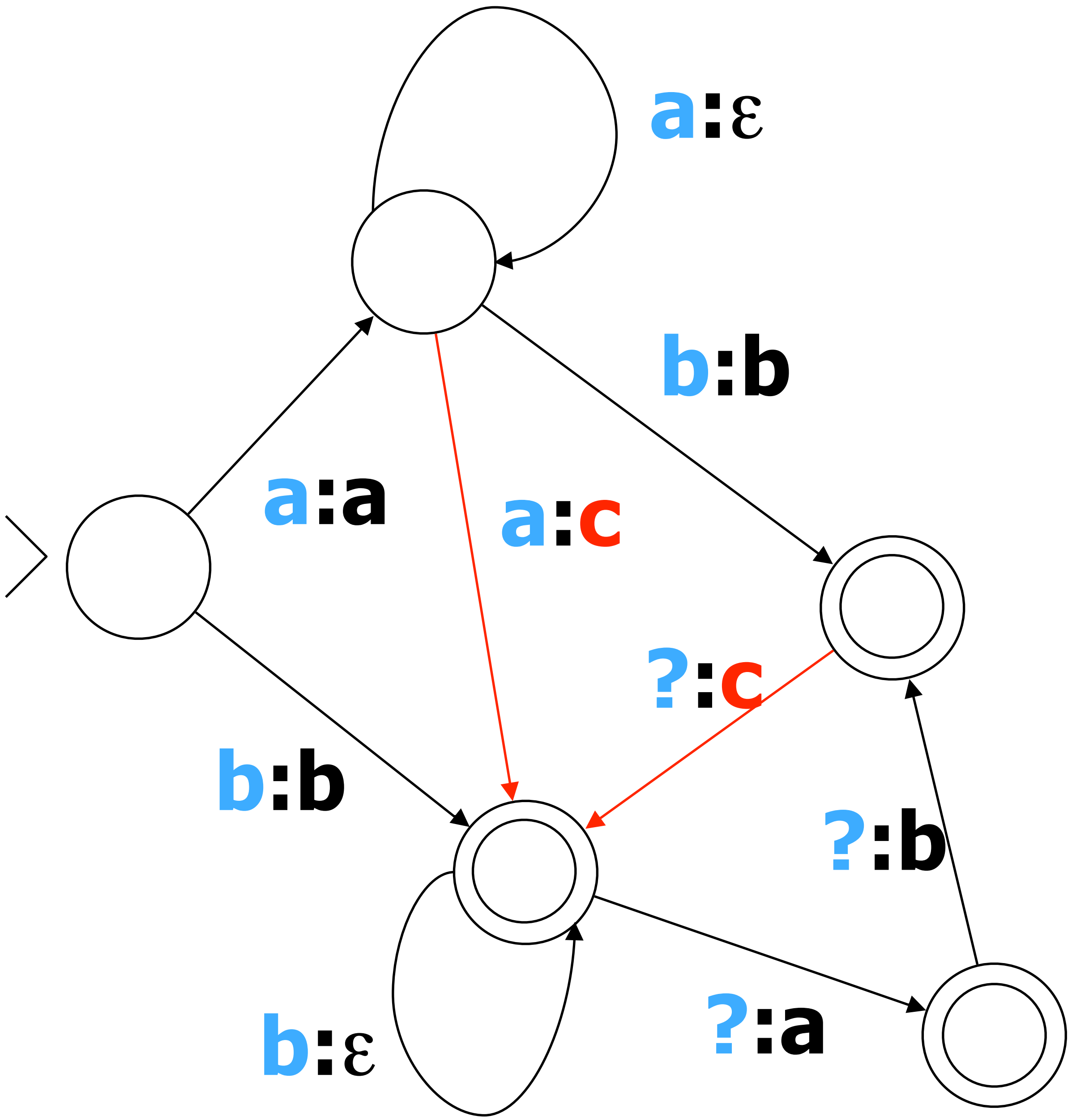


Regular Relation (of strings)



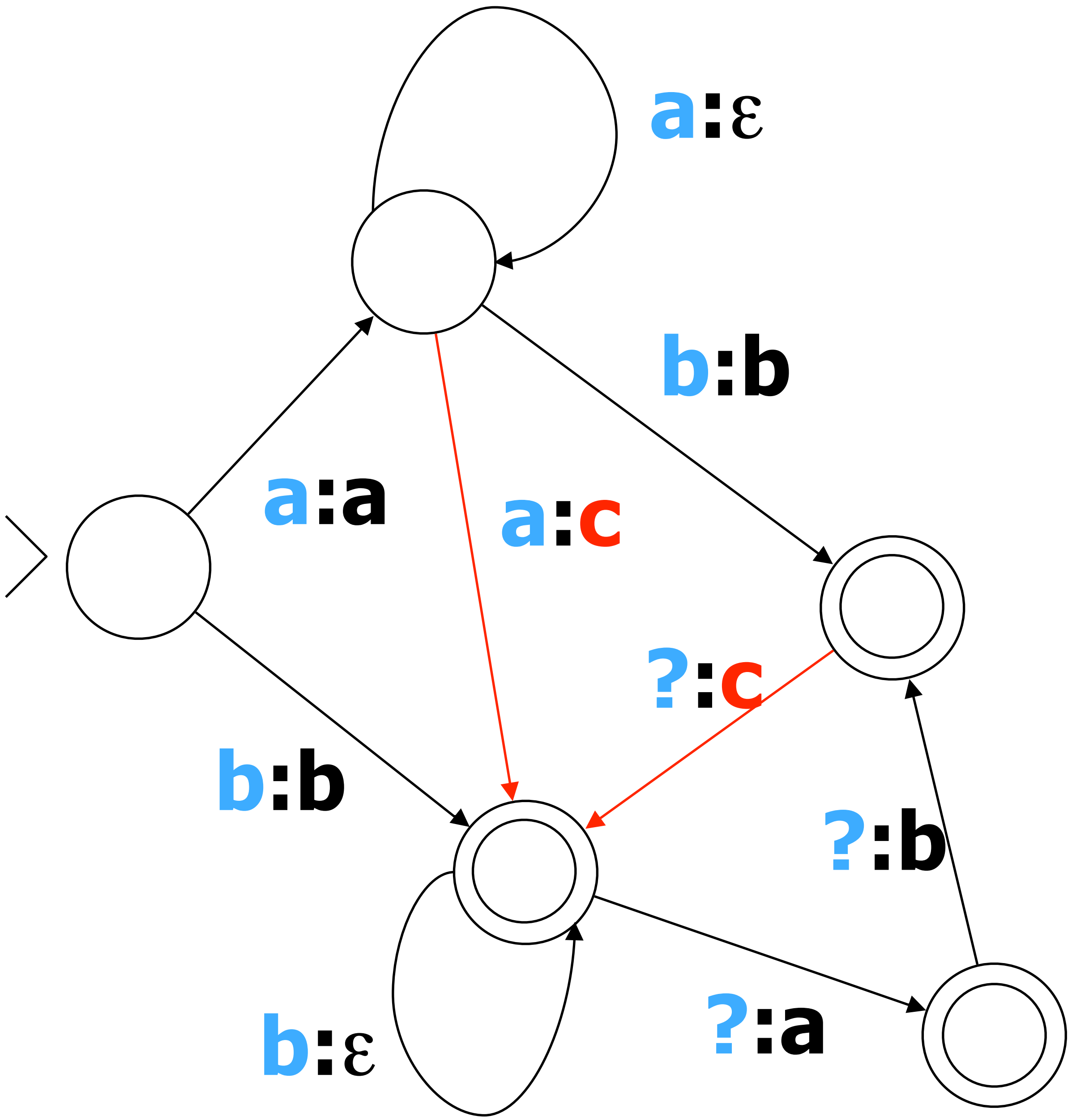
Regular Relation (of strings)

- Can weight the arcs: → vs. →



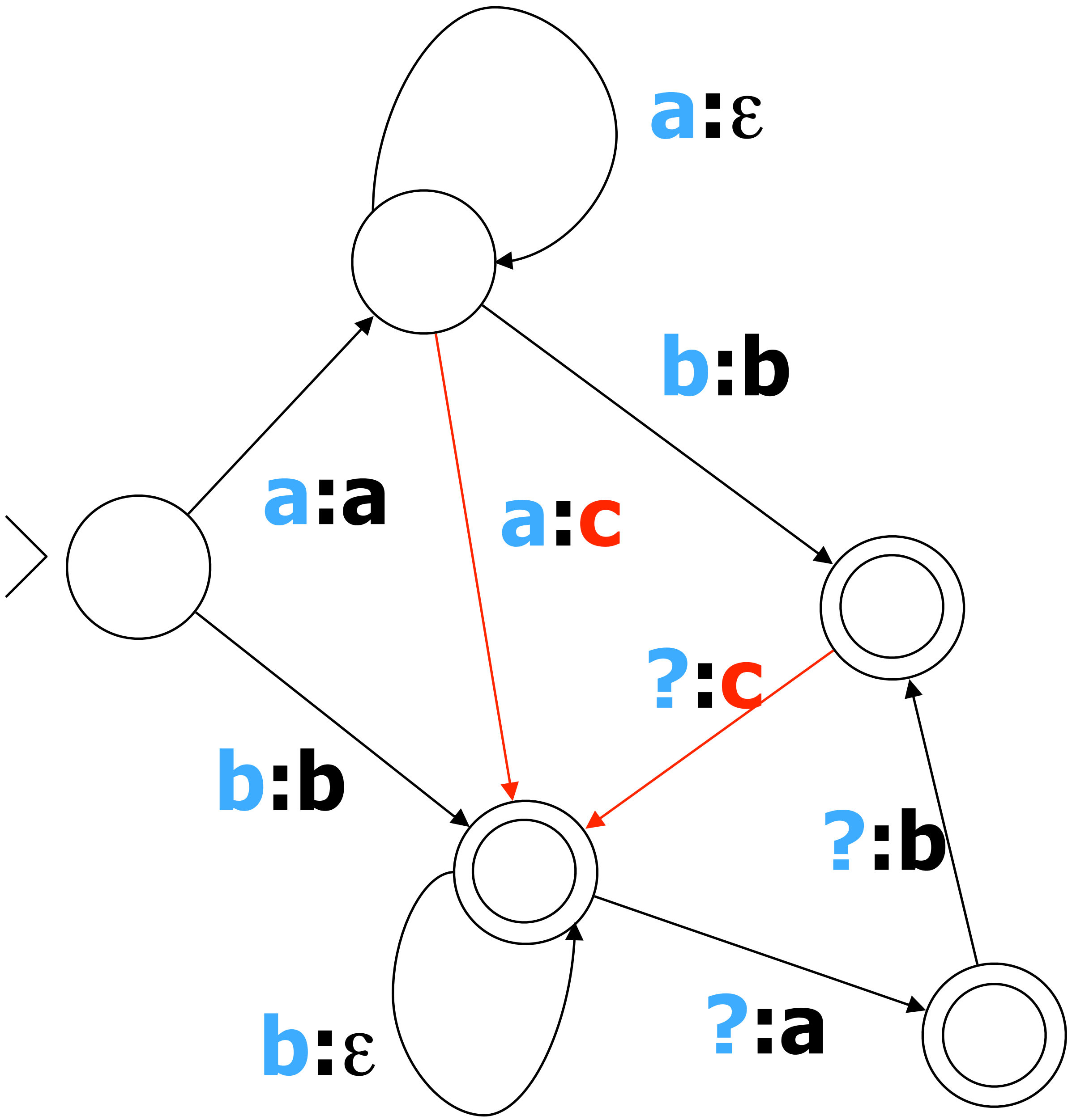
Regular Relation (of strings)

- Can weight the arcs: → vs. →
- b → {b}   a → {}



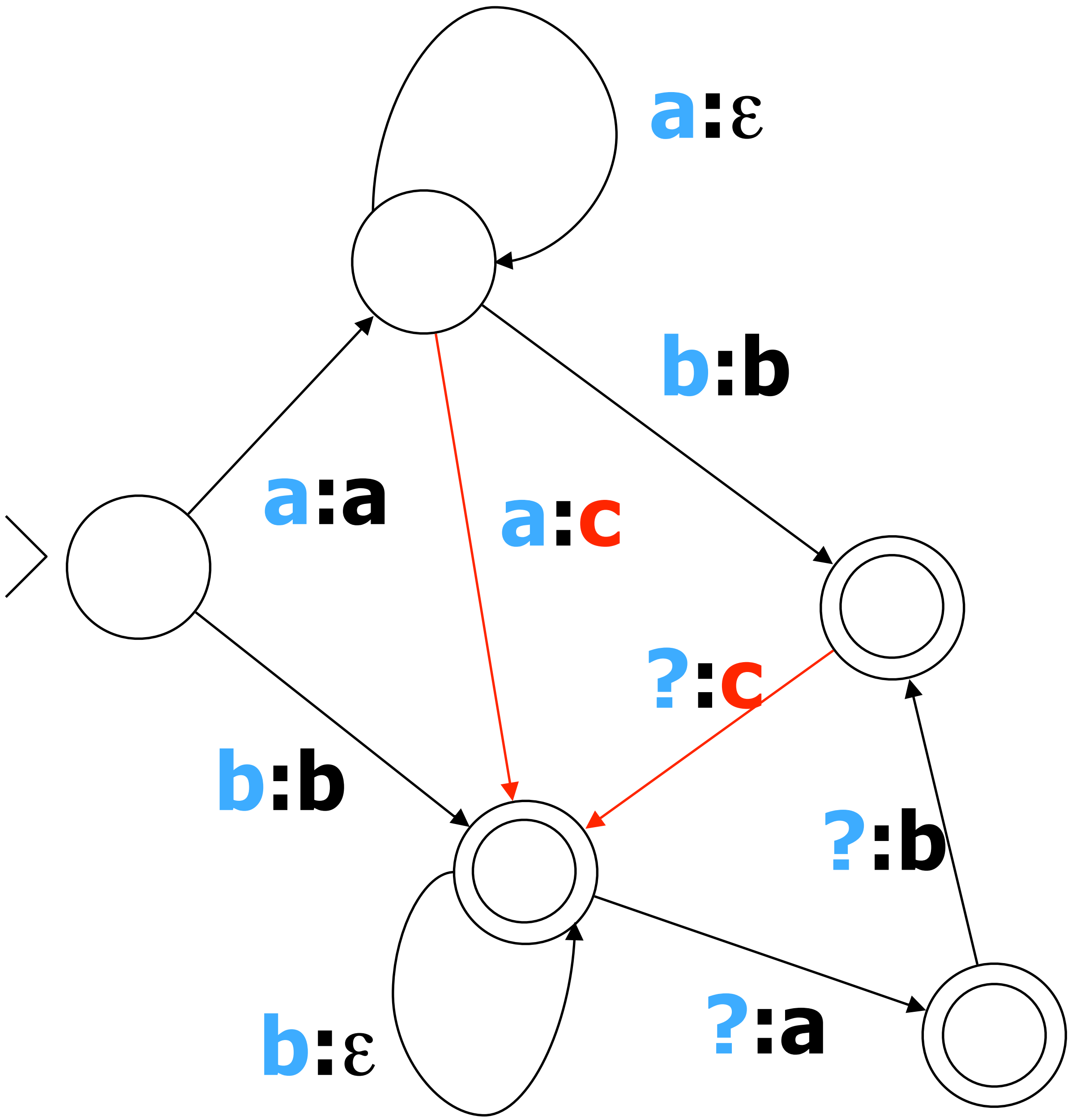
Regular Relation (of strings)

- Can weight the arcs: → vs. →
- b → {b}   a → {}
- aaaaa → {ac, aca, acab, acabc}



Regular Relation (of strings)

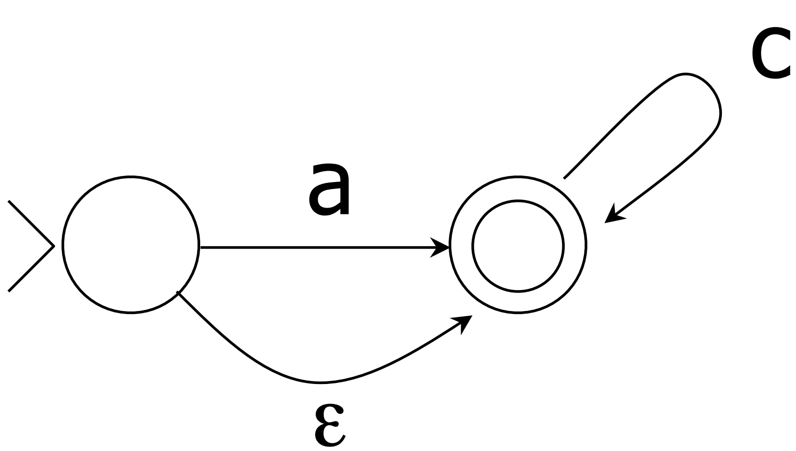
- Can weight the arcs: → vs. →
- b → {b}   a → {}
- aaaaa → {ac, aca, acab, acabc}
- How to find best outputs?

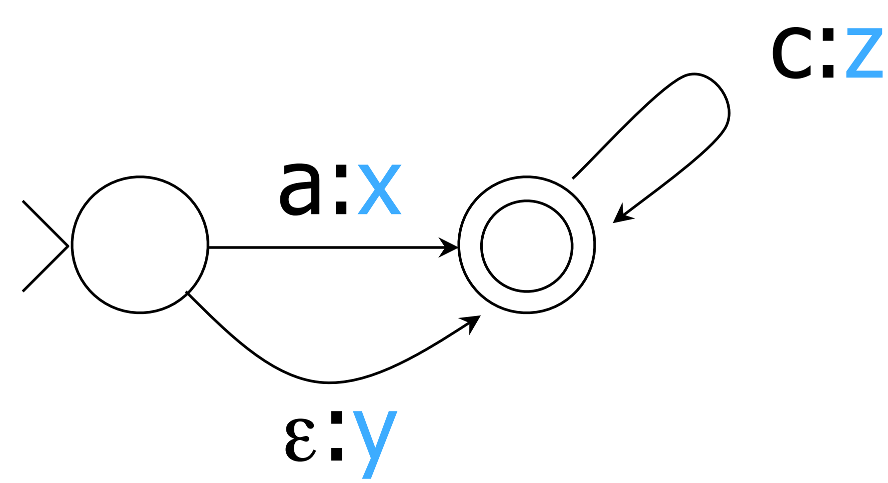


Function from strings to ...

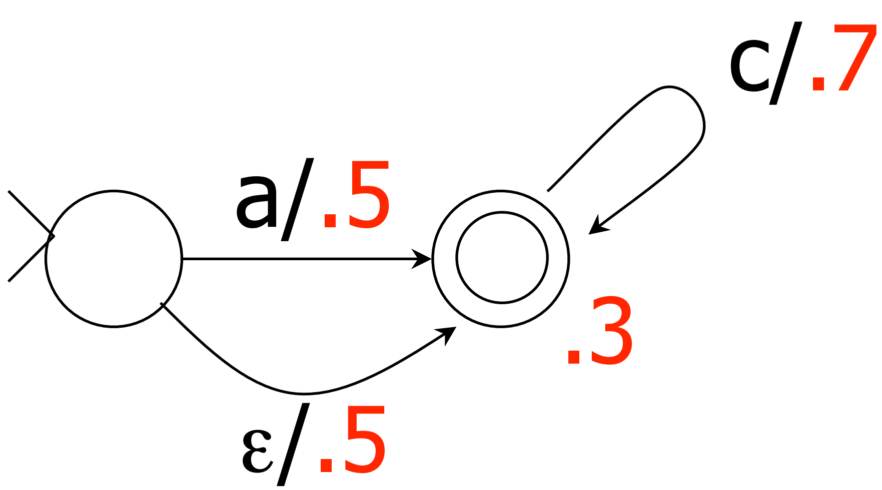
Acceptors (FSAs) **Transducers (FSTs)**

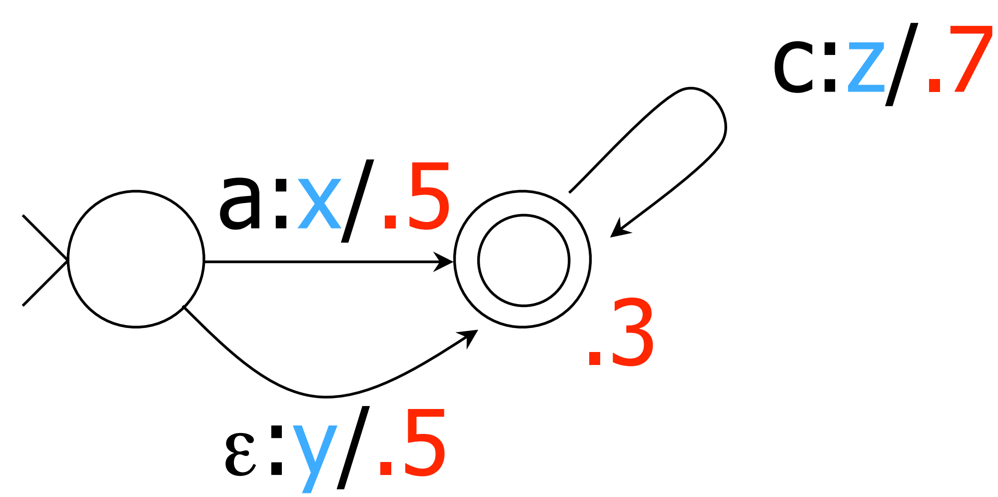
Unweighted





Weighted



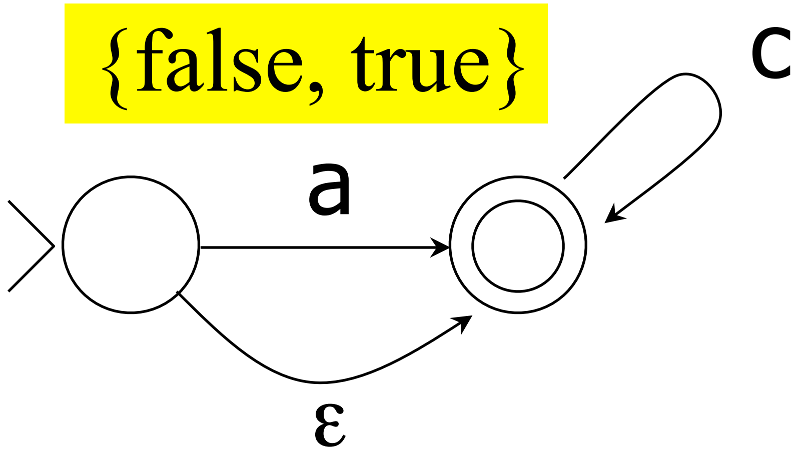


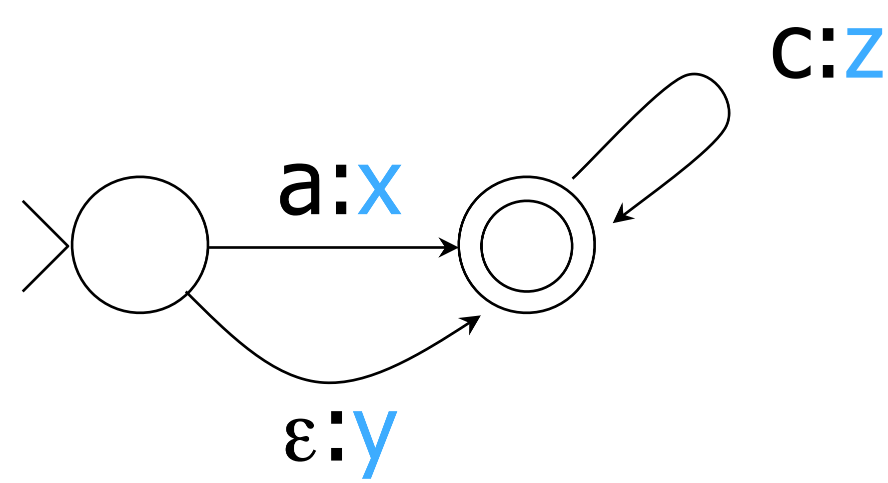
Function from strings to ...

Acceptors (FSAs) | **Transducers (FSTs)**

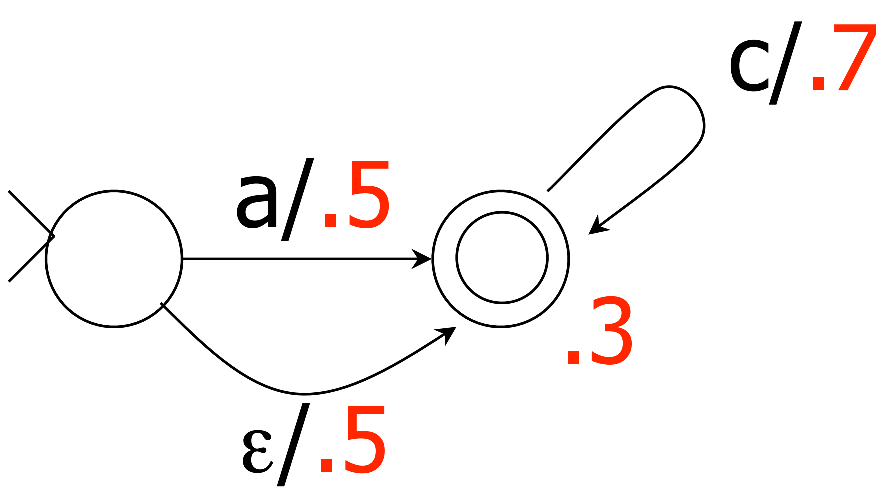
{false, true}

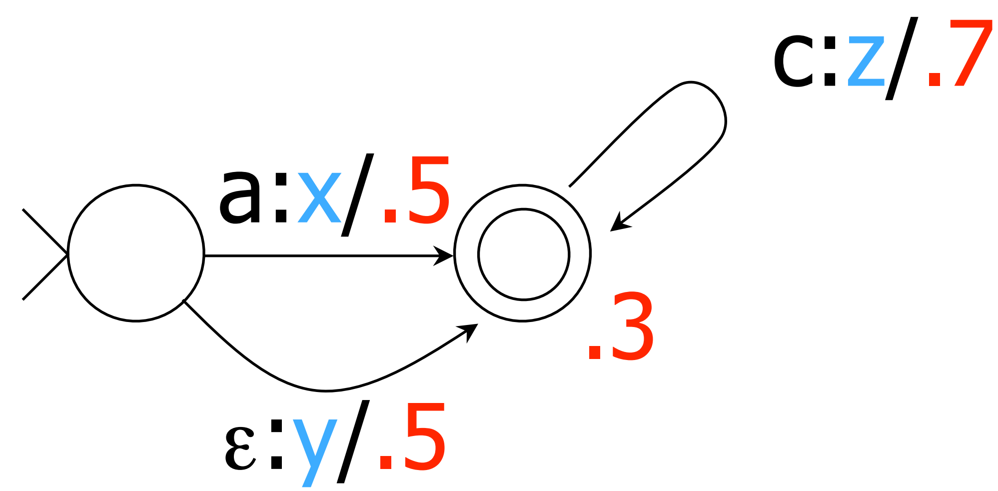
Unweighted





Weighted

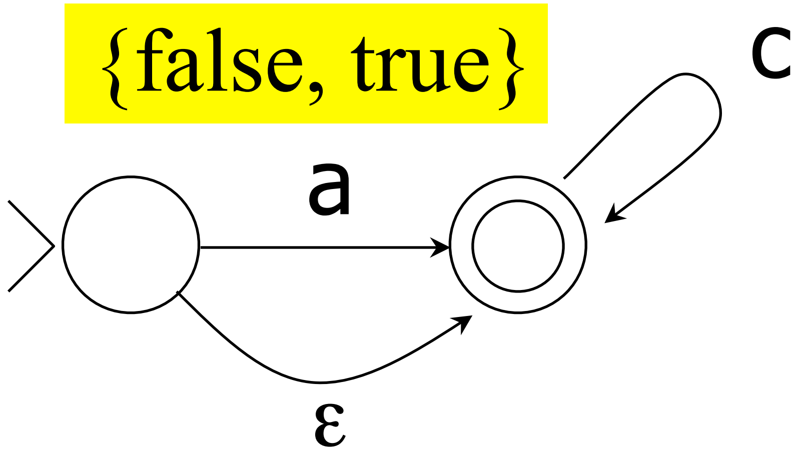


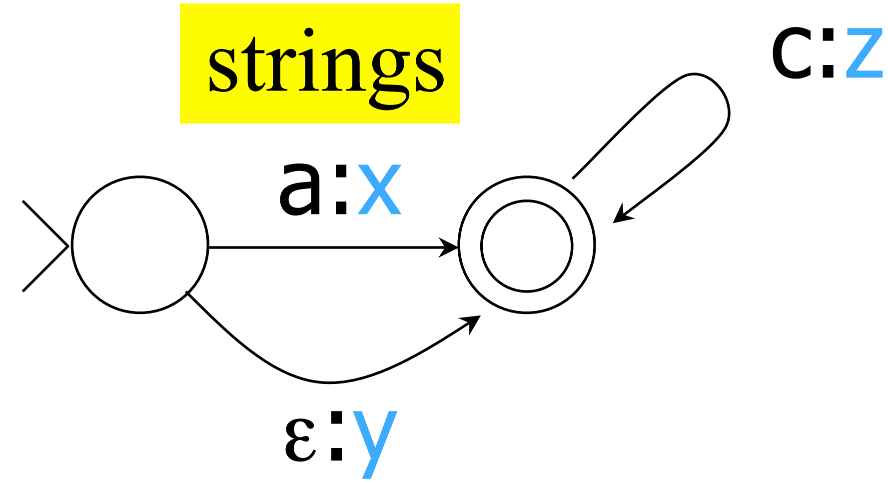


Function from strings to ...

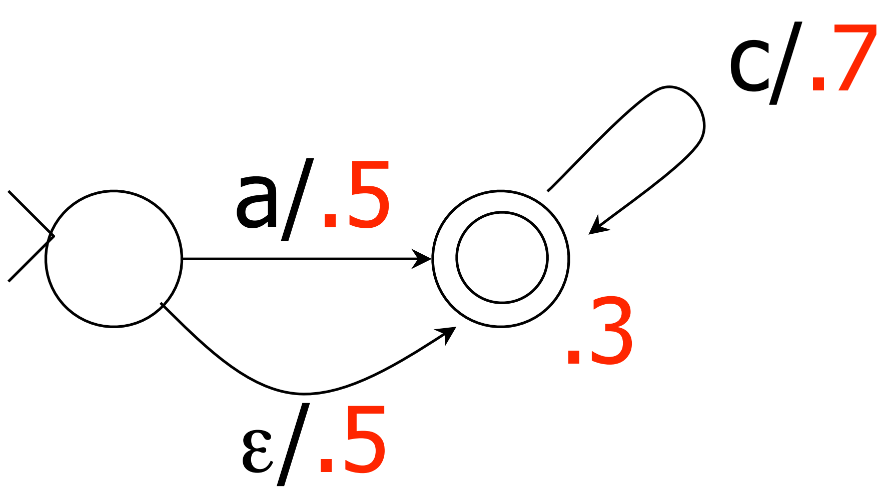
Acceptors (FSAs) | **Transducers (FSTs)**

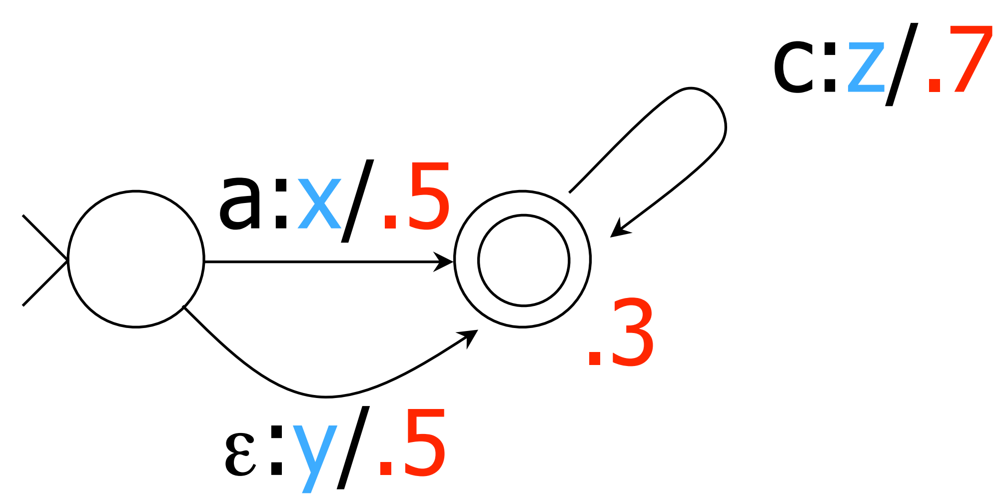
Unweighted





Weighted

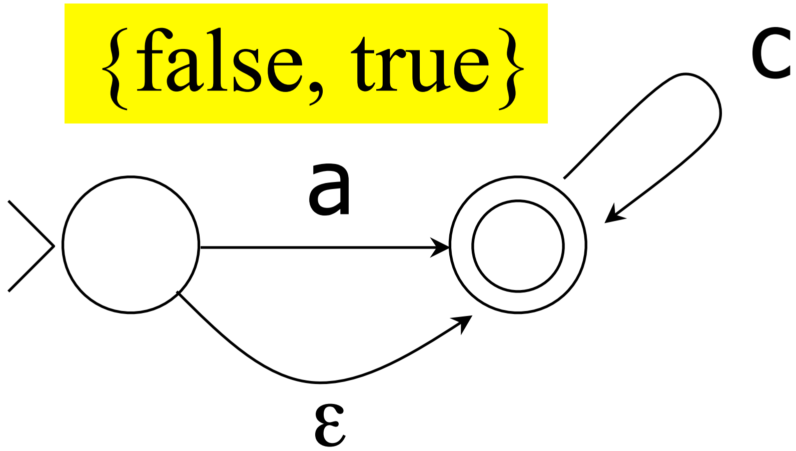




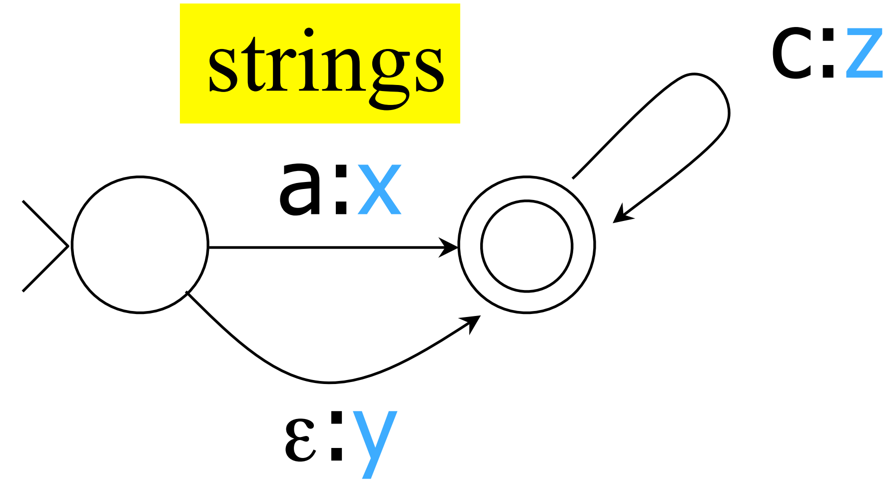
Function from strings to ...

	Acceptors (FSAs)	**Transducers (FSTs)**
Unweighted	{false, true}	strings
Weighted	numbers	

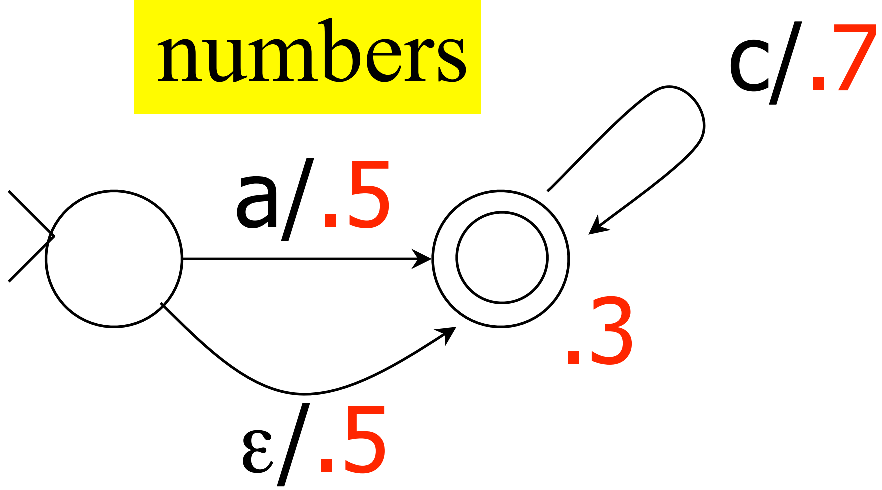
Unweighted, Acceptors (FSAs):



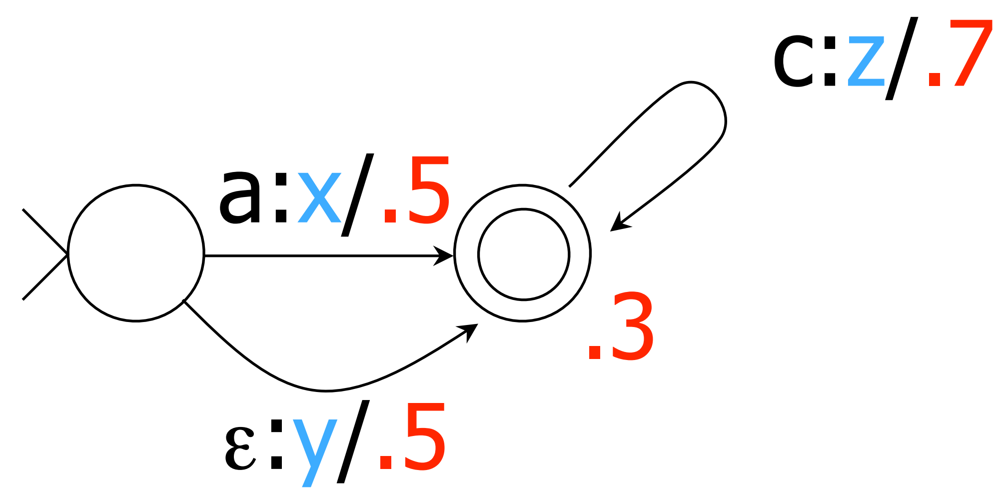
Unweighted, Transducers (FSTs):



Weighted, Acceptors (FSAs):



Weighted, Transducers (FSTs):



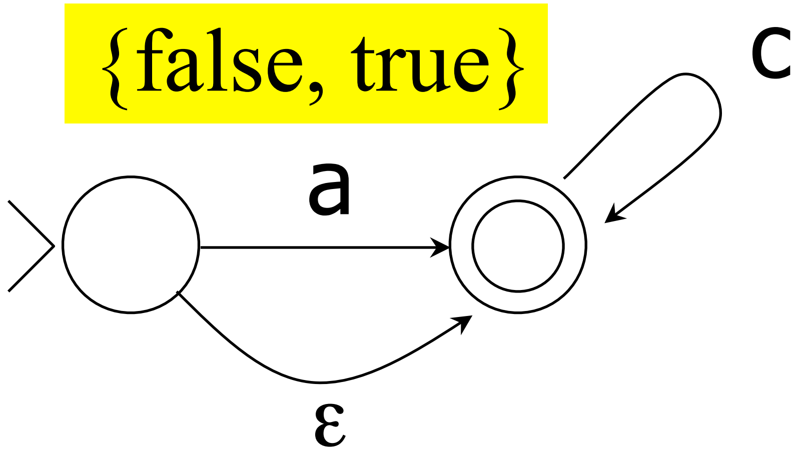
Function from strings to ...

Acceptors (FSAs)

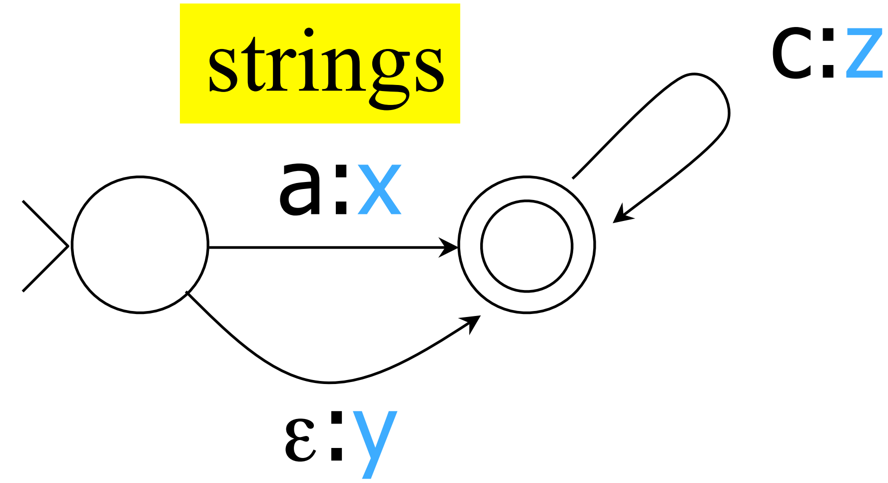
Transducers (FSTs)

Unweighted

{false, true}

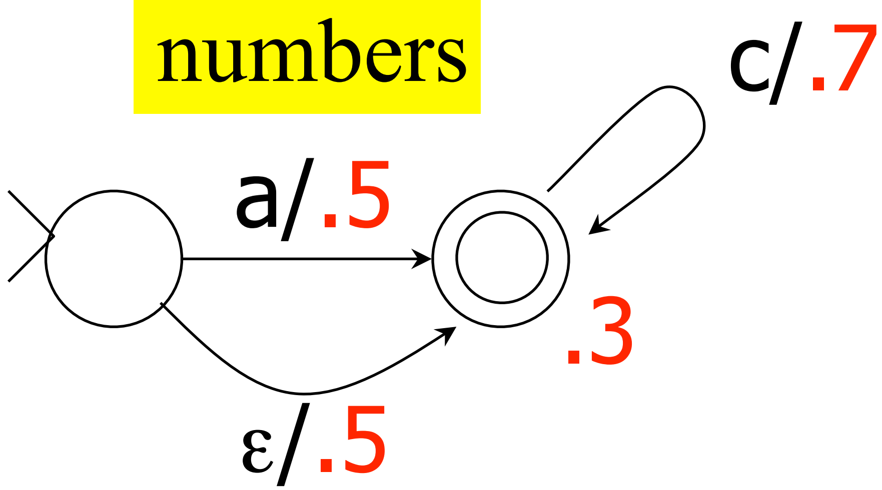


strings

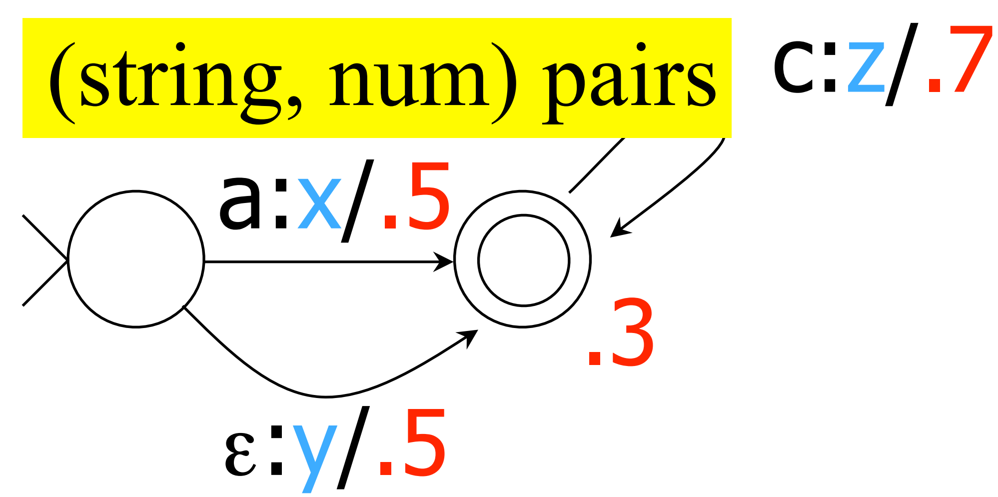


Weighted

numbers



(string, num) pairs



Sample functions

	Acceptors (FSAs)	Transducers (FSTs)
Unweighted	{false, true}	strings
Weighted	numbers	(string, num) pairs

Sample functions

	Acceptors (FSAs)	Transducers (FSTs)
Unweighted	{false, true} Grammatical?	strings
Weighted	numbers	(string, num) pairs

Sample functions

	Acceptors (FSAs)	Transducers (FSTs)
Unweighted	{false, true} Grammatical?	strings
Weighted	numbers How grammatical? Better, how likely?	(string, num) pairs

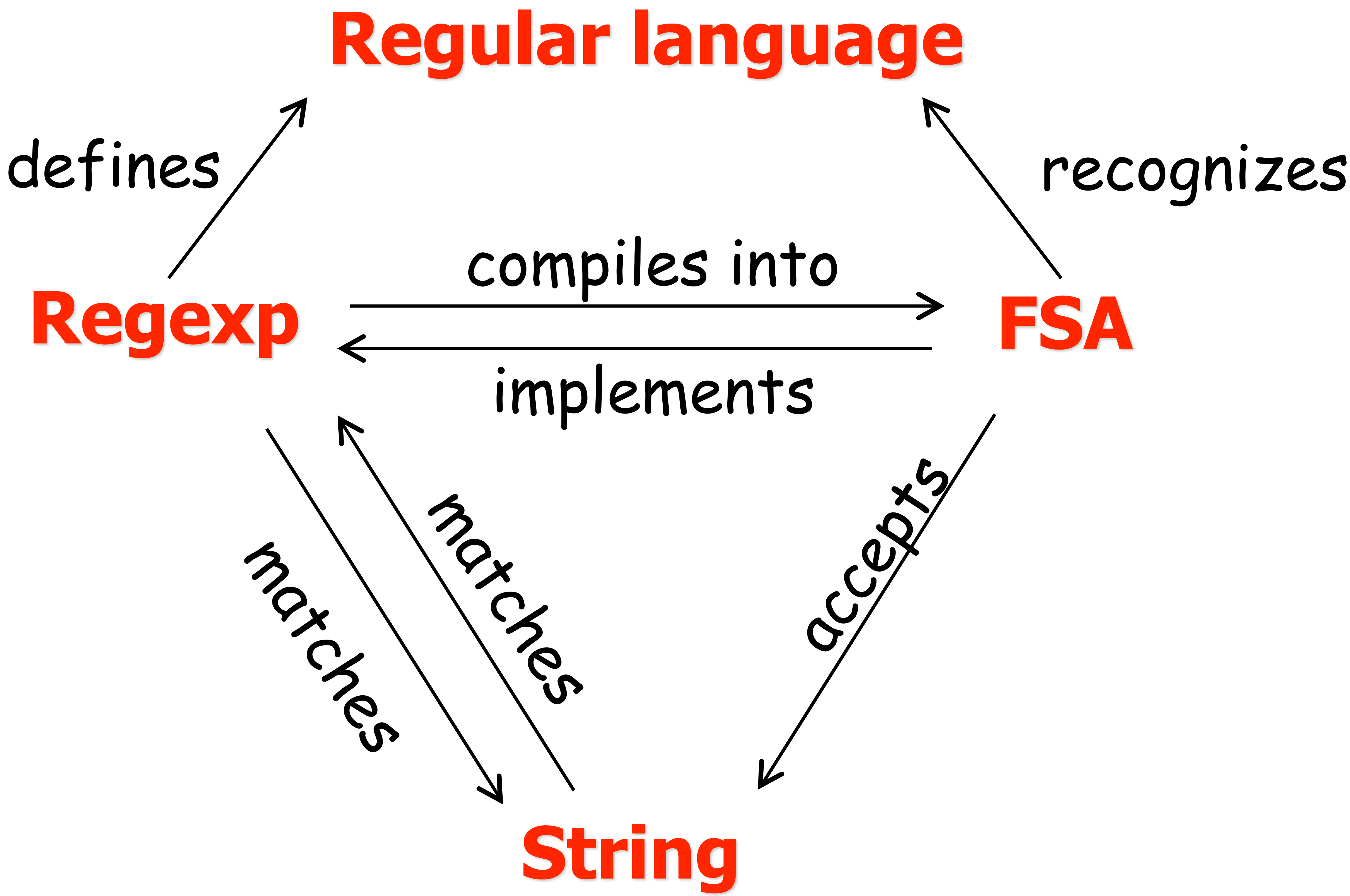
Sample functions

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

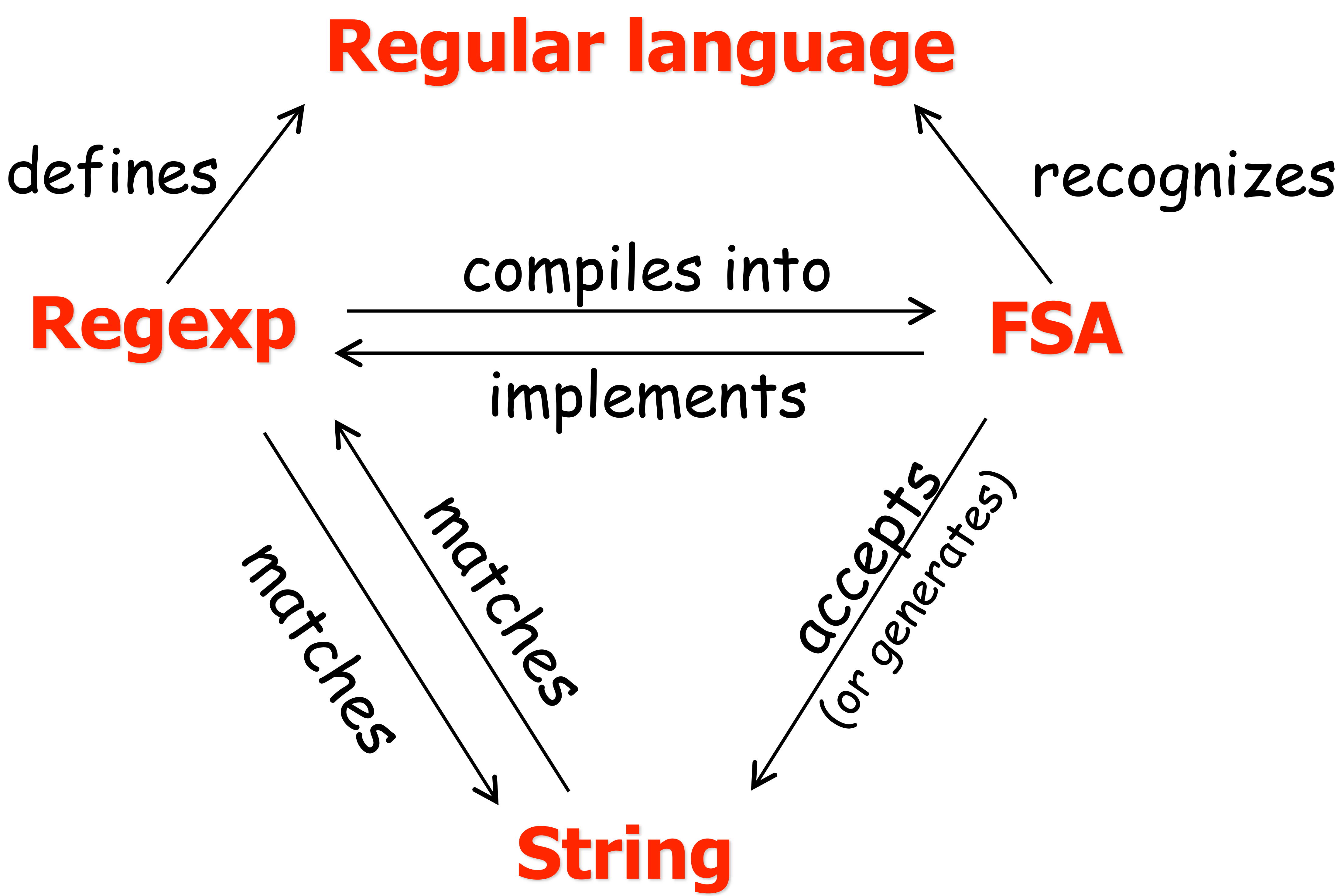
Sample functions

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

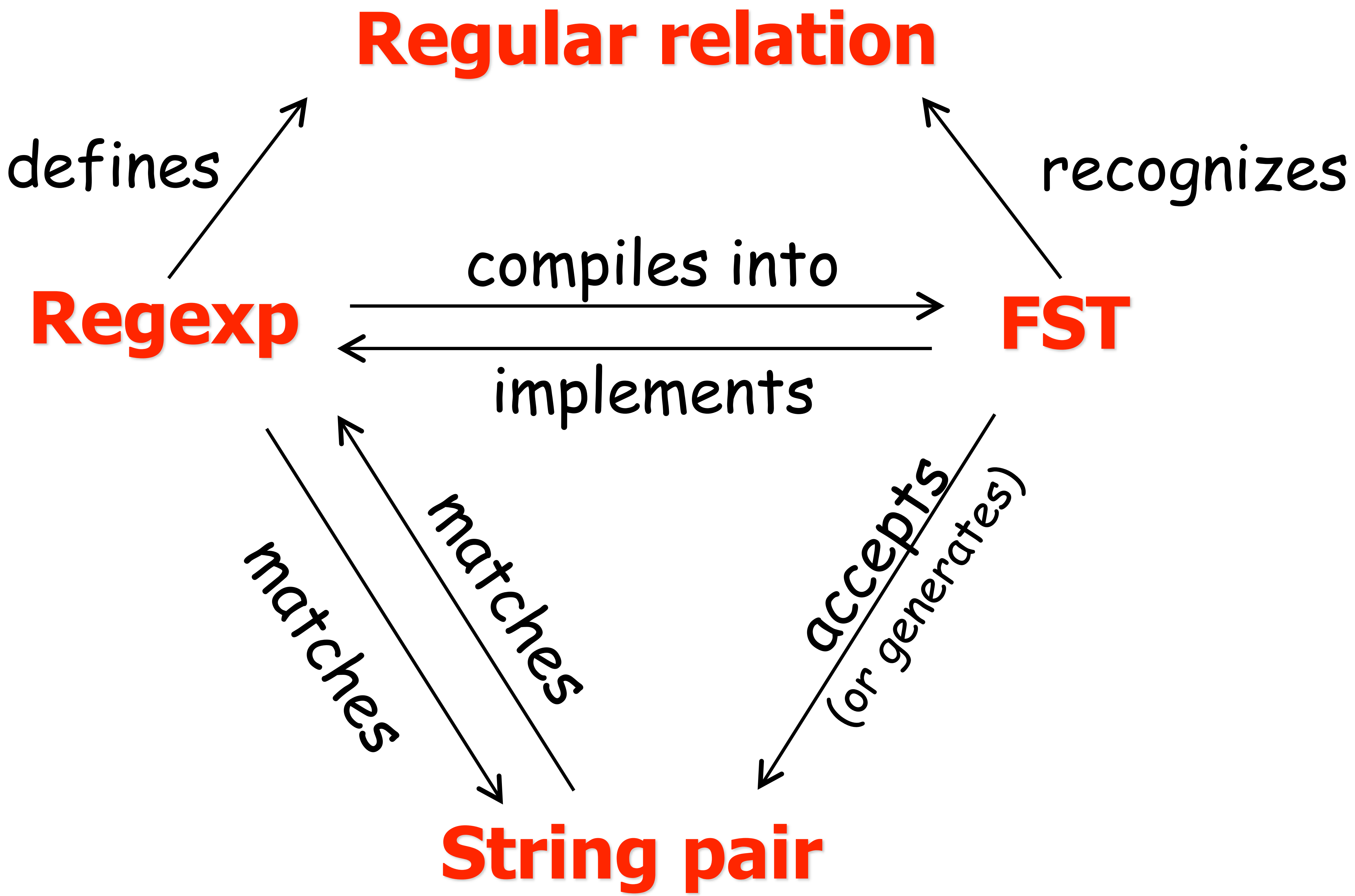
Terminology (acceptors)



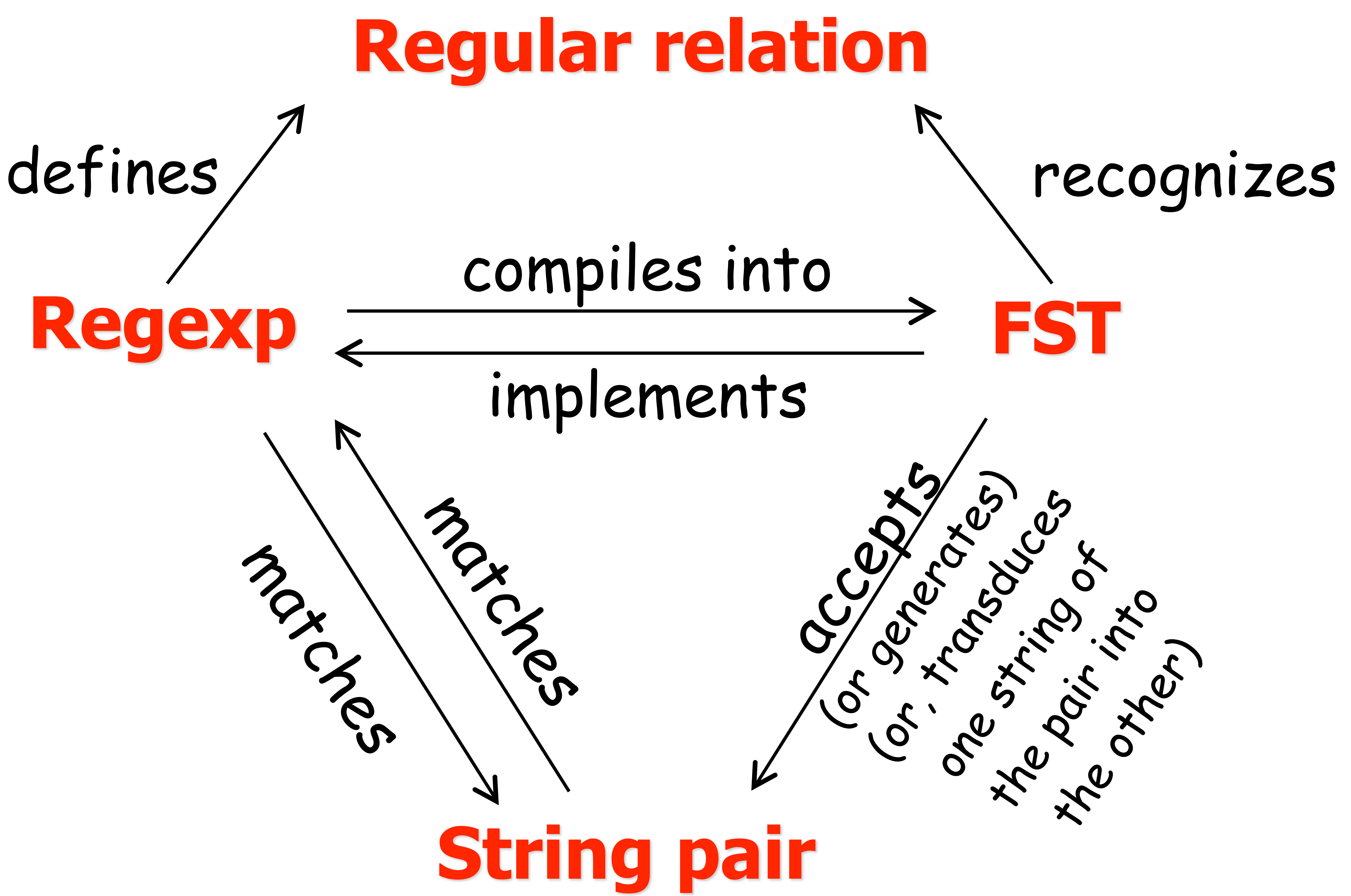
Terminology (acceptors)



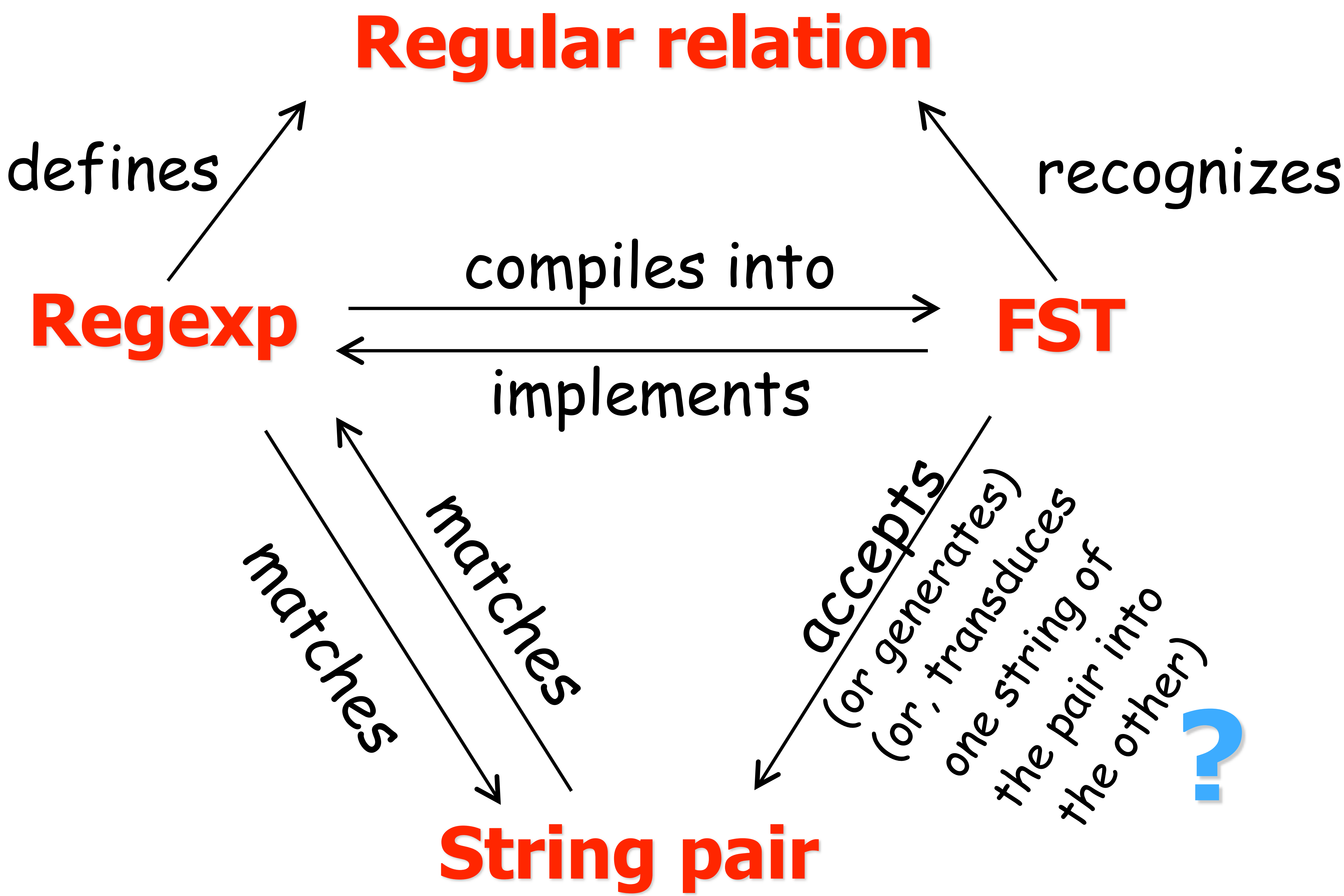
Terminology (transducers)



Terminology (transducers)

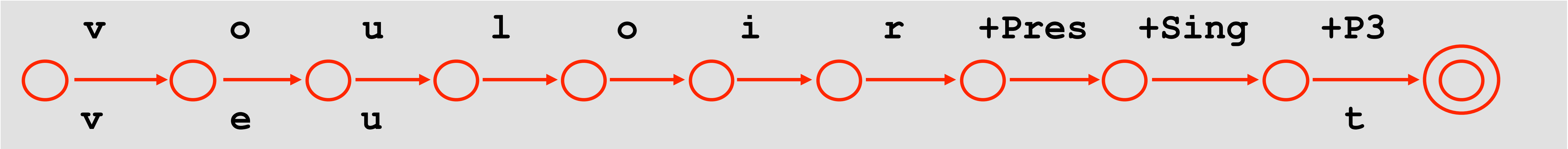


Terminology (transducers)



Perspectives on 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

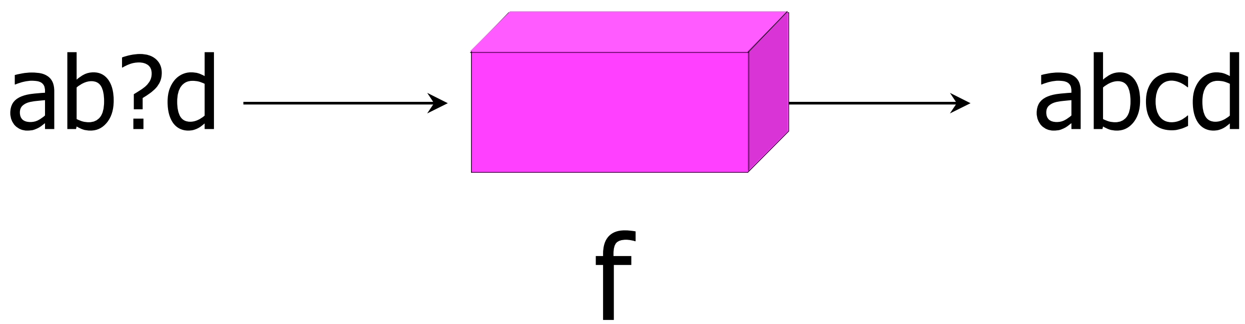


FST just defines the regular relation (mathematical object: set of pairs).
What's "input" and "output" depends on what one asks about the relation.
The 0, 1, or 2 given string(s) constrain which paths you can use.

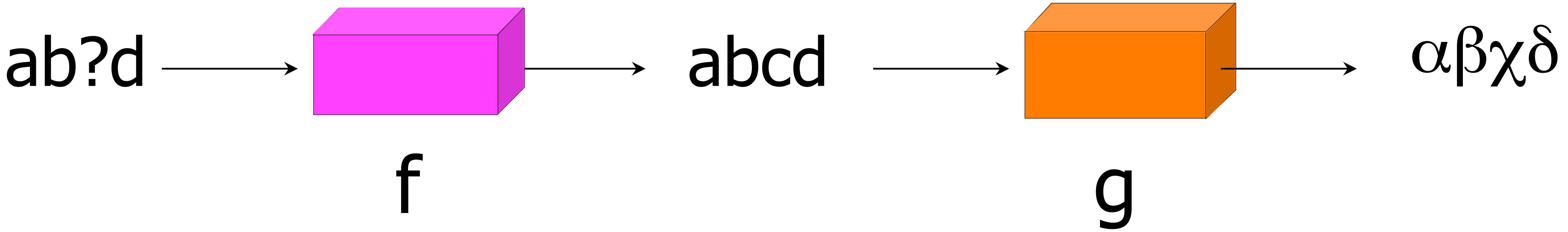
Functions

ab?d $\longrightarrow$

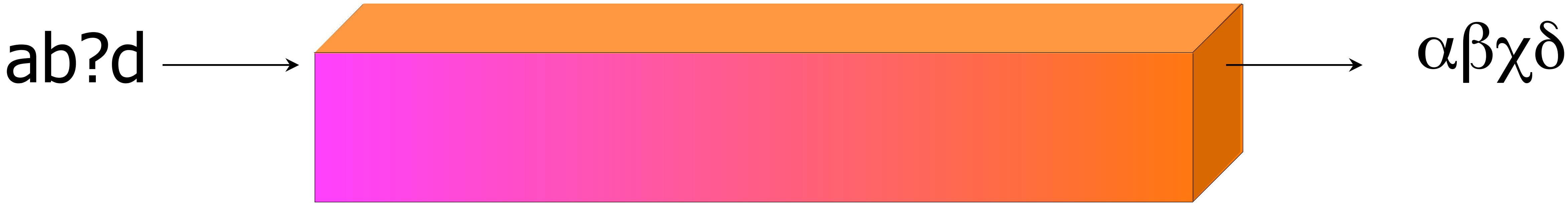
Functions



Functions



Functions



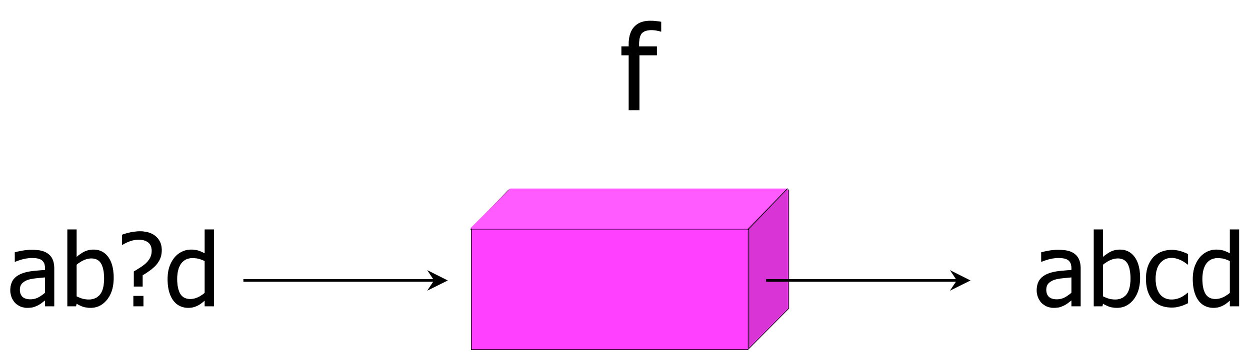
Function composition: f ° g

Functions

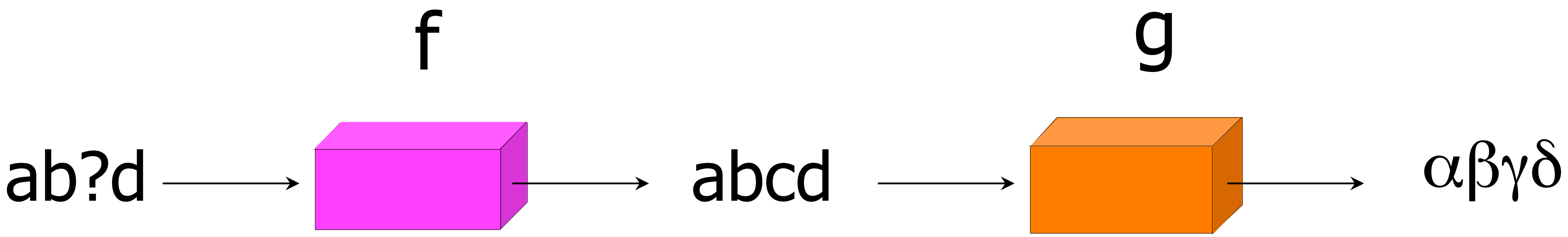
Function composition: $f \circ g$

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

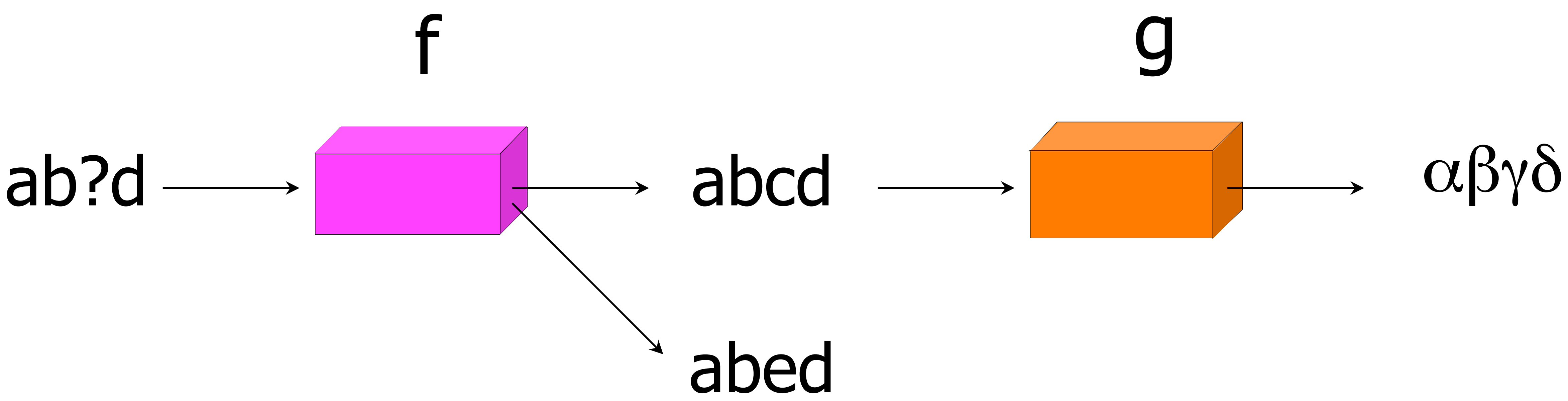
From Functions to Relations



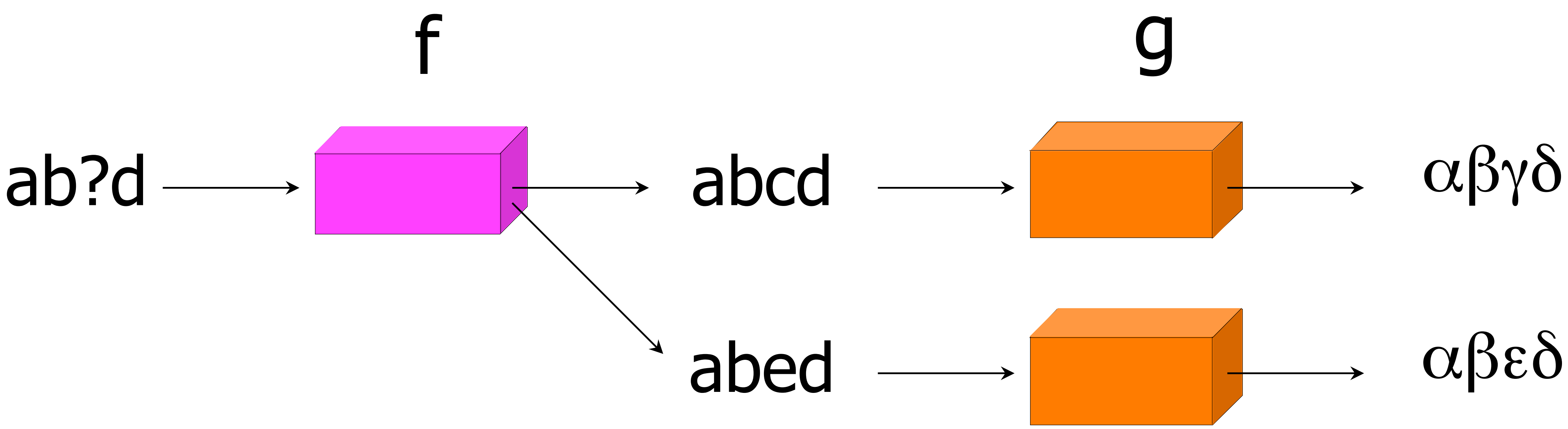
From Functions to Relations



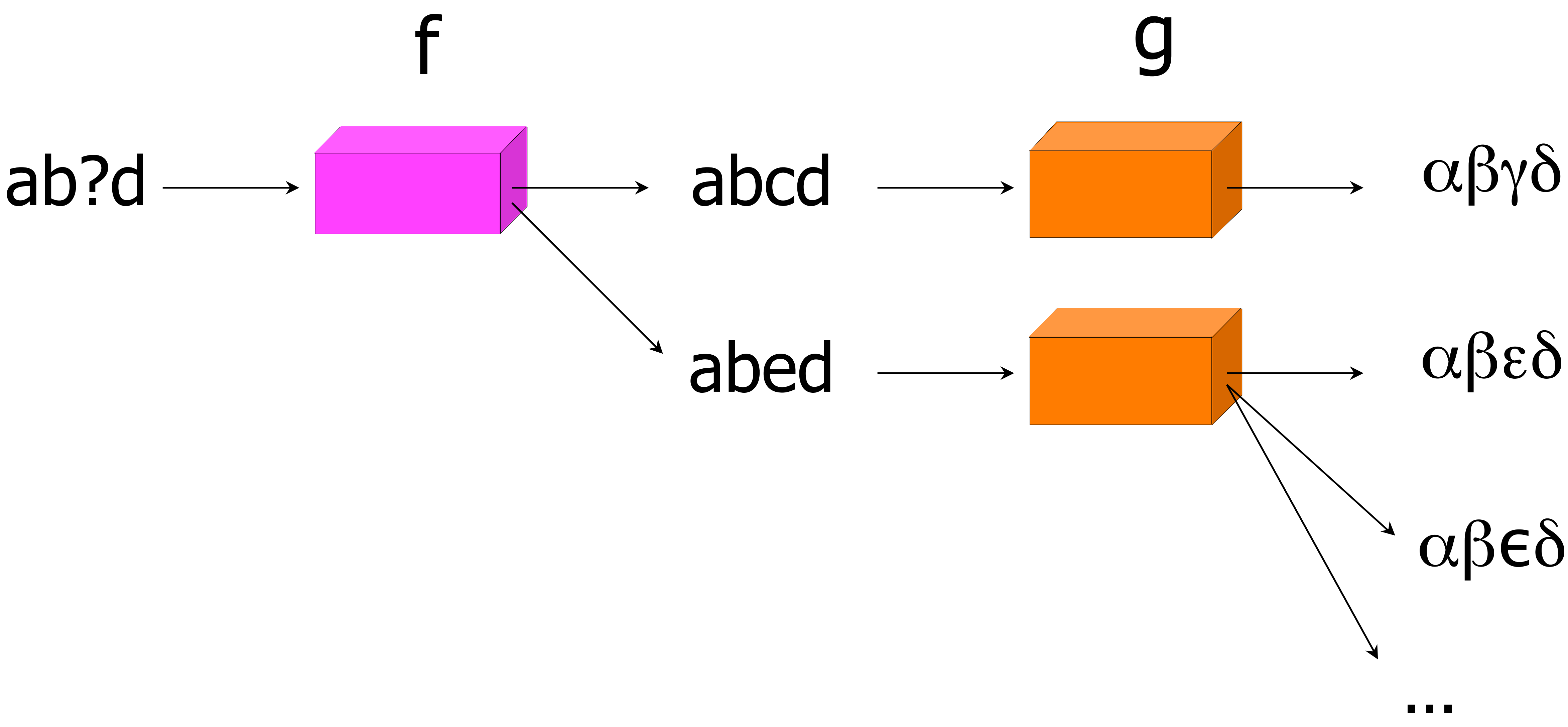
From Functions to Relations



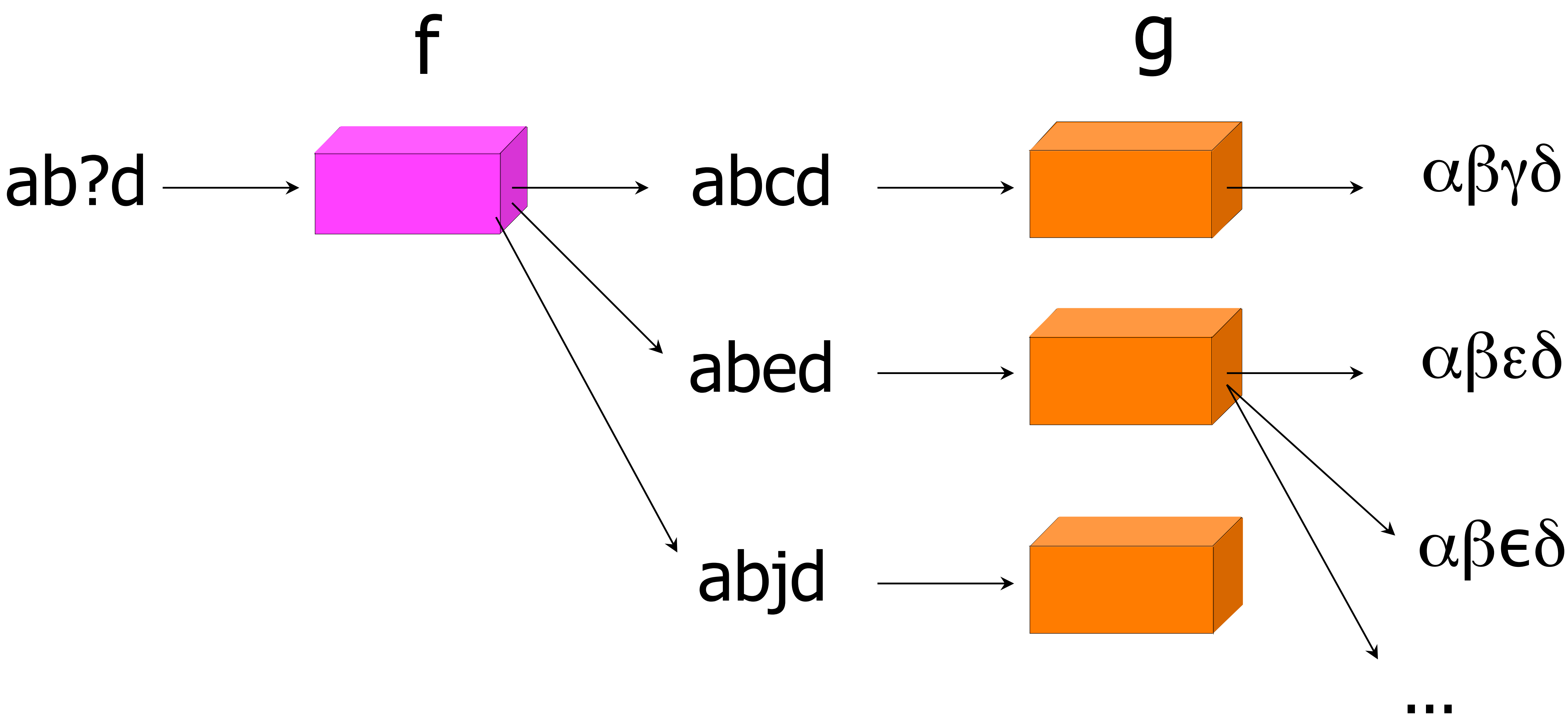
From Functions to Relations



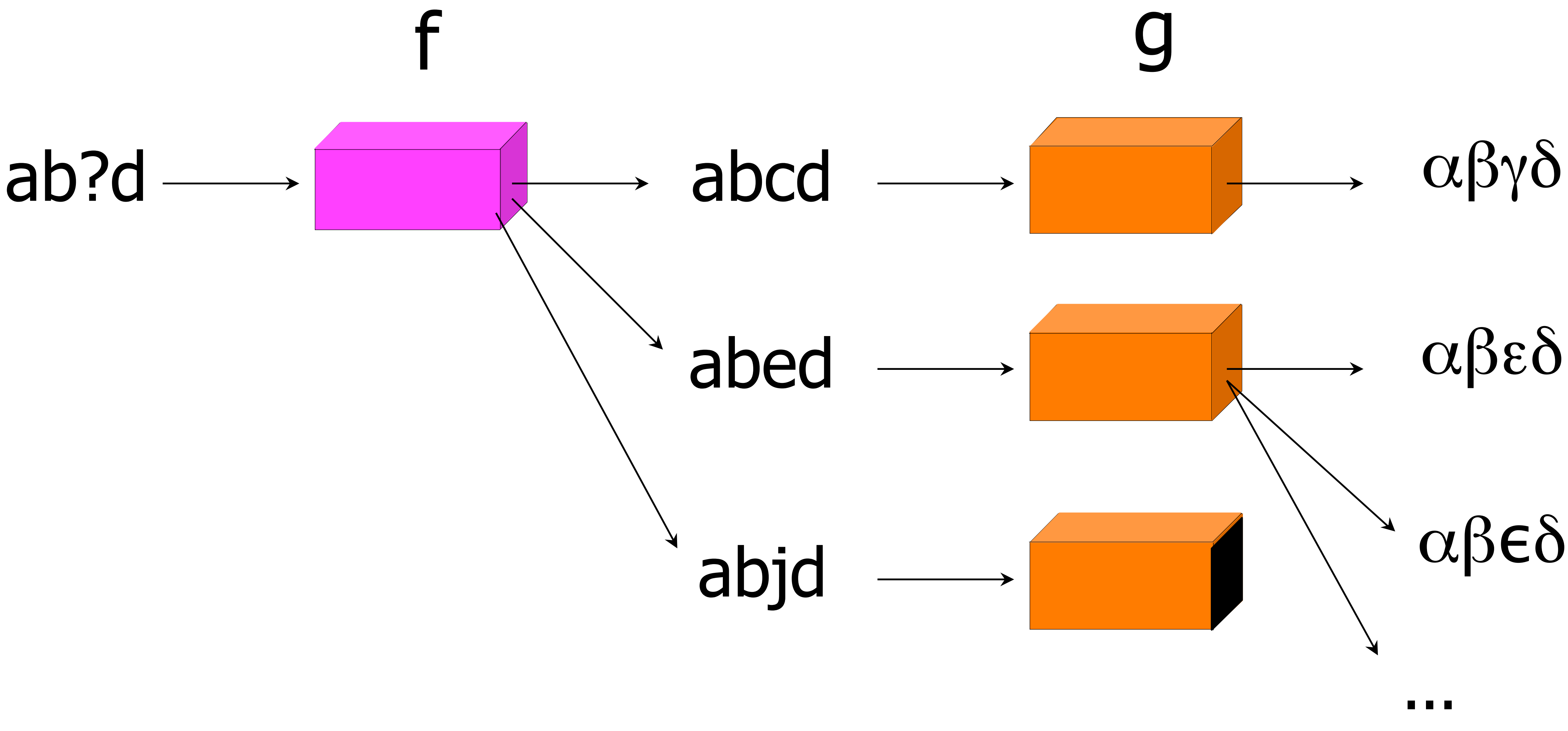
From Functions to Relations



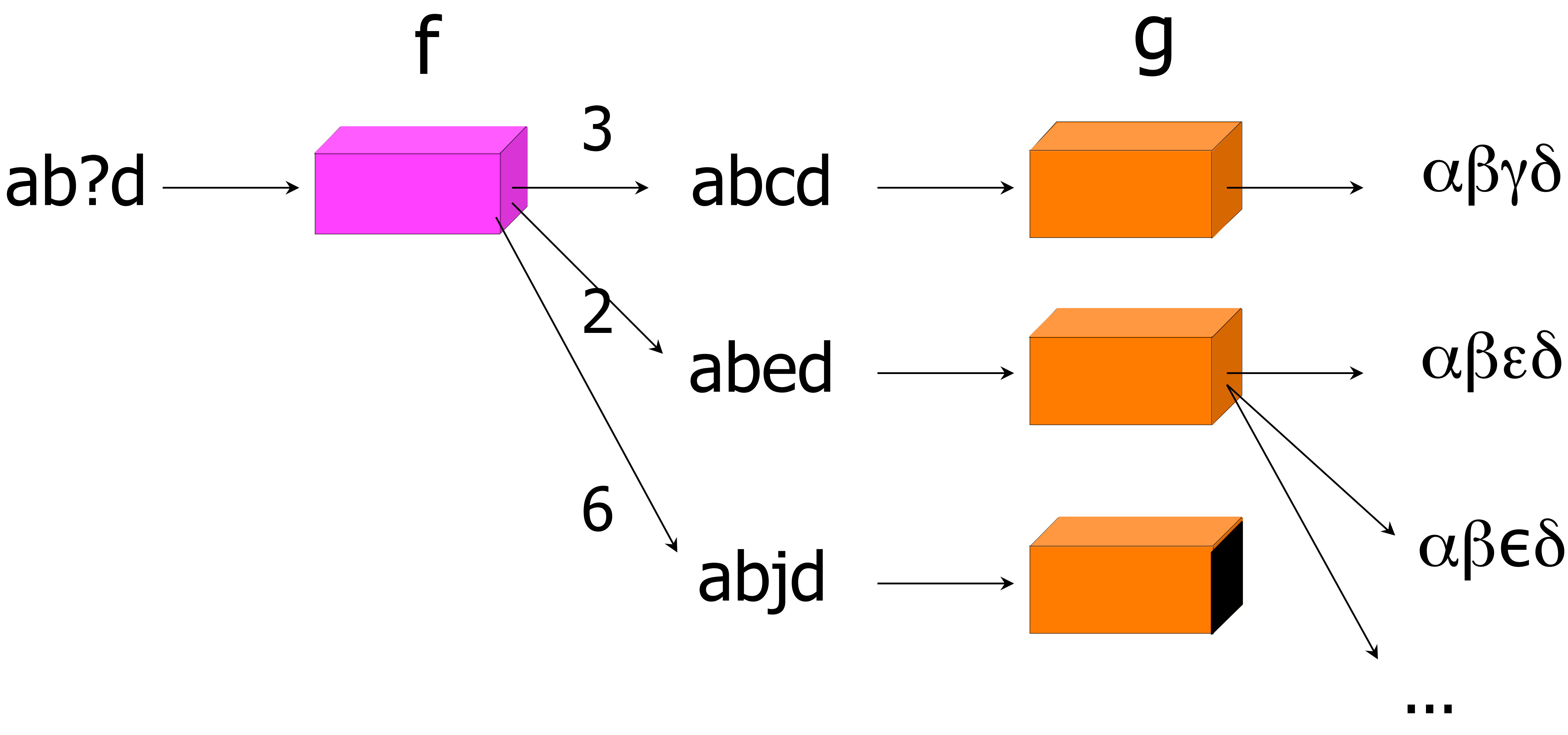
From Functions to Relations



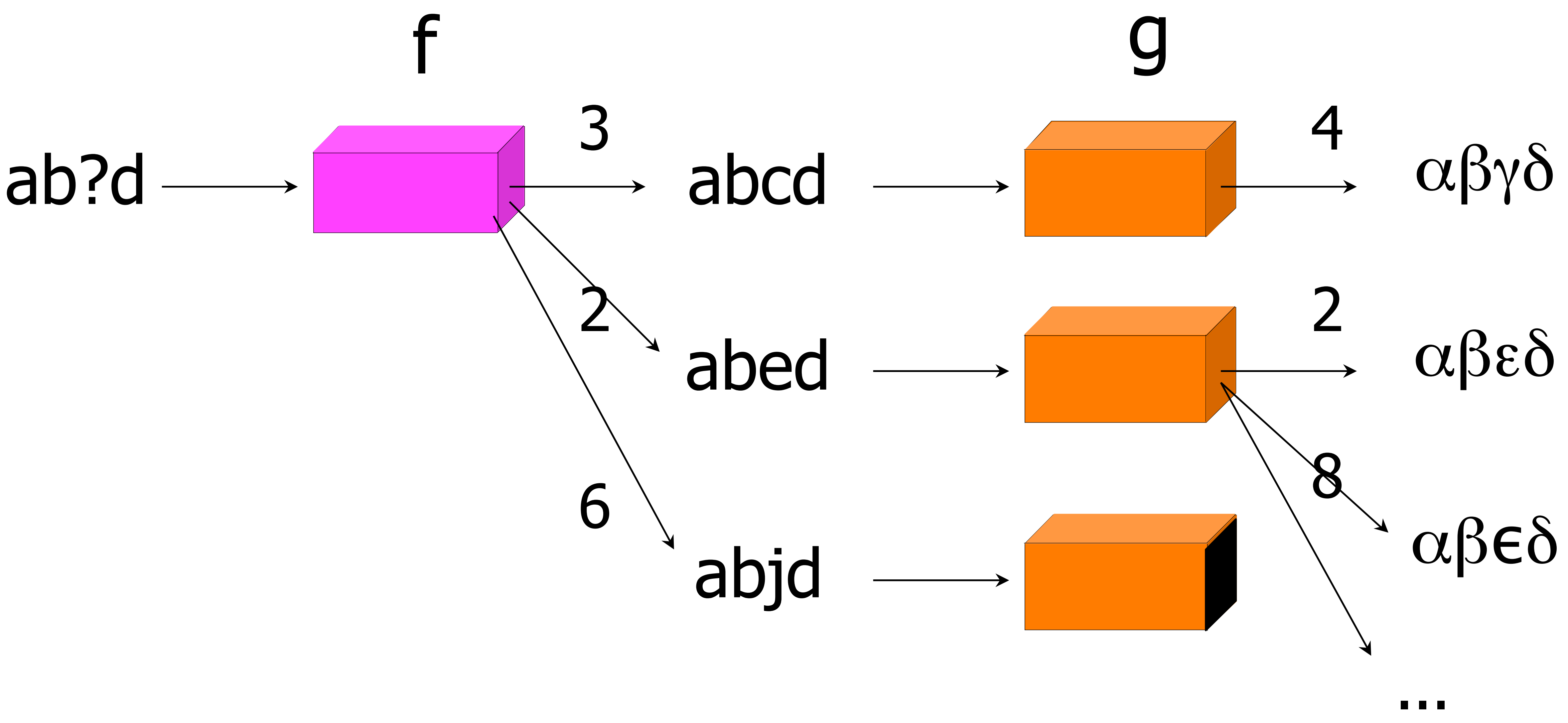
From Functions to Relations



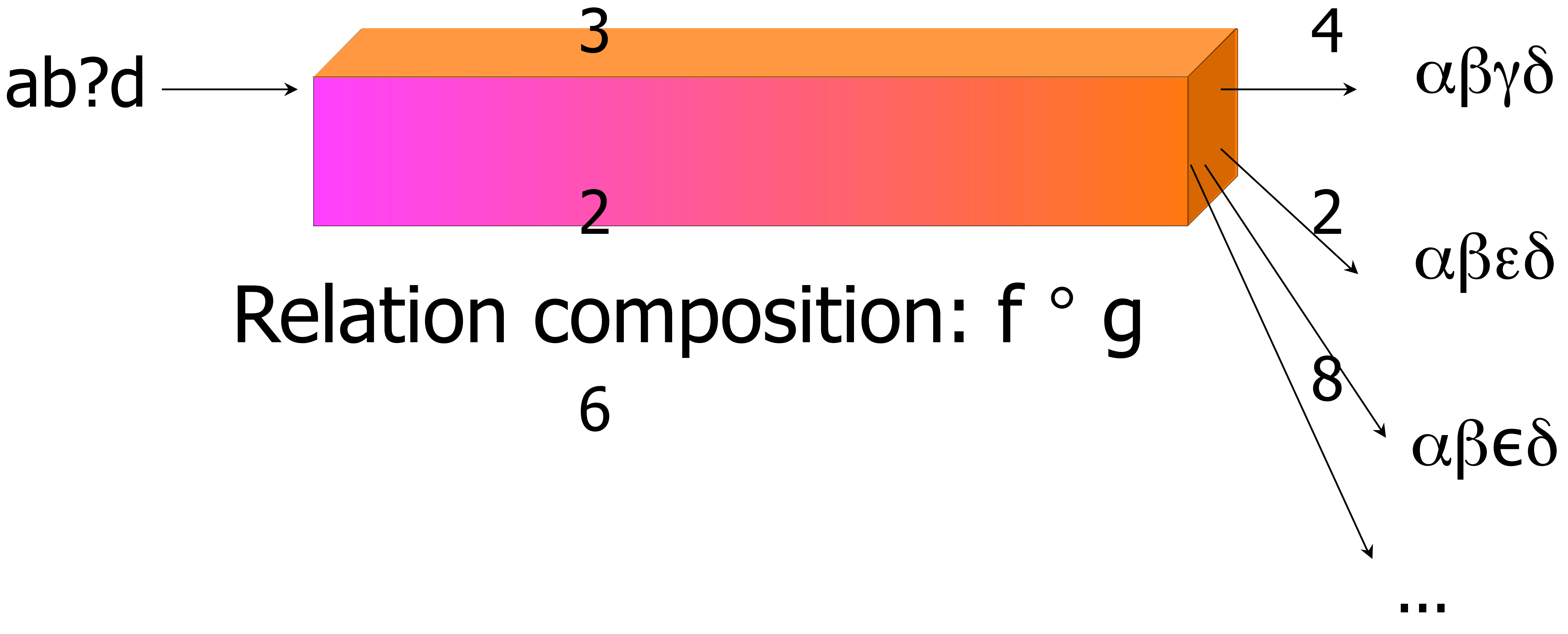
From Functions to Relations



From Functions to Relations

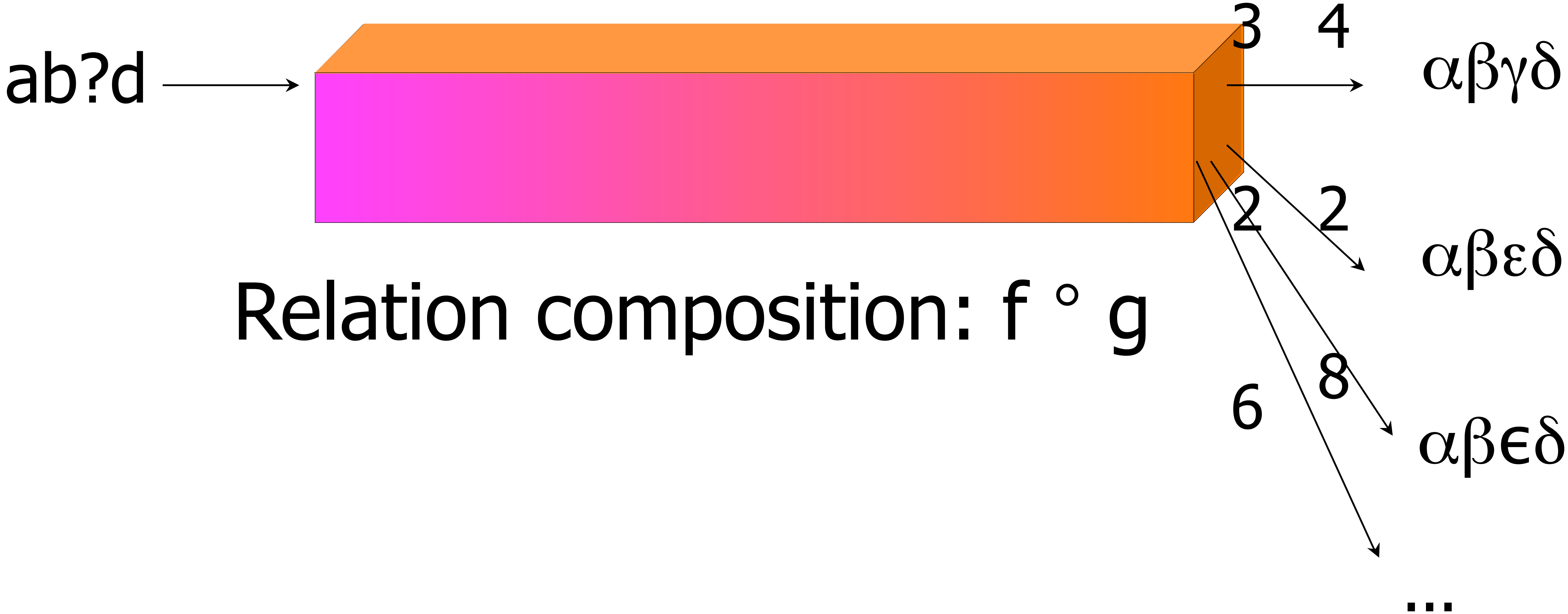


From Functions to Relations



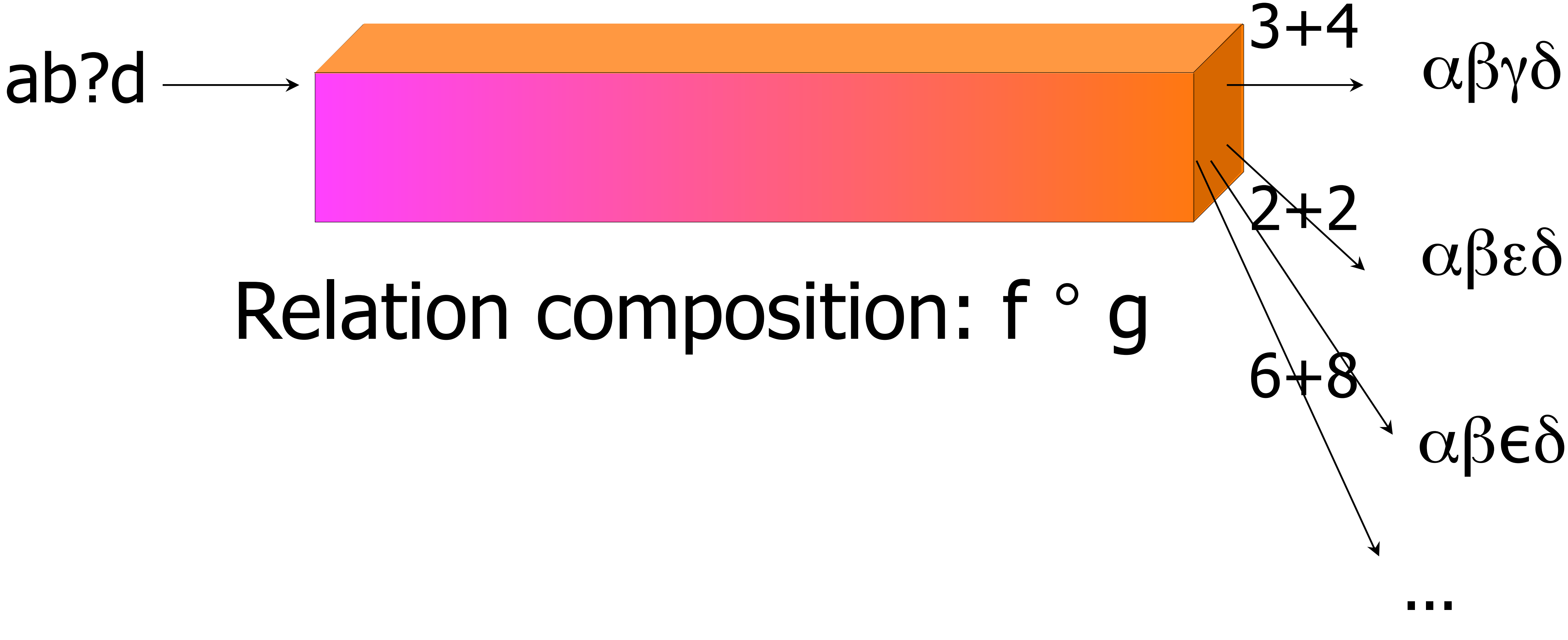
Relation composition: f ° g

From Functions to Relations



Relation composition: $f \circ g$

From Functions to Relations



From Functions to Relations

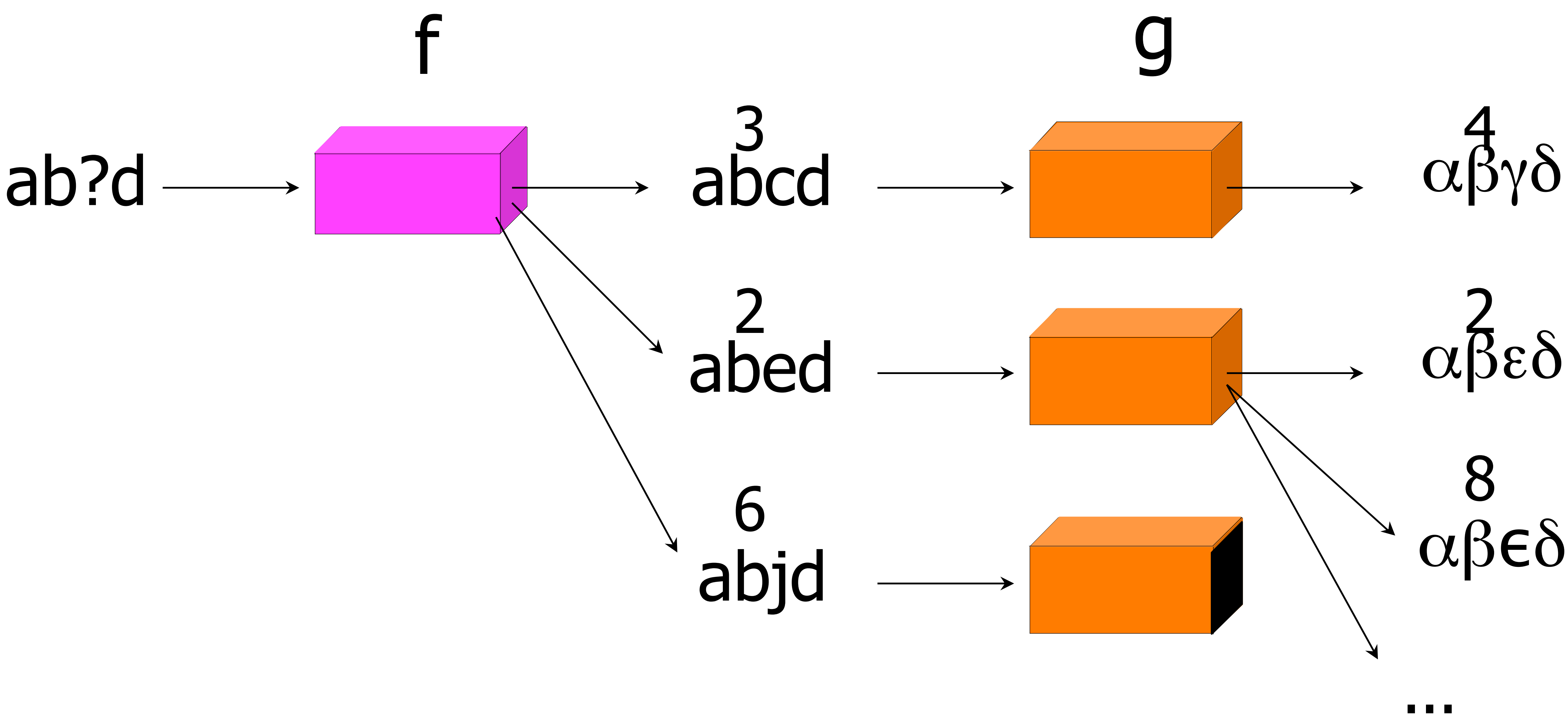
Pick min-cost or max-prob output

From Functions to Relations

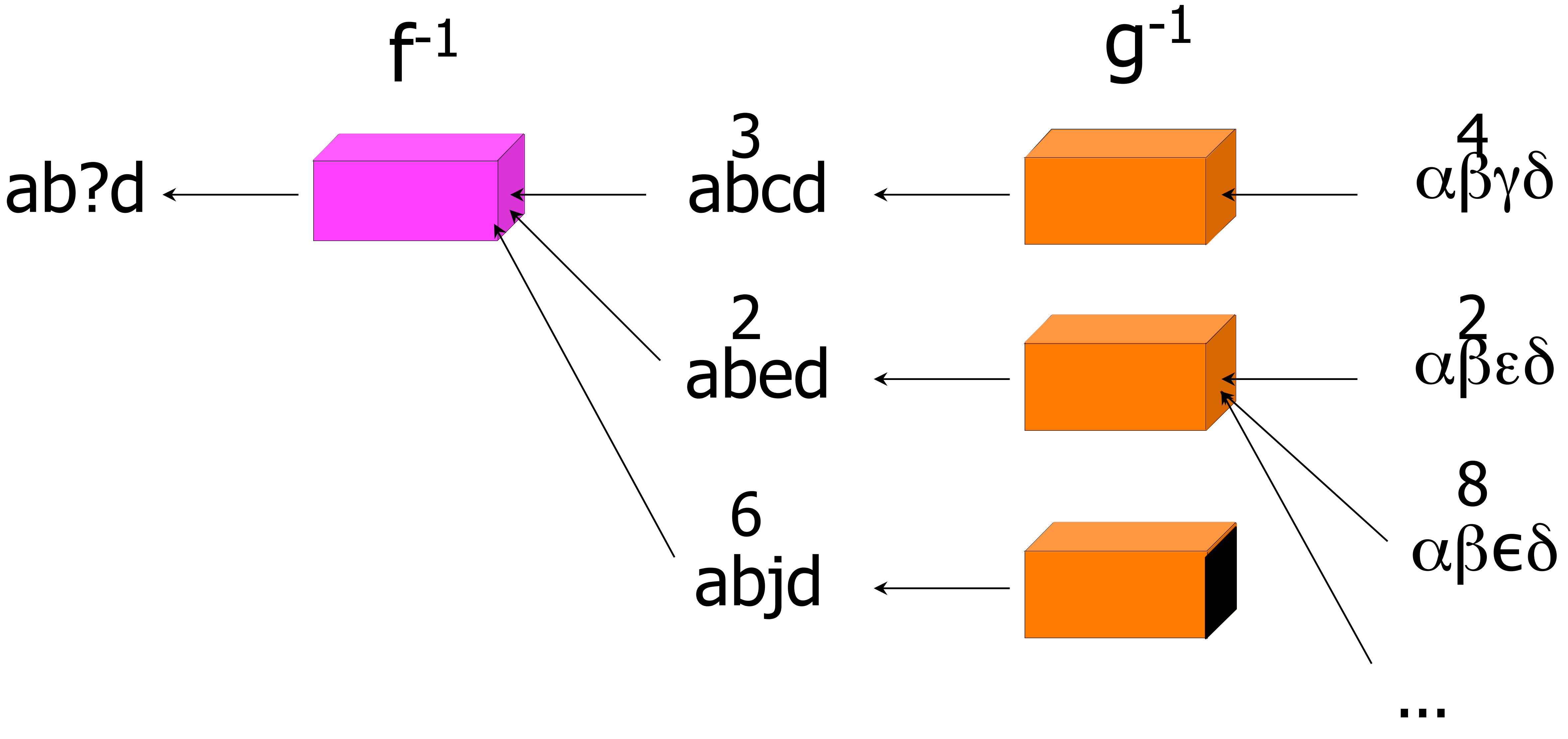
Pick min-cost or max-prob output

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

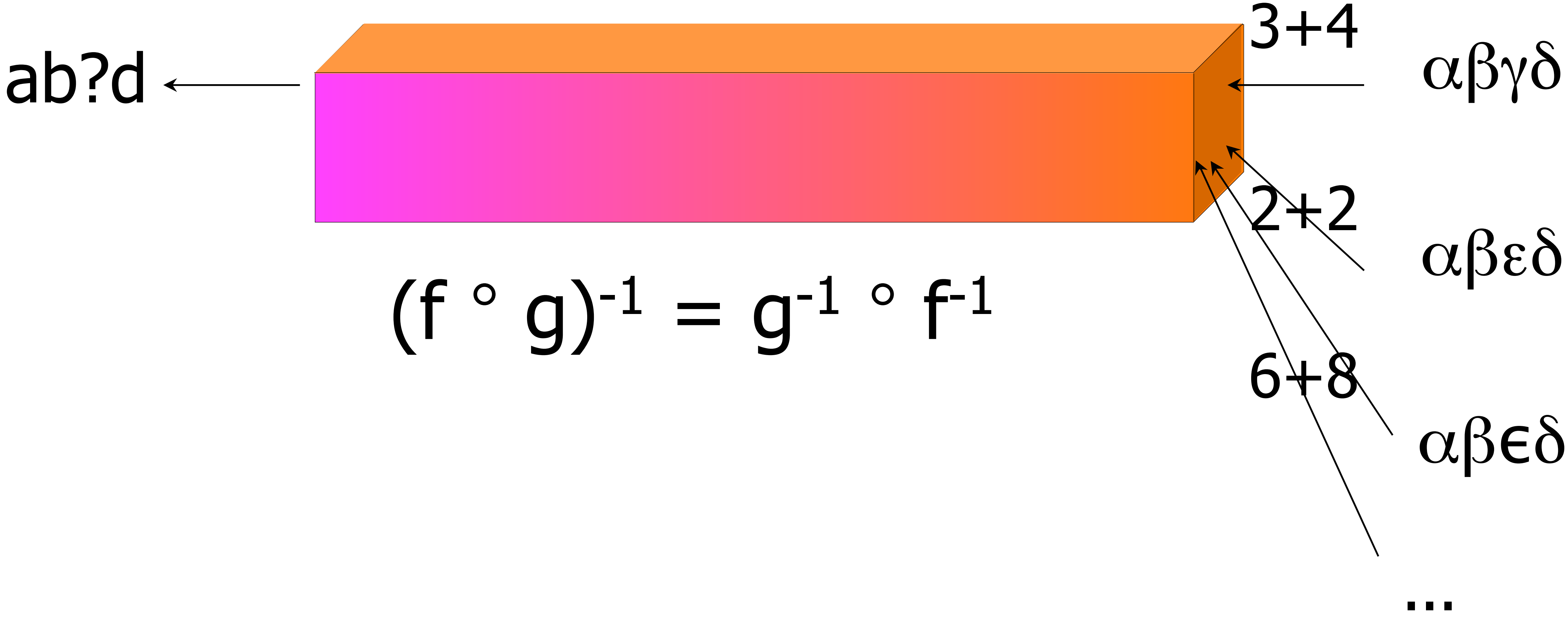
Inverting Relations



Inverting Relations



Inverting Relations



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

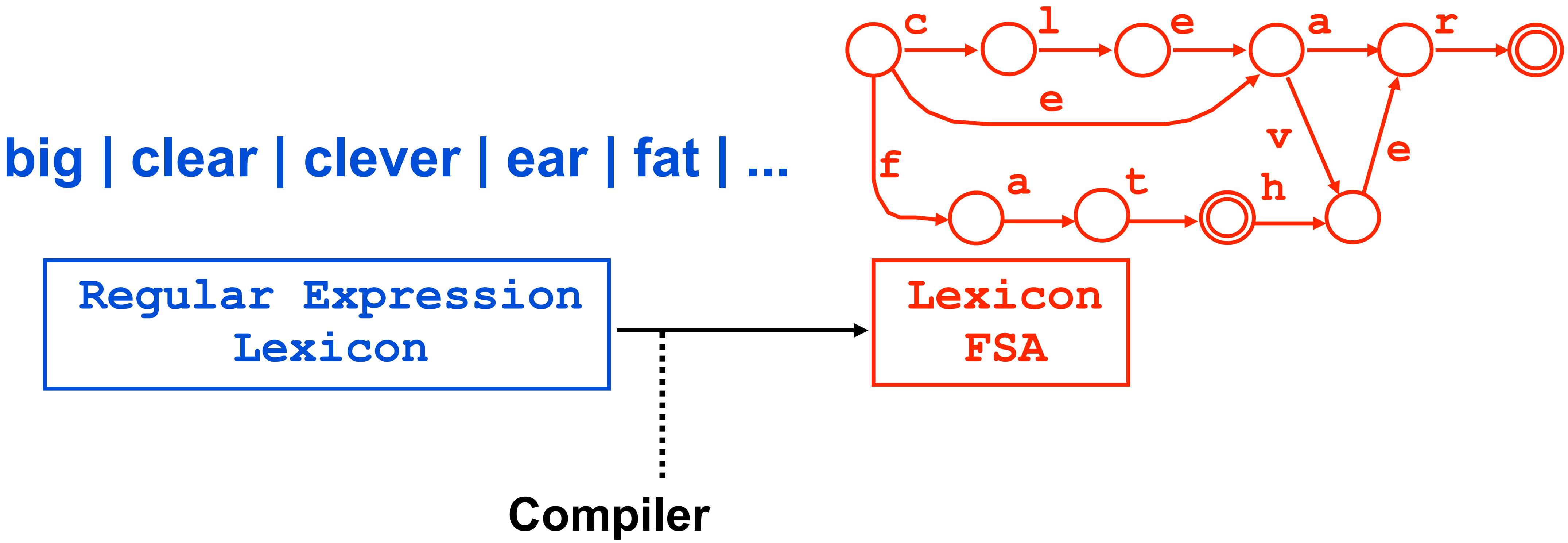
slide courtesy of L. Karttunen (modified)

Building a lexical transducer

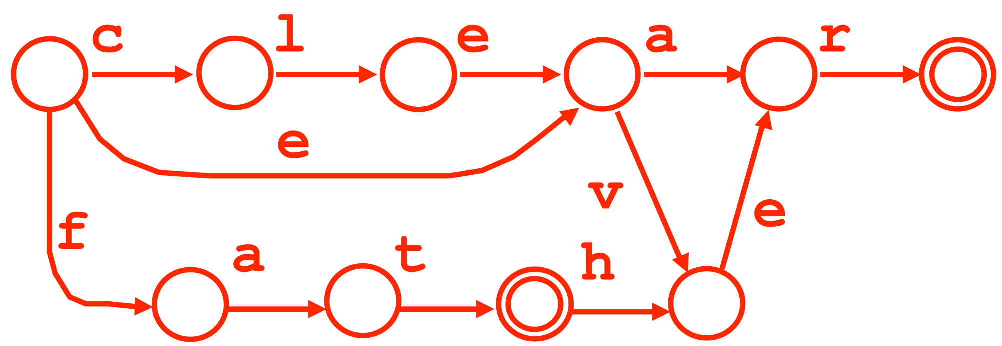
big | clear | clever | ear | fat | ...

Regular Expression
Lexicon

Building a lexical transducer



Building a lexical transducer



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

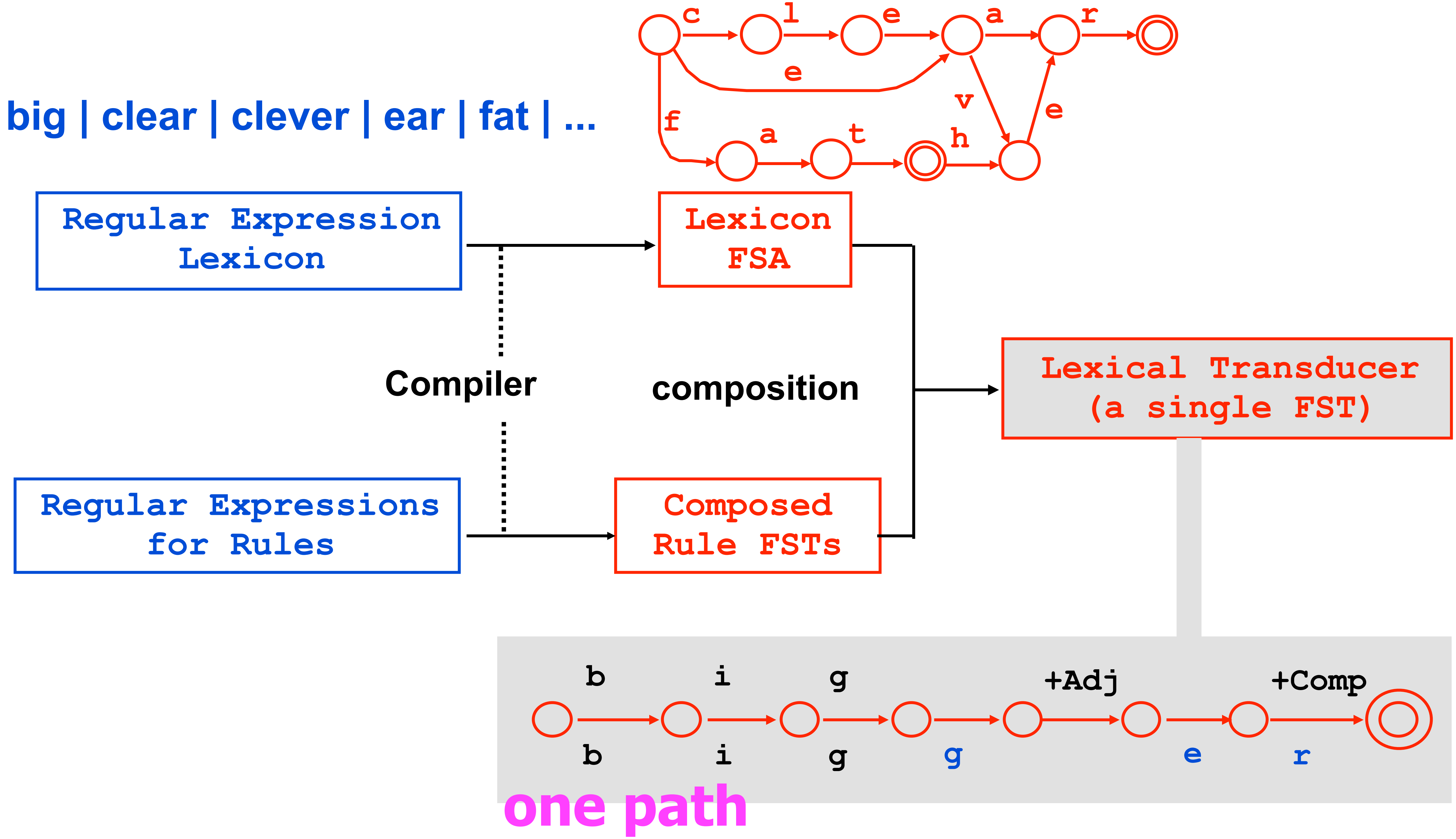
Regular Expression Lexicon → Lexicon FSA

Compiler

Regular Expressions for Rules → Composed Rule FSTs

slide courtesy of L. Karttunen (modified)

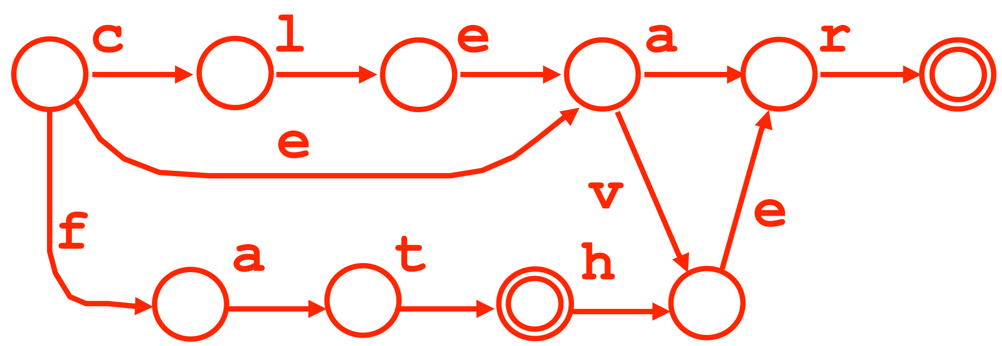
Building a lexical transducer



slide courtesy of L. Karttunen (modified)

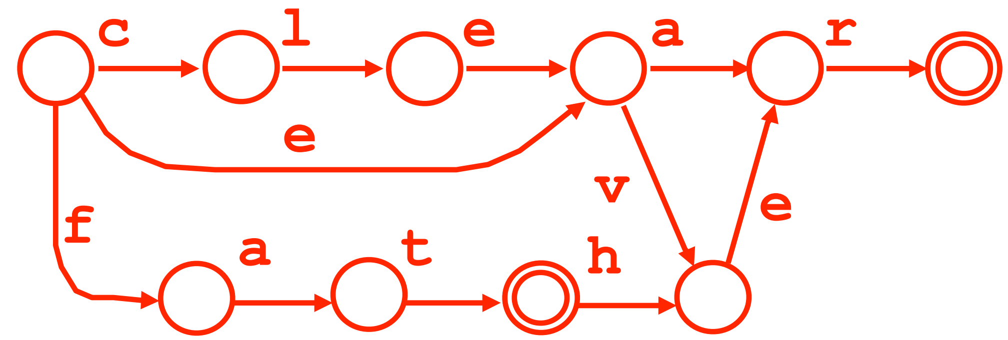
Building a lexical transducer

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

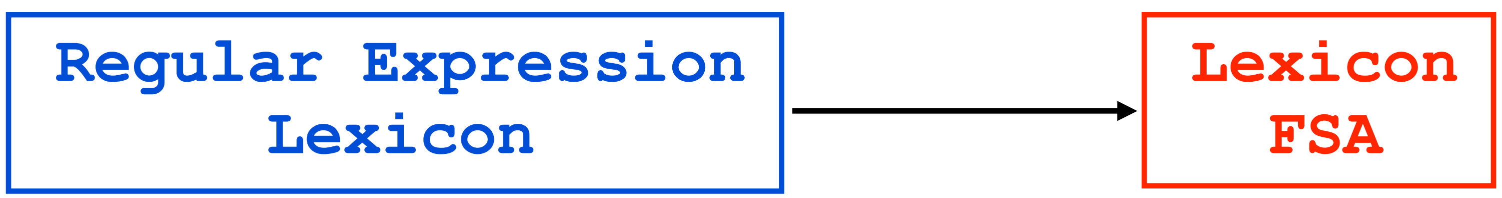


Regular Expression Lexicon → Lexicon FSA

Building a lexical transducer



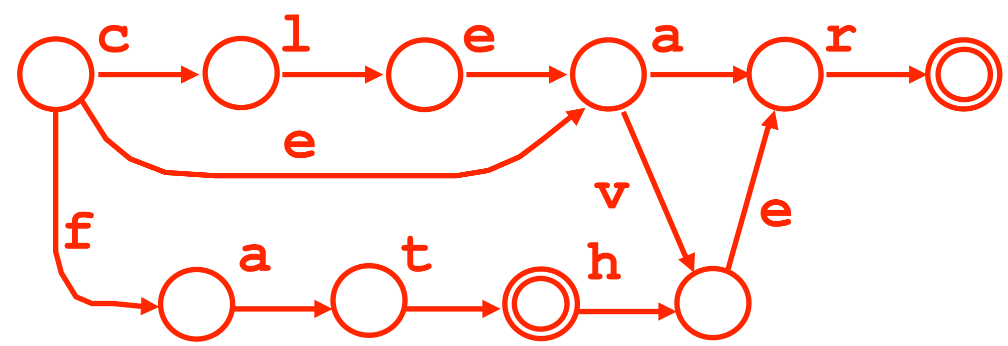
big | clear | clever | ear | fat | ...



- Actually, the lexicon must contain elements like
  big +Adj +Comp

Building a lexical transducer

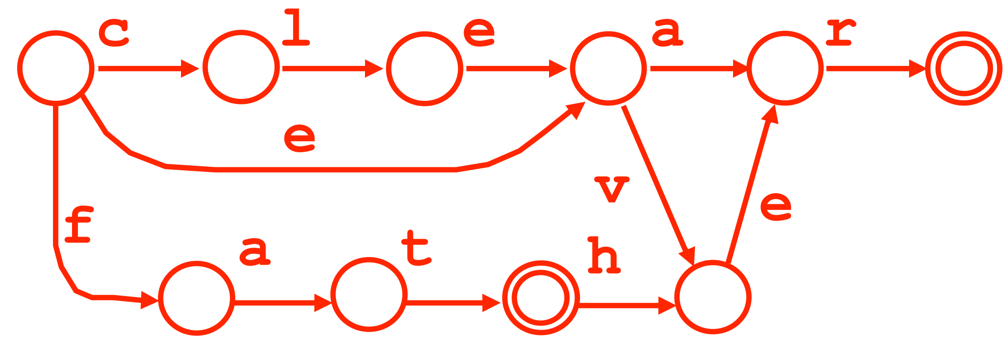
big | clear | clever | ear | fat | ...



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 (ε | +Comp | +Sup) ← *adjectives*
  | (ear | father | ...) +Noun (+Sing | +Pl) ← *nouns*
  | ... ← *...*

Building a lexical transducer



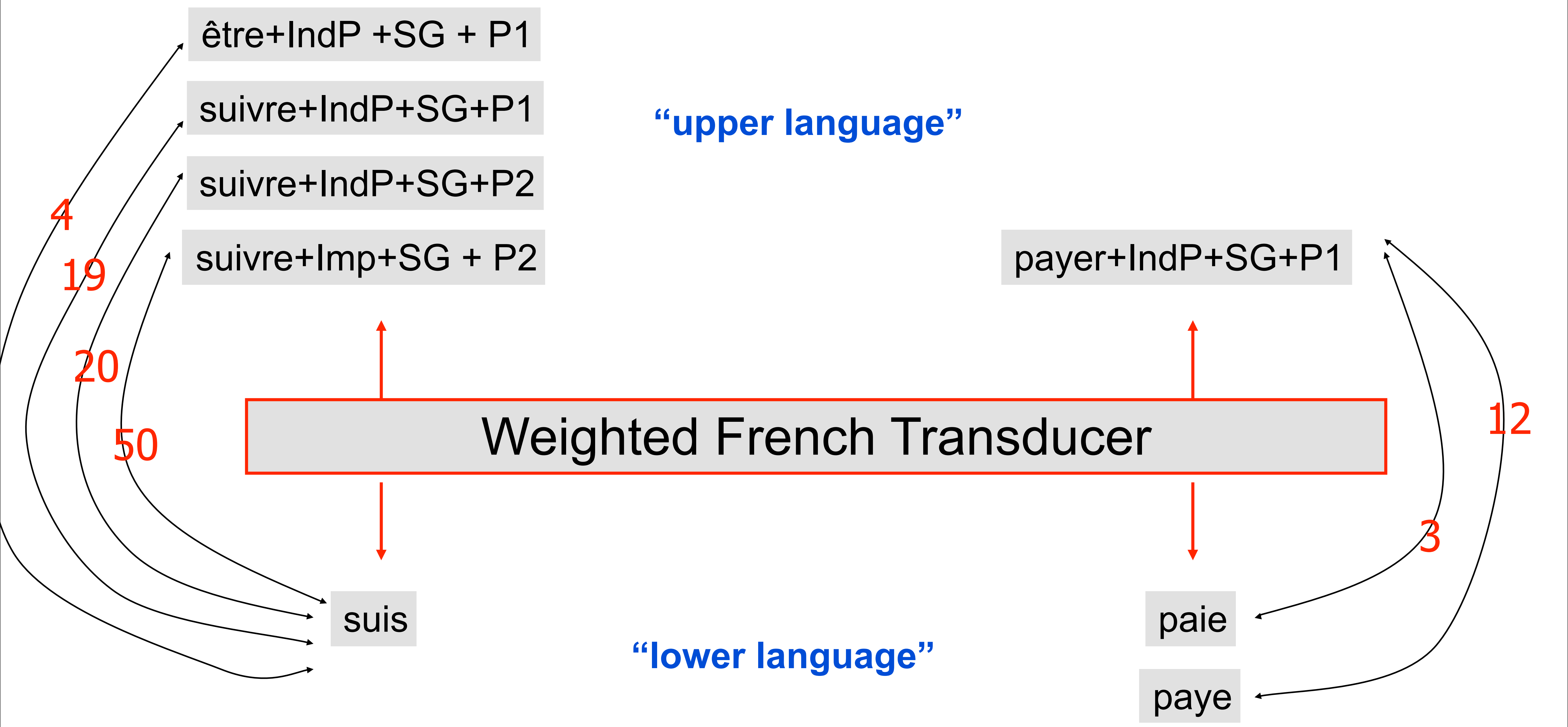
big | clear | clever | ear | fat | ...

`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 (ε | +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



Constructing Regular Languages

Xerox Regex Notation (Paper)

	concatenation	EF
`* +`	iteration	E*, E+
`|`	union	E | F
`&`	intersection	E & F
`~ \ -`	complementation, minus	~E, \x, F-E
`.x.`	crossproduct	E .x. F
`.o.`	composition	E .o. F
`.u`	upper (input) language	E.u "domain"
`.l`	lower (output) language	E.l "range"

Common Regular Expression Operators (in XFST notation)

concatenation EF

$$EF = \{ef : e \in E, f \in F\}$$

ef denotes the concatenation of 2 strings.

EF denotes the concatenation of 2 languages.

- To pick a string in EF, pick $e \in E$ and $f \in F$ and concatenate them.
- To find out whether $w \in EF$, look for at least one way to split w into two "halves," $w = ef$, such that $e \in E$ and $f \in F$.

A **language** is a set of strings.

It is a **regular language** if there exists an FSA that accepts all the strings in the language, and no other strings.

If E and F denote regular languages, than so does EF. (We will have to prove this by finding the FSA for EF!)

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
* +	iteration	E*, E+

$$E^* = \{e_1e_2 \ldots e_n : n \geq 0, e_1 \in E, \ldots e_n \in E\}$$

- To pick a string in E*, pick any number of strings in E and concatenate them.
- To find out whether $w \in E^*$, look for at least one way to split w into 0 or more sections, $e_1e_2 \ldots e_n$, all of which are in E.

$$E+ = \{e_1e_2 \ldots e_n : n>0, e_1 \in E, \ldots e_n \in E\} = EE^*$$

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
* +	iteration	E*, E+
|	union	E | F

$$E \mid F = \{w: w \in E \text{ or } w \in F\} = E \cup F$$

- To pick a string in E | F, pick a string from either E or F.
- To find out whether $w \in E \mid F$, check whether $w \in E$ or $w \in F$.

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
`*` `+`	iteration	E*, E+
`|`	union	E | F
`&`	intersection	E & F

$$E \,\&\, F = \{w: w \in E \text{ and } w \in F\} = E \cap F$$

- To pick a string in E & F, pick a string from E that is also in F.
- To find out whether $w \in E \,\&\, F$, check whether $w \in E$ and $w \in F$.

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
* +	iteration	E*, E+
|	union	E | F
&	intersection	E & F
~ \ -	complementation, minus	~E, \x, F-E

$\sim E = \{e: e \notin E\} = \Sigma^* - E$

$E - F = \{e: e \in E \text{ and } e \notin F\} = E \,\&\, \sim F$

$\backslash E = \Sigma - E$ (any single character not in E)

Σ is set of all letters; so Σ* is set of all strings; ?* in XFST

Regular Expressions

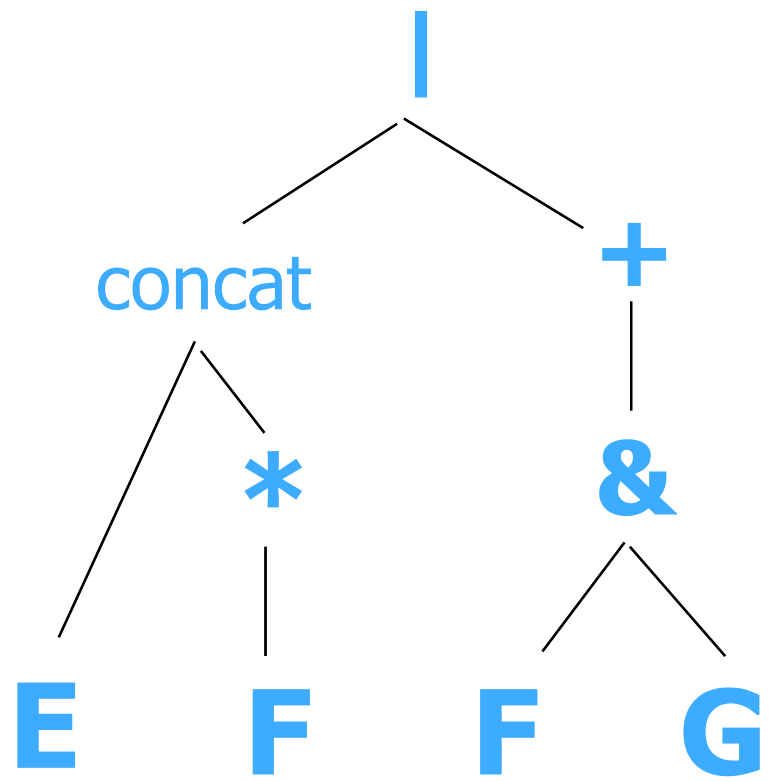
A **language** is a set of strings.

It is a **regular language** if there exists an FSA that accepts all the strings in the language, and no other strings.

If E and F denote regular languages, than so do EF, etc.

Regular expression: EF*|(F & G)+

Syntax:



Semantics:

Denotes a regular language.
As usual, can build semantics compositionally bottom-up.

E, F, G must be regular languages.
As a base case, e denotes {e} (a language containing a single string), so ef*|(f&g)+ is regular.

Regular Expressions for Regular Relations

A **language** is a set of strings.

It is a **regular language** if there exists an FSA that accepts all the strings in the language, and no other strings.

If E and F denote regular languages, than so do EF, etc.

A **relation** is a set of pairs – here, pairs of strings.

It is a **regular relation** if here exists an FST that accepts all the pairs in the language, and no other pairs.

If E and F denote regular relations, then so do EF, etc.

$$EF = \{(ef, e'f') : (e,e') \in E, (f,f') \in F\}$$

Can you guess the definitions for E*, E+, E | F, E & F when E and F are regular relations?

Surprise: E & F isn't necessarily regular in the case of relations; so not supported.

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
`* +`	iteration	E*, E+
`|`	union	E | F
`&`	intersection	E & F
`~ \ -`	complementation, minus	~E, \x, F-E
`.x.`	crossproduct	E .x. F

$$E \text{ .x. } F = \{(e,f) : e \in E, f \in F\}$$

- Combines two regular <u>languages</u> into a regular <u>relation</u>.

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
`* +`	iteration	E*, E+
`|`	union	E | F
`&`	intersection	E & F
`~ \ -`	complementation, minus	~E, \x, F-E
`.x.`	crossproduct	E .x. F
`.o.`	composition	E .o. F

$$E\ .o.\ F = \{(e,f) : \exists m.\ (e,m) \in E,\ (m,f) \in F\}$$

- Composes two regular relations into a regular relation.
- As we've seen, this generalizes ordinary function composition.

Common Regular Expression Operators (in XFST notation)

	concatenation	EF
`* +`	iteration	E*, E+
`|`	union	E | F
`&`	intersection	E & F
`~ \ -`	complementation, minus	~E, \x, F-E
`.x.`	crossproduct	E .x. F
`.o.`	composition	E .o. F
`.u`	upper (input) language	E.u “domain”

$$E.u = \{e : \exists m.\ (e,m) \in E\}$$

Common Regular Expression Operators (in XFST notation)

	concatenation	EF	
`* +`	iteration	E*, E+	
`|`	union	E | F	
`&`	intersection	E & F	
`~ \ -`	complementation, minus	~E, \x, F-E	
`.x.`	crossproduct	E .x. F	
`.o.`	composition	E .o. F	
`.u`	upper (input) language	E.u	“domain”
`.l`	lower (output) language	E.l	“range”

Finite-State Programming

Finite-state “programming”

Function	Function on (set of) strings
Source code	Regular expression
Object code	Finite state machine
Compiler	Regexp compiler
Optimization of object code	Determinization, minimization, pruning

Finite-state "programming"

Function composition	(Weighted) composition
Higher-order function	Operator
Function inversion (available in Prolog)	Function inversion
Structured programming	Ops + small regexps

Finite-state "programming"

Parallelism	Apply to set of strings
Nondeterminism	Nondeterminism
Stochasticity	Prob.-weighted arcs

slide courtesy of L. Karttunen (modified)

Some Xerox Extensions

`$`	containment
`=>`	restriction
`-> @->`	replacement

Make it easier to describe complex languages and relations without extending the formal power of finite-state systems.

Containment

Containment

`$[ab*c]`

"Must contain a substring that matches `ab*c`."

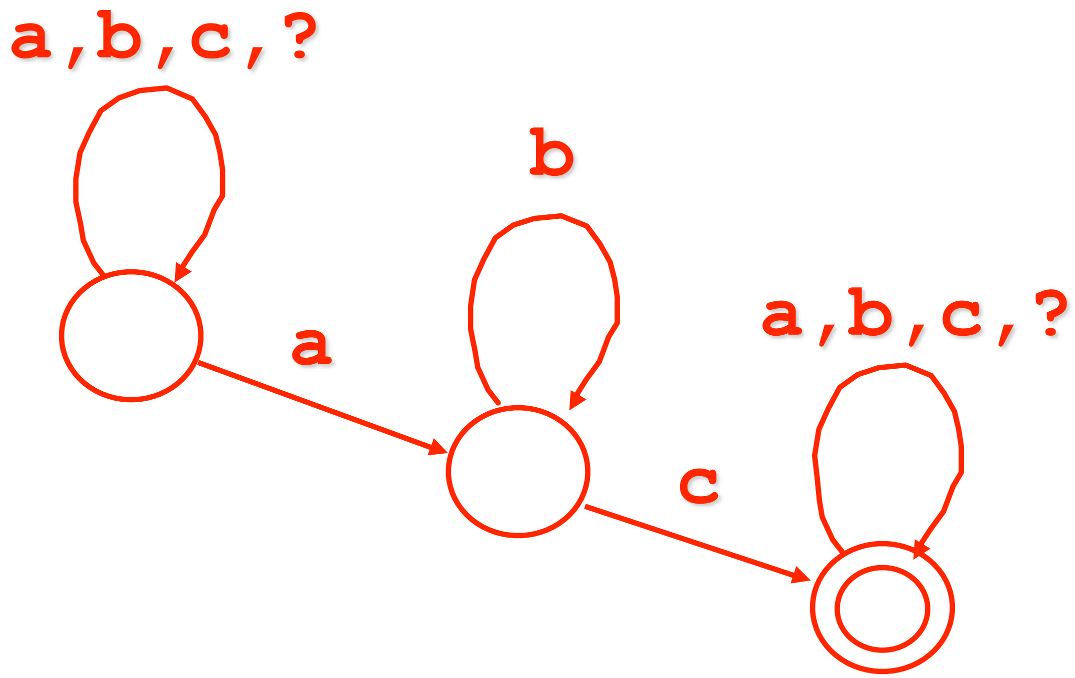
Accepts **`xxxacyy`**
Rejects **`bcba`**

Containment

`$[ab*c]`

“Must contain a substring that matches **`ab*c`**.”

Accepts **`xxxacyy`**
Rejects **`bcba`**

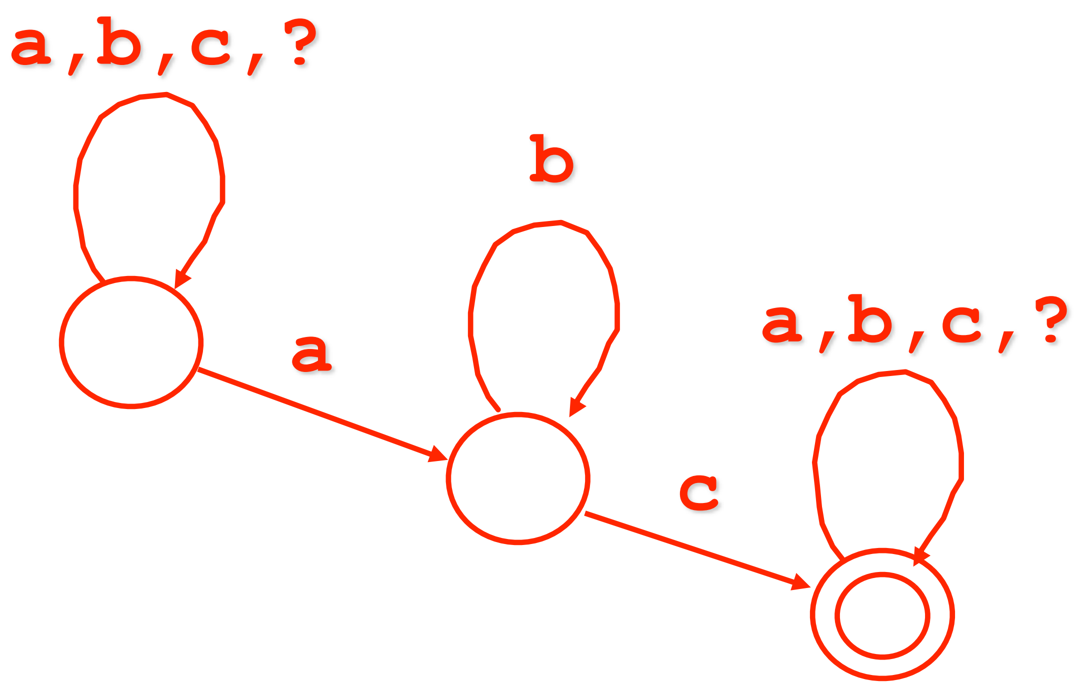


Containment

`$[ab*c]`

"Must contain a substring that matches `ab*c`."

Accepts **xxxacyy**
Rejects **bcba**



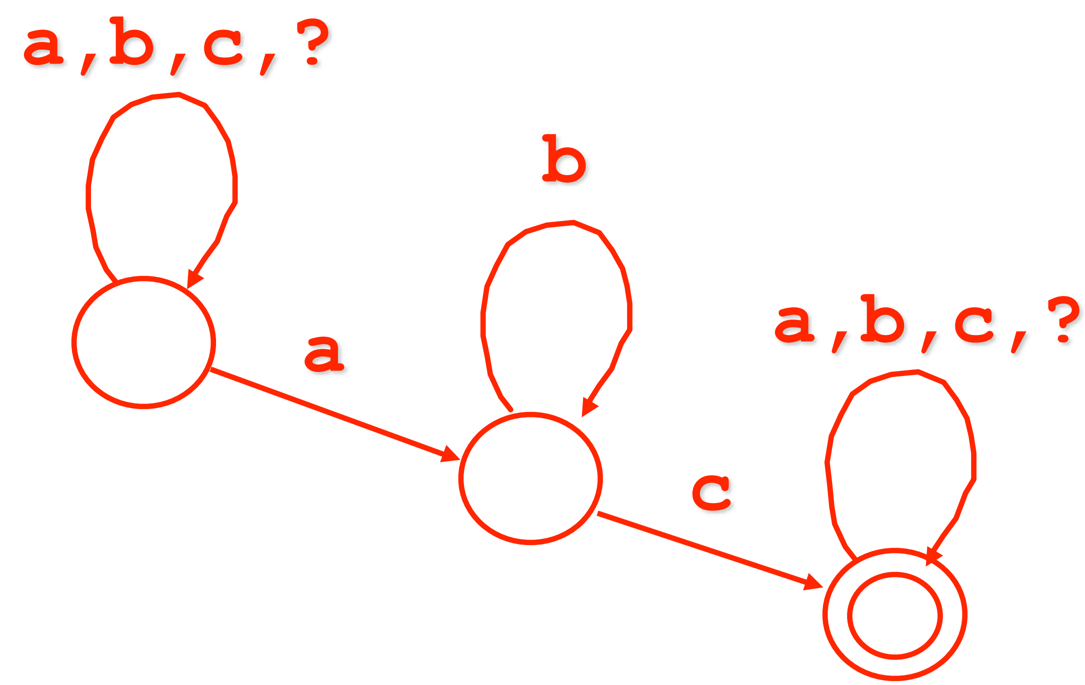
`?* [ab*c] ?*`

Equivalent expression

Containment

$[ab*c]

"Must contain a substring that matches ab*c."



Accepts **xxxacyy**
Rejects **bcba**

Warning: ? in regexps means "any character at all."
But ? in machines means "any character not explicitly mentioned anywhere in the machine."

?* [ab*c] ?*

Equivalent expression

slide courtesy of L. Karttunen (modified)

Restriction

Restriction

`a => b _ c`

"Any `a` must be preceded by `b` and followed by `c`."

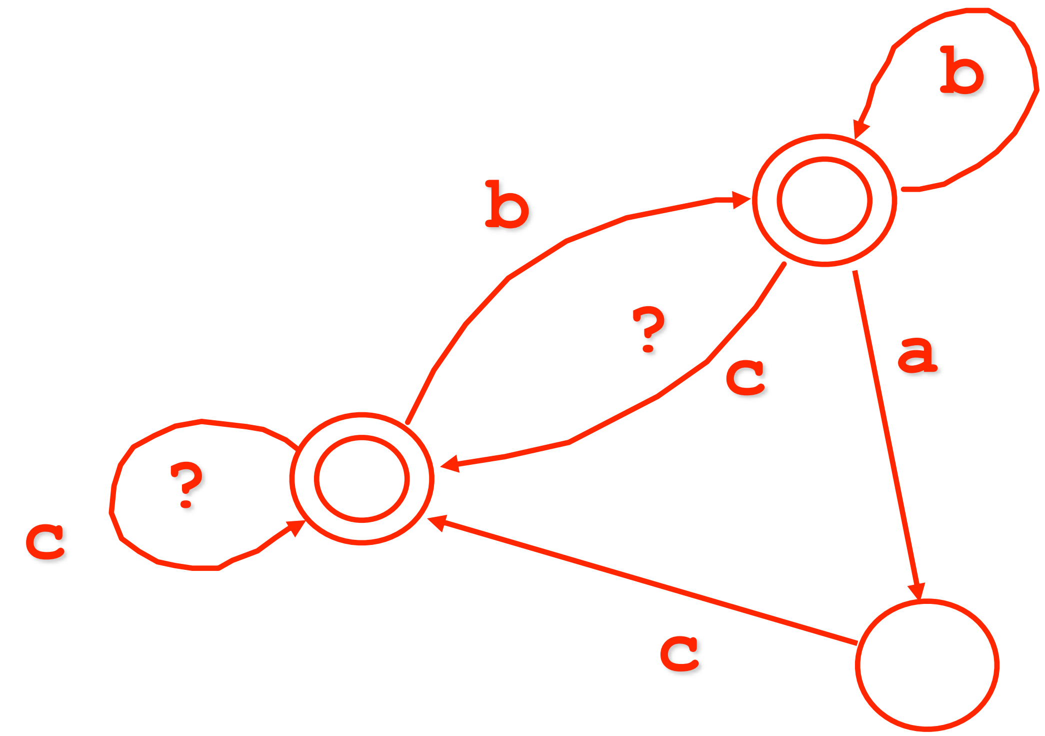
Accepts **`bacbbacde`**
Rejects **`bac`<u>`a`</u>**

Restriction

a => b _ c

"Any **a** must be preceded by **b** and followed by **c**."

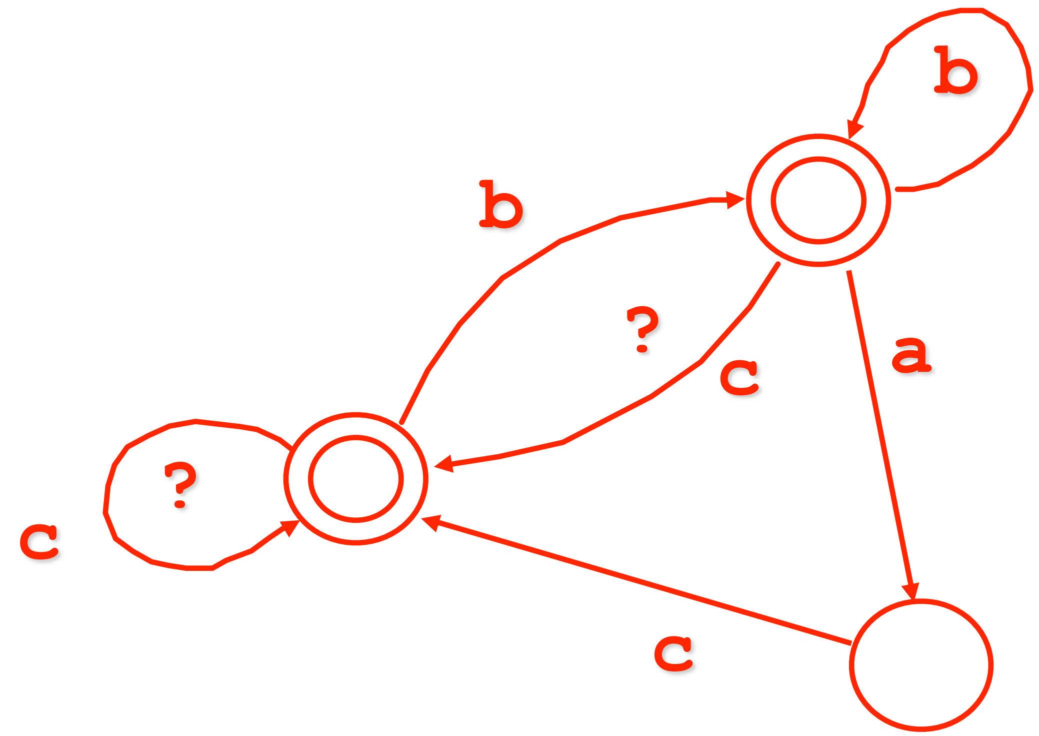
Accepts **bacbbacde**
Rejects **bac<u>a</u>**



slide courtesy of L. Karttunen (modified)

Restriction

`a => b _ c`



"Any `a` must be preceded by `b` and followed by `c`."

Accepts **`bacbbacde`**
Rejects **`baca`**

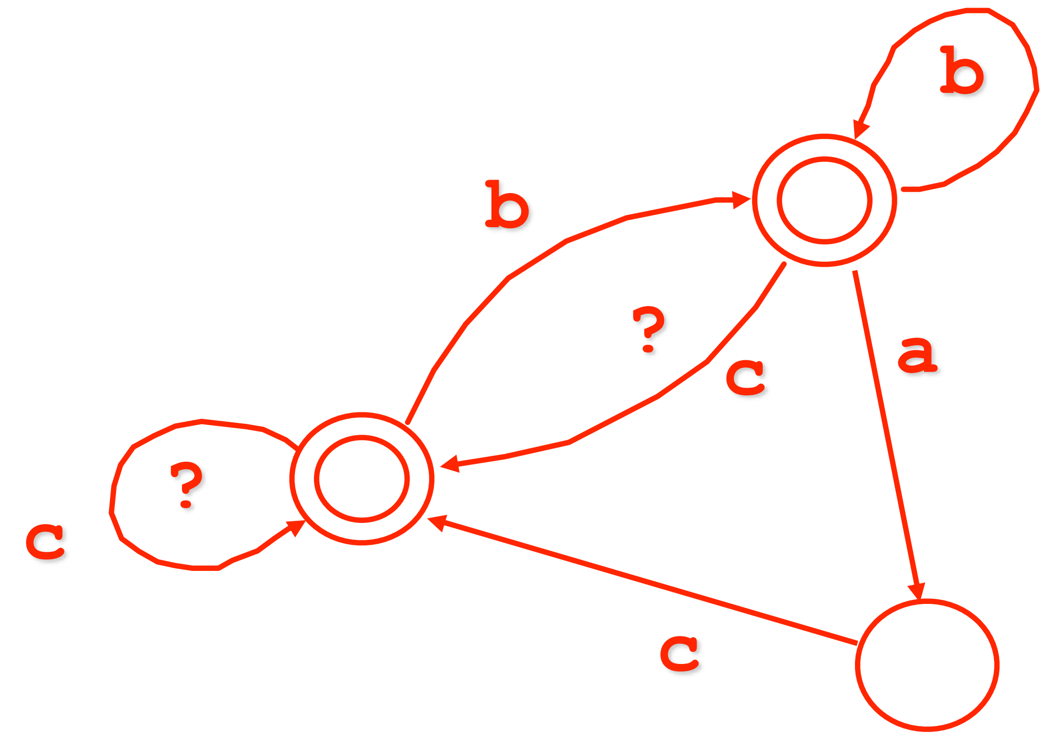
`~[~[?* b] a ?*] & ~[?* a ~[c ?*]]`

Equivalent expression

Restriction

`a => b _ c`

"Any `a` must be preceded by `b` and followed by `c`."



Accepts **`bacbbacde`**

Rejects **`baca`**

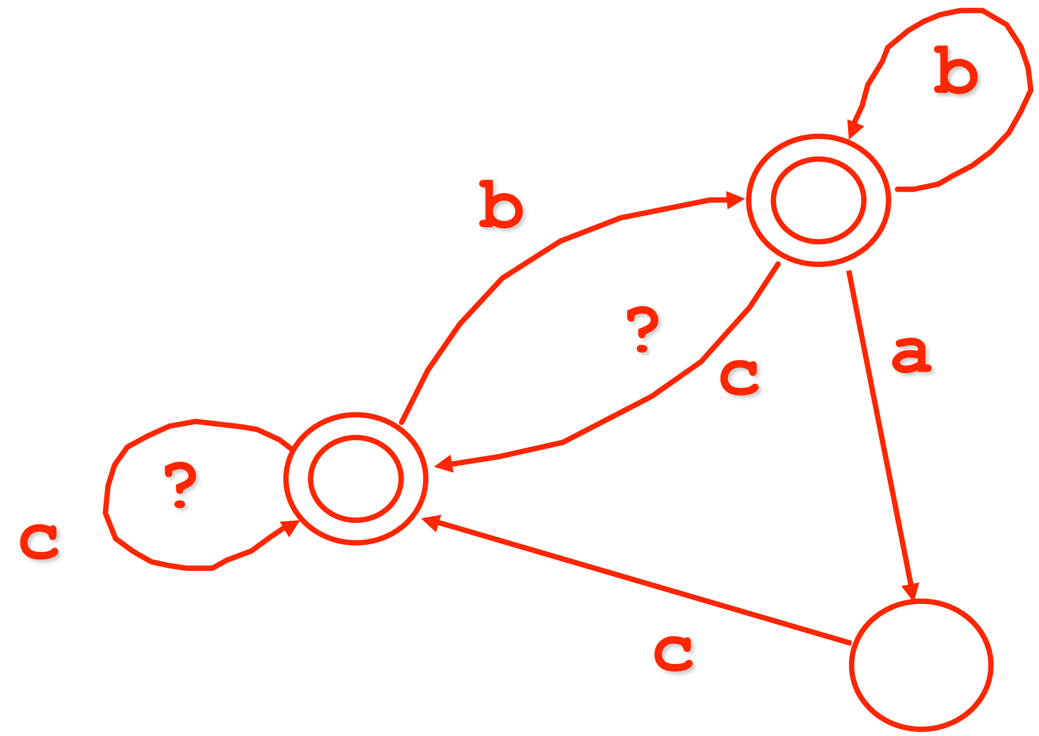
contains `a` not preceded by `b`

`~[~[?* b] a ?*] & ~[?* a ~[c ?*]]`

Equivalent expression

Restriction

`a => b _ c`



"Any **a** must be preceded by **b** and followed by **c**."

Accepts **bacbbacde**
Rejects **bac<u>a</u>**

contains **a** not preceded by **b**

contains **a** not followed by **c**

`~[ ~[?* b] a ?* ] & ~[ ?* a ~[c ?*] ]`

Equivalent expression

Replacement

Replacement

`a b -> b a`

"Replace 'ab' by 'ba'."

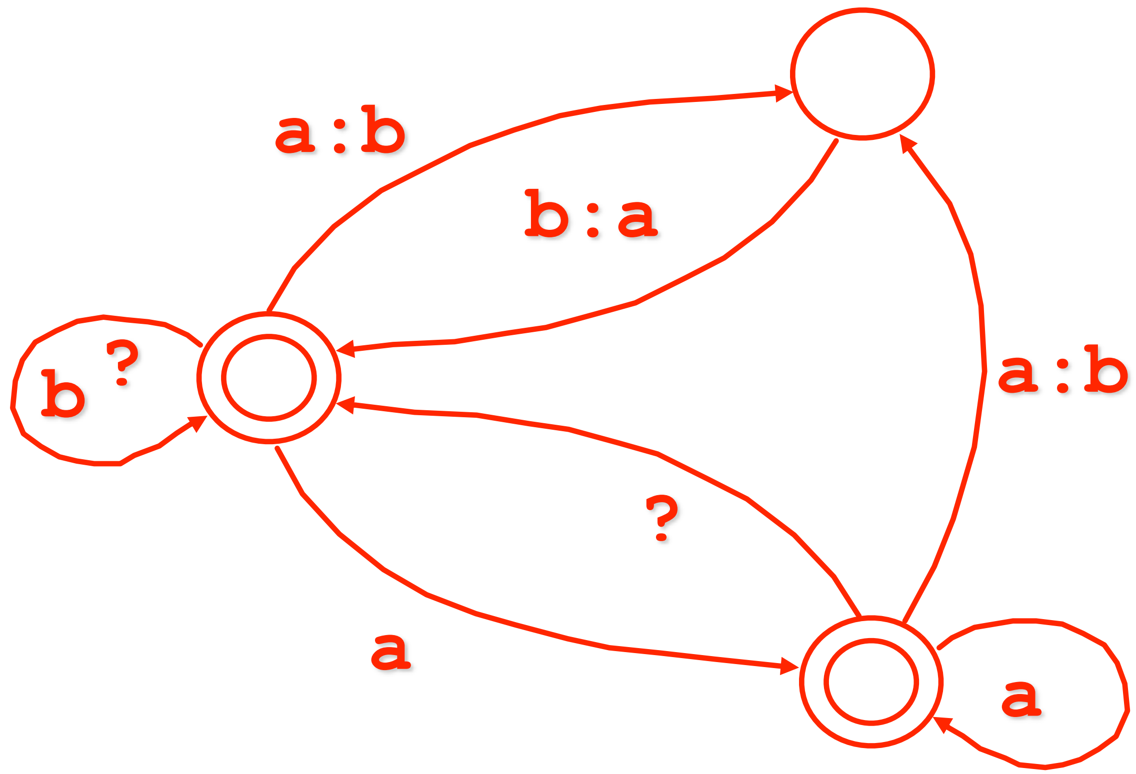
Transduces <u>ab</u>cdb<u>ab</u>a
to <u>ba</u>cdb<u>ba</u>a

Replacement

`a b -> b a`

"Replace 'ab' by 'ba'."

Transduces abcdbaba
to bacdbbaa



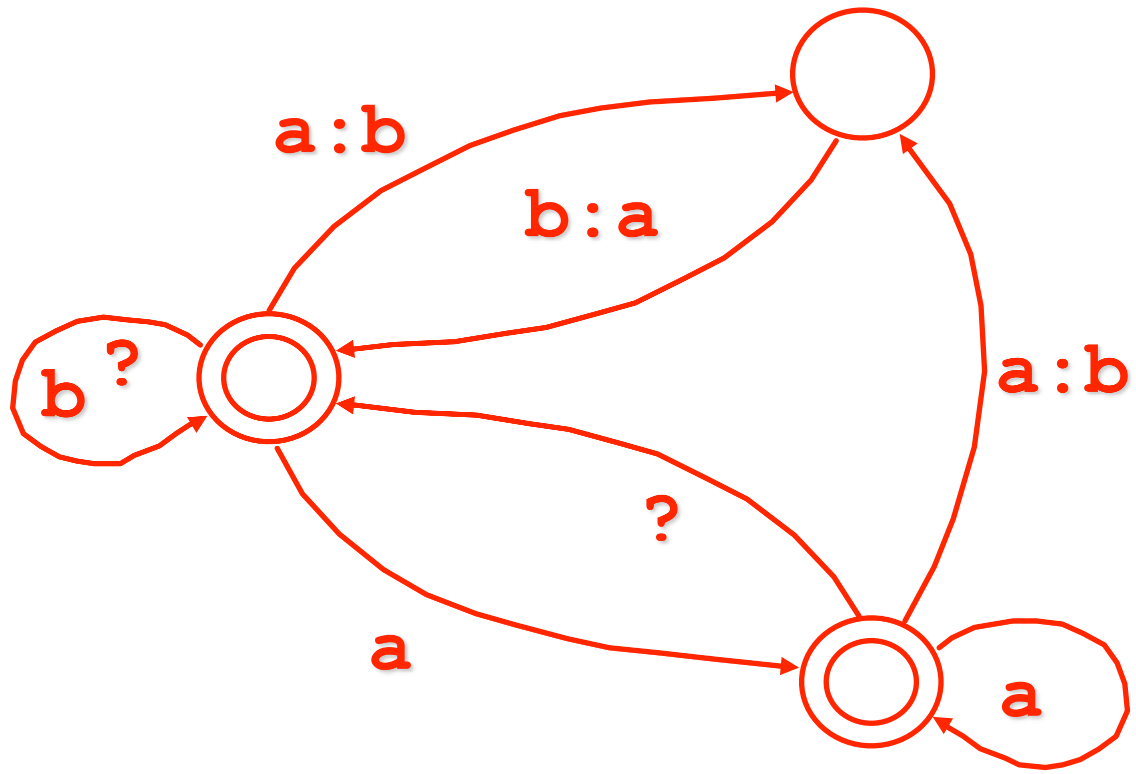
slide courtesy of L. Karttunen (modified)

Replacement

`a b -> b a`

"Replace 'ab' by 'ba'."

Transduces abcdbaba
to bacdbbaa



`[~$[a b] [[a b] .x. [b a]]]* ~$[a b]`

Equivalent expression

Replacement is Nondeterministic

Replacement is Nondeterministic

```
a b -> b a | x
```

"Replace 'ab' by 'ba' or 'x', nondeterministically."

Transduces **<u>ab</u>cdb<u>ab</u>a**

to {**<u>ba</u>cdb<u>ba</u>a**, **<u>ba</u>cdb<u>x</u>a**, **<u>x</u>cdb<u>ba</u>a**, **<u>x</u>cdb<u>x</u>a**}

Replacement is Nondeterministic

`[ a b -> b a | x ] .o. [ x => _ c ]`

"Replace 'ab' by 'ba' or 'x', nondeterministically."

Transduces **abcdbaba**

to {**bacdbbaa**, **bacdbxa**, **xcdbbaa**, **xcdbxa**}

Replacement is Nondeterministic

```
[ a b -> b a | x ] .o. [ x => _ c ]
```

“Replace ‘ab’ by ‘ba’ or ‘x’, nondeterministically.”

Transduces **abcdbaba**
to {**bacdbbaa**, ~~**bacdbxa**~~, **xcdbbaa**, ~~**xcdbxa**~~}

slide courtesy of L. Karttunen (modified)

Replacement is Nondeterministic

```
a b | b | b a | a b a -> x
```

applied to “aba”

Four overlapping substrings match; we haven’t told it which one to replace so it chooses nondeterministically

```
a b a              a b a      a b a
  a b a
a x a              a  x        x  a
  x
```

More Replace Operators

- Optional replacement: `a b (->) b a`
- Directed replacement
  - guarantees a unique result by constraining the factorization of the input string by
    - Direction of the match (rightward or leftward)
    - Length (longest or shortest)

`@->` Left-to-right, Longest-match Replacement

`a b | b | b a | a b a @-> x`

applied to “aba”

~~a b a / a x a~~ **a b a / x**

~~a b a / a x~~

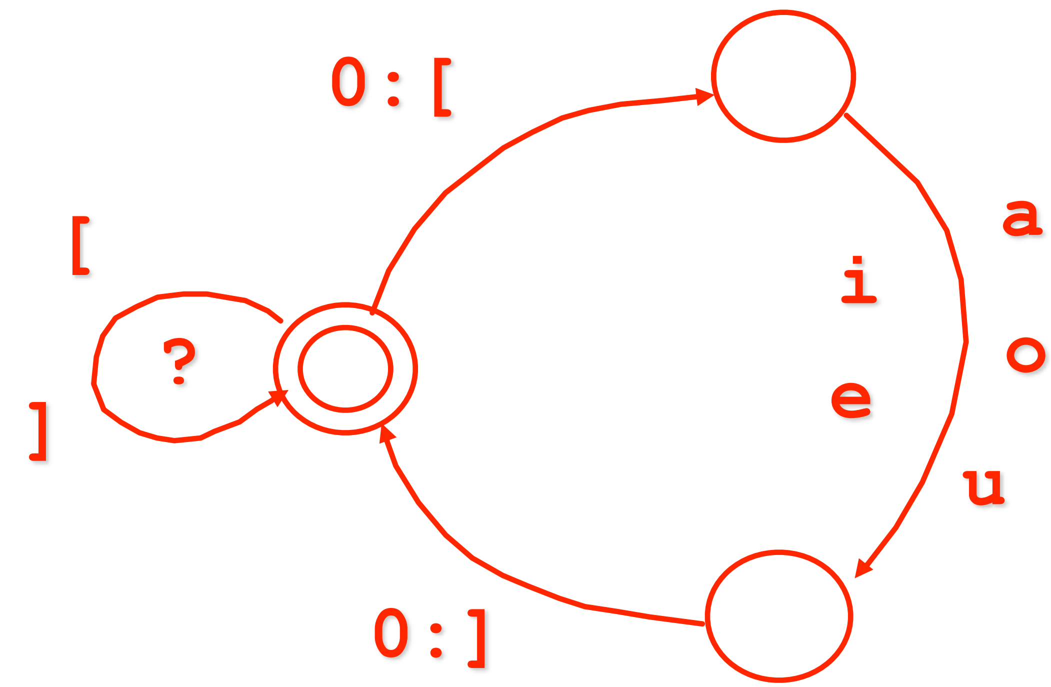
~~a b a / x a~~

a b a / ______

- `@->` **left-to-right, longest match (cf. perl s///)**
- `@>` **left-to-right, shortest match**
- `->@` **right-to-left, longest match**
- `>@` **right-to-left, shortest match**

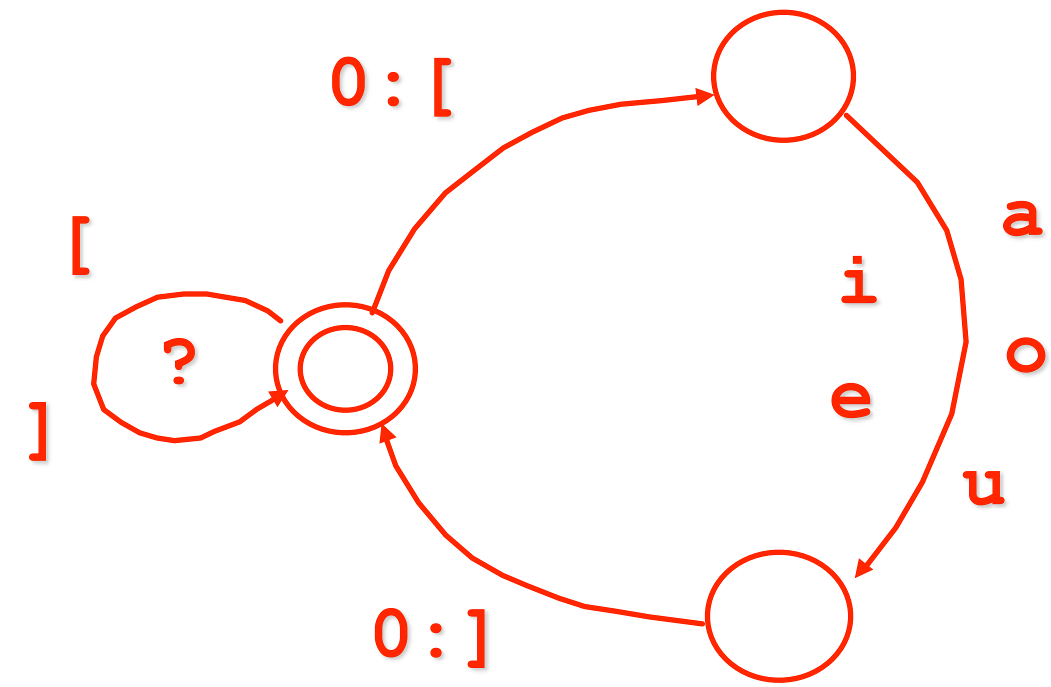
Using "…" for marking

`a|e|i|o|u -> [ ... ]`



Using "…" for marking

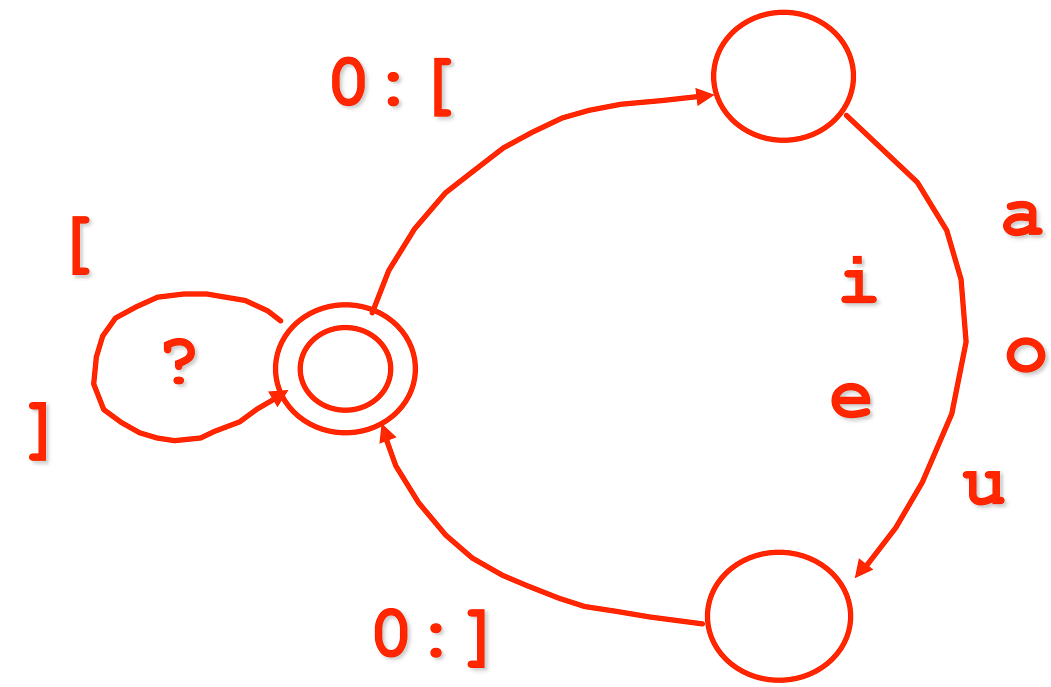
a|e|i|o|u -> [...]



p o t a t o
p[o]t[a]t[o]

Using "…" for marking

`a|e|i|o|u -> [ ... ]`



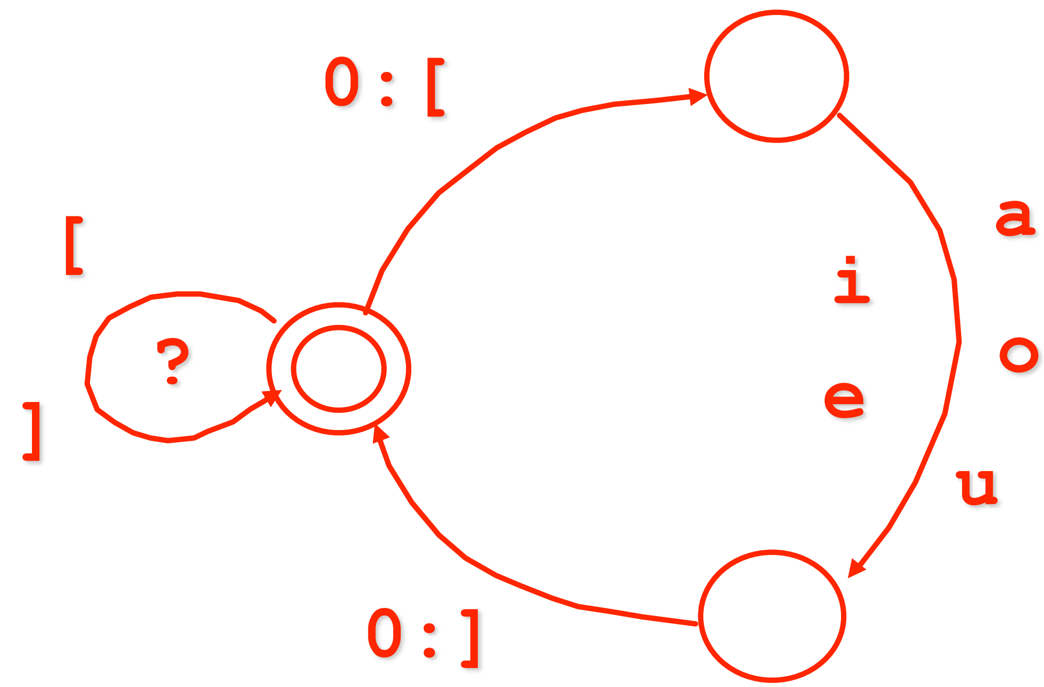

```
p o t a t o
p[o]t[a]t[o]
```

Note: actually have to write as `-> %[ ... %]`
or `-> "[" ... "]"`
since [] are parens in the regexp language

slide courtesy of L. Karttunen (modified)

Using "…" for marking

```
a|e|i|o|u -> [ ... ]
```




```
p o t a t o
p[o]t[a]t[o]
```

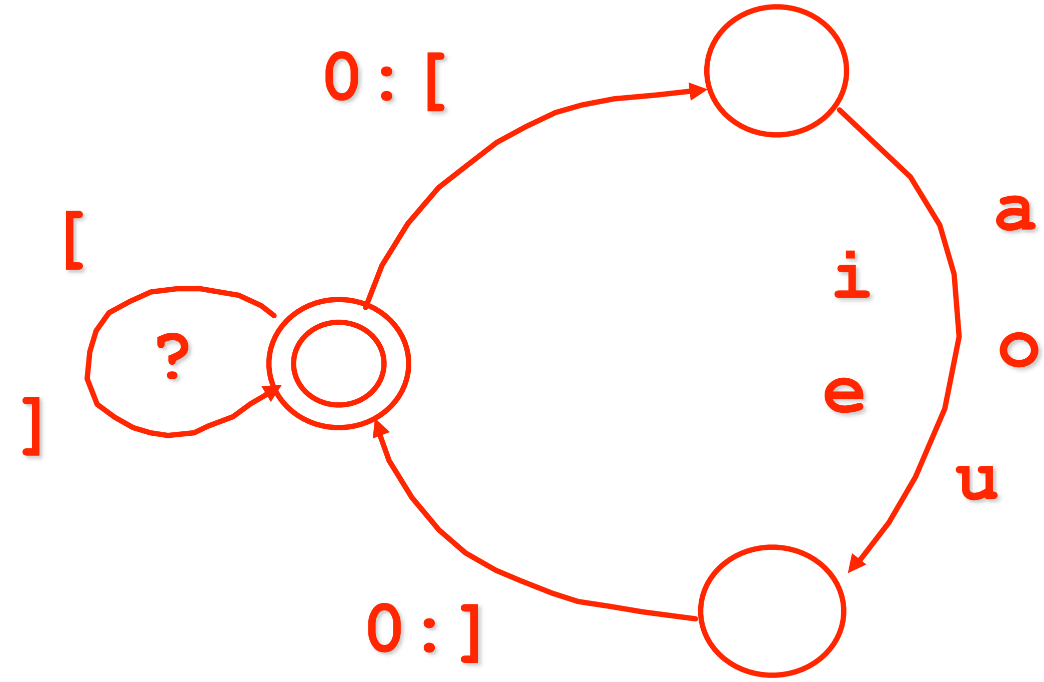
Which way does the FST transduce potatoe?

```
p o t a t o  e           p o t a t o e
p[o]t[a]t[o][e]   vs.    p[o]t[a]t[o e]
```

slide courtesy of L. Karttunen (modified)

Using "…" for marking

a|e|i|o|u -> [...]



p o t a t o
p[o]t[a]t[o]

Which way does the FST transduce potatoe?

p o t a t o e
p[o]t[a]t[o][e]

vs.

p o t a t o e
p[o]t[a]t[o e]

How would you change it to get the other answer?

slide courtesy of L. Karttunen

Example: Finnish Syllabification

Example: Finnish Syllabification

```
define C [ b | c | d | f ...
define V [ a | e | i | o | u | y | ä | ...
```

slide courtesy of L. Karttunen

Example: Finnish Syllabification

```
define C [ b | c | d | f ...
define V [ a | e | i | o | u | y | ä | ...
```

```
[C* V+ C*] @-> ... "-" || _ [C V]
```

“Insert a hyphen after the longest instance of the `C* V+ C*` pattern in front of a `C V` pattern.”

slide courtesy of L. Karttunen

Example: Finnish Syllabification

```
define C [ b | c | d | f ...
define V [ a | e | i | o | u | y | ä | ...
```

```
[C* V+ C*] @-> ... "-" || _ [C V]
```

"Insert a hyphen after the longest instance of the `C* V+ C*` pattern in front of a `C V` pattern."

```
s t r u k   t u   r a   l i s   m i
s t r u k - t u - r a - l i s - m i
```

slide courtesy of L. Karttunen

Example: Finnish Syllabification

```
define C [ b | c | d | f ...
define V [ a | e | i | o | u | y | ä | ...
```

```
[C* V+ C*] @-> ... "-" || _ [C V]
```

"Insert a hyphen after the longest instance of the
C* V+ C* pattern ~~in front of a C V pattern.~~" why?

```
s t r u k   t u   r a   l i s   m i
s t r u k - t u - r a - l i s - m i
```

slide courtesy of L. Karttunen

Conditional Replacement

slide courtesy of L. Karttunen

Conditional Replacement

`A -> B` `L _ R`

Replacement Context

The relation that replaces **`A`** by **`B`** between **`L`** and **`R`** leaving everything else unchanged.

Conditional Replacement

`A -> B` `L _ R`

Replacement Context

The relation that replaces **`A`** by **`B`** between **`L`** and **`R`** leaving everything else unchanged.

Sources of complexity:

- Replacements and contexts may overlap
- Alternative ways of interpreting "between left and right."

Hand-Coded Example: Parsing Dates

slide courtesy of L. Karttunen

Today is [Tuesday, July 25, 2000].

Hand-Coded Example: Parsing Dates

slide courtesy of L. Karttunen

Today is [Tuesday, July 25, 2000].

Best result

Today is Tuesday, [July 25, 2000].

Today is [Tuesday, July 25], 2000.

Today is Tuesday, [July 25], 2000.

Today is [Tuesday], July 25, 2000.

Bad results

Hand-Coded Example: Parsing Dates

slide courtesy of L. Karttunen

Today is [Tuesday, July 25, 2000].

Best result

Today is Tuesday, [July 25, 2000].

Today is [Tuesday, July 25], 2000.

Today is Tuesday, [July 25], 2000.

Today is [Tuesday], July 25, 2000.

Bad results

Need left-to-right, longest-match constraints.

Source code: Language of Dates

slide courtesy of L. Karttunen

Source code: Language of Dates

```
Day = Monday | Tuesday |  ... | Sunday
Month = January | February | ... | December
Date = 1 | 2 | 3 | ... | 3 1
Year = %0To9 (%0To9 (%0To9 (%0To9))) - %0?*
                              from 1 to 9999
```

slide courtesy of L. Karttunen

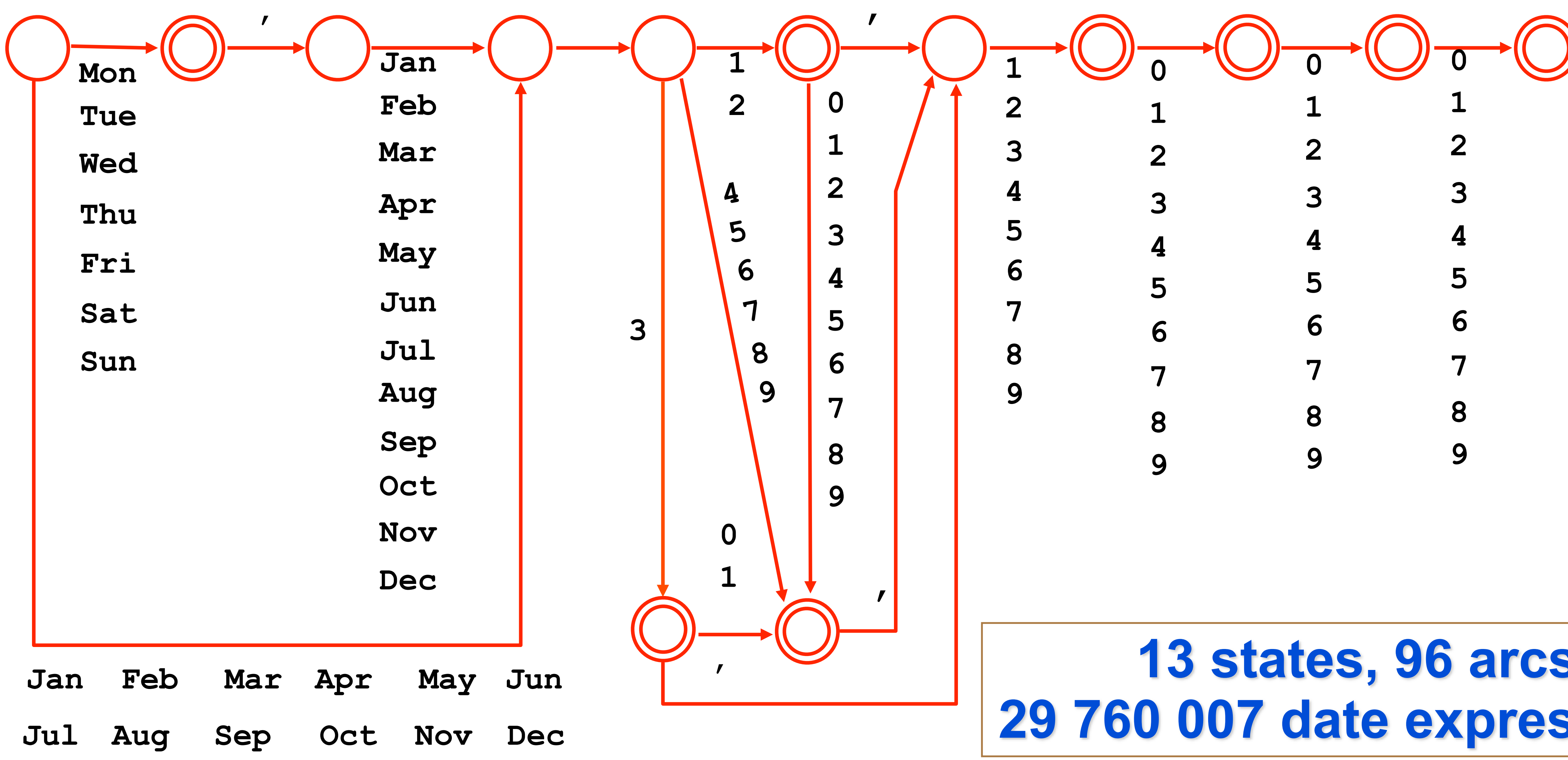
Source code: Language of Dates

```
Day = Monday | Tuesday |  ... | Sunday
Month = January | February | ... | December
Date = 1 | 2 | 3 | ... | 3 1
Year = %0To9 (%0To9 (%0To9 (%0To9))) - %0?*
                                 from 1 to 9999

AllDates =  Day | (Day ", ") Month " " Date (",
  " Year))
```

Object code:
All Dates from 1/1/1 to 12/31/9999

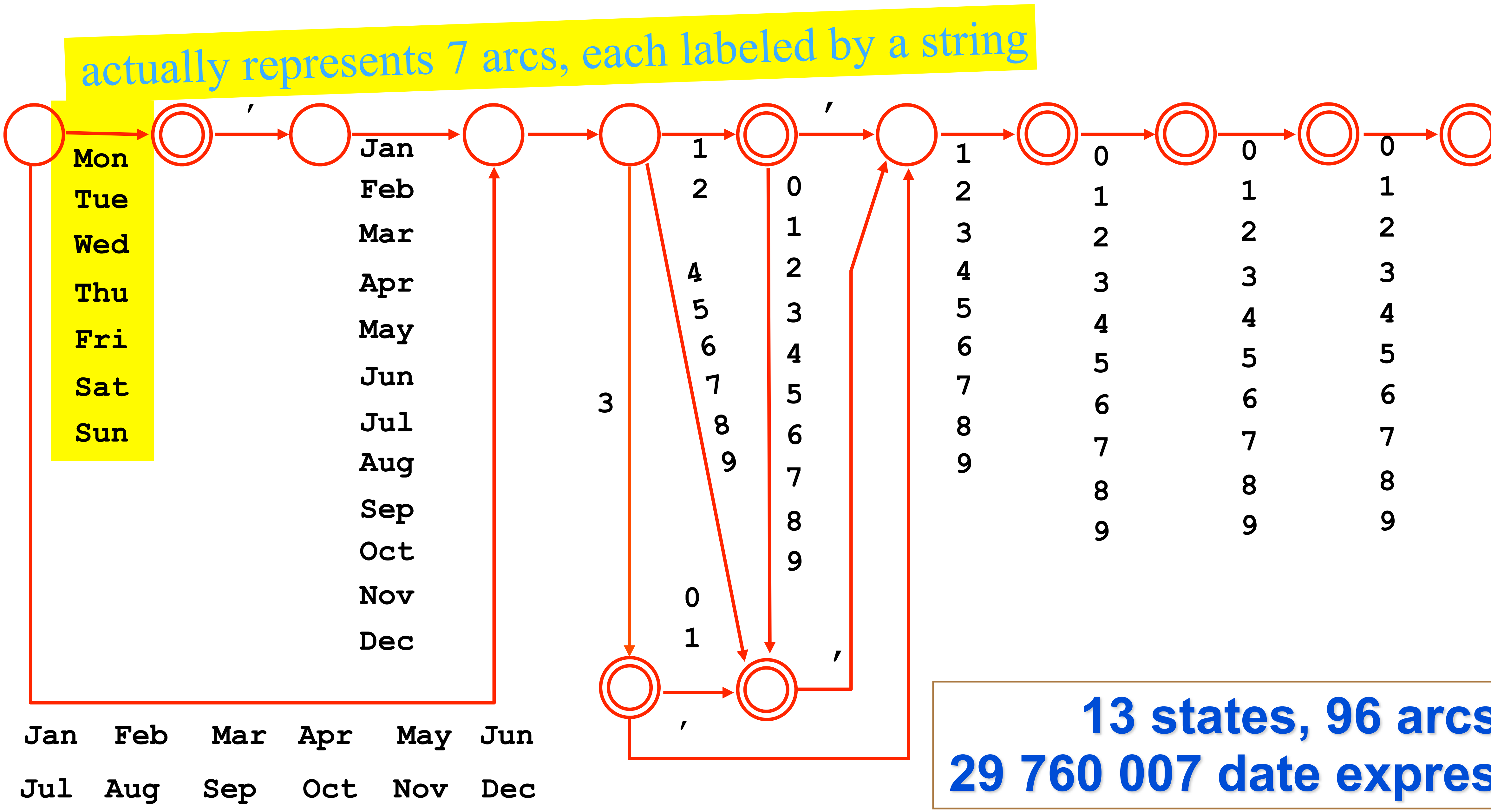
slide courtesy of L. Karttunen



13 states, 96 arcs
29 760 007 date expressions

slide courtesy of L. Karttunen

Object code: All Dates from 1/1/1 to 12/31/9999



13 states, 96 arcs
29 760 007 date expressions

slide courtesy of L. Karttunen (modified)

Parser for Dates

slide courtesy of L. Karttunen (modified)

Parser for Dates

```
AllDates @-> "[DT " ... "]"
```

Parser for Dates

`AllDates @-> "[DT " ... "]"`

Compiles into an unambiguous transducer (23 states, 332 arcs).

slide courtesy of L. Karttunen (modified)

Parser for Dates

```
AllDates @-> "[DT " ... "]"
```

Compiles into an unambiguous transducer (23 states, 332 arcs).

Today is [DT Tuesday, July 25, 2000] because yesterday was [DT Monday] and it was [DT July 24] so tomorrow must be [DT Wednesday, July 26] and not [DT July 27] as it says on the program.

slide courtesy of L. Karttunen (modified)

Parser for Dates

```
AllDates @-> "[DT " ... "]"
```

Compiles into an unambiguous transducer (23 states, 332 arcs).

Xerox left-to-right replacement operator

Today is [DT Tuesday, July 25, 2000] because yesterday was [DT Monday] and it was [DT July 24] so tomorrow must be [DT Wednesday, July 26] and not [DT July 27] as it says on the program.

slide courtesy of L. Karttunen

Problem of Reference

Valid dates

`Tuesday, July 25, 2000`

`Tuesday, February 29, 2000`

`Monday, September 16, 1996`

Invalid dates

`Wednesday, April 31, 1996`

`Thursday, February 29, 1900`

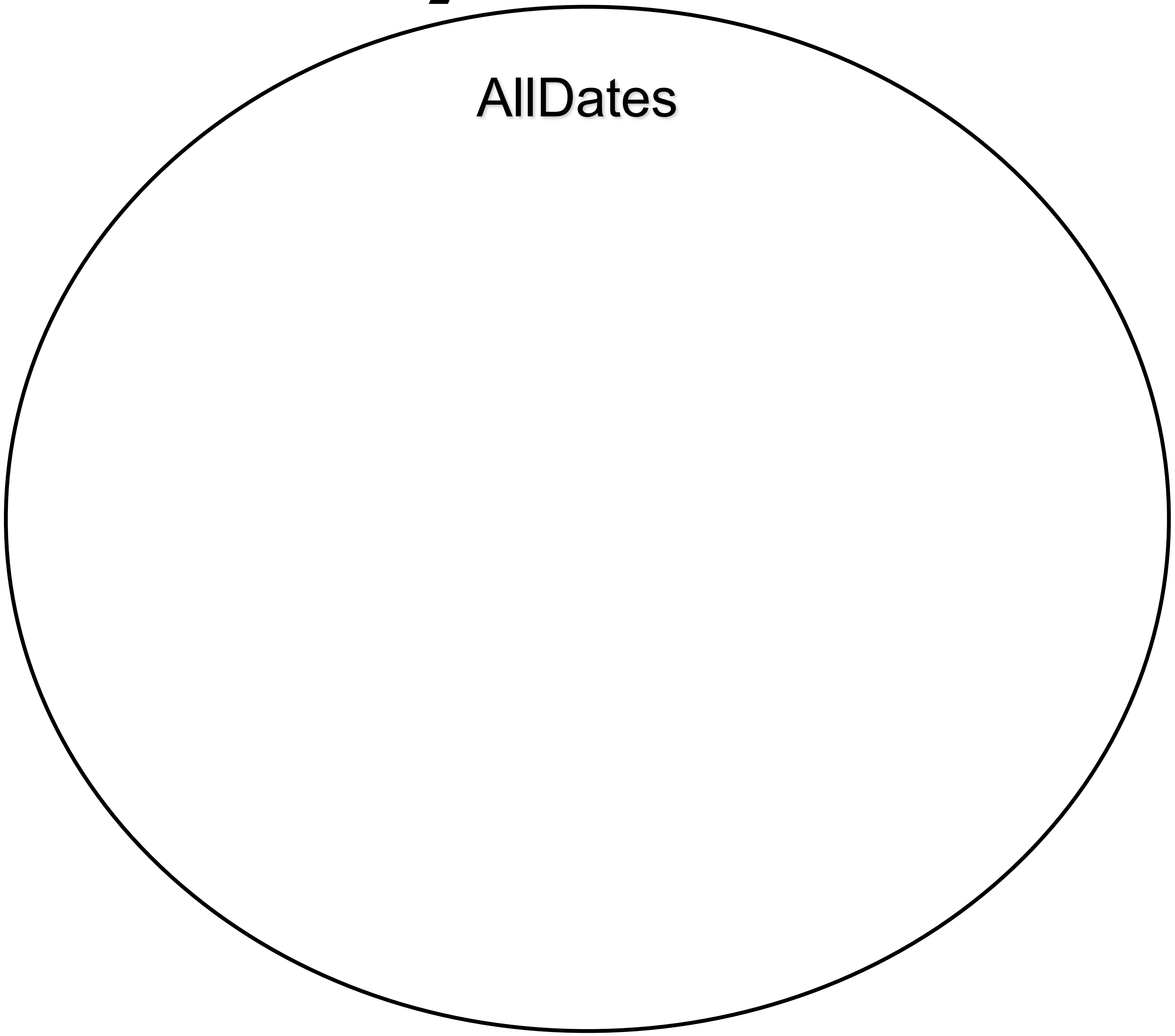
`Tuesday, July 26, 2000`

slide courtesy of L. Karttunen (modified)

Refinement by Intersection

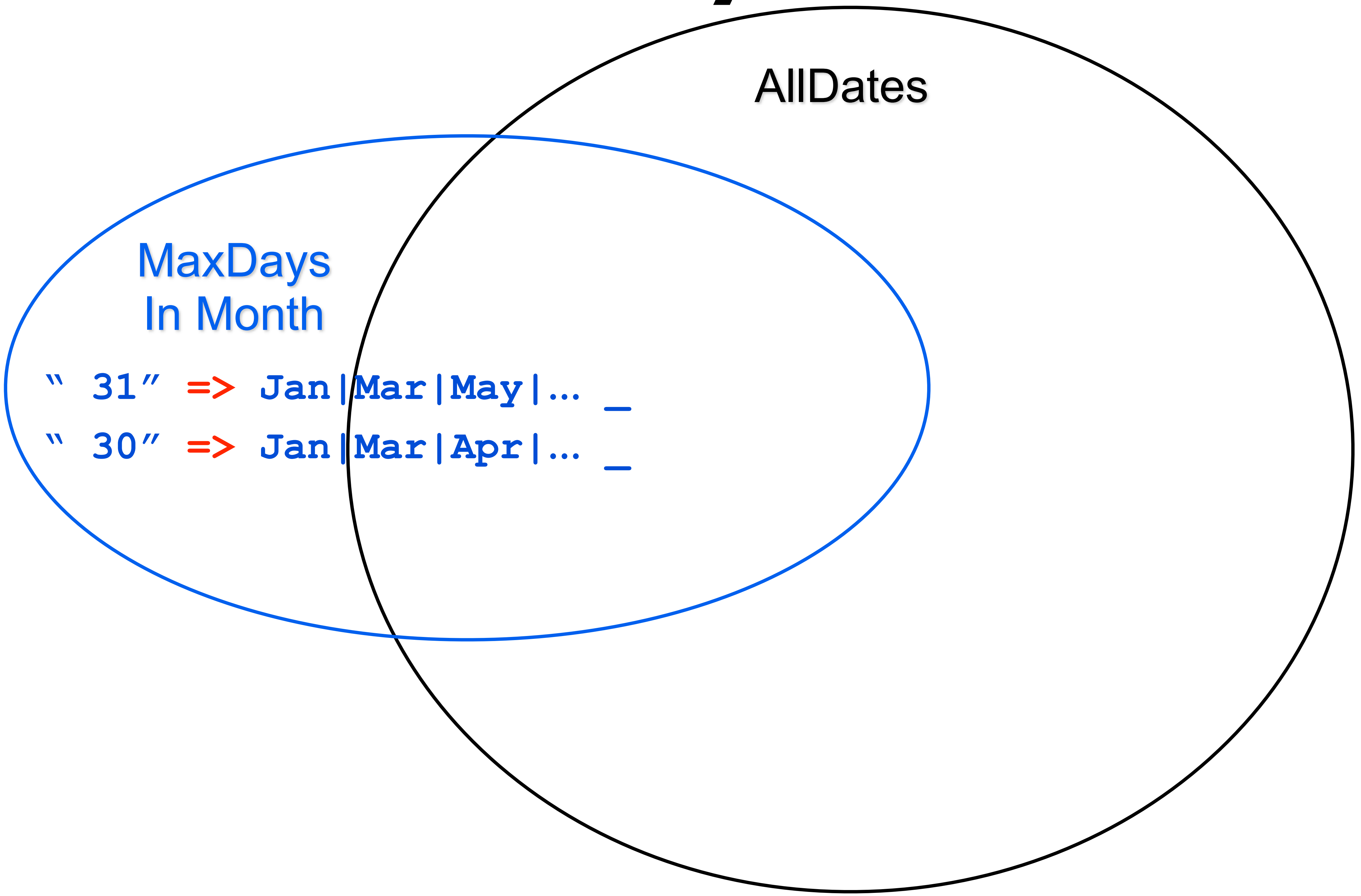
slide courtesy of L. Karttunen (modified)

Refinement by Intersection



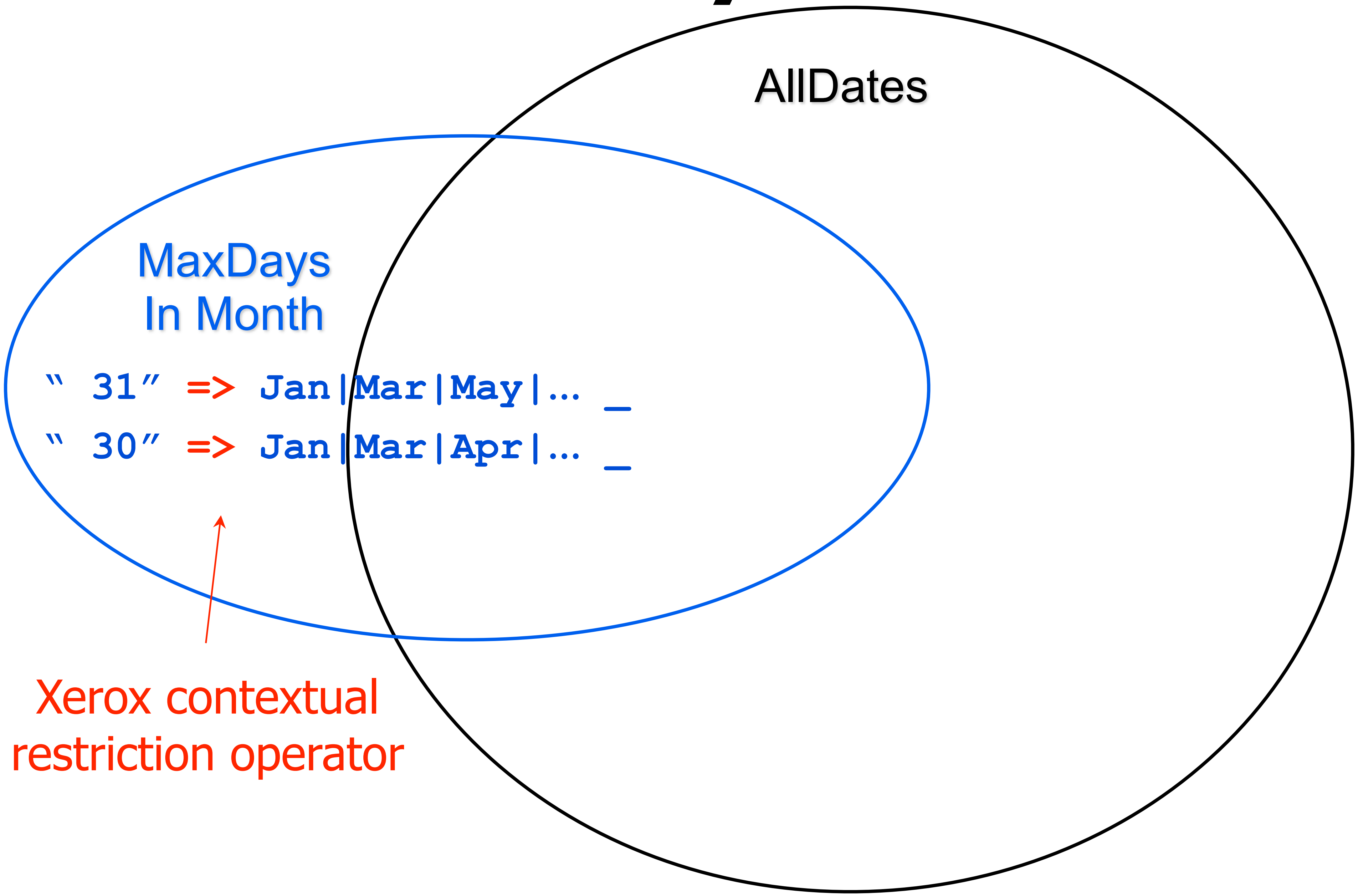
slide courtesy of L. Karttunen (modified)

Refinement by Intersection



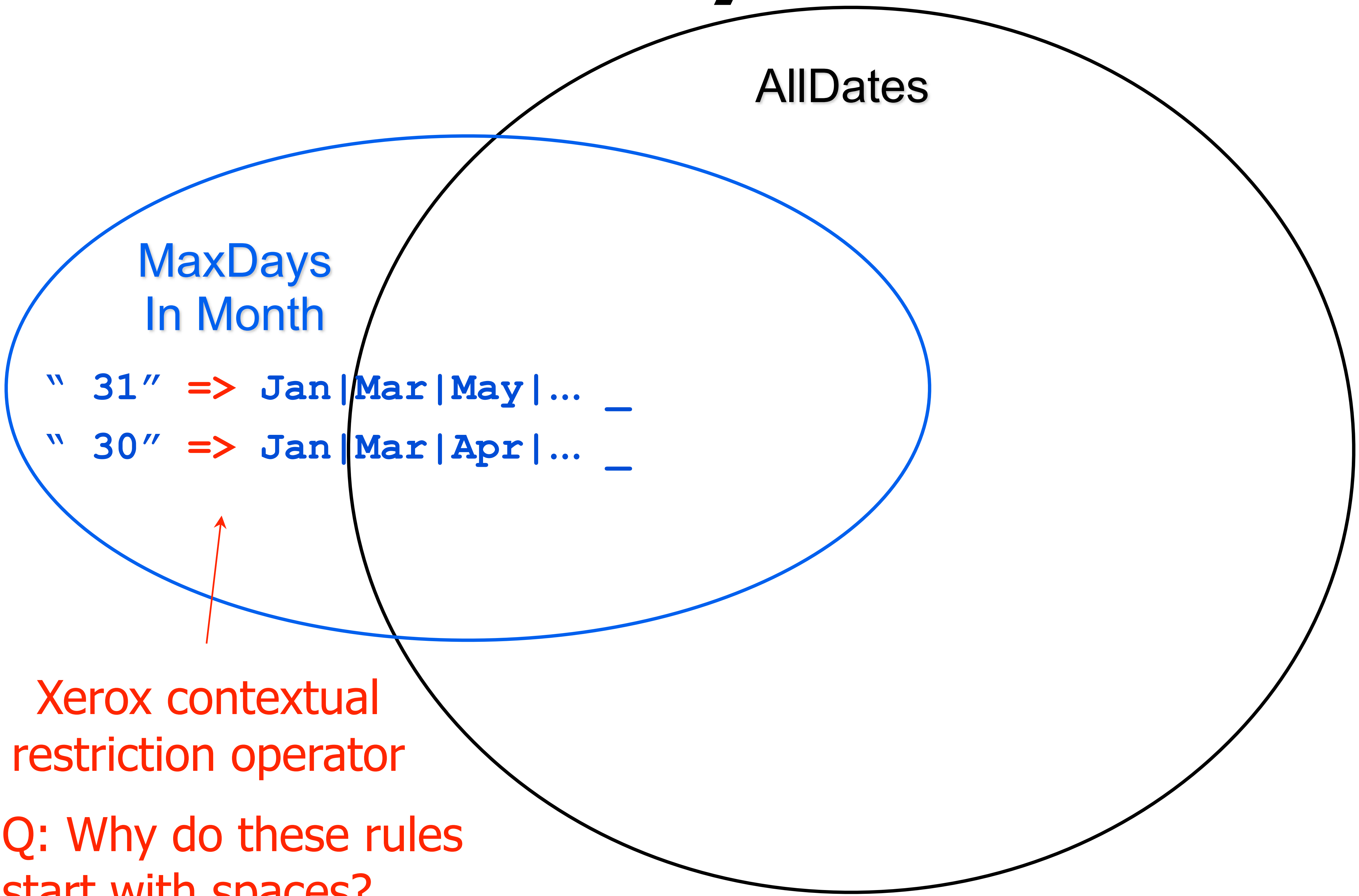
slide courtesy of L. Karttunen (modified)

Refinement by Intersection



slide courtesy of L. Karttunen (modified)

Refinement by Intersection

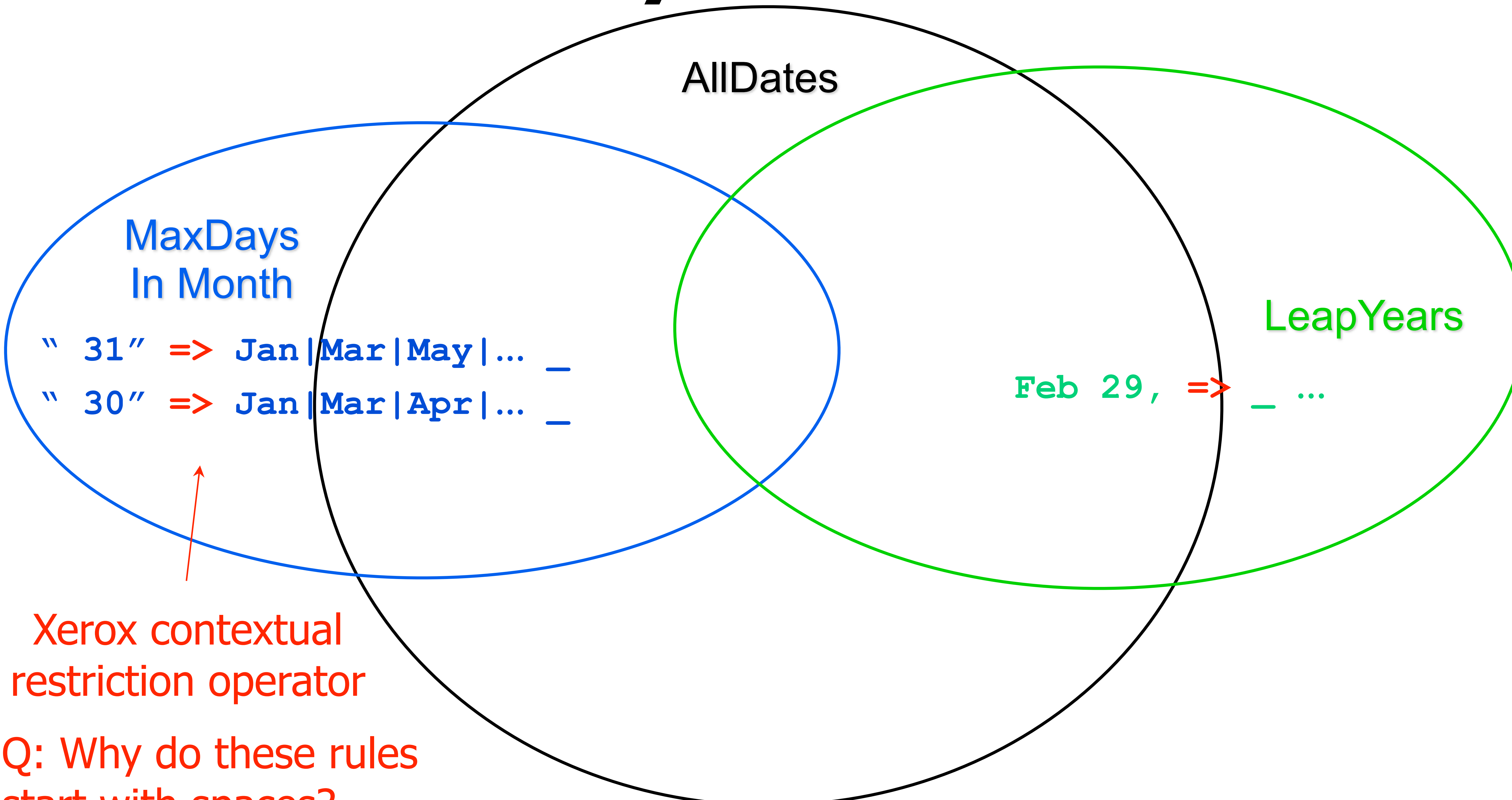


Xerox contextual restriction operator

Q: Why do these rules start with spaces? (And is it enough?)

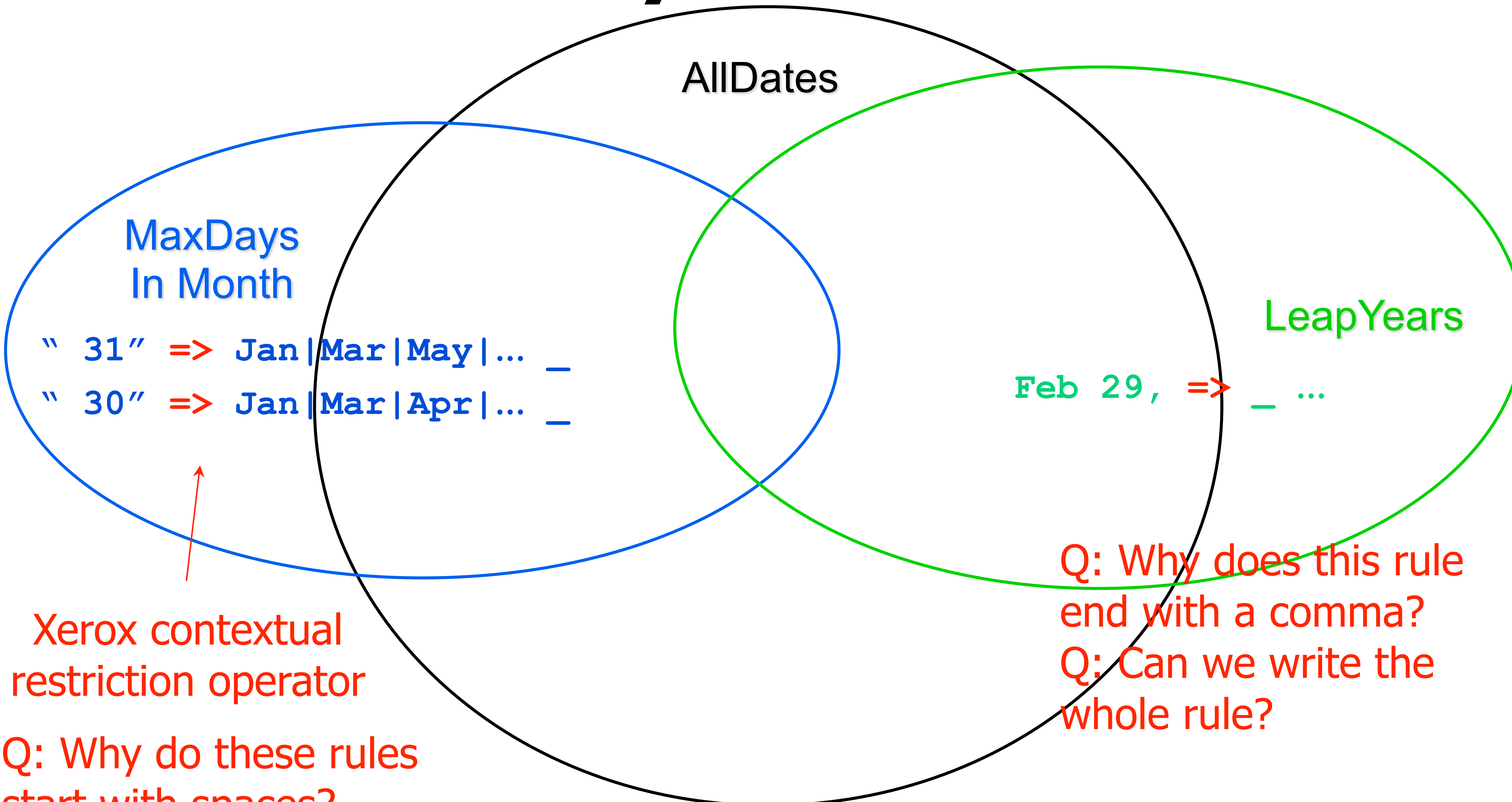
slide courtesy of L. Karttunen (modified)

Refinement by Intersection



Q: Why do these rules start with spaces? (And is it enough?)

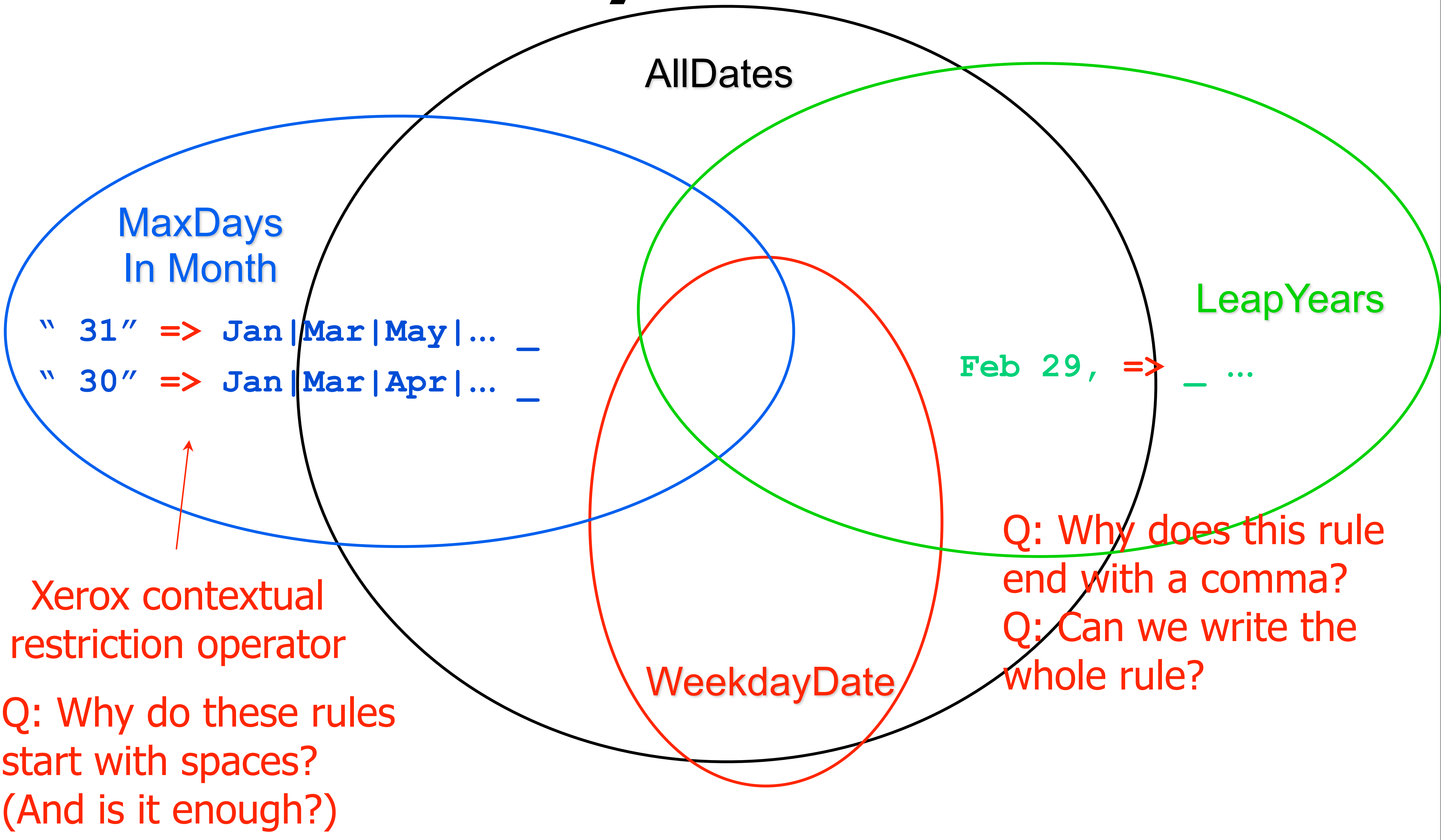
Refinement by Intersection



Q: Why do these rules start with spaces? (And is it enough?)

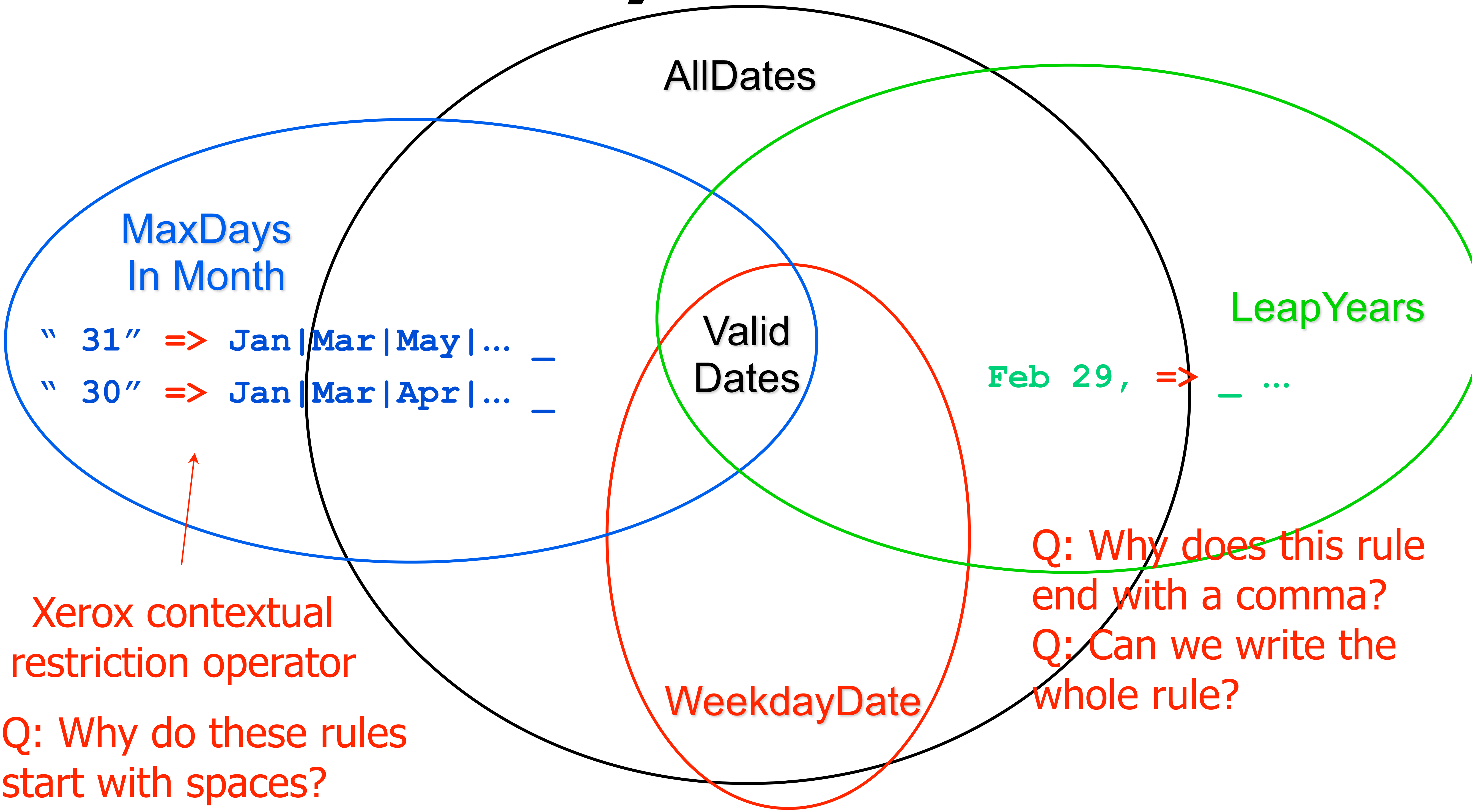
slide courtesy of L. Karttunen (modified)

Refinement by Intersection



slide courtesy of L. Karttunen (modified)

Refinement by Intersection



Xerox contextual restriction operator

Q: Why do these rules start with spaces? (And is it enough?)

Q: Why does this rule end with a comma?

Q: Can we write the whole rule?

slide courtesy of L. Karttunen

Defining Valid Dates

```
AllDates
&
MaxDaysInMonth
&
LeapYears
&
WeekdayDates
```

```
= ValidDates
```

AllDates: 13 states, 96 arcs
29 760 007 date expressions

ValidDates: 805 states, 6472 arcs
7 307 053 date expressions

slide courtesy of L. Karttunen

Parser for Valid and Invalid Dates

```
[AllDates - ValidDates] @-> "[ID " ... "]"
                        ,
      ValidDates @-> "[VD " ... "]"
```

2688 states,
20439 arcs

Today is [VD Tuesday, July 25, 2000],
not [ID Tuesday, July 26, 2000].

valid date

invalid date

Parser for Valid and Invalid Dates

```
[AllDates - ValidDates] @-> "[ID " ... "]"
                                ,
      ValidDates @-> "[VD " ... "]"
```

2688 states,
20439 arcs

Comma creates a single FST that does left-to-right longest match against either pattern

Today is [VD Tuesday, July 25, 2000],
not [ID Tuesday, July 26, 2000].

valid date

invalid date

More Engineering Applications

More Engineering Applications

- Markup

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation
  - Spelling correction / edit distance

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
  - Transliteration / back-transliteration

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
  - Transliteration / back-transliteration
  - Machine translation?

More Engineering Applications

- Markup
  - Dates, names, places, noun phrases; spelling/grammar errors?
  - Hyphenation
  - Informative templates for information extraction (FASTUS)
  - Word segmentation (use probabilities!)
  - Part-of-speech tagging (use probabilities – maybe!)
- Translation
  - Spelling correction / edit distance
  - Phonology, morphology: series of little fixups? constraints?
  - Speech
  - Transliteration / back-transliteration
  - Machine translation?
- Learning …

FASTUS – Information Extraction

Appelt et al, 1992-?

Input: Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan. The joint venture, Bridgestone Sports Taiwan Co., capitalized at 20 million new Taiwan dollars, will start production in January 1990 with …

Output:

Relationship:	TIE-UP
Entities:	"Bridgestone Sports Co."
	"A local concern"
	"A Japanese trading house"
Joint Venture Company:	"Bridgestone Sports Taiwan Co."
Amount:	NT$20000000

FASTUS: Successive Markups

(details on subsequent slides)

Tokenization

.o.

Multiwords

.o.

Basic phrases (noun groups, verb groups …)

.o.

Complex phrases

.o.

Semantic Patterns

.o.

Merging different references

FASTUS: Tokenization

- Spaces, hyphens, etc.
- wouldn't → would not
- their → them 's
- company. → company .
  but
  Co. → Co.

FASTUS: Multiwords

- "set up"
- "joint venture"
- "San Francisco Symphony Orchestra," "Canadian Opera Company"

- … use a specialized regexp to match musical groups.
- ... what kind of regexp would match company names?

FASTUS : Basic phrases

Output looks like this (no nested brackets!):

… [NG it] [VG had set_up] [NP a joint_venture] [Prep in] …

Company Name:	Bridgestone Sports Co.
Verb Group:	said
Noun Group:	Friday
Noun Group:	it
Verb Group:	had set up
Noun Group:	a joint venture
Preposition:	in
Location:	Taiwan
Preposition:	with
Noun Group:	a local concern

FASTUS: Noun Groups

Build FSA to recognize phrases like

- approximately 5 kg
- more than 30 people
- the newly elected president
- the largest leftist political force
- a government and commercial project

Use the FSA for left-to-right longest-match markup

What does FSA look like? See next slide …

FASTUS: Noun Groups

Described with a kind of non-recursive CFG …

(a regexp can include names that stand for other regexps)

NG → Pronoun | Time-NP | Date-NP

NG → (Det) (Adjs) HeadNouns

…

Adjs → sequence of adjectives maybe with commas, conjunctions, adverbs

…

Det → DetNP | DetNonNP

DetNP → detailed expression to match "the only five, another three, this, many, hers, all, the most …"

…

FASTUS: Semantic patterns

BusinessRelationship =
NounGroup(Company/ies) VerbGroup(Set-up)
NounGroup(JointVenture) with NounGroup(Company/ies)
| …

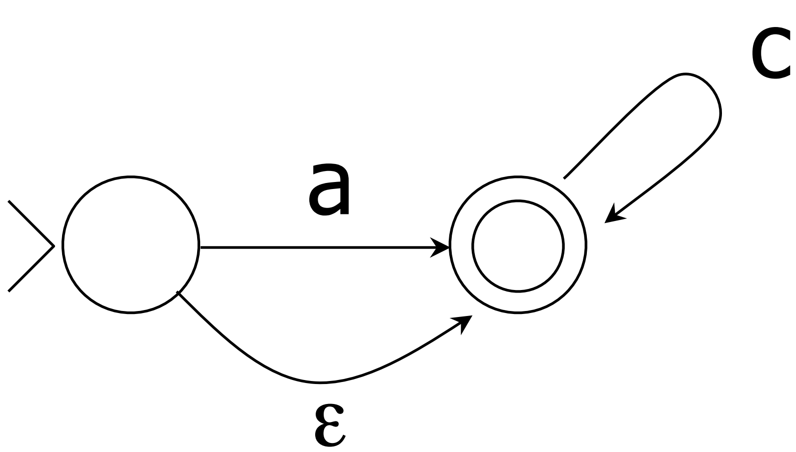
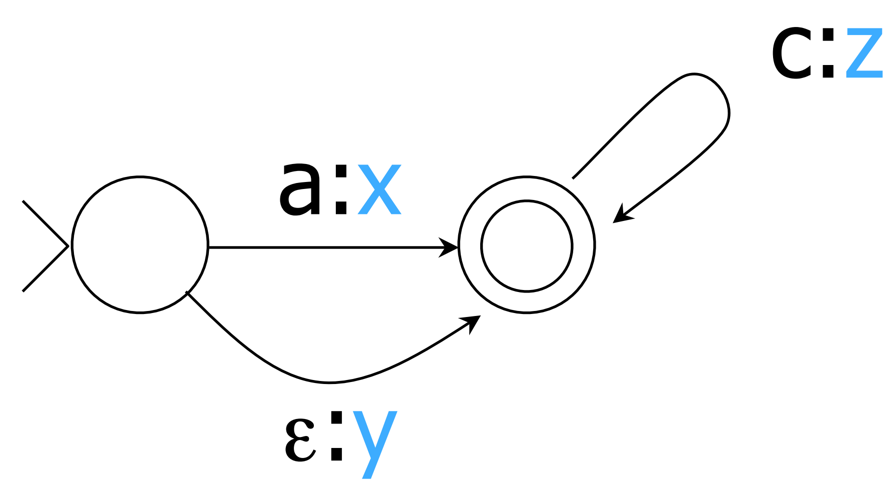
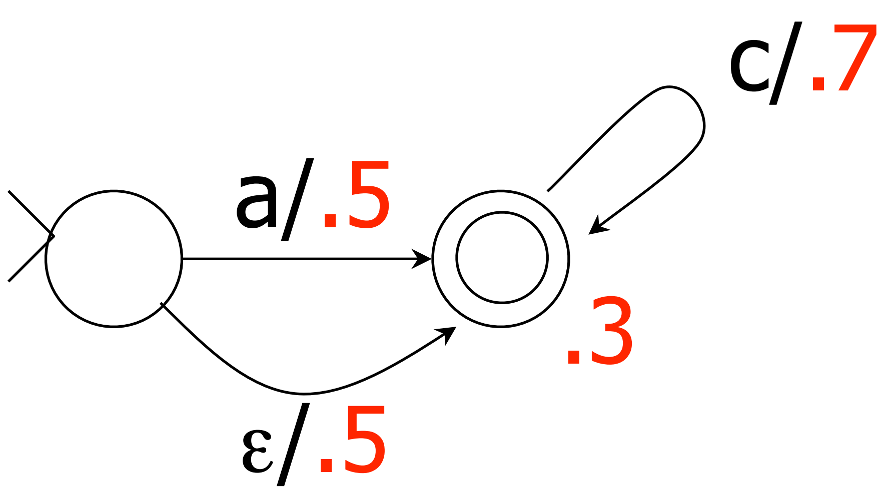
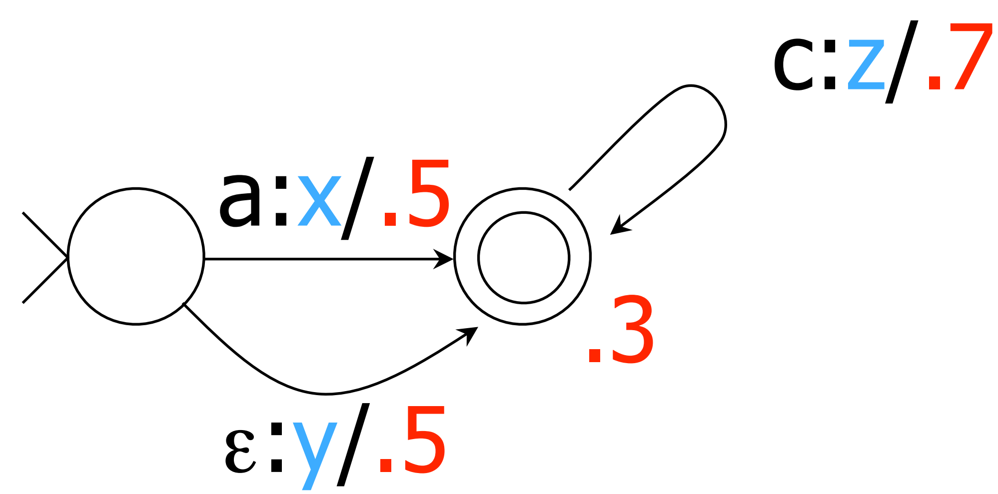
ProductionActivity =
VerbGroup(Produce) NounGroup(Product)

NounGroup(Company/ies) → NounGroup & …
is made easy by the processing done at a previous level

Use this for spotting references to put in the database.

Weighted FSMs

Function from strings to ...

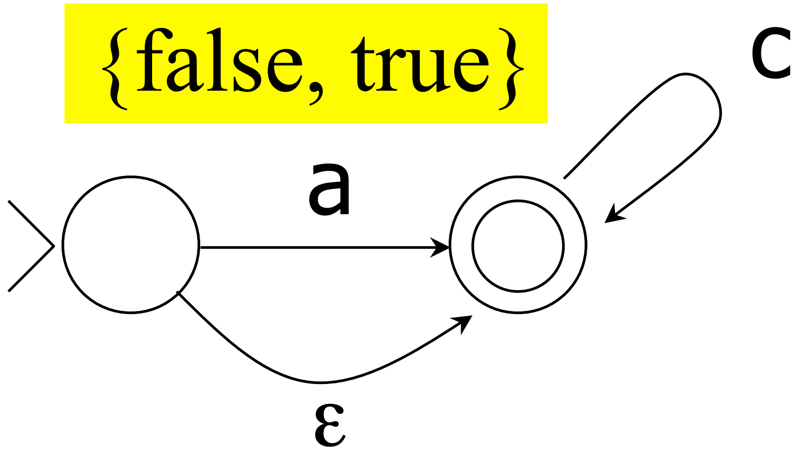
	Acceptors (FSAs)	Transducers (FSTs)
Unweighted		
Weighted		

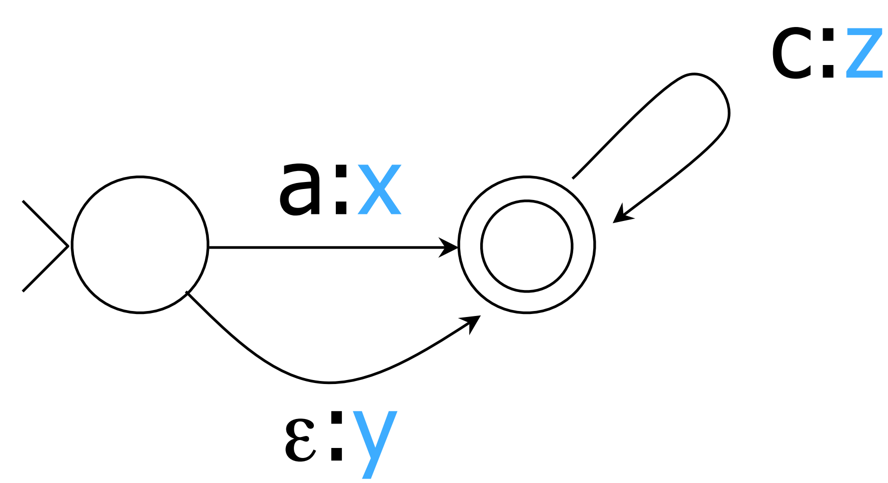
Function from strings to ...

Acceptors (FSAs)

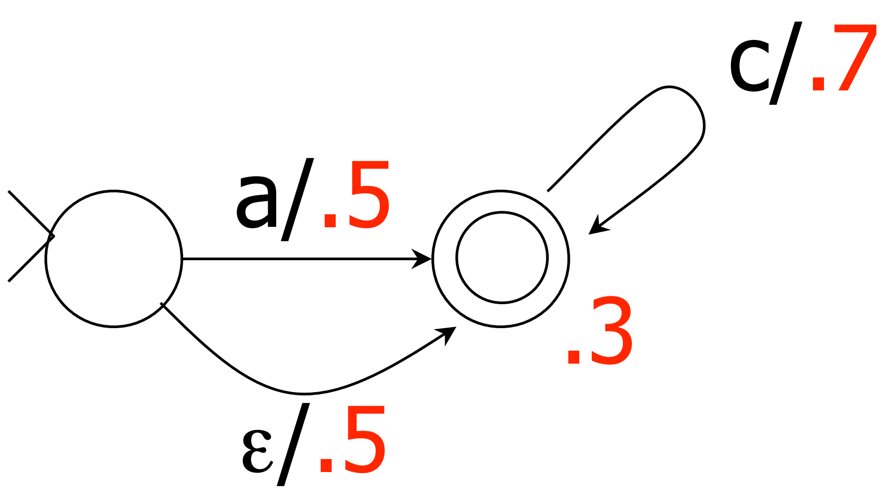
Transducers (FSTs)

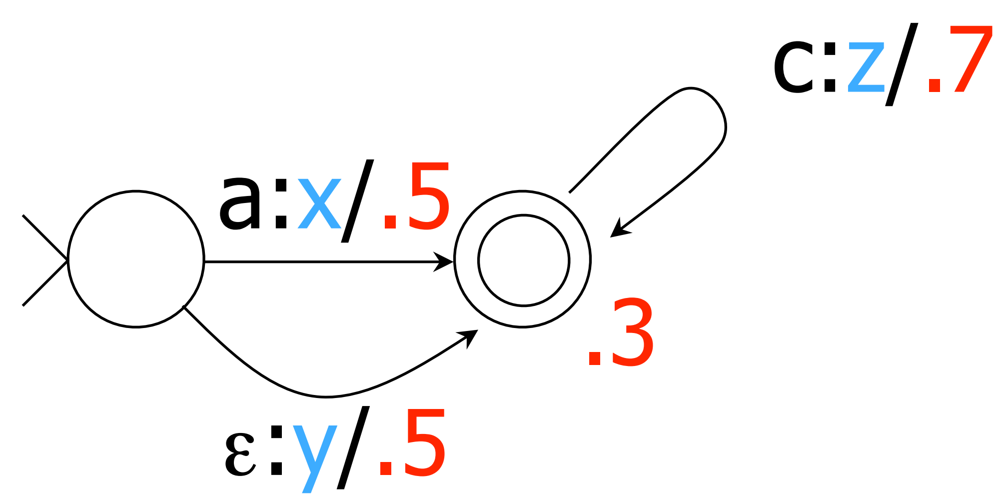
Unweighted



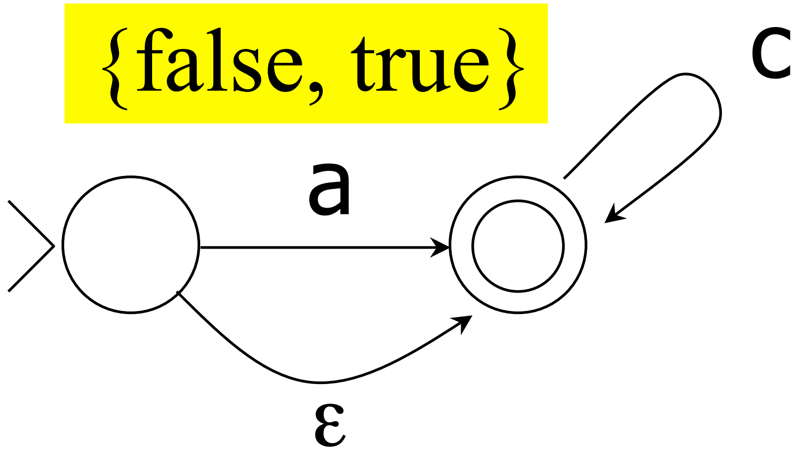
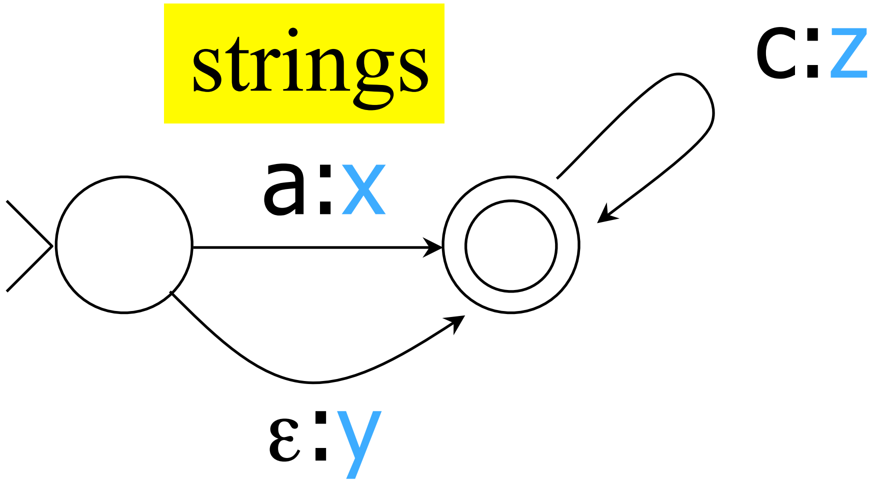
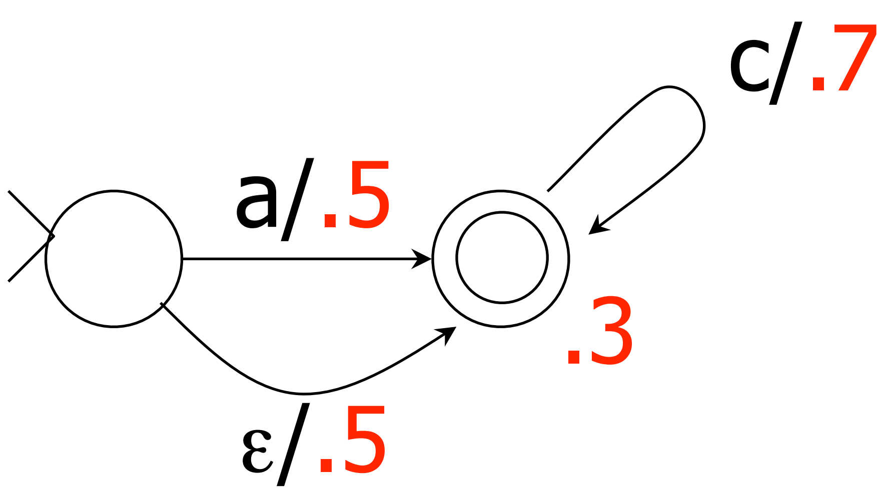
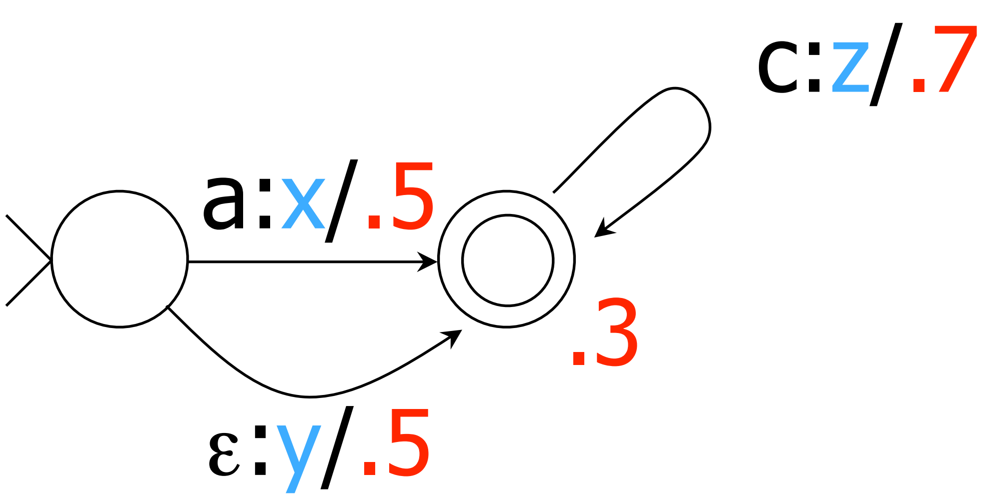


Weighted





Function from strings to ...

	Acceptors (FSAs)	Transducers (FSTs)
	{false, true}	strings
Unweighted		
Weighted		

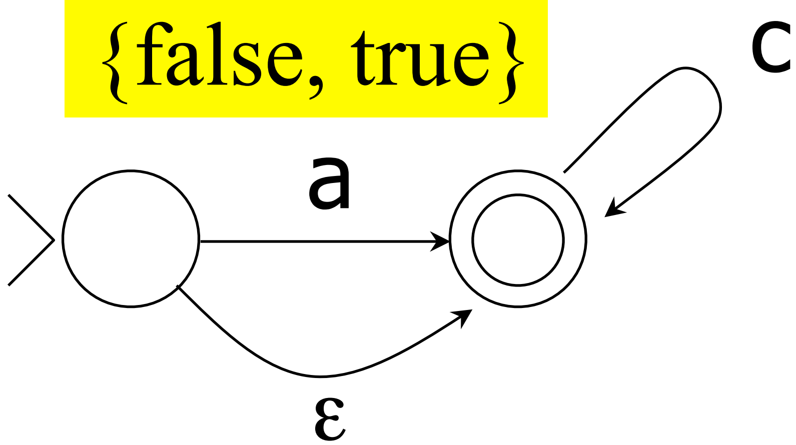
Function from strings to ...

Acceptors (FSAs)

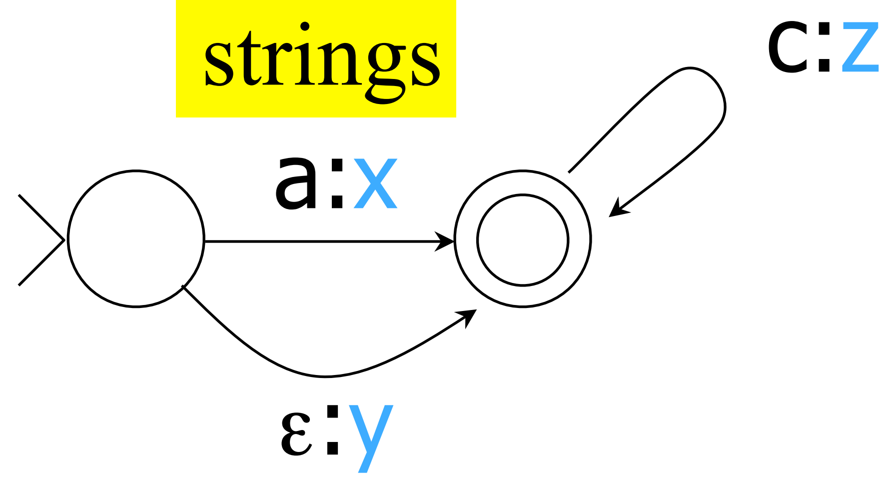
Transducers (FSTs)

Unweighted

{false, true}

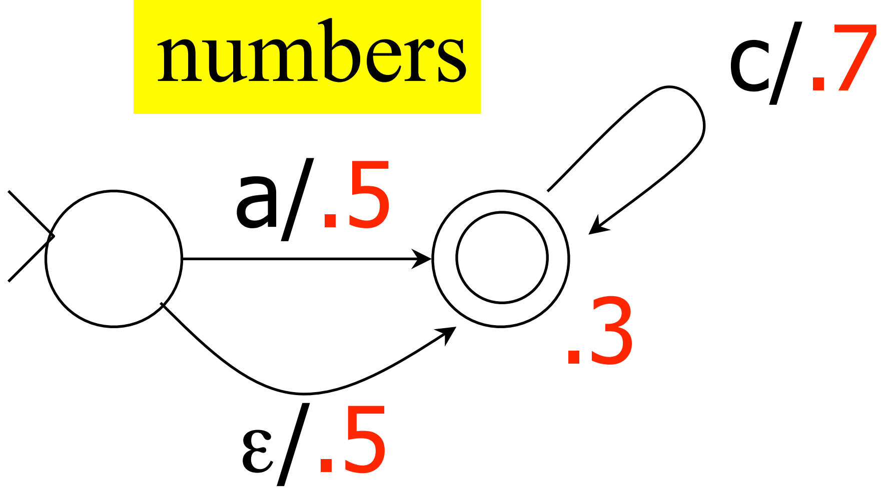


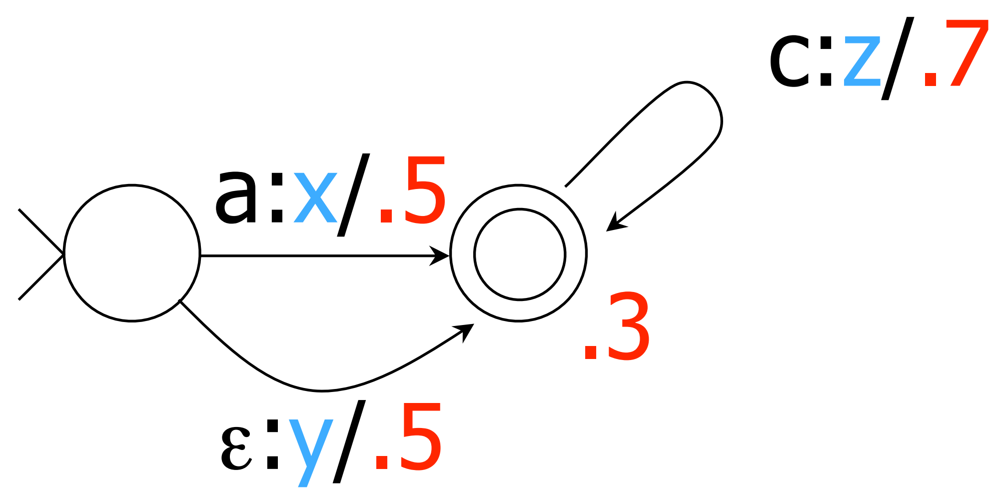
strings



Weighted

numbers

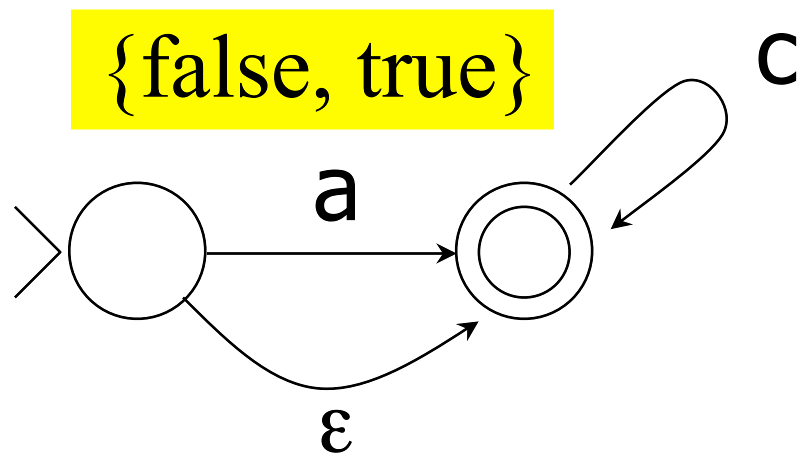


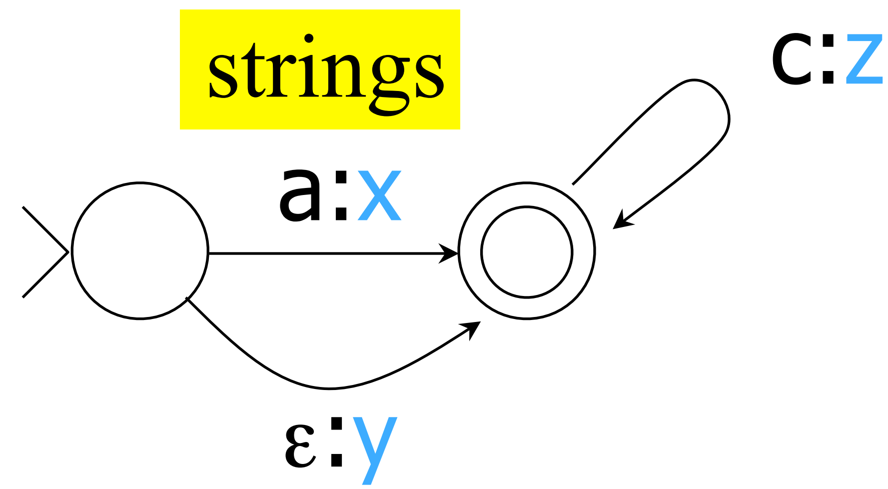


Function from strings to ...

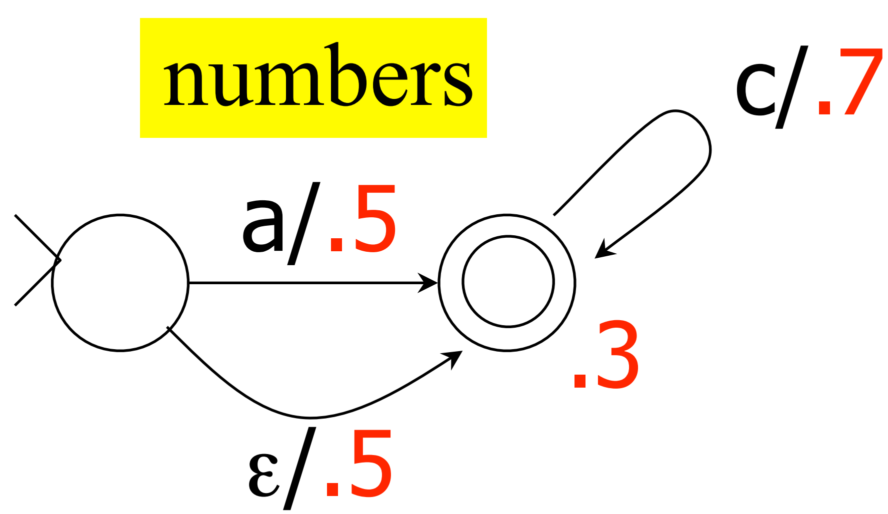
Acceptors (FSAs) | **Transducers (FSTs)**

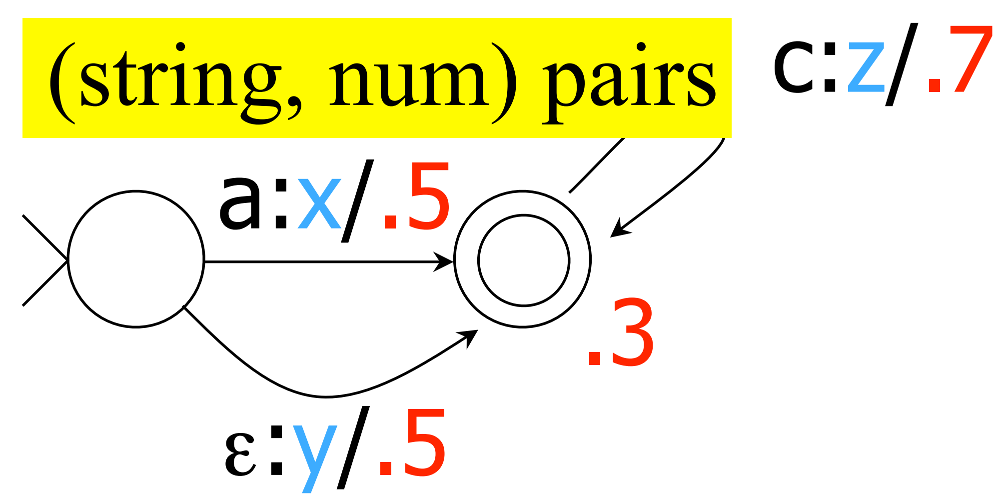
Unweighted





Weighted





Weighted Relations

- If we have a language [or relation], we can ask it:
  Do you contain this string [or string pair]?

- If we have a **weighted** language [or relation], we ask:
  What **weight** do you assign to this string [or string pair]?

- Pick a semiring: all our **weights** will be in that semiring.
  - **What?!**

Semirings

	Set	$\oplus$	$\otimes$	0	1
Prob	R	+	x	0	1
Max	R	max	x	0	1
Log	$R \cup \{\pm\infty\}$	log+	+	$-\infty$	0
"Tropical"	$R \cup \{\pm\infty\}$	max	+	$-\infty$	0
Shortest path	$R \cup \{\pm\infty\}$	min	+	∞	0
Boolean	{0,1}	$\vee$	$\wedge$	F	T
String	Σ	longest common prefix	concat	∞	ε

Weighted Relations

- If we have a language [or relation], we can ask it:
  Do you contain this string [or string pair]?

- If we have a **weighted** language [or relation], we ask:
  What **weight** do you assign to this string [or string pair]?

- Pick a semiring: all our **weights** will be in that semiring.
  - **Don't panic!** We will cover this again when we get to HMMs and parsing.
  - The unweighted case is the boolean semring {true, false}.
  - If a string is not in the language, it has weight $⓪$.
  - If an FST or regular expression can choose among multiple ways to match, use $\oplus$ to combine the weights of the different choices.
  - If an FST or regular expression matches by matching multiple substrings, use $\otimes$ to combine those different matches.
  - Remember, $\oplus$ is like "or" and $\otimes$ is like "and"!

Which Semiring Operators are Needed?

	concatenation	EF
`* +`	iteration	E*, E+
`|`	union	E | F
`~ \ -`	complementation, minus	~E, \x, E-F
`&`	intersection	E & F
`.x.`	crossproduct	E .x. F
`.o.`	composition	E .o. F
`.u`	upper (input) language	E.u “domain”
`.l`	lower (output) language	E.l “range”

Which Semiring Operators are Needed?

concatenation — EF

* + iteration — E*, E+

| union — ⊕ to sum over 2 choices — E | F

~ \ - complementation, minus — ~E, \x, E-F

& intersection — E & F

.x. crossproduct — E .x. F

.o. composition — E .o. F

.u upper (input) language — E.u “domain”

.l lower (output) language — E.l “range”

Which Semiring Operators are Needed?

	concatenation		EF
* +	iteration		E*, E+
|	union	⊕ to sum over 2 choices	E | F
~ \ -	complementation, minus		~E, \x, E-F
&	intersection	⊗ to combine the matches against E and F	E & F
.x.	crossproduct		E .x. F
.o.	composition		E .o. F
.u	upper (input) language		E.u "domain"
.l	lower (output) language		E.l "range"

Common Regular Expression Operators (in XFST notation)

| union | $\oplus$ to sum over 2 choices | E \| F |

$$E \mid F = \{w : w \in E \text{ or } w \in F\} = E \cup F$$

- Weighted case: Let's write $E(w)$ to denote the weight of w in the weighted language E.

$$(E|F)(w) = E(w) \oplus F(w)$$

Which Semiring Operators are Needed?

	concatenation		EF
`* +`	iteration		E*, E+
`|`	union	⊕ to sum over 2 choices	E | F
`~ \ -`	complementation, minus		~E, \x, E-F
`&`	intersection		E & F
`.x.`	crossproduct		E .x. F
`.o.`	composition		E .o. F
`.u`	upper (input) language		E.u "domain"
`.l`	lower (output) language		E.l "range"

Which Semiring Operators are Needed?

	concatenation		EF
* +	iteration		E*, E+
|	union	⊕ to sum over 2 choices	E | F
~ \ -	complementation, minus		~E, \x, E-F
&	intersection	⊗ to combine the matches against E and F	E & F
.x.	crossproduct		E .x. F
.o.	composition		E .o. F
.u	upper (input) language		E.u "domain"
.l	lower (output) language		E.l "range"

Which Semiring Operators are Needed?

<table>
<tr><td></td><td>concatenation</td><td rowspan="2">need both ⊕ and ⊗</td><td>EF</td></tr>
<tr><td><code>* +</code></td><td>iteration</td><td>E*, E+</td></tr>
<tr><td><code>|</code></td><td>union</td><td>⊕ to sum over 2 choices</td><td>E | F</td></tr>
<tr><td><code>~ \ -</code></td><td>complementation, minus</td><td></td><td>~E, \x, E-F</td></tr>
<tr><td><code>&</code></td><td>intersection</td><td rowspan="2">⊗ to combine the matches against E and F</td><td>E & F</td></tr>
<tr><td><code>.x.</code></td><td>crossproduct</td><td>E .x. F</td></tr>
<tr><td><code>.o.</code></td><td>composition</td><td></td><td>E .o. F</td></tr>
<tr><td><code>.u</code></td><td>upper (input) language</td><td></td><td>E.u “domain”</td></tr>
<tr><td><code>.l</code></td><td>lower (output) language</td><td></td><td>E.l “range”</td></tr>
</table>

Which Semiring Operators are Needed?

- concatenation EF
- \+ iteration E*, E+

need both ⊕ and ⊗

$$EF = \{ef: e \in E, f \in F\}$$

- Weighted case must match two things (⊗), but there's a choice (⊕) about which two things:

$$(EF)(w) = \bigoplus_{\substack{e,f \text{ such} \\ \text{that } w=ef}} (E(e) \otimes F(f))$$

Which Semiring Operators are Needed?

concatenation — EF

`* +` iteration — E*, E+

need both ⊕ and ⊗ (concatenation, iteration)

`|` union — E | F

⊕ to sum over 2 choices

`~ \ -` complementation, minus — ~E, \x, E-F

`&` intersection — E & F

`.x.` crossproduct — E .x. F

⊗ to combine the matches against E and F (intersection, crossproduct)

`.o.` composition — E .o. F

`.u` upper (input) language — E.u “domain”

`.l` lower (output) language — E.l “range”

Which Semiring Operators are Needed?

<table>
<tr><td></td><td>concatenation</td><td rowspan="2">need both ⊕ and ⊗</td><td>EF</td><td></td></tr>
<tr><td>* +</td><td>iteration</td><td>E*, E+</td><td></td></tr>
<tr><td>|</td><td>union</td><td>⊕ to sum over 2 choices</td><td>E | F</td><td></td></tr>
<tr><td>~ \ -</td><td>complementation, minus</td><td></td><td>~E, \x, E-F</td><td></td></tr>
<tr><td>&</td><td>intersection</td><td rowspan="2">⊗ to combine the matches against E and F</td><td>E & F</td><td></td></tr>
<tr><td>.x.</td><td>crossproduct</td><td>E .x. F</td><td></td></tr>
<tr><td>.o.</td><td>composition</td><td>both ⊕ and ⊗ (why?)</td><td>E .o. F</td><td></td></tr>
<tr><td>.u</td><td>upper (input) language</td><td></td><td>E.u</td><td>“domain”</td></tr>
<tr><td>.l</td><td>lower (output) language</td><td></td><td>E.l</td><td>“range”</td></tr>
</table>

Which Semiring Operators are Needed?

	concatenation	need both $\oplus$ and $\otimes$	EF	
`*` `+`	iteration		E*, E+	
`|`	union	$\oplus$ to sum over 2 choices	E | F	
`~` `\` `-`	complementation, minus		~E, \x, E-F	
`&`	intersection	$\otimes$ to combine the matches against E and F	E & F	
`.x.`	crossproduct		E .x. F	
`.o.`	composition	both $\oplus$ and $\otimes$ (why?)	E .o. F	
`.u`	upper (input) language	$\oplus$	E.u	“domain”
`.l`	lower (output) language	$\oplus$	E.l	“range”

Definition of FSTs

Definition of FSTs

- [Red material shows differences from FSAs.]

Definition of FSTs

- [Red material shows differences from FSAs.]
- Simple view:
  - An FST is simply a finite directed graph, with some labels.
  - It has a designated initial state and a set of final states.
  - Each edge is labeled with an "upper string" (in Σ^*).

Definition of FSTs

- [Red material shows differences from FSAs.]
- Simple view:
  - An FST is simply a finite directed graph, with some labels.
  - It has a designated initial state and a set of final states.
  - Each edge is labeled with an "upper string" (in Σ^*).
  - Each edge is also labeled with a "lower string" (in Δ^*).
  - [Upper/lower are sometimes regarded as input/output.]

Definition of FSTs

- [Red material shows differences from FSAs.]
- Simple view:
  - An FST is simply a finite directed graph, with some labels.
  - It has a designated initial state and a set of final states.
  - Each edge is labeled with an “upper string” (in Σ^*).
  - Each edge is also labeled with a “lower string” (in Δ^*).
  - [Upper/lower are sometimes regarded as input/output.]
  - Each edge and final state is also labeled with a semiring weight.

Definition of FSTs

- [Red material shows differences from FSAs.]
- Simple view:
  - An FST is simply a finite directed graph, with some labels.
  - It has a designated initial state and a set of final states.
  - Each edge is labeled with an "upper string" (in Σ^*).
  - Each edge is also labeled with a "lower string" (in Δ^*).
  - [Upper/lower are sometimes regarded as input/output.]
  - Each edge and final state is also labeled with a semiring weight.
- More traditional definition specifies an FST via these:
  - a state set Q
  - initial state i
  - set of final states F
  - input alphabet Σ (also define Σ^*, $\Sigma+$, $\Sigma?$)
  - output alphabet Δ

Definition of FSTs

- [Red material shows differences from FSAs.]
- Simple view:
  - An FST is simply a finite directed graph, with some labels.
  - It has a designated initial state and a set of final states.
  - Each edge is labeled with an "upper string" (in Σ^*).
  - Each edge is also labeled with a "lower string" (in Δ^*).
  - [Upper/lower are sometimes regarded as input/output.]
  - Each edge and final state is also labeled with a semiring weight.
- More traditional definition specifies an FST via these:
  - a state set Q
  - initial state i
  - set of final states F
  - input alphabet Σ (also define Σ^*, $\Sigma+$, $\Sigma?$)
  - output alphabet Δ
  - transition function d: Q x Σ? --> 2^Q

Definition of FSTs

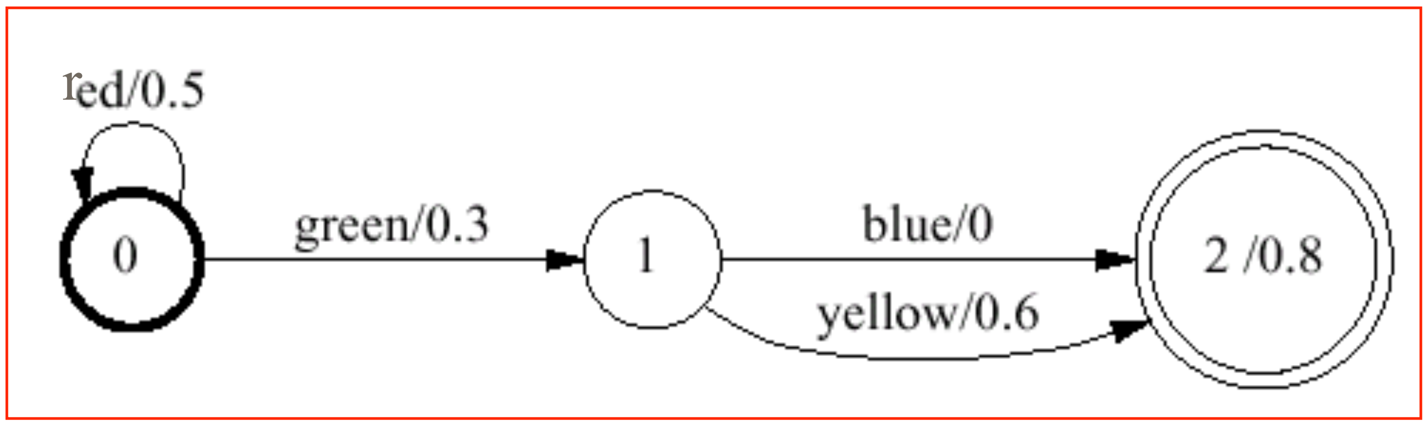
- [Red material shows differences from FSAs.]
- Simple view:
  - An FST is simply a finite directed graph, with some labels.
  - It has a designated initial state and a set of final states.
  - Each edge is labeled with an "upper string" (in Σ^*).
  - Each edge is also labeled with a "lower string" (in Δ^*).
  - [Upper/lower are sometimes regarded as input/output.]
  - Each edge and final state is also labeled with a semiring weight.
- More traditional definition specifies an FST via these:
  - a state set Q
  - initial state i
  - set of final states F
  - input alphabet Σ (also define Σ^*, $\Sigma+$, $\Sigma?$)
  - output alphabet Δ
  - transition function d: Q x Σ? --> 2^Q
  - output function s: Q x Σ? x Q --> Δ?

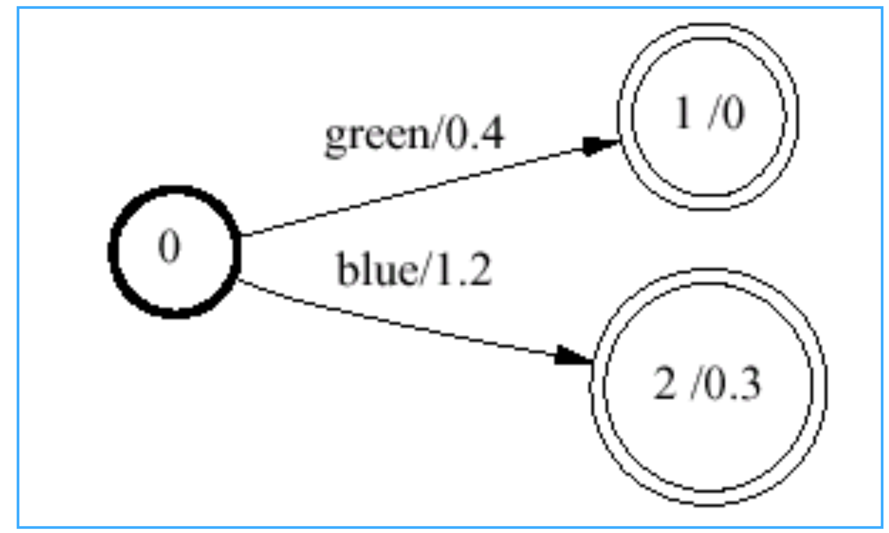
How to implement?

	concatenation	EF
`* +`	iteration	E*, E+
`|`	union	E | F
`~ \ -`	complementation, minus	~E, \x, E-F
`&`	intersection	E & F
`.x.`	crossproduct	E .x. F
`.o.`	composition	E .o. F
`.u`	upper (input) language	E.u “domain”
`.l`	lower (output) language	E.l “range”

example courtesy of M. Mohri

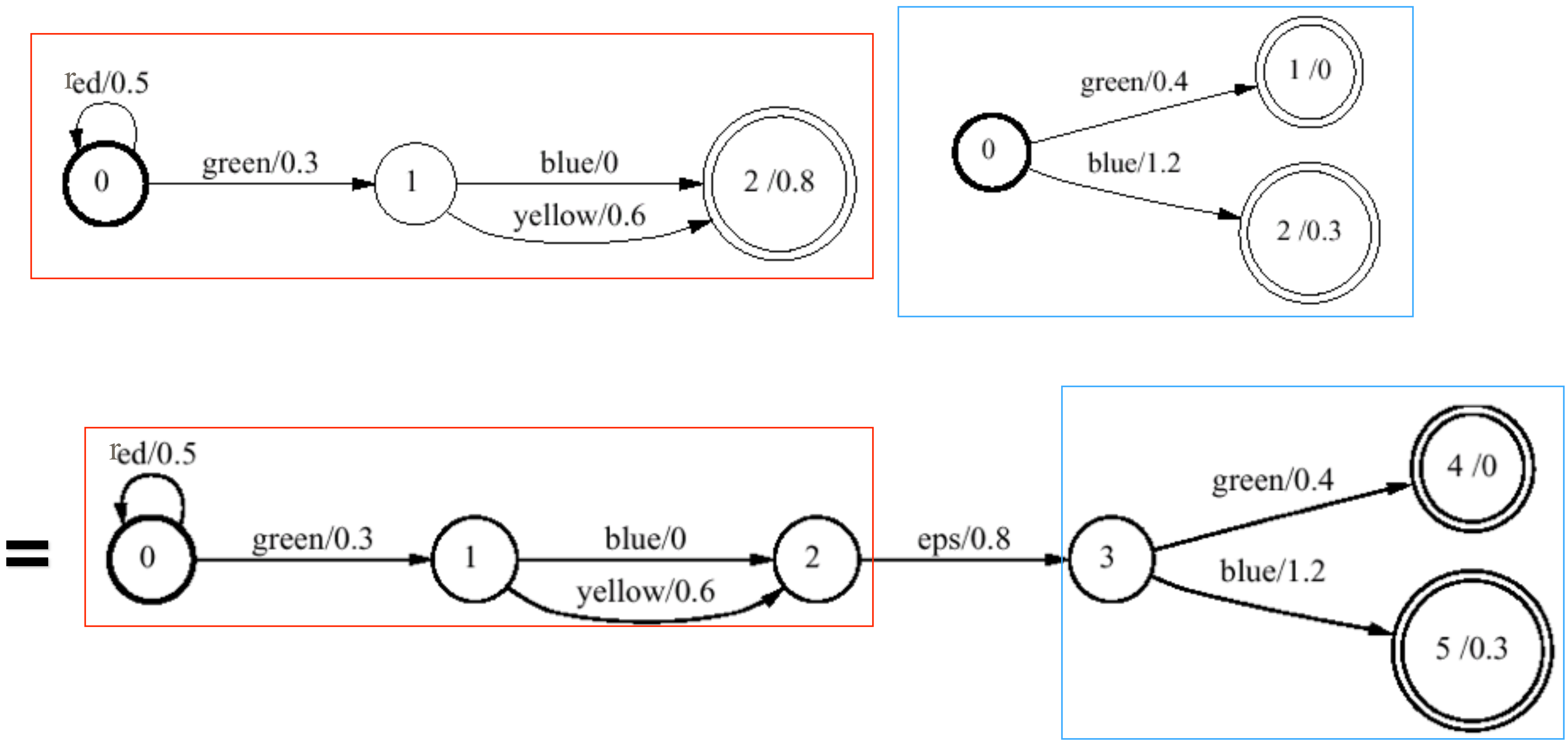
Concatenation





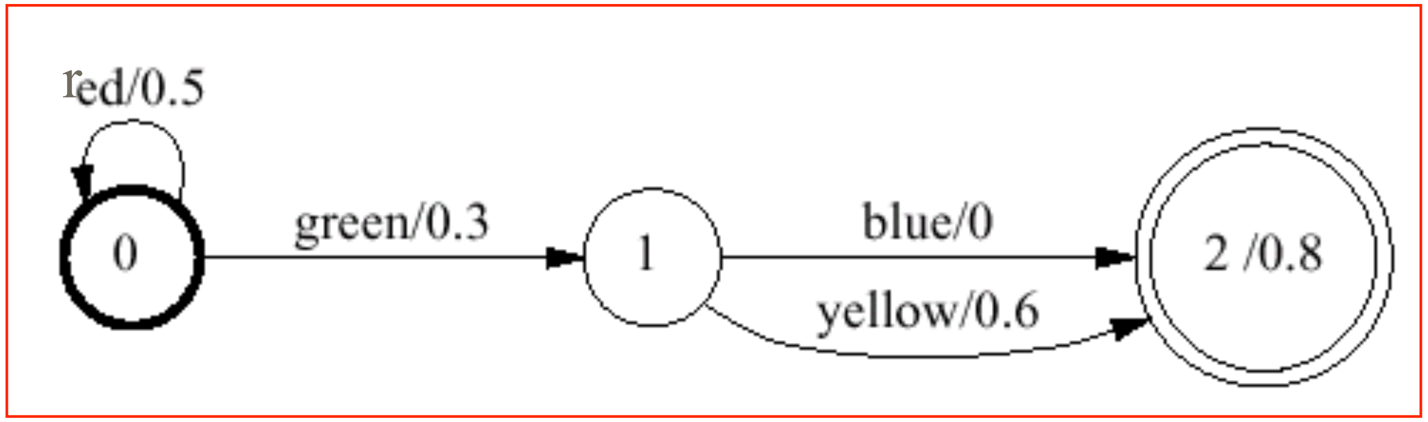
example courtesy of M. Mohri

Concatenation

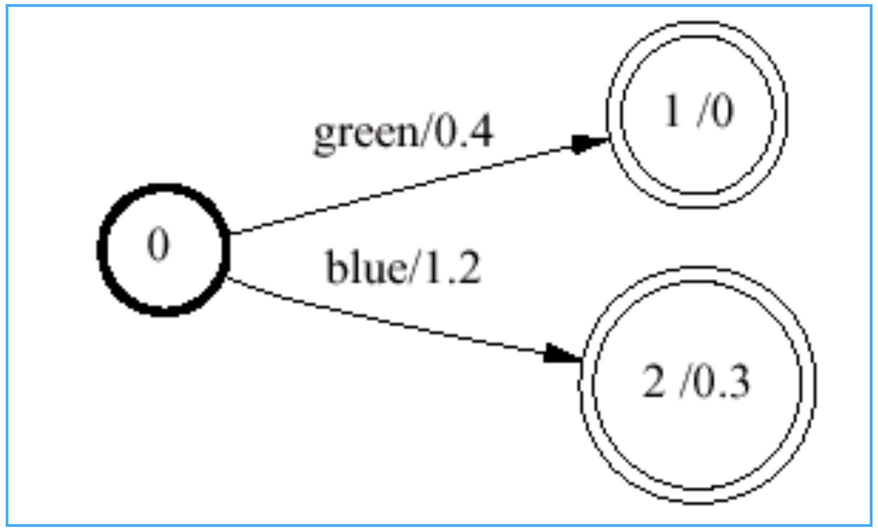


example courtesy of M. Mohri

Union

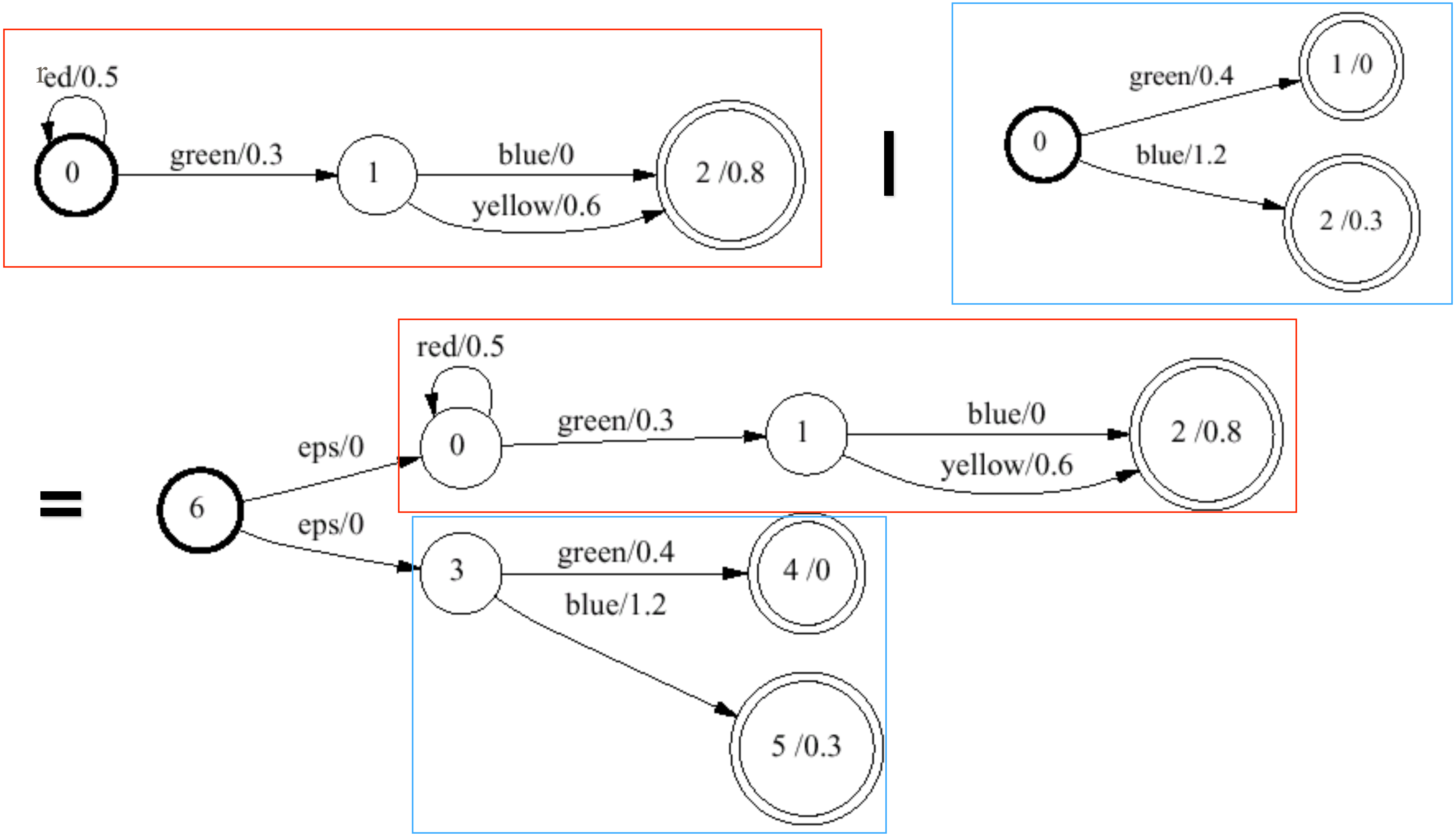


|



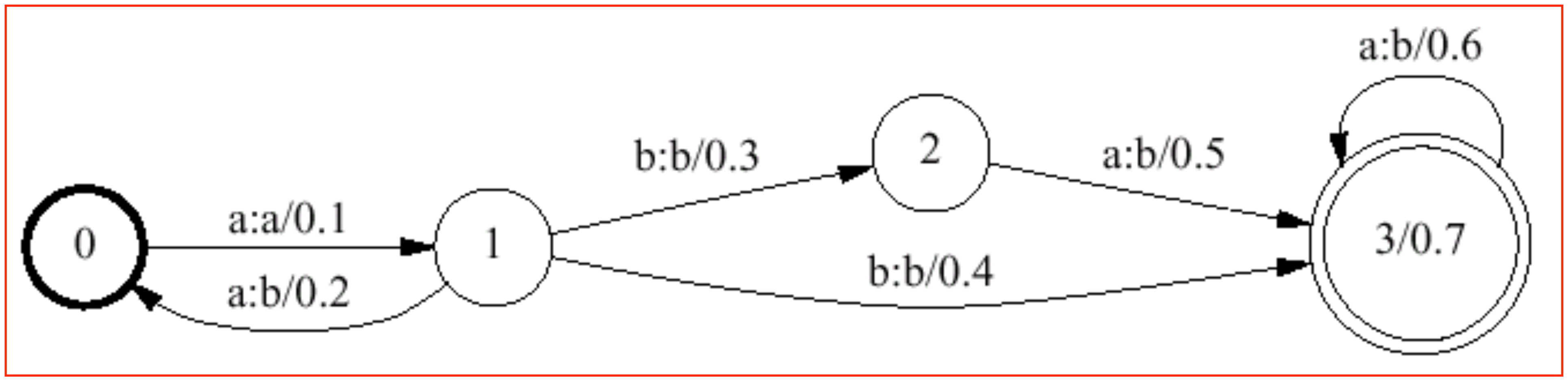
example courtesy of M. Mohri

Union



example courtesy of M. Mohri

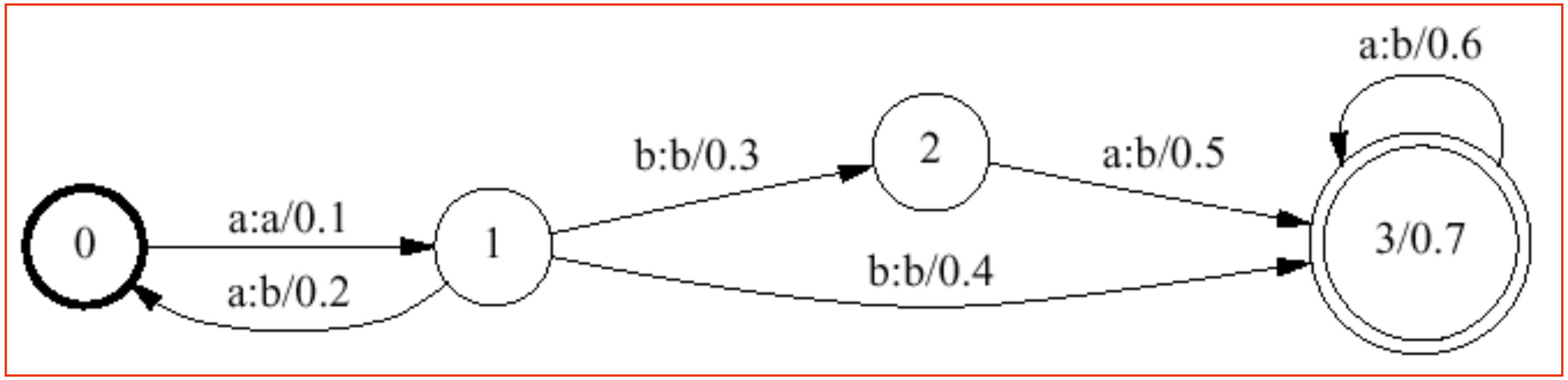
Closure (this example has outputs too)



*

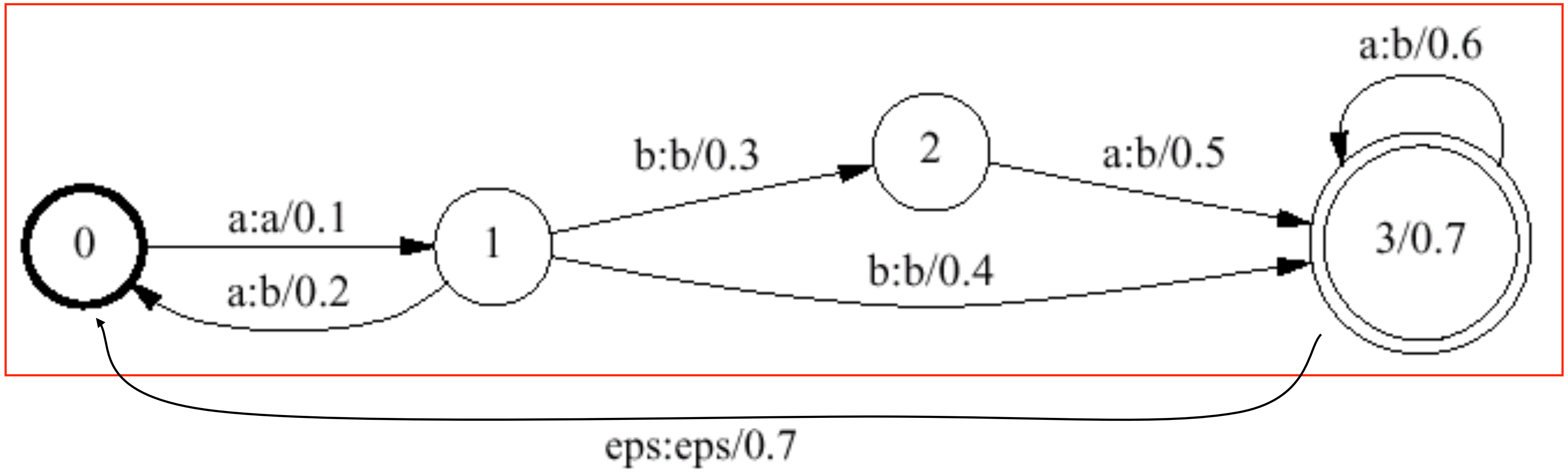
example courtesy of M. Mohri

Closure (this example has outputs too)



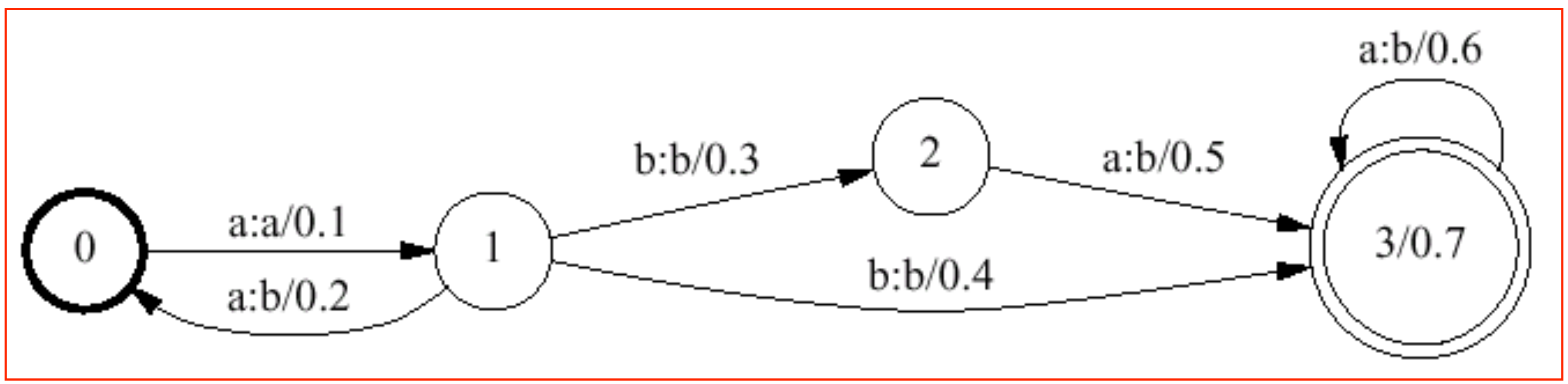
*

=



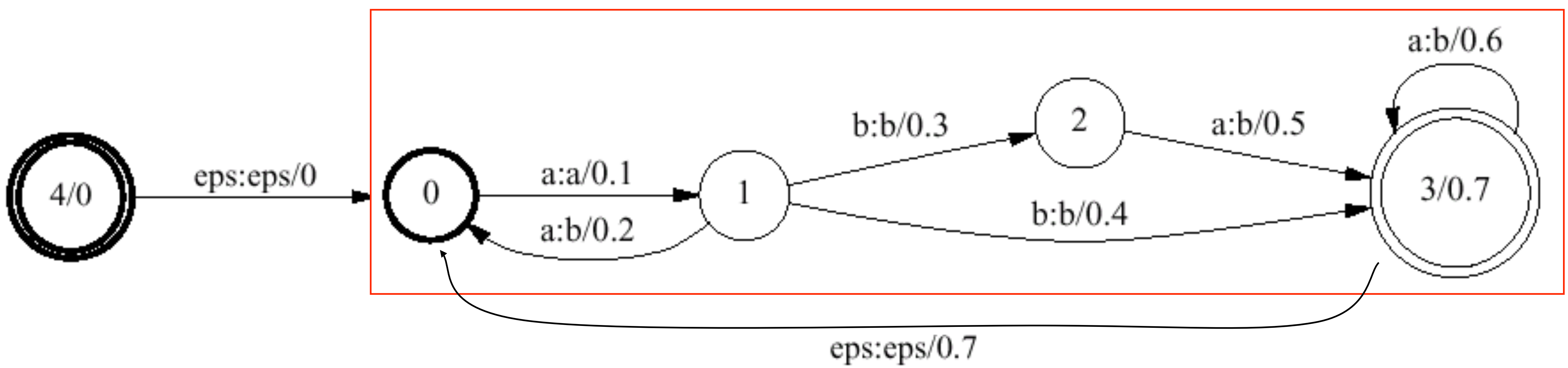
example courtesy of M. Mohri

Closure (this example has outputs too)



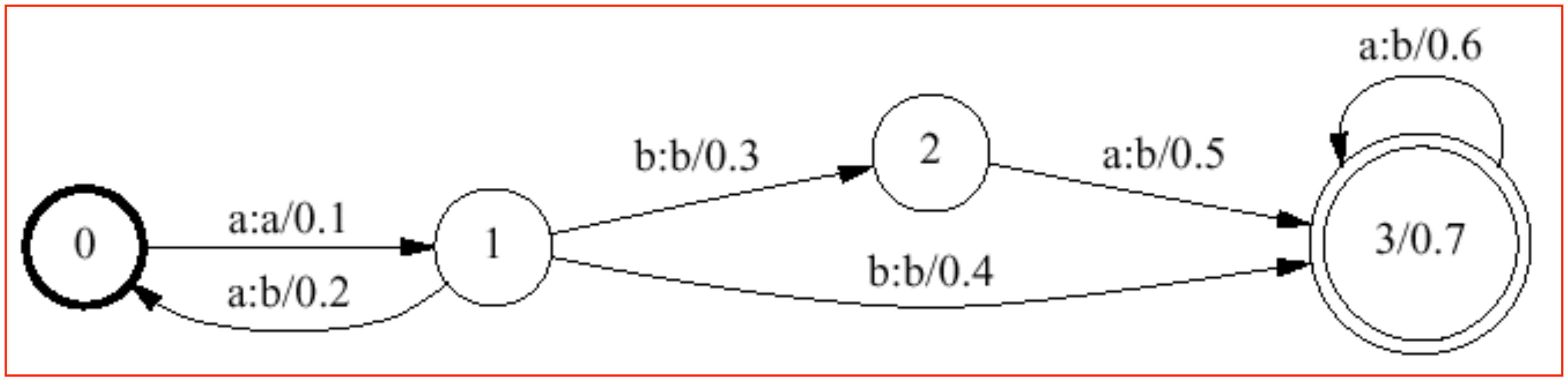
*

=



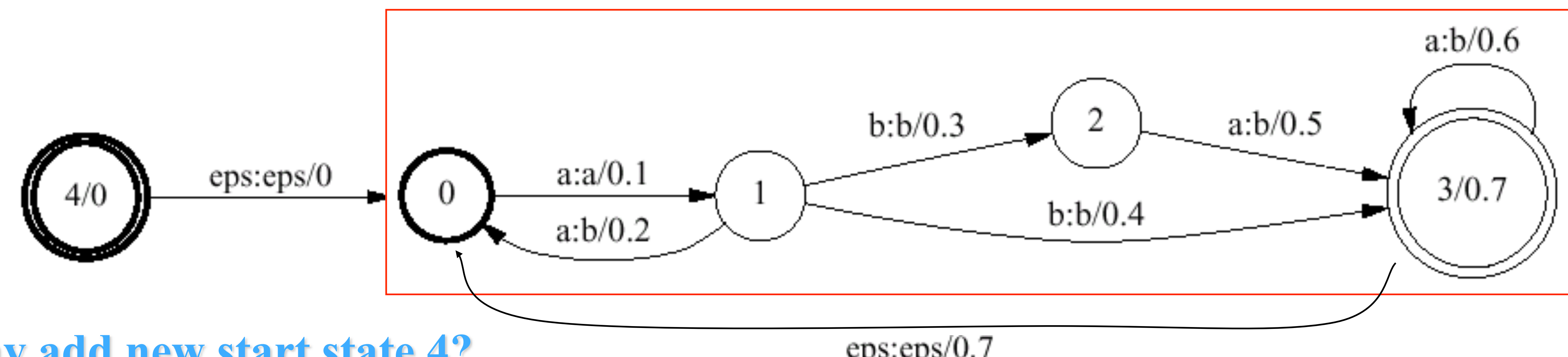
example courtesy of M. Mohri

Closure (this example has outputs too)



*

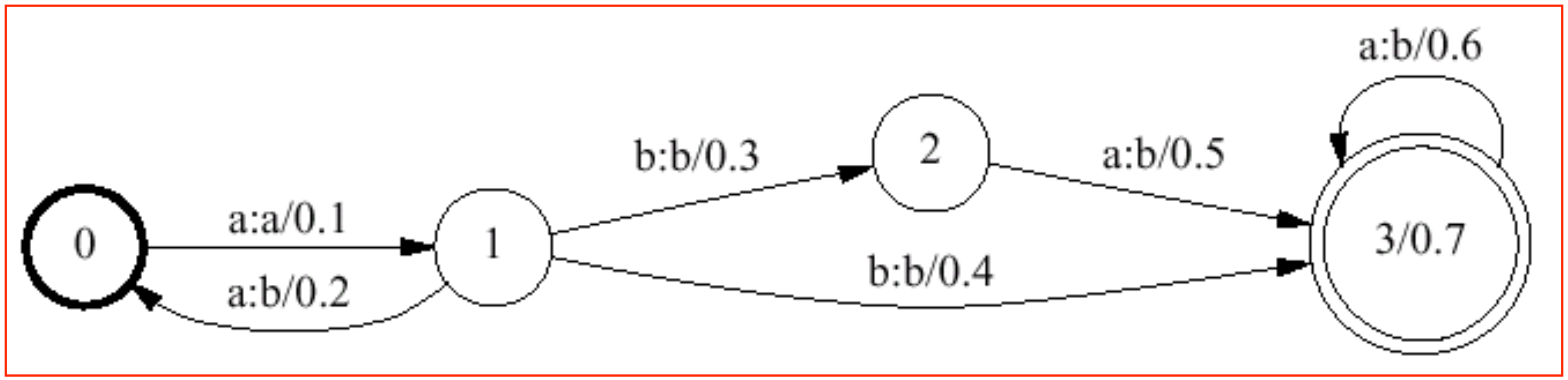
=



why add new start state 4?
why not just make state 0 final?

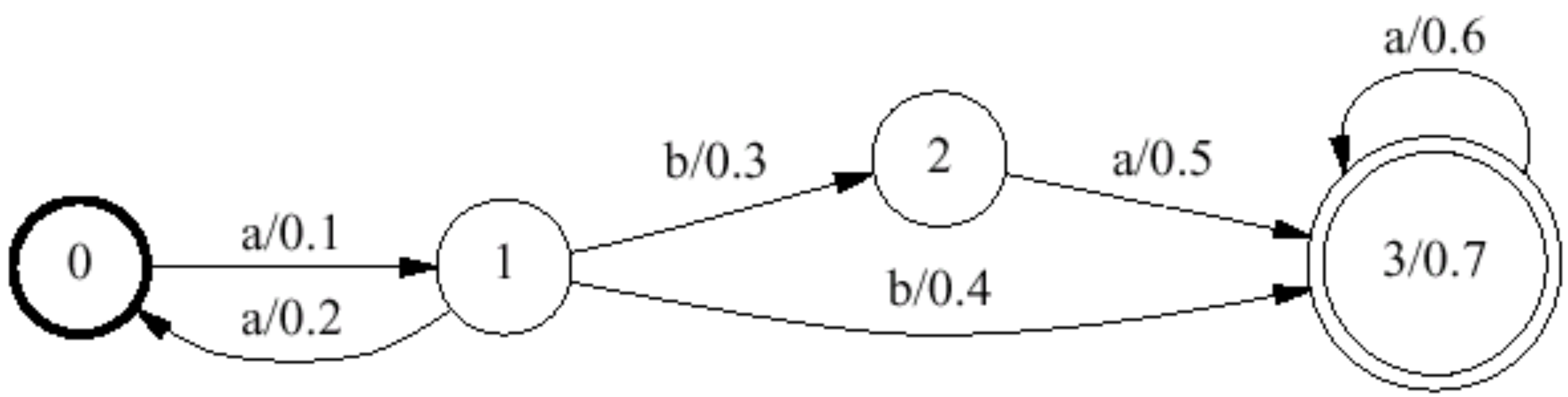
example courtesy of M. Mohri

Upper language (domain)

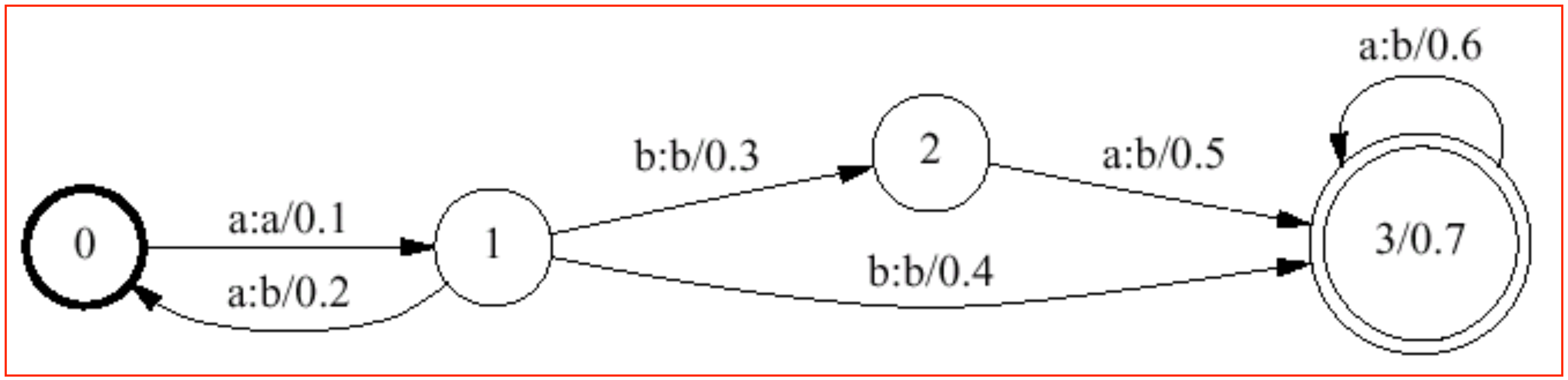


.u

=

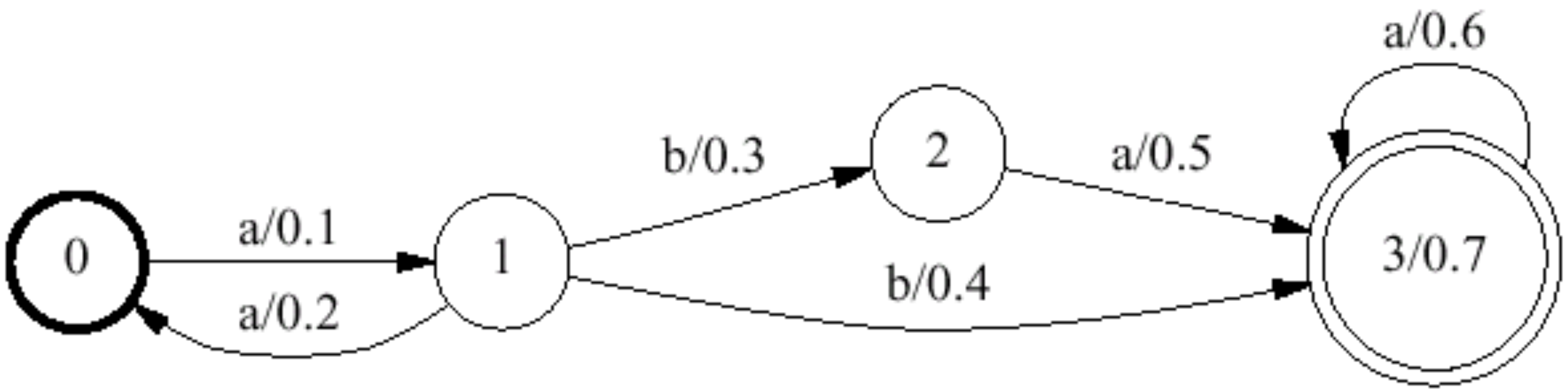


Upper language (domain)



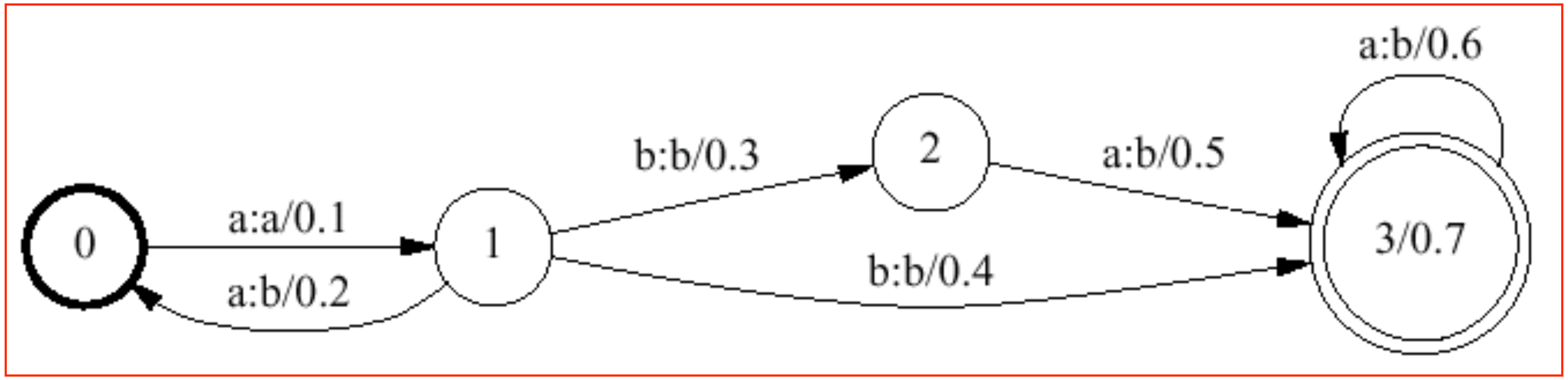
.u

=



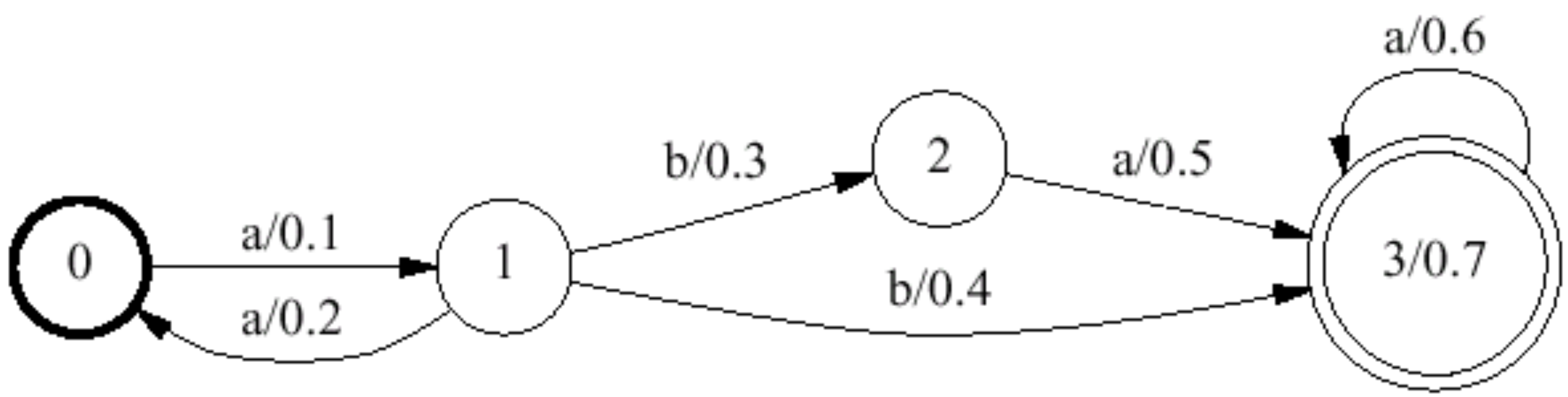
similarly construct lower language .l

Upper language (domain)



.u

=



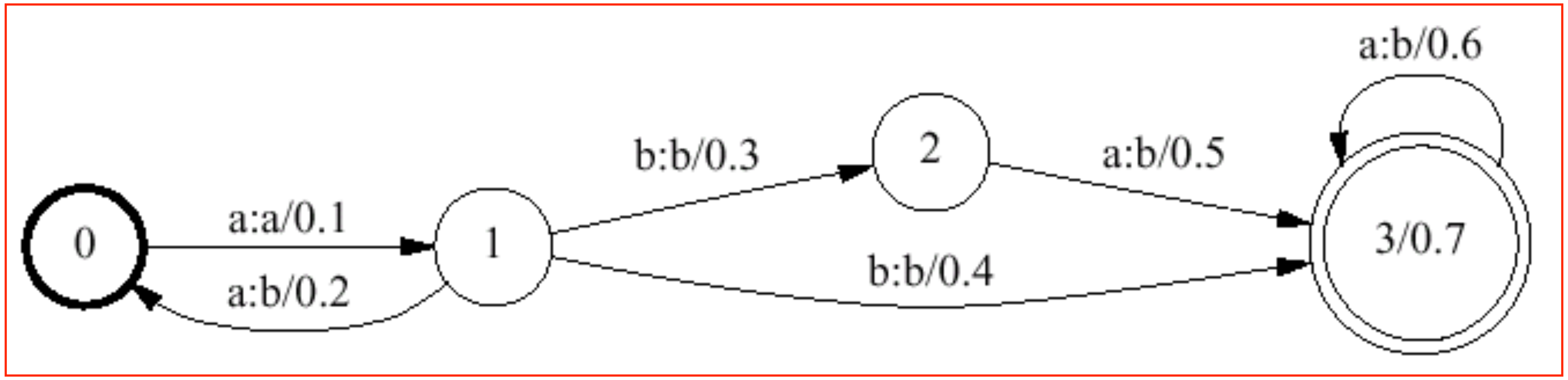
similarly construct lower language .l
also called input & output languages

example courtesy of M. Mohri

Reversal

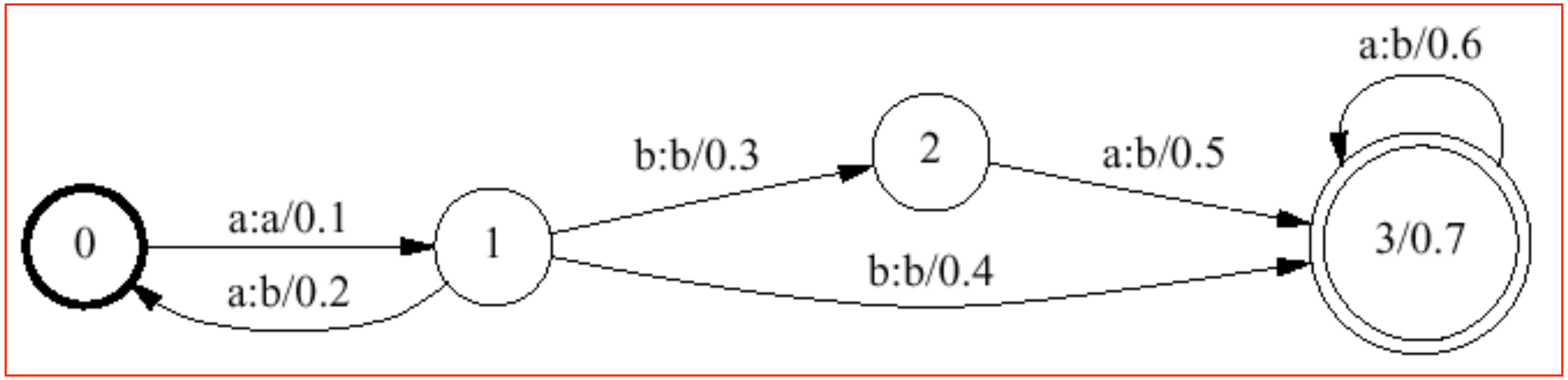
example courtesy of M. Mohri

Reversal



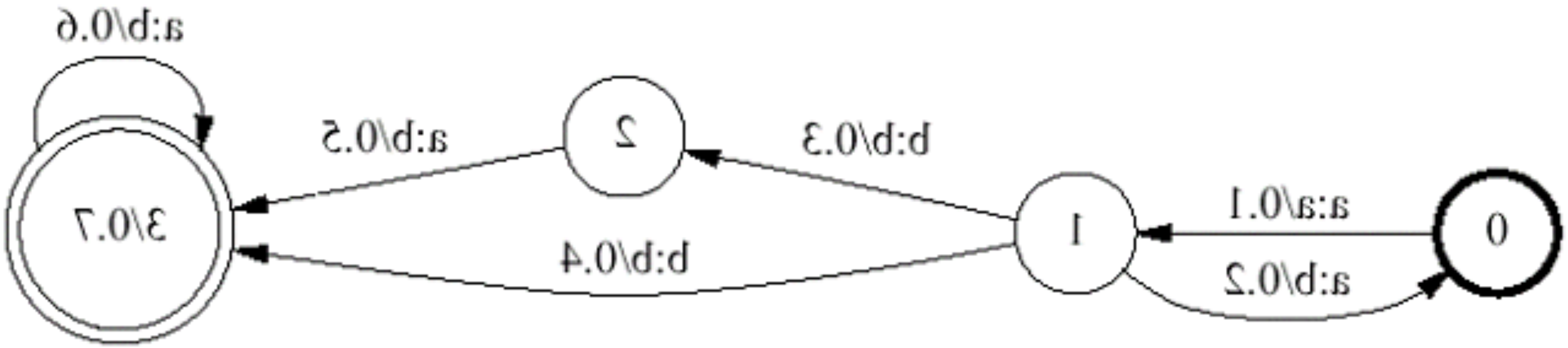
.r

Reversal



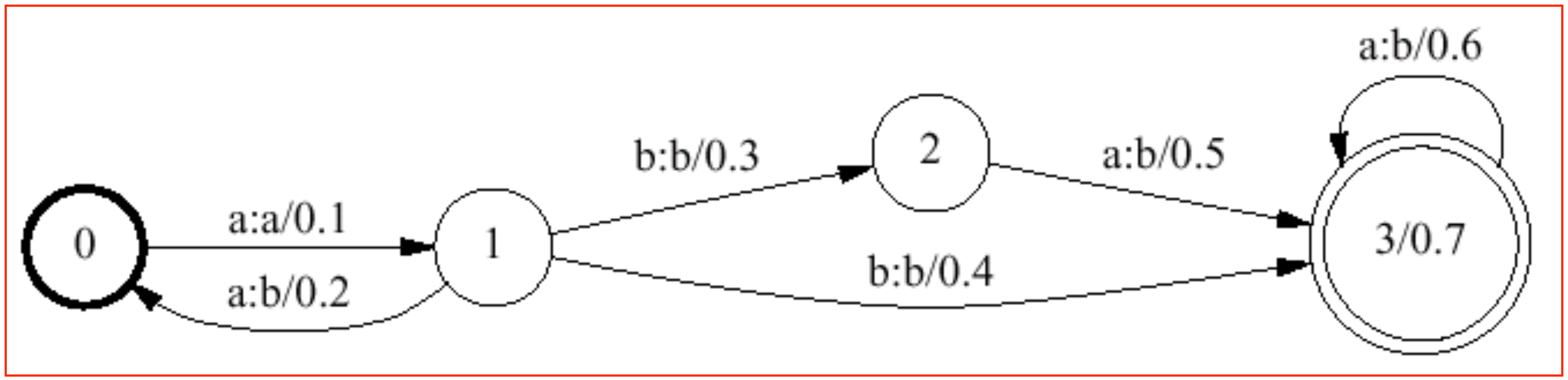
.r

=



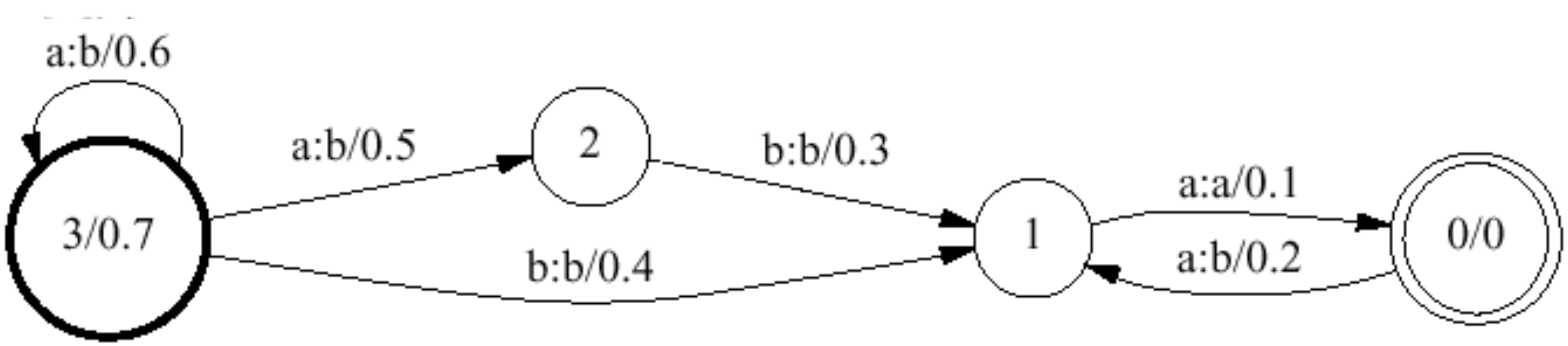
example courtesy of M. Mohri

Reversal



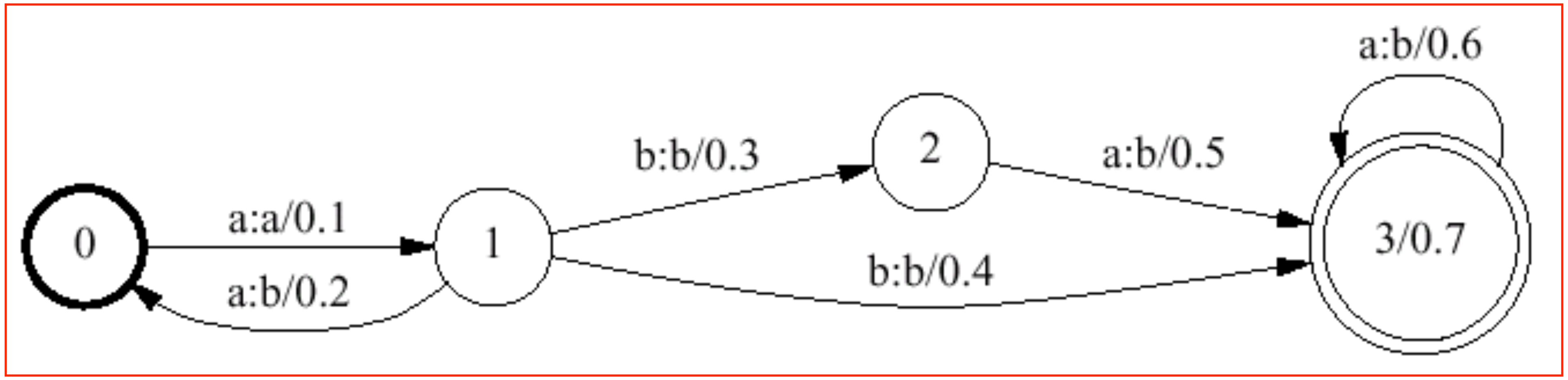
.r

=



example courtesy of M. Mohri

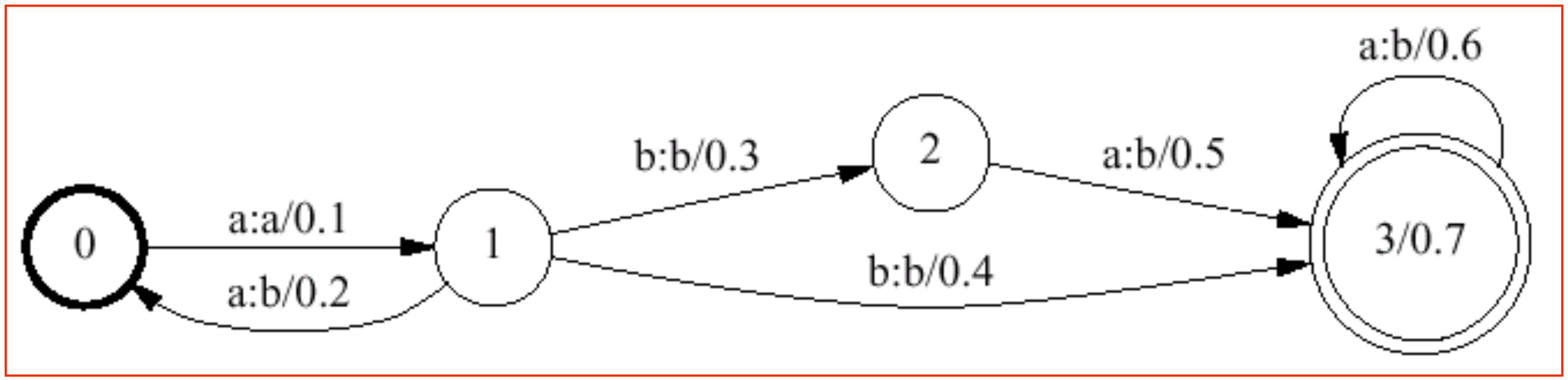
Inversion



.i

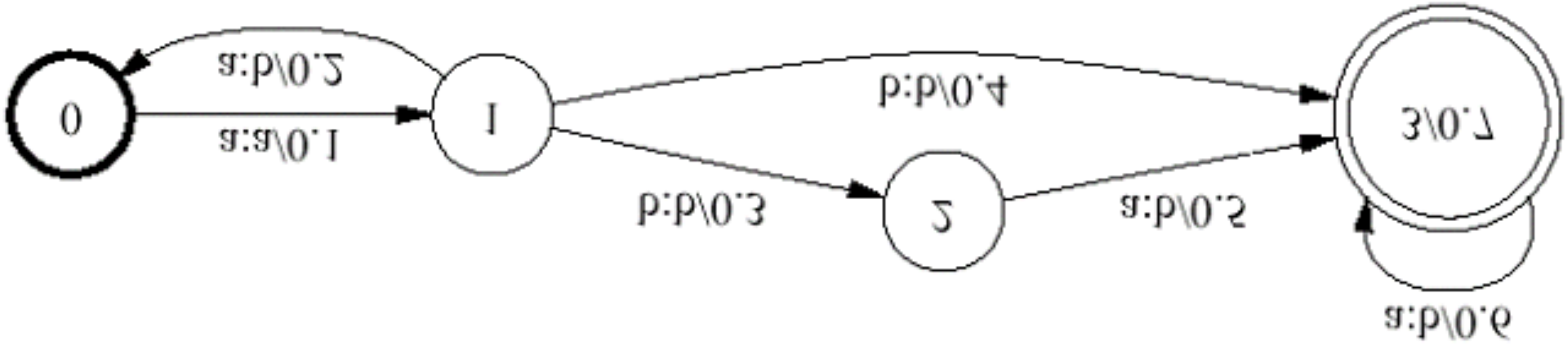
example courtesy of M. Mohri

Inversion

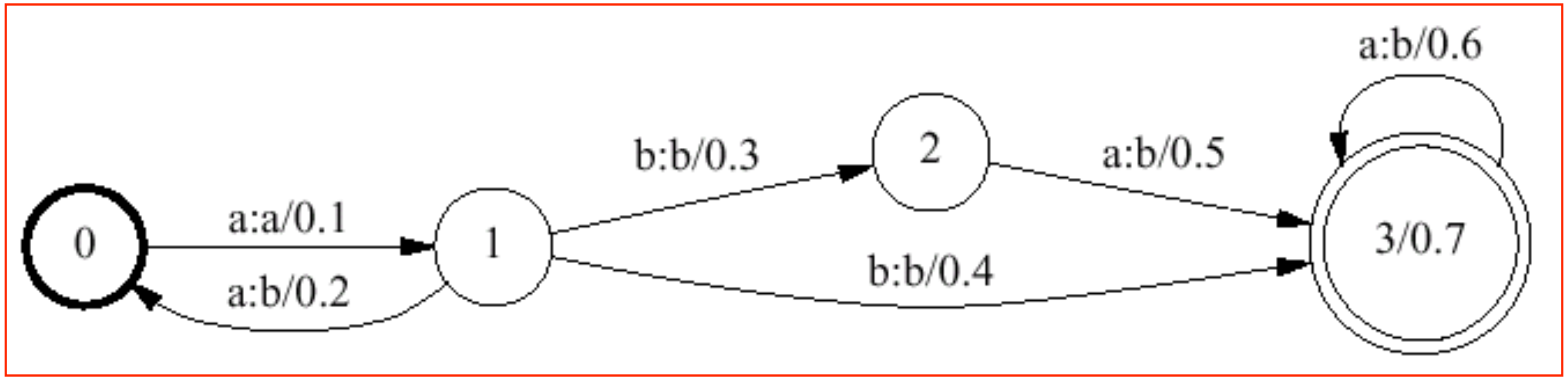


.i

=

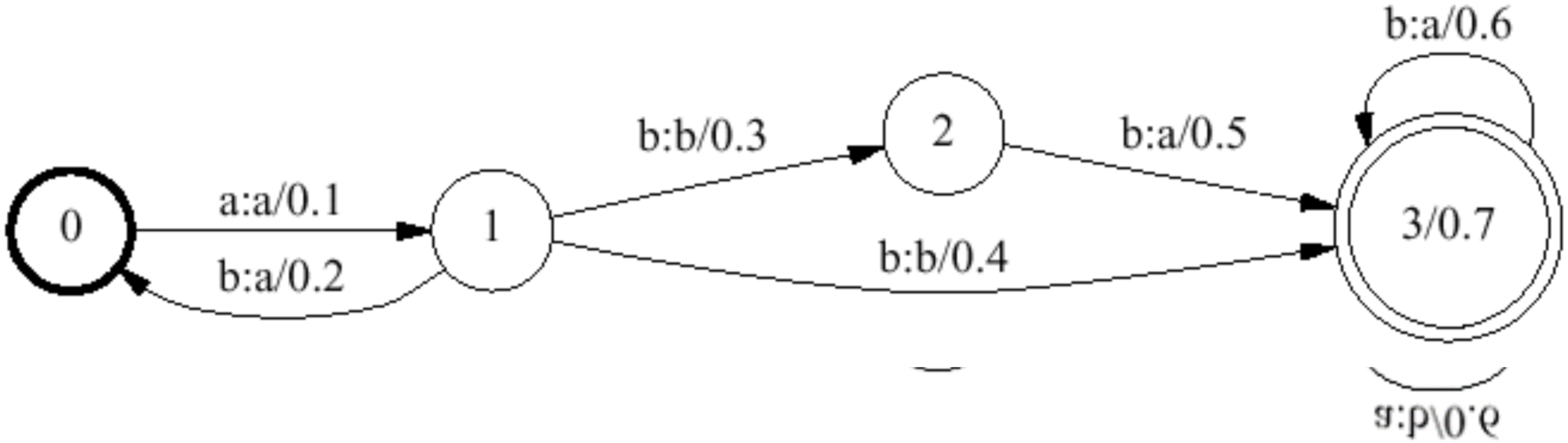


Inversion


a:b/0.6
b:b/0.3
2
a:b/0.5
0
a:a/0.1
1
3/0.7
a:b/0.2
b:b/0.4

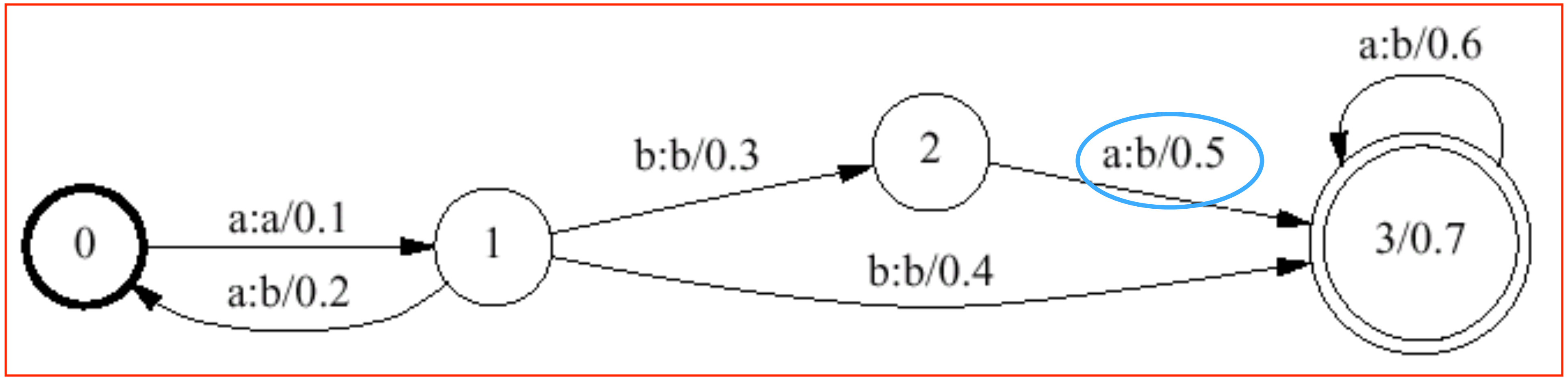
.i

=


b:a/0.6
b:b/0.3
2
b:a/0.5
0
a:a/0.1
1
3/0.7
b:a/0.2
b:b/0.4

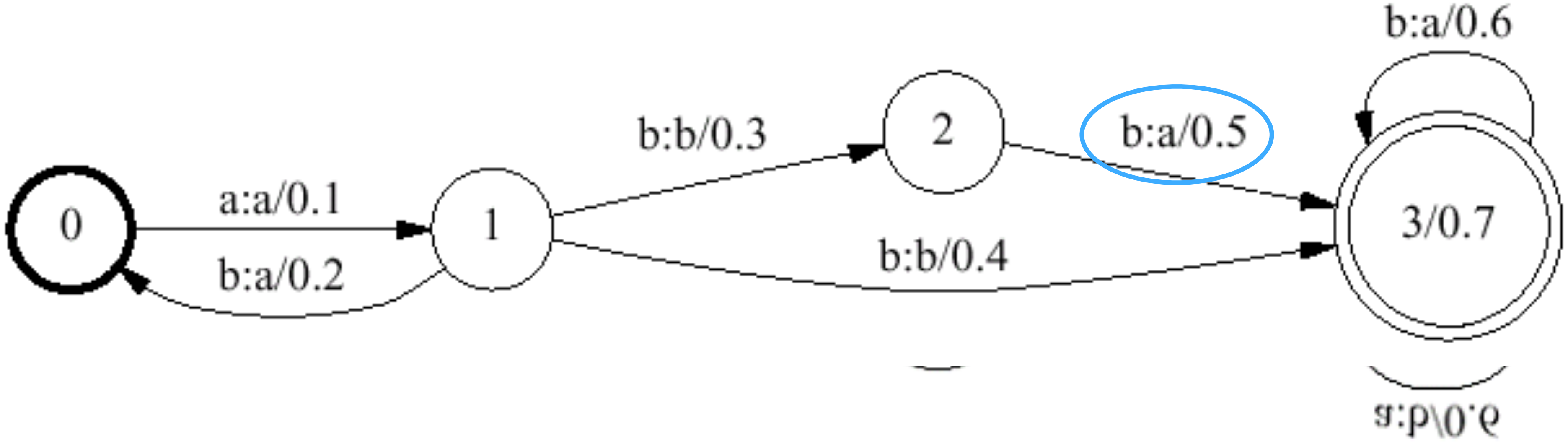
example courtesy of M. Mohri

Inversion



.i

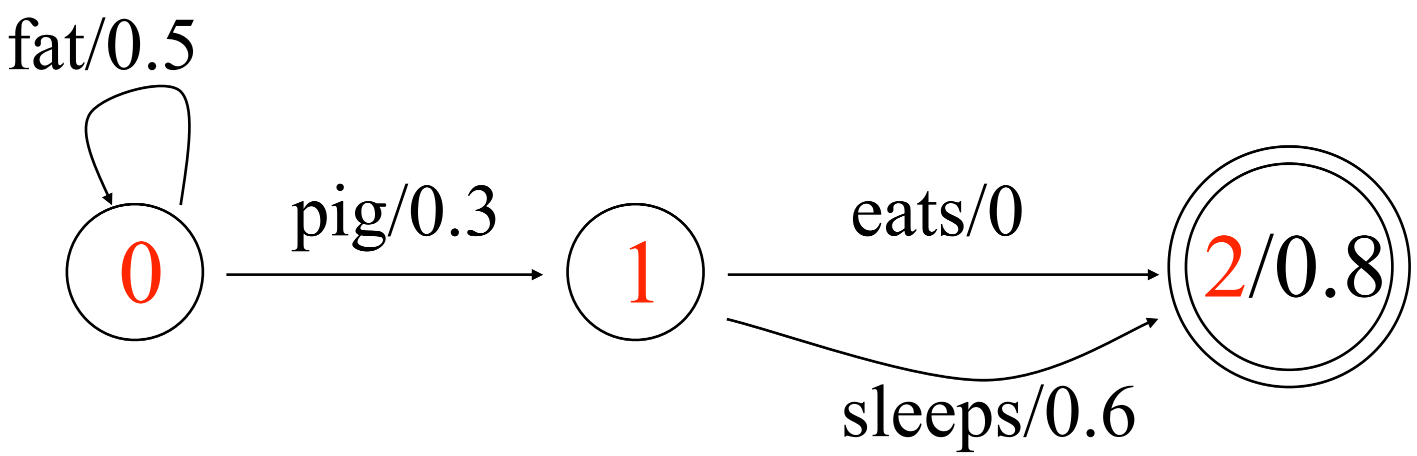
=



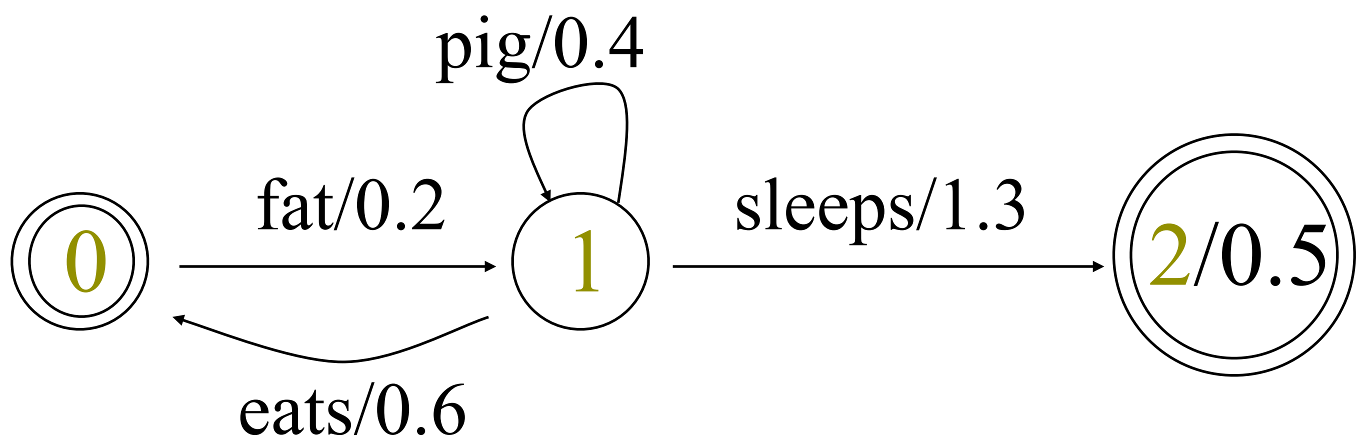
Complementation

- Given a machine M, represent all strings not accepted by M
- Just change final states to non-final and vice-versa
- Works only if machine has been determinized and completed first

Intersection



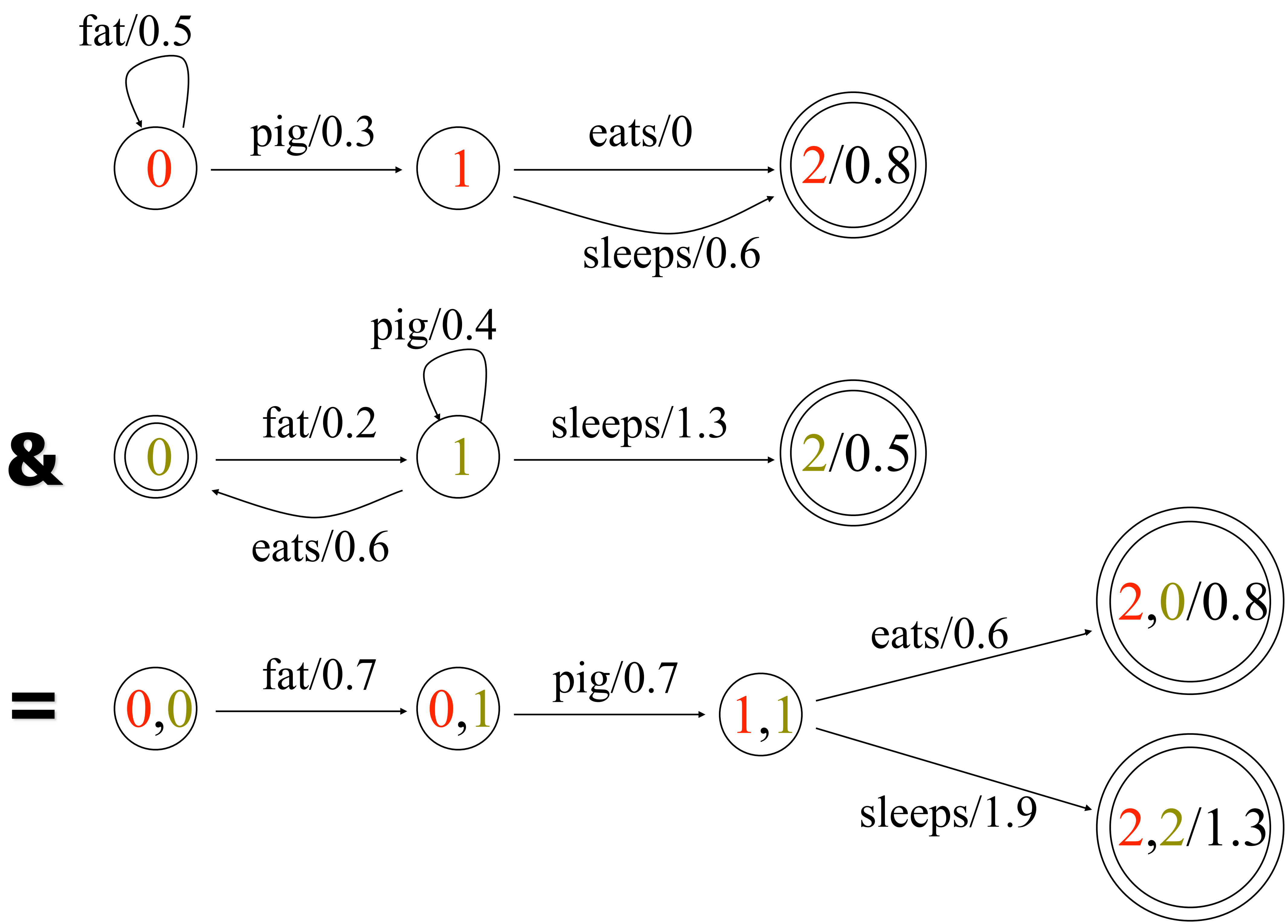
&



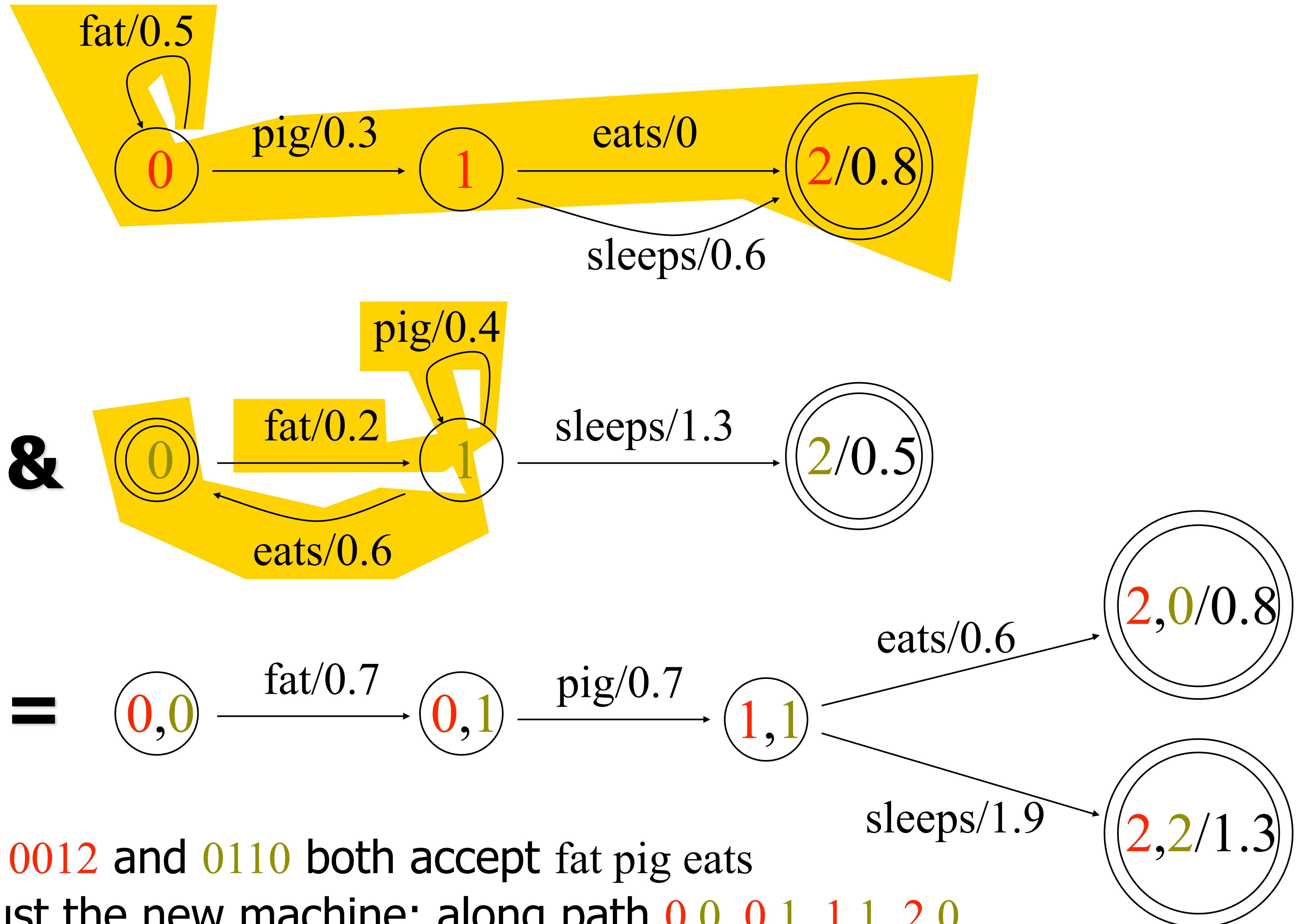
=

example adapted from M. Mohri

Intersection



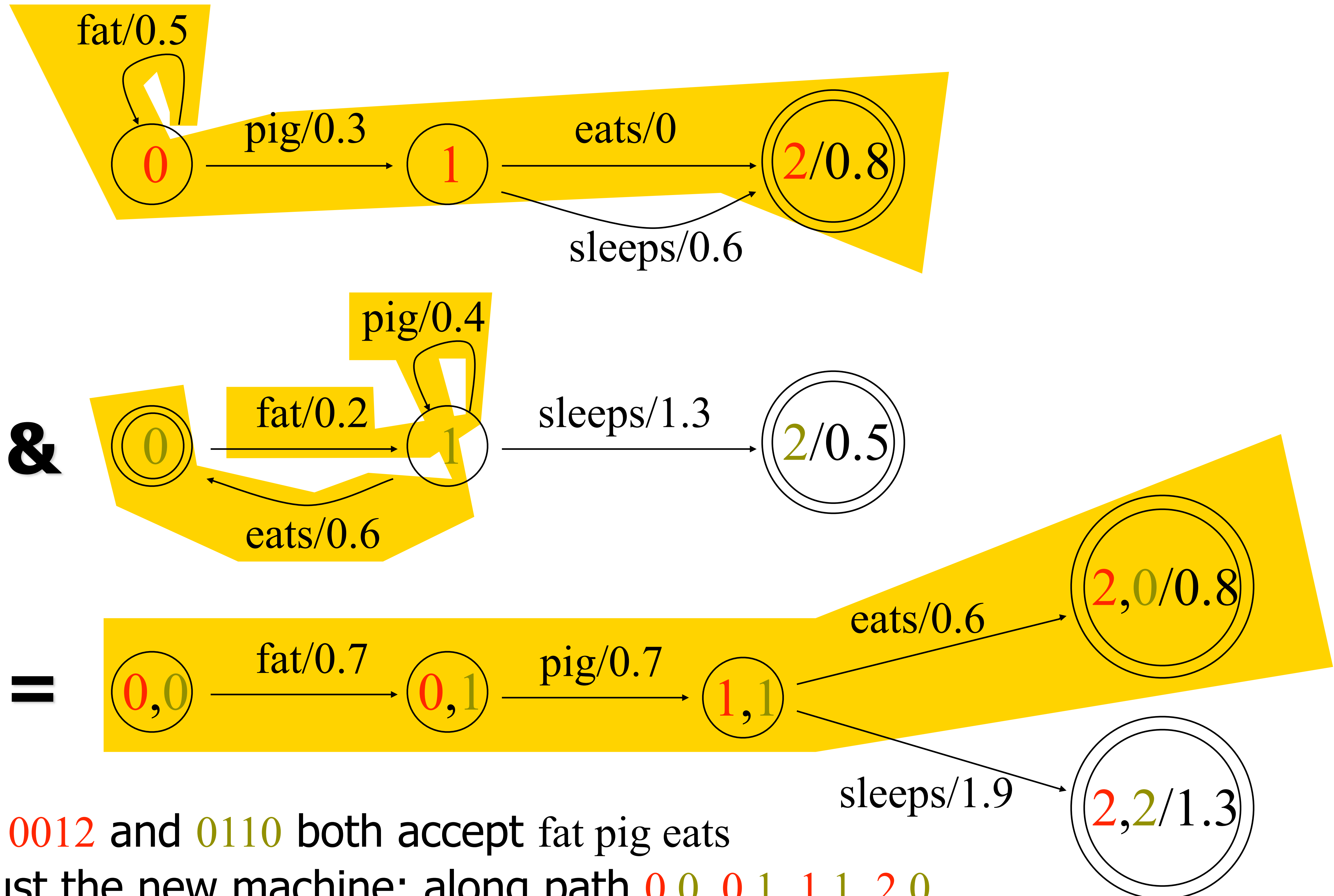
Intersection



Paths 0012 and 0110 both accept fat pig eats

So must the new machine: along path 0,0 0,1 1,1 2,0

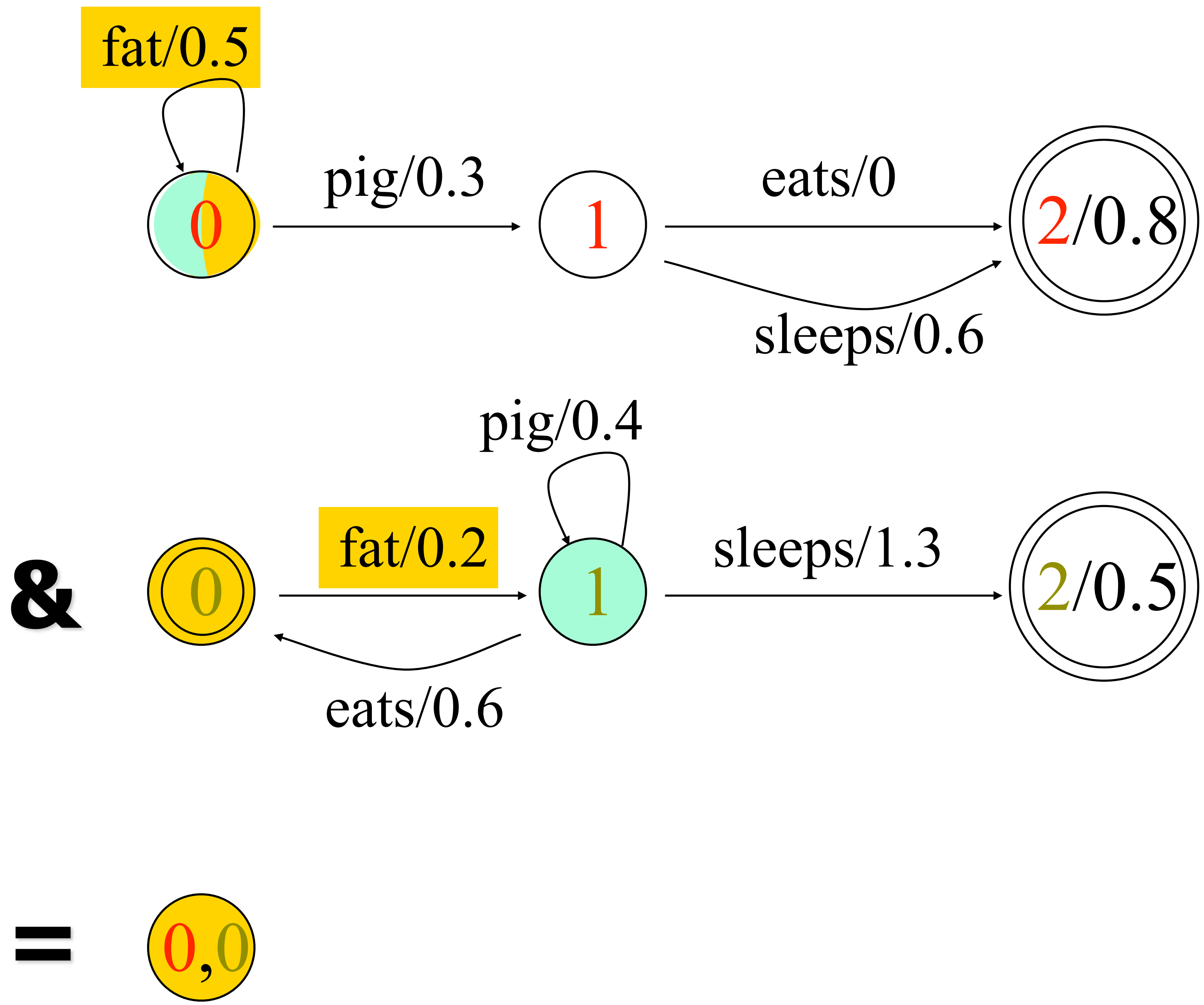
Intersection



Paths 0012 and 0110 both accept fat pig eats

So must the new machine: along path 0,0 0,1 1,1 2,0

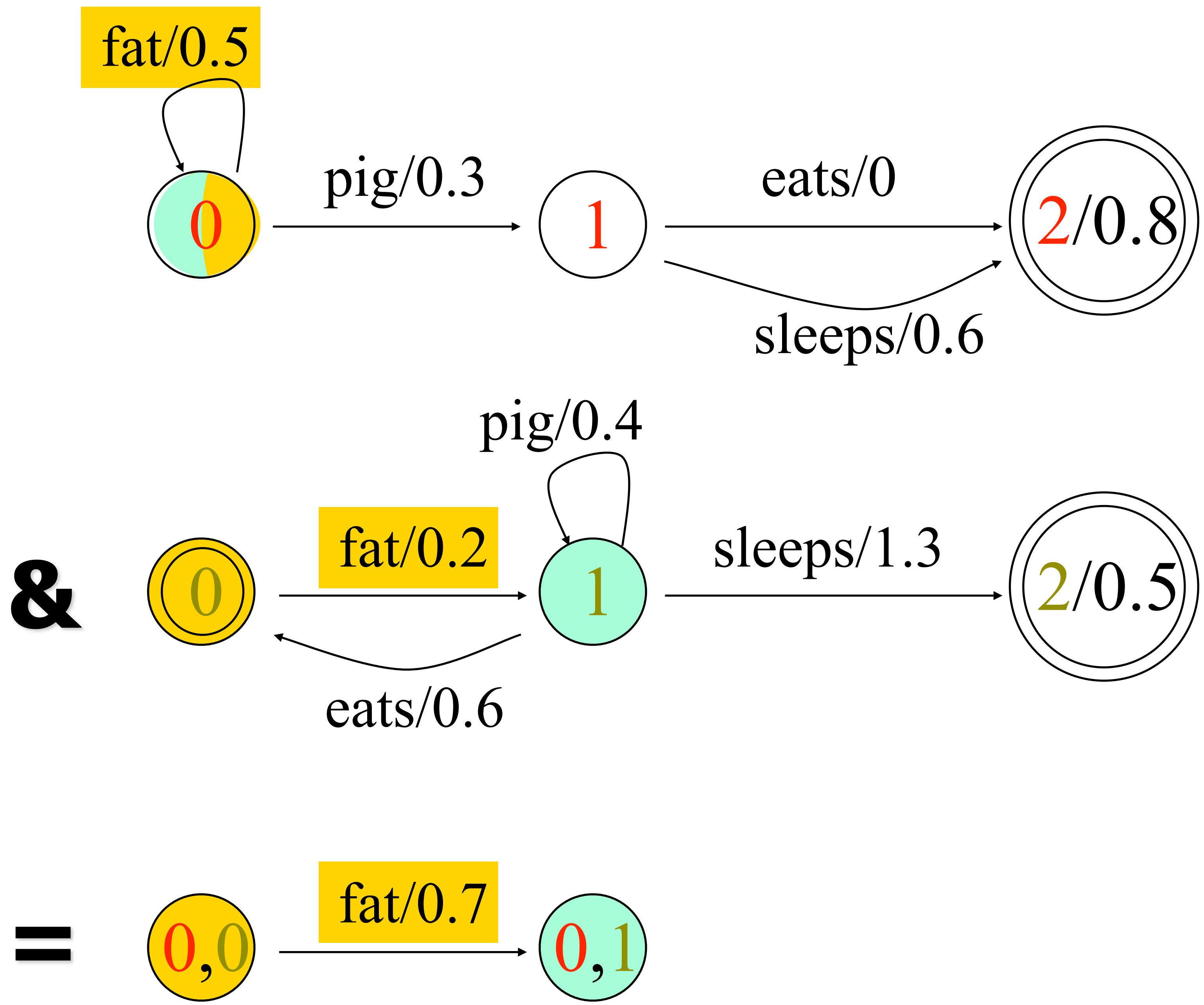
Intersection



Paths 00 and 01 both accept fat

So must the new machine: along path 0,0 0,1

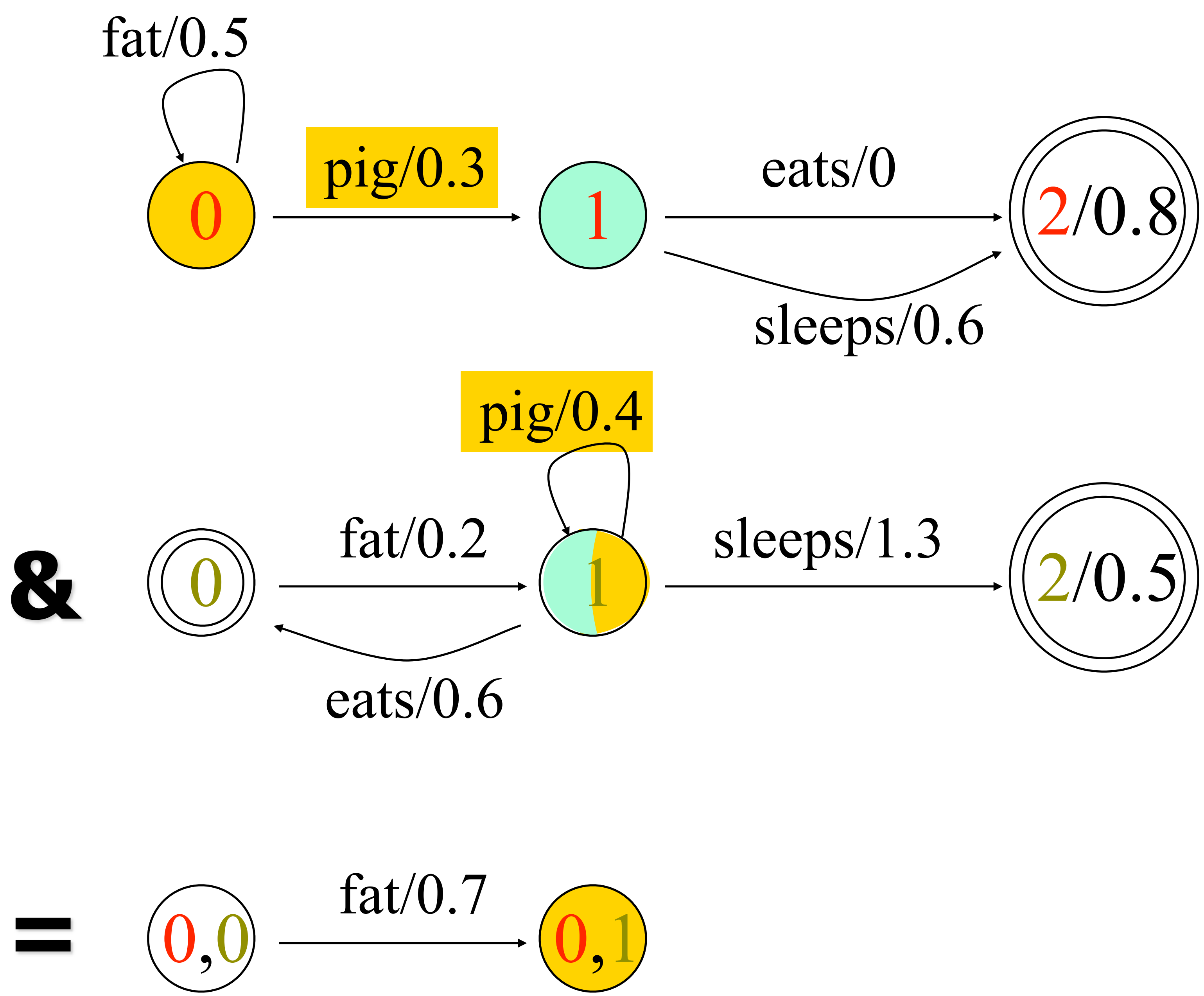
Intersection



Paths 00 and 01 both accept fat

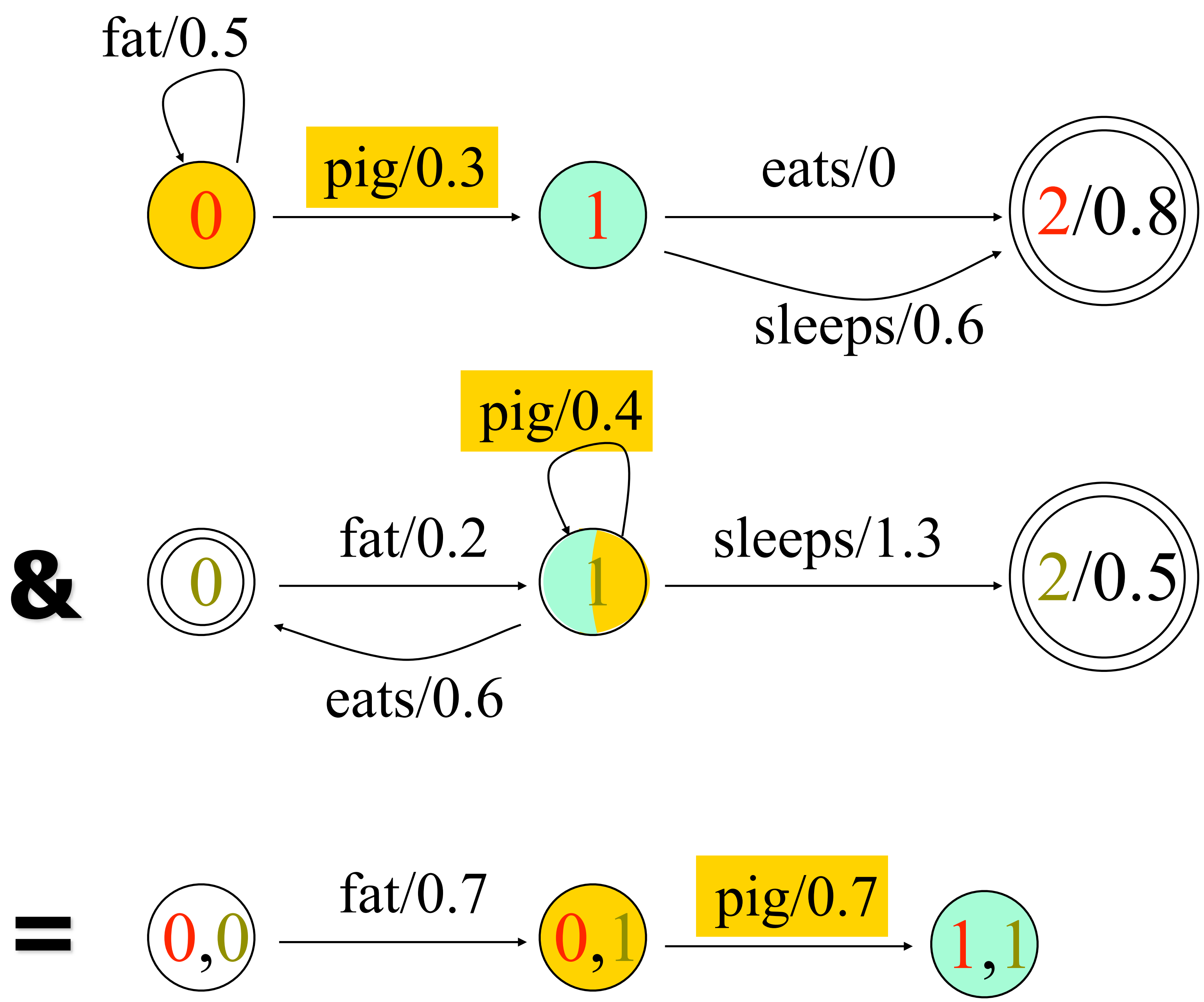
So must the new machine: along path 0,0 0,1

Intersection



Paths 00 and 11 both accept pig
So must the new machine: along path 0,1 1,1

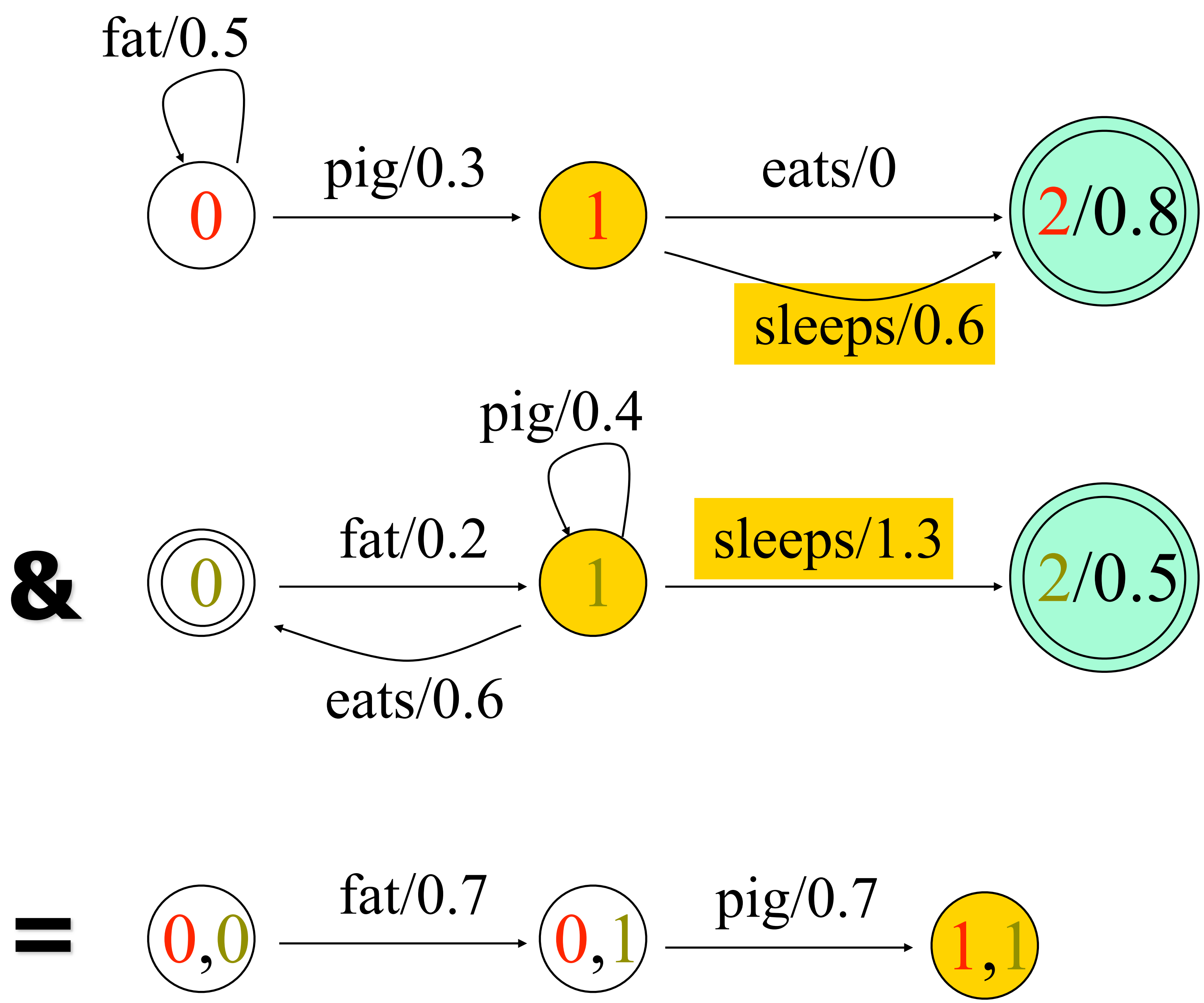
Intersection



Paths 00 and 11 both accept pig

So must the new machine: along path 0,1 1,1

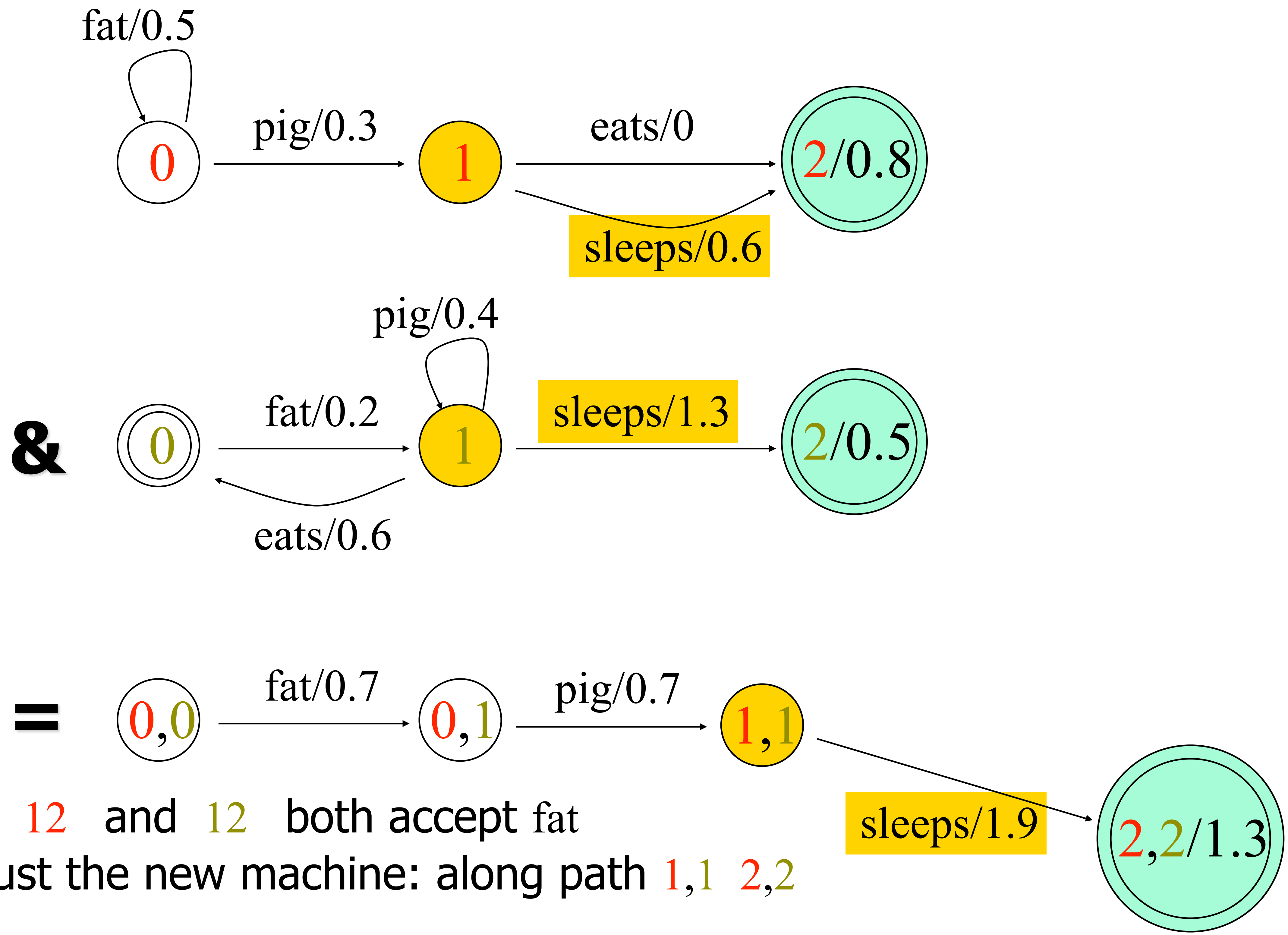
Intersection



Paths 12 and 12 both accept fat

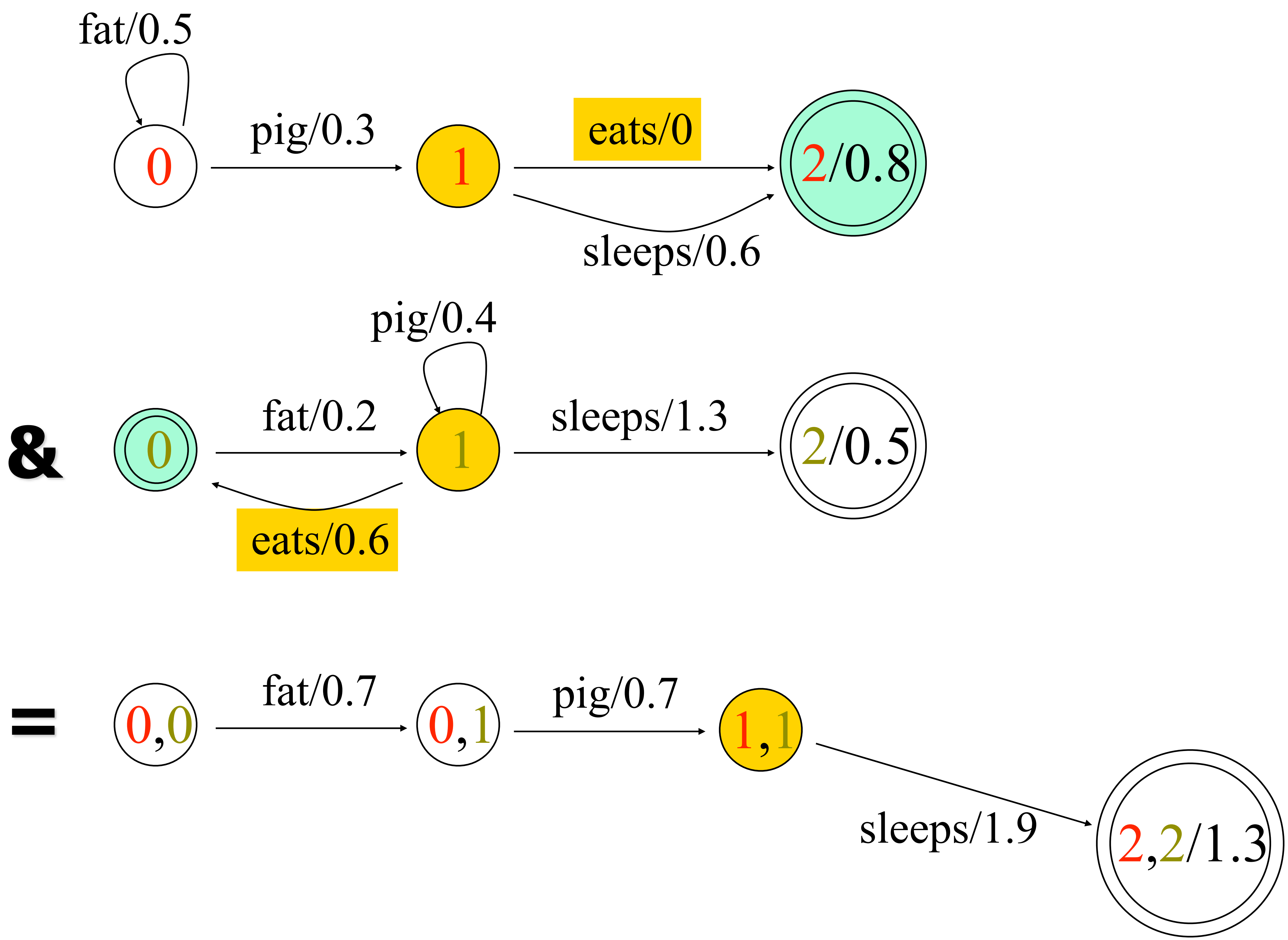
So must the new machine: along path 1,1 2,2

Intersection

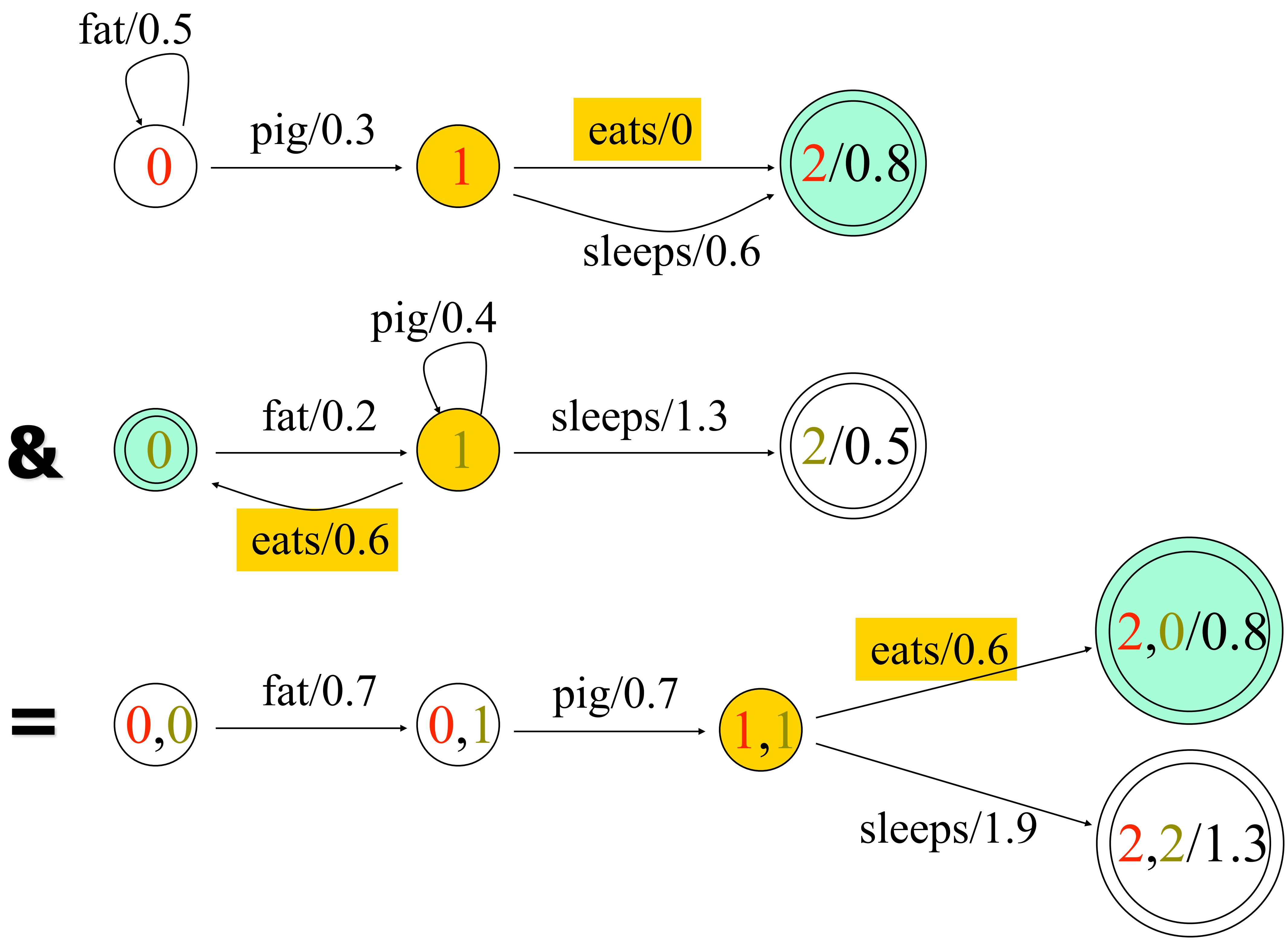


Paths 12 and 12 both accept fat
So must the new machine: along path 1,1 2,2

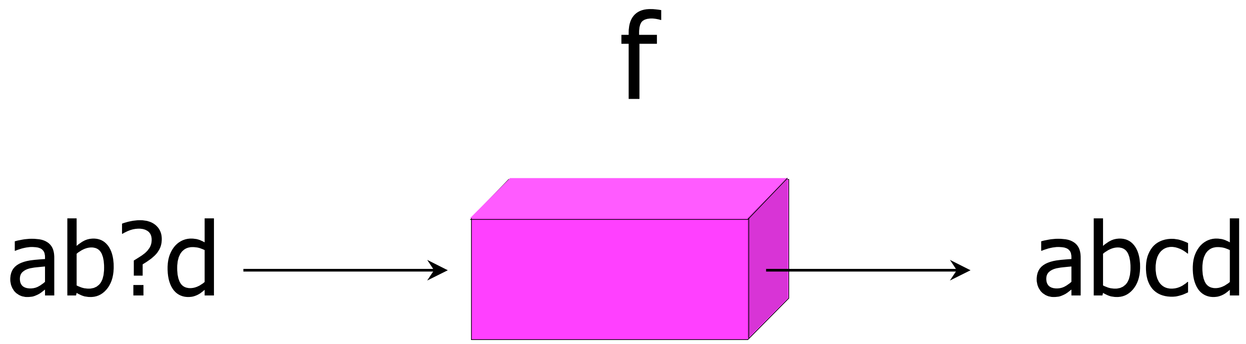
Intersection



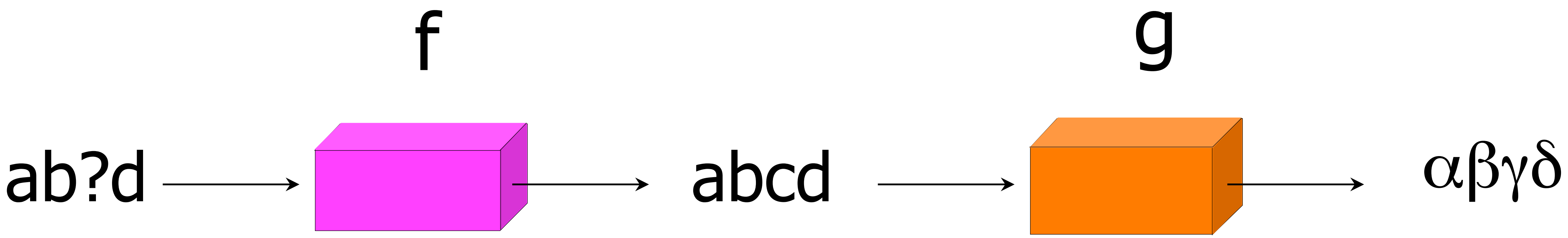
Intersection



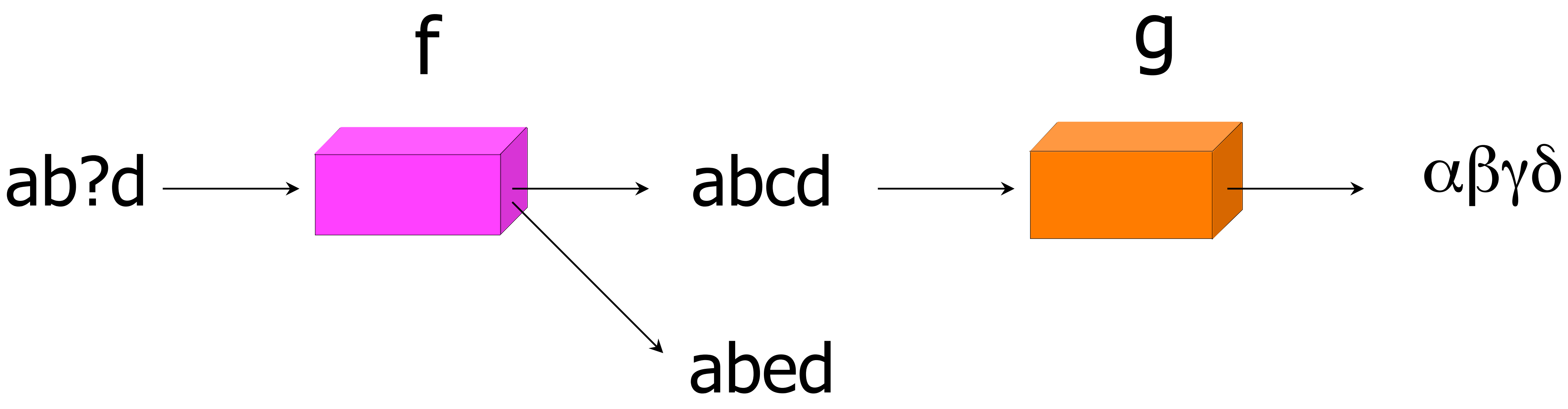
What Composition Means



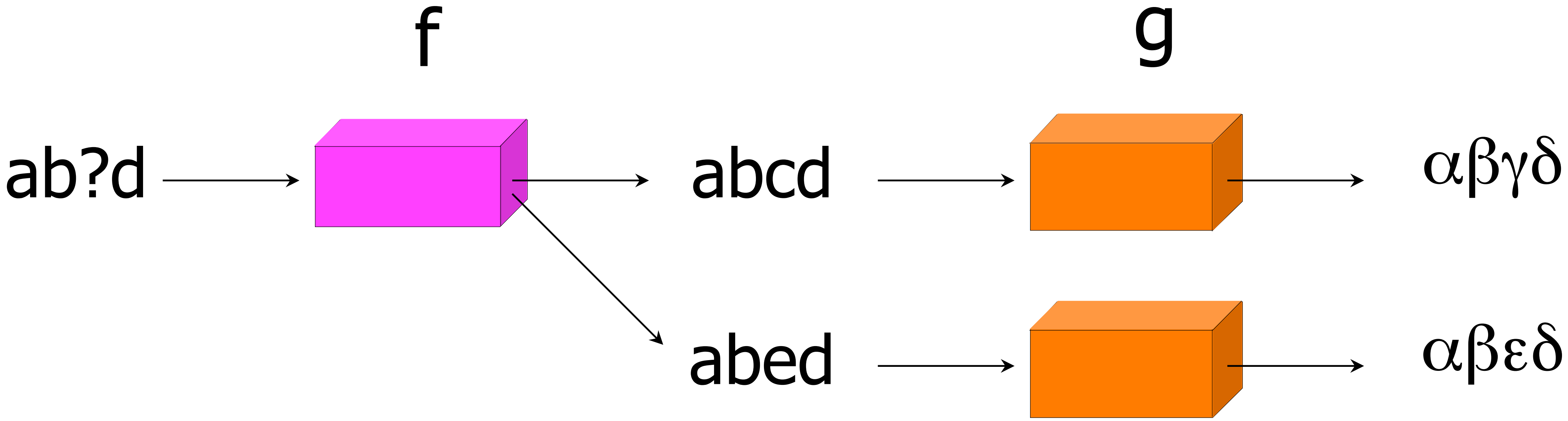
What Composition Means



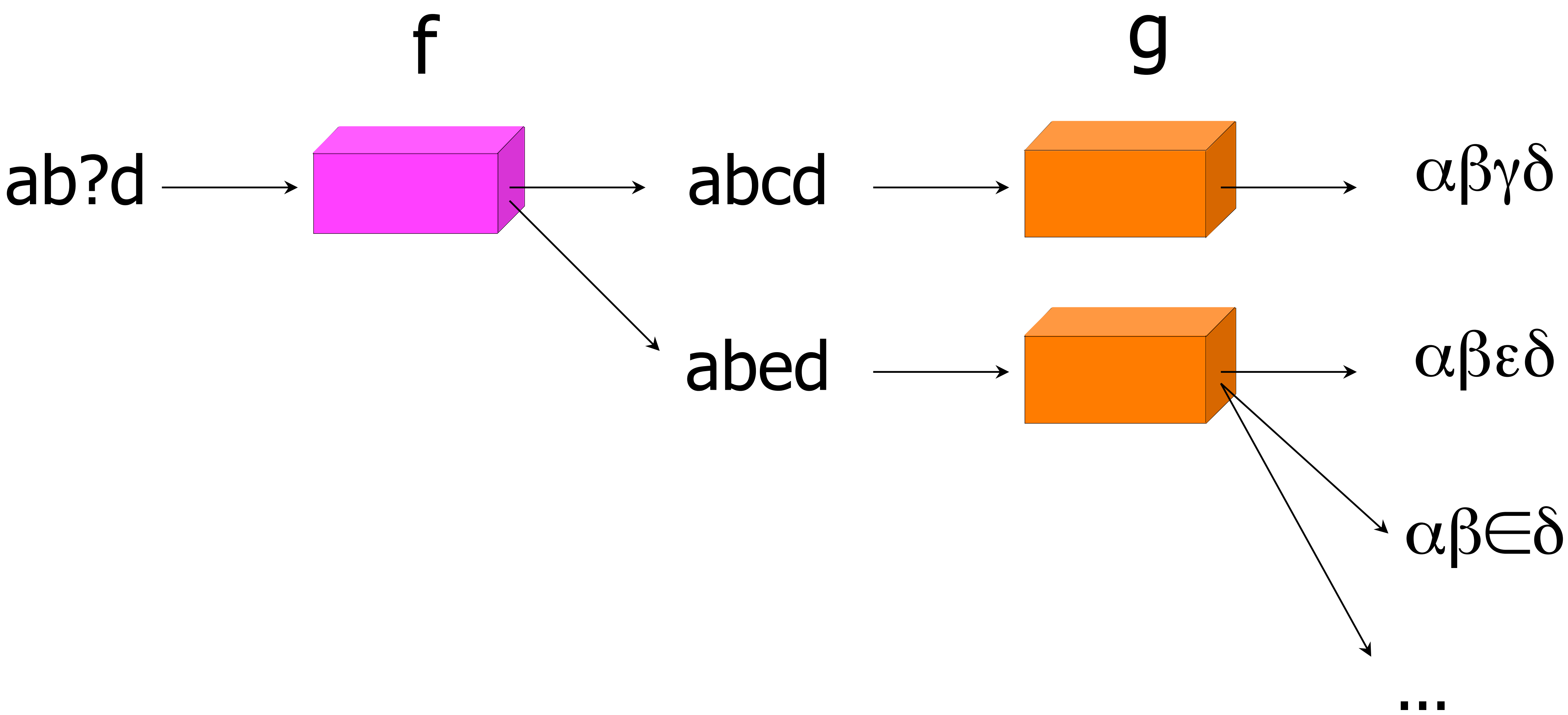
What Composition Means



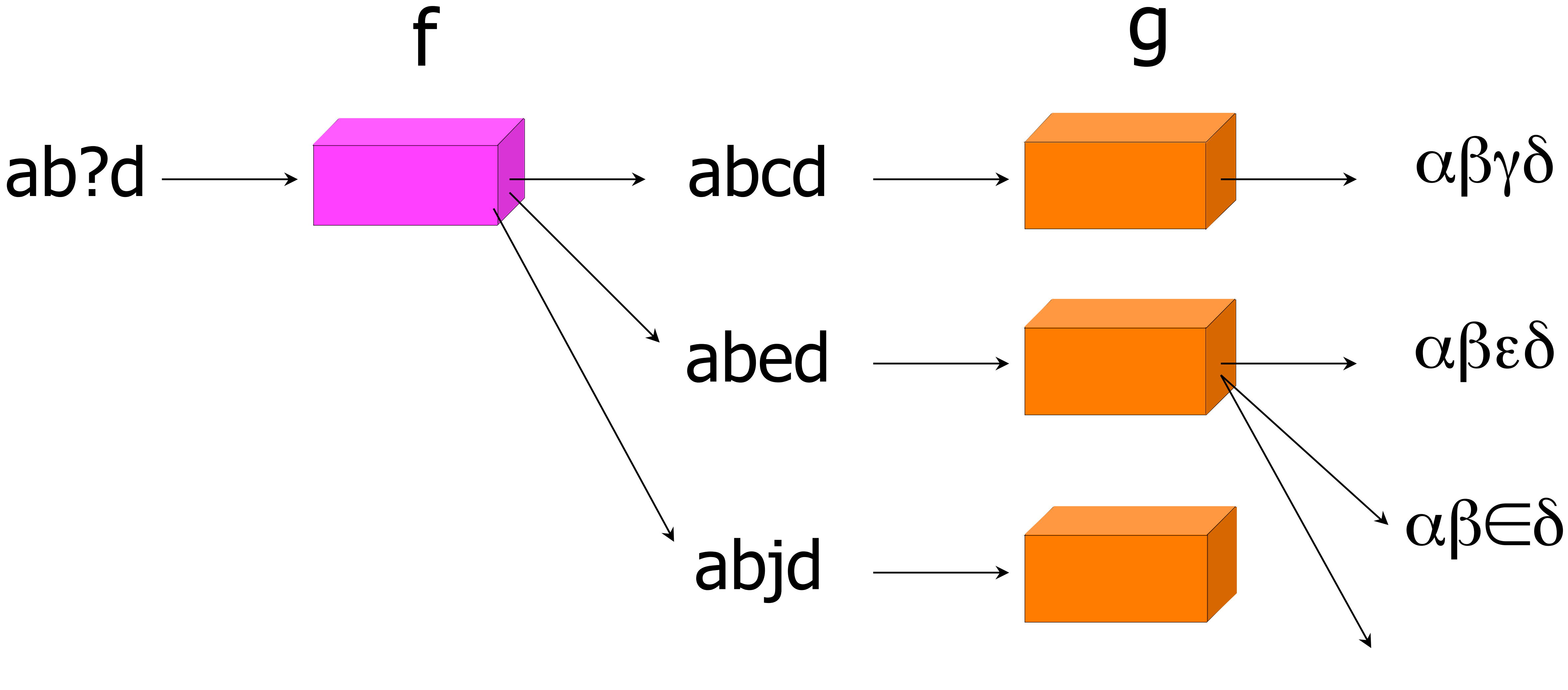
What Composition Means



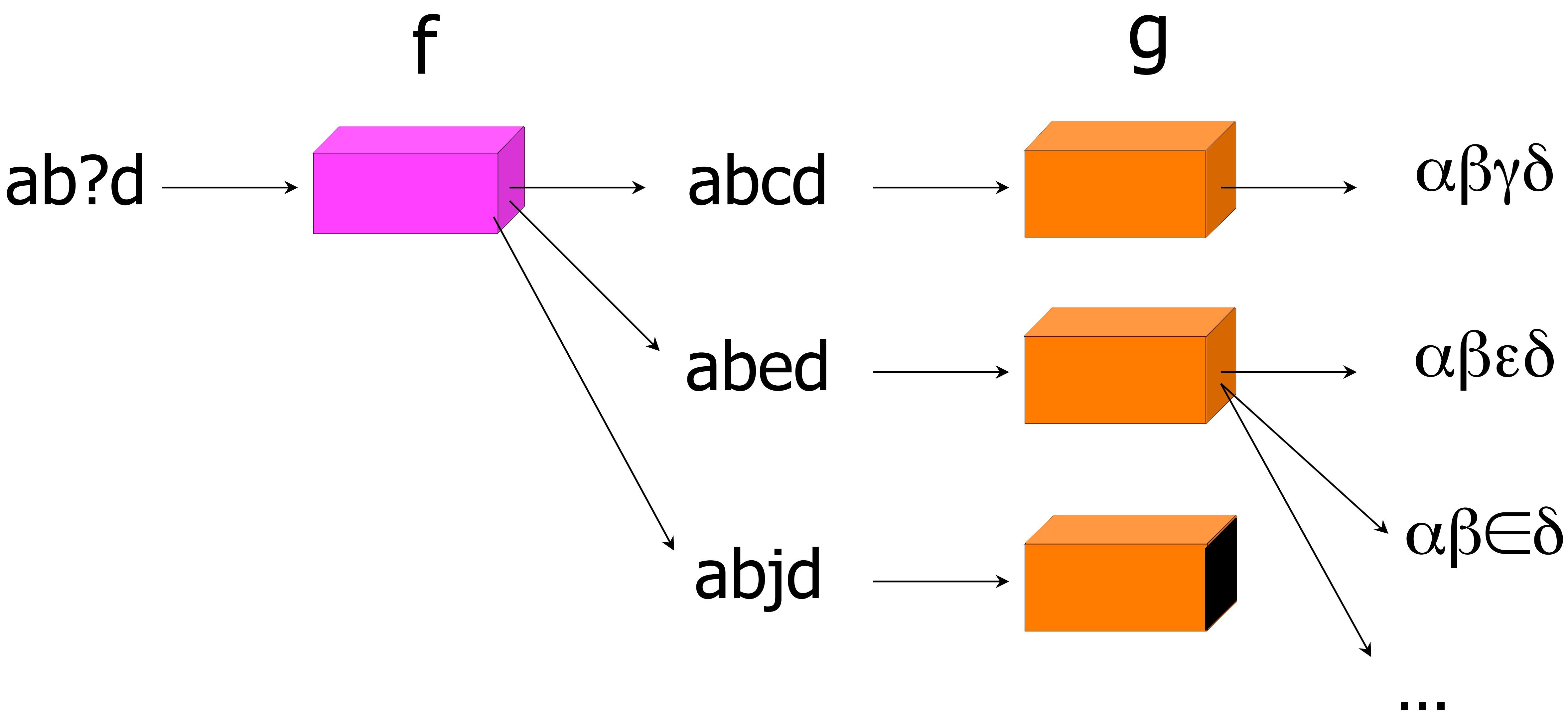
What Composition Means



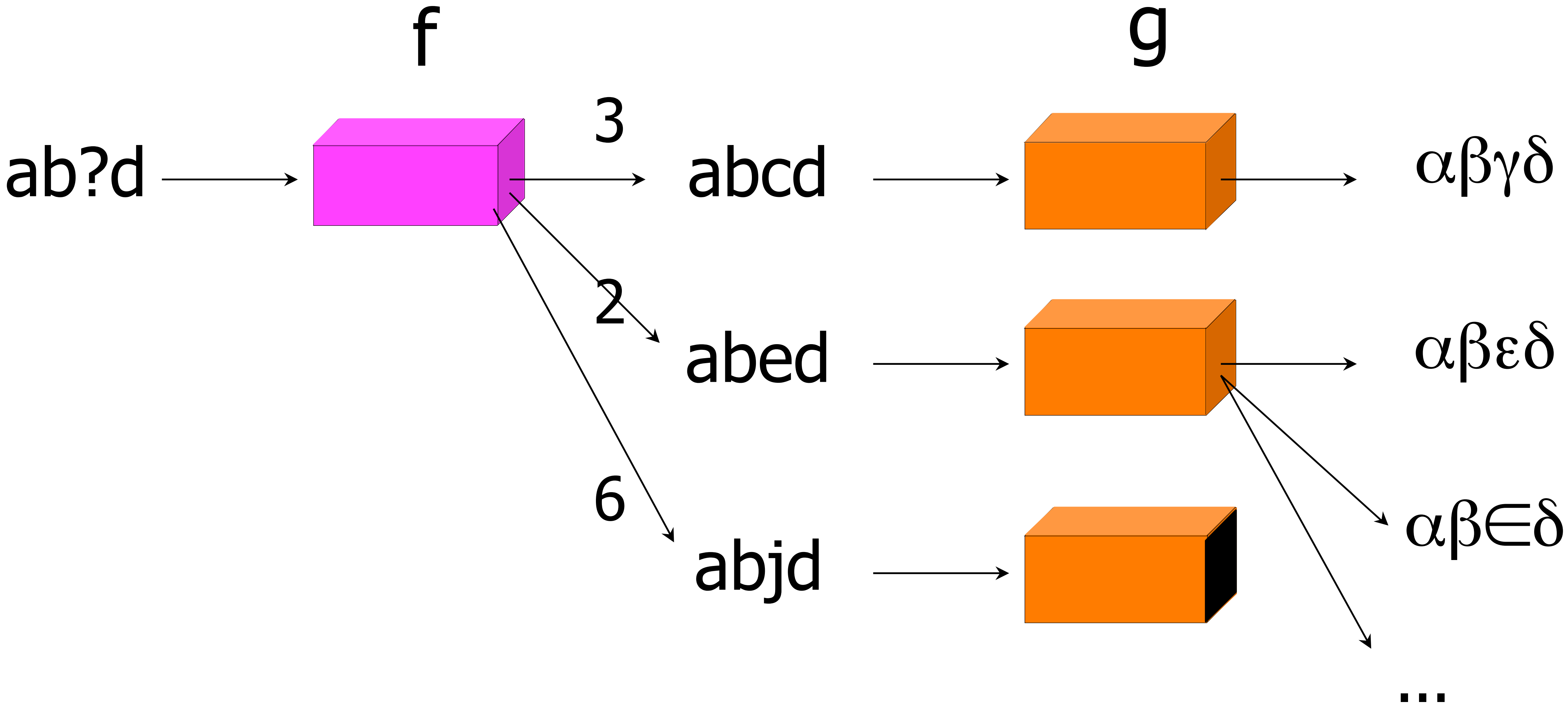
What Composition Means



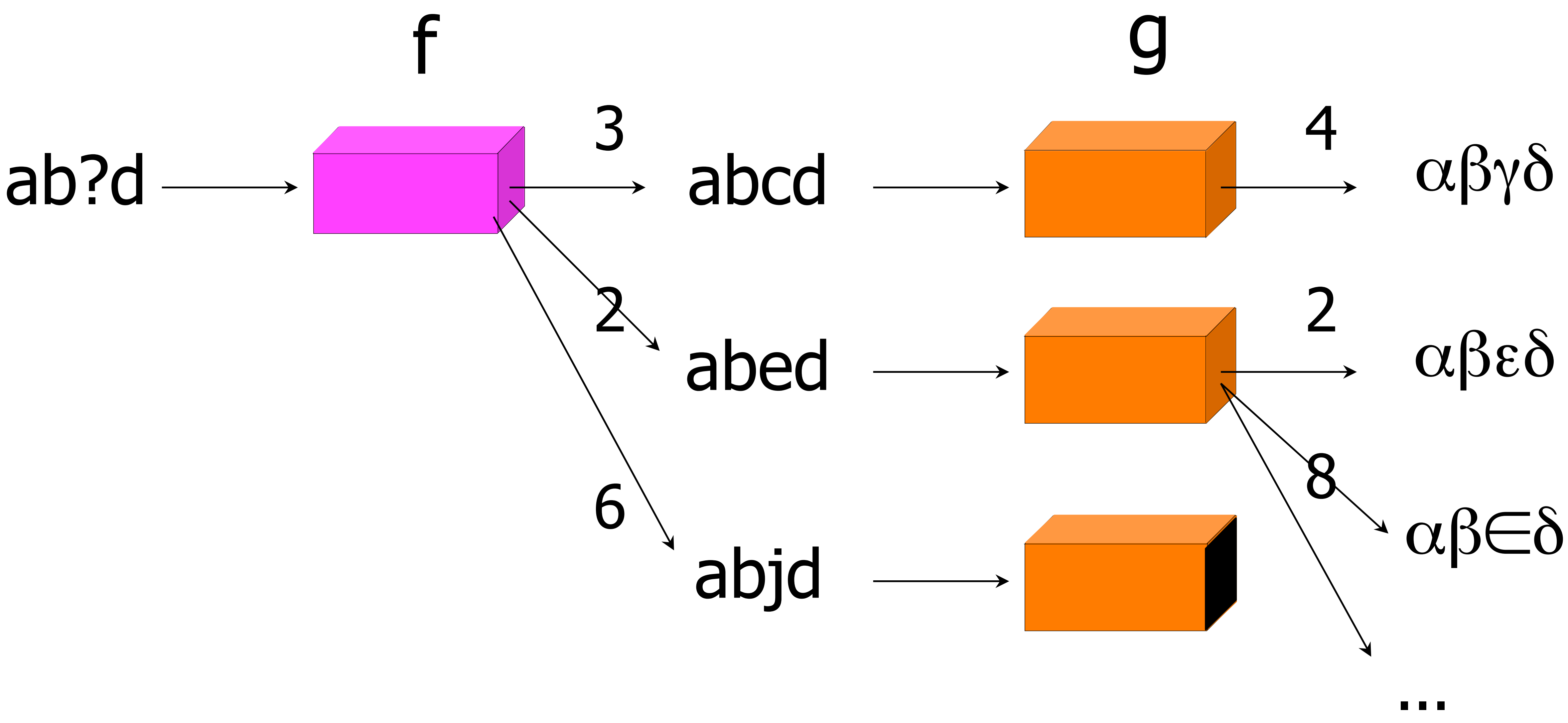
What Composition Means



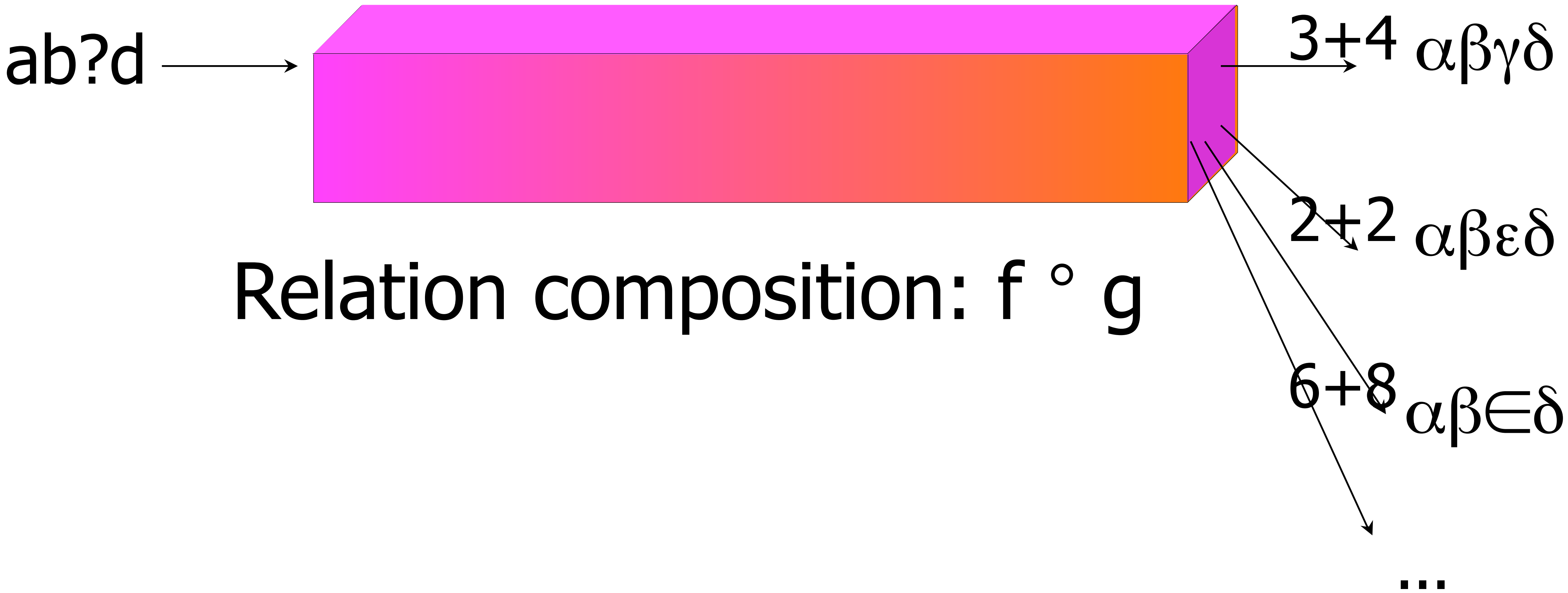
What Composition Means



What Composition Means



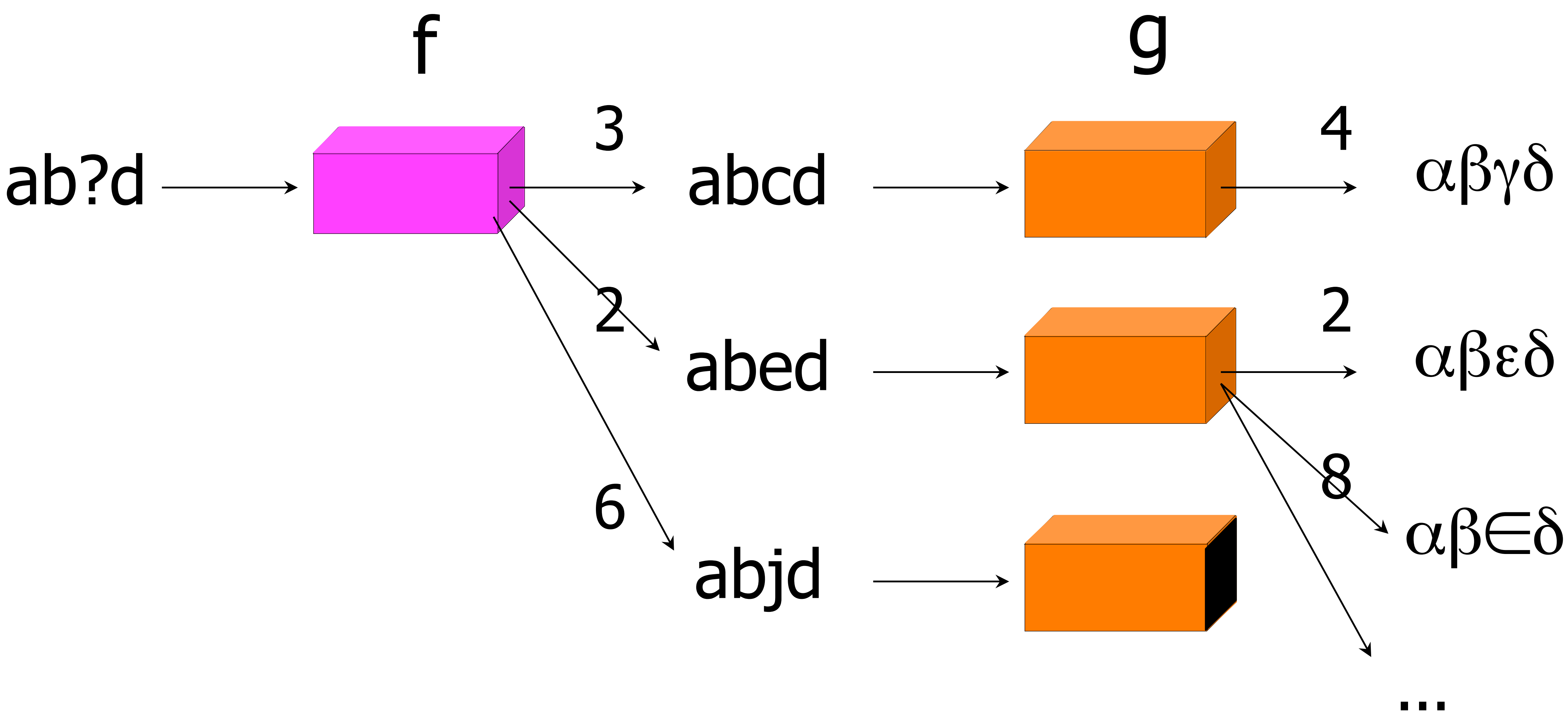
What Composition Means



Relation composition: f ° g

Relation = set of pairs

$$\left\{ \begin{array}{l} ab?d \to abcd \\ ab?d \to abed \\ ab?d \to abjd \\ \quad \ldots \end{array} \right\} \qquad \left\{ \begin{array}{l} abcd \to \alpha\beta\gamma\delta \\ abed \to \alpha\beta\varepsilon\delta \\ abed \to \alpha\beta{\in}\delta \\ \quad \ldots \end{array} \right\}$$

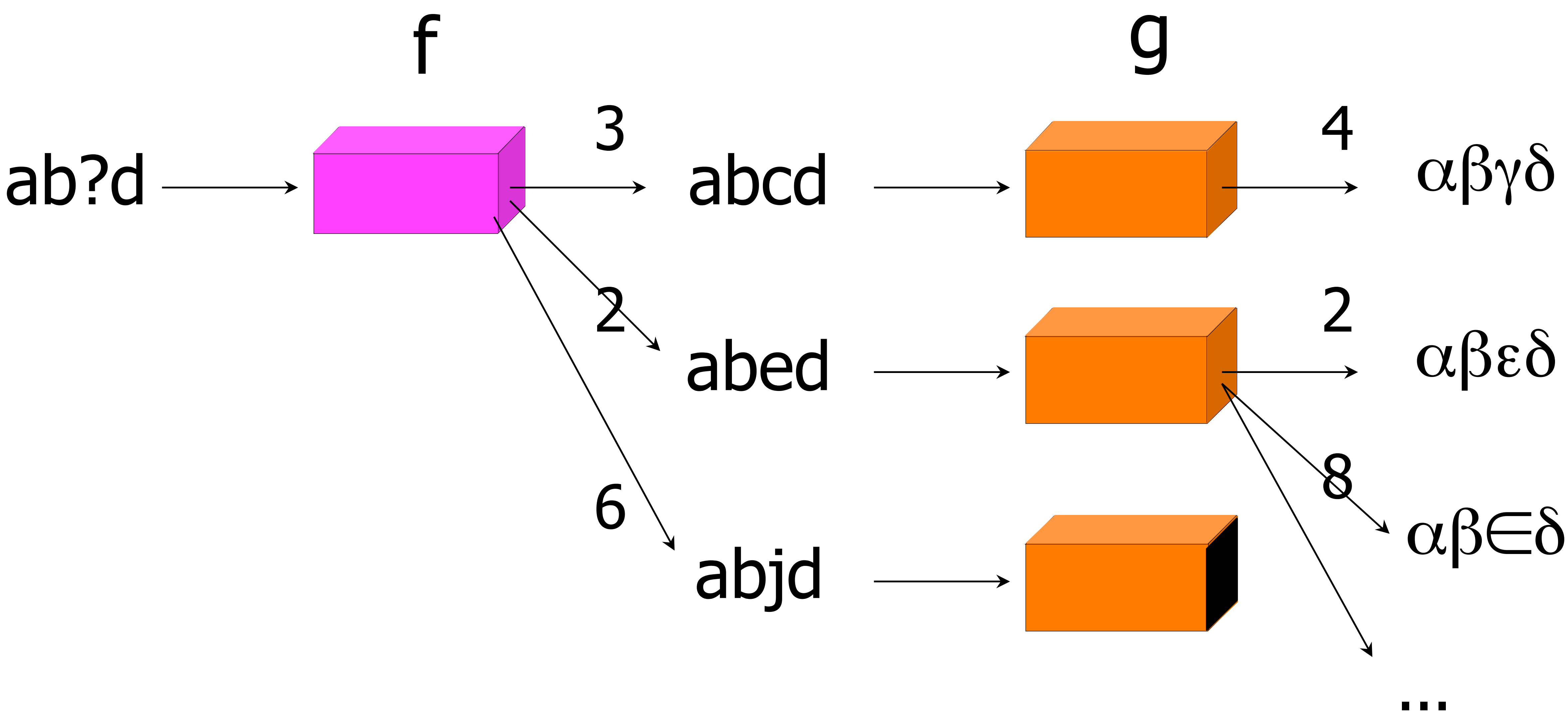


Relation = set of pairs

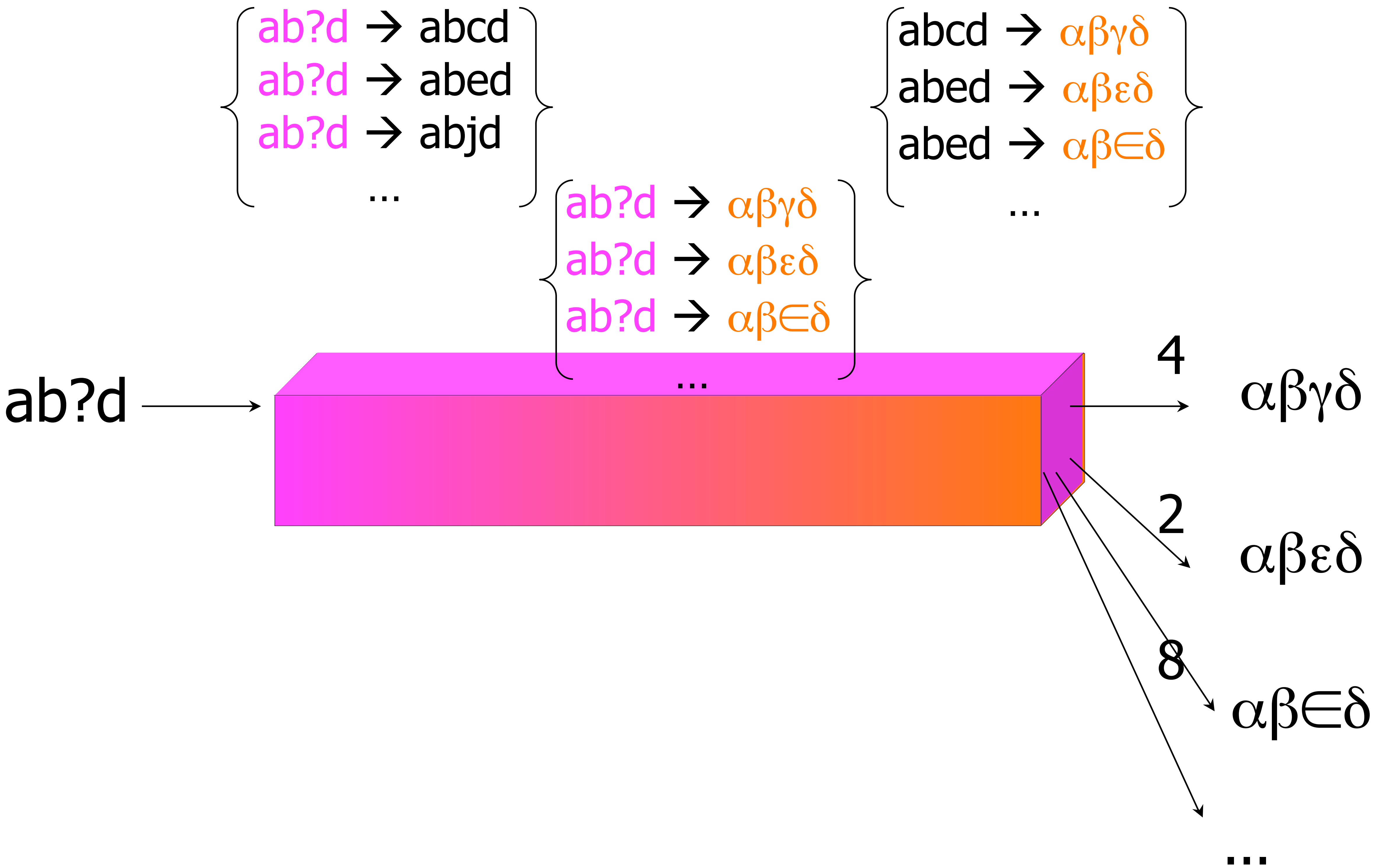
does not contain any pair of the form abjd → …

$$\left\{\begin{array}{l} \text{ab?d} \rightarrow \text{abcd} \\ \text{ab?d} \rightarrow \text{abed} \\ \text{ab?d} \rightarrow \text{abjd} \\ \quad\ldots \end{array}\right\}$$

$$\left\{\begin{array}{l} \text{abcd} \rightarrow \alpha\beta\gamma\delta \\ \text{abed} \rightarrow \alpha\beta\varepsilon\delta \\ \text{abed} \rightarrow \alpha\beta\in\delta \\ \quad\ldots \end{array}\right\}$$



Relation = set of pairs



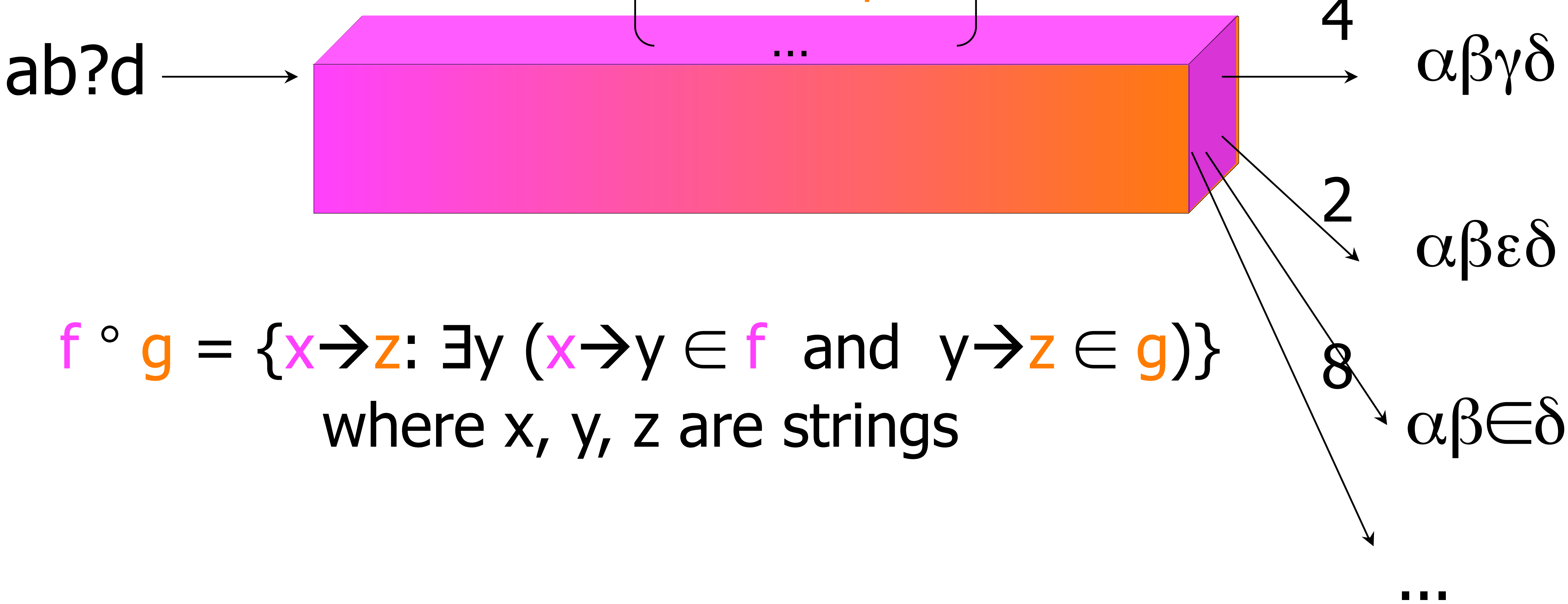
Relation = set of pairs

$$\left\{\begin{array}{l} ab?d \rightarrow abcd \\ ab?d \rightarrow abed \\ ab?d \rightarrow abjd \\ \quad\ldots \end{array}\right\}$$

$$\left\{\begin{array}{l} abcd \rightarrow \alpha\beta\gamma\delta \\ abed \rightarrow \alpha\beta\varepsilon\delta \\ abed \rightarrow \alpha\beta\in\delta \\ \quad\ldots \end{array}\right\}$$

$f \circ g$

$$\left\{\begin{array}{l} ab?d \rightarrow \alpha\beta\gamma\delta \\ ab?d \rightarrow \alpha\beta\varepsilon\delta \\ ab?d \rightarrow \alpha\beta\in\delta \\ \quad\ldots \end{array}\right\}$$

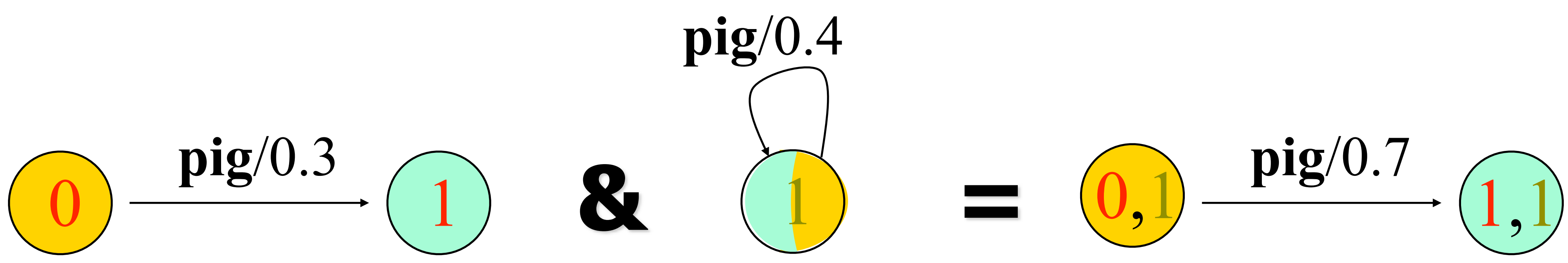


$f \circ g = \{x \rightarrow z : \exists y\ (x \rightarrow y \in f \text{ and } y \rightarrow z \in g)\}$

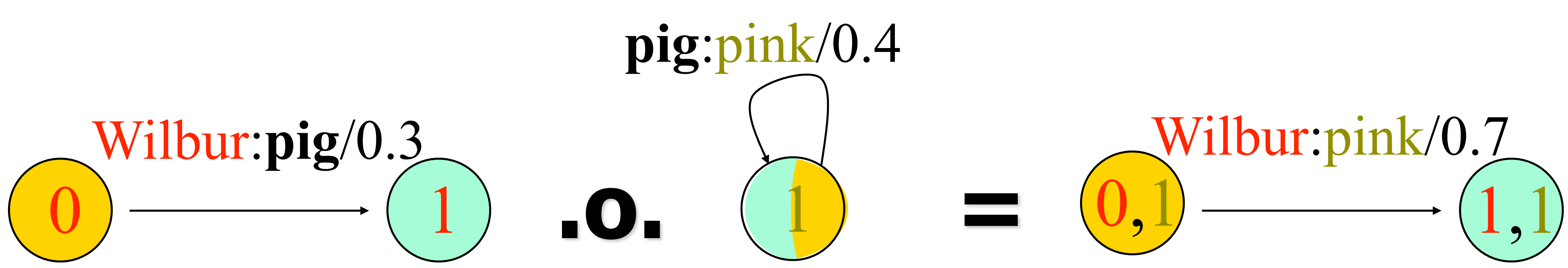
where x, y, z are strings

Intersection vs. Composition

Intersection

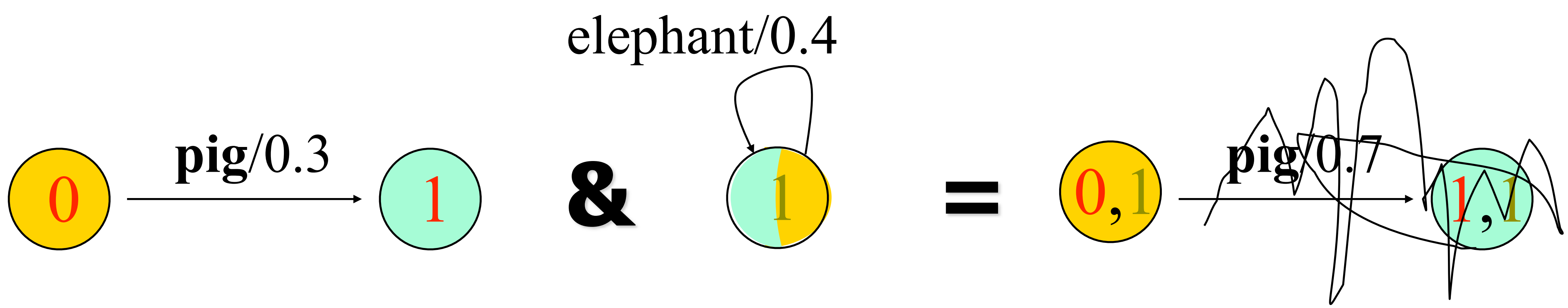


Composition

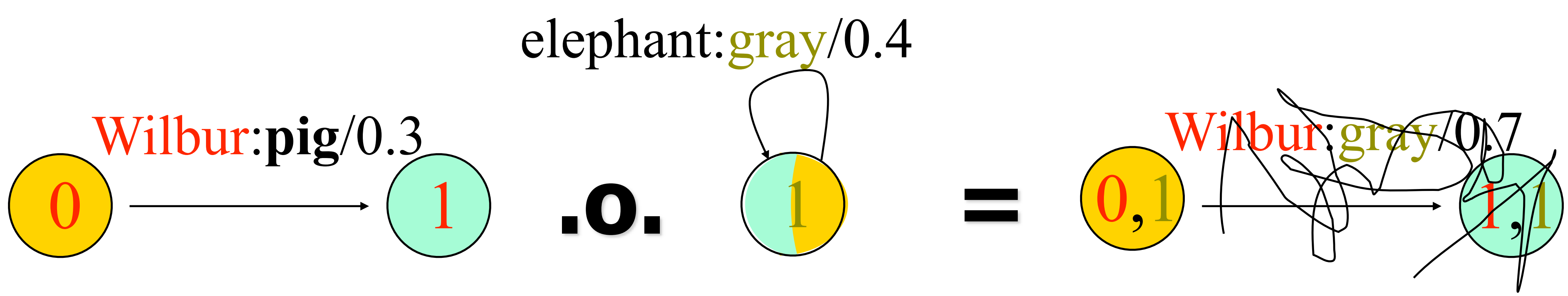


Intersection vs. Composition

Intersection mismatch

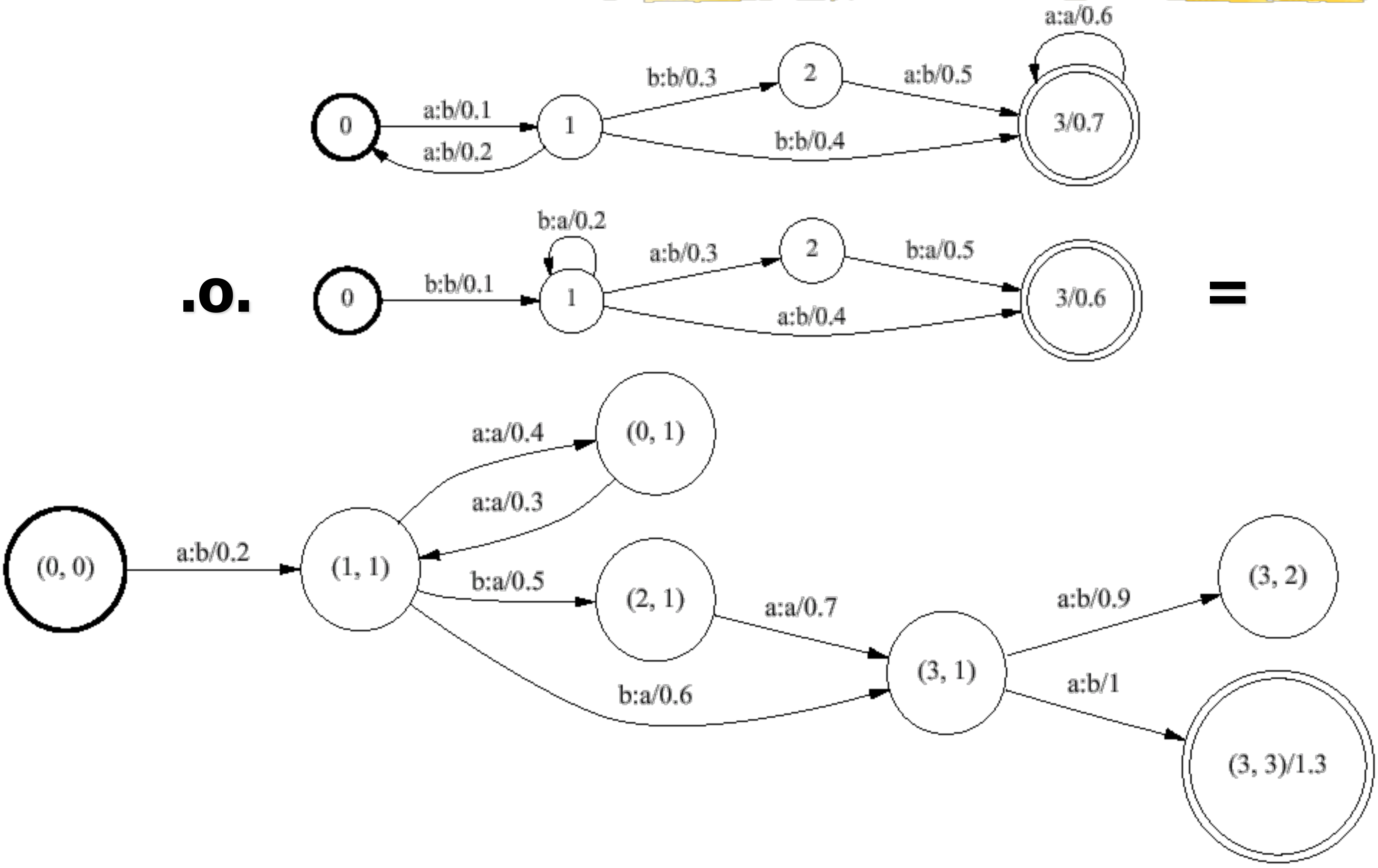


Composition mismatch

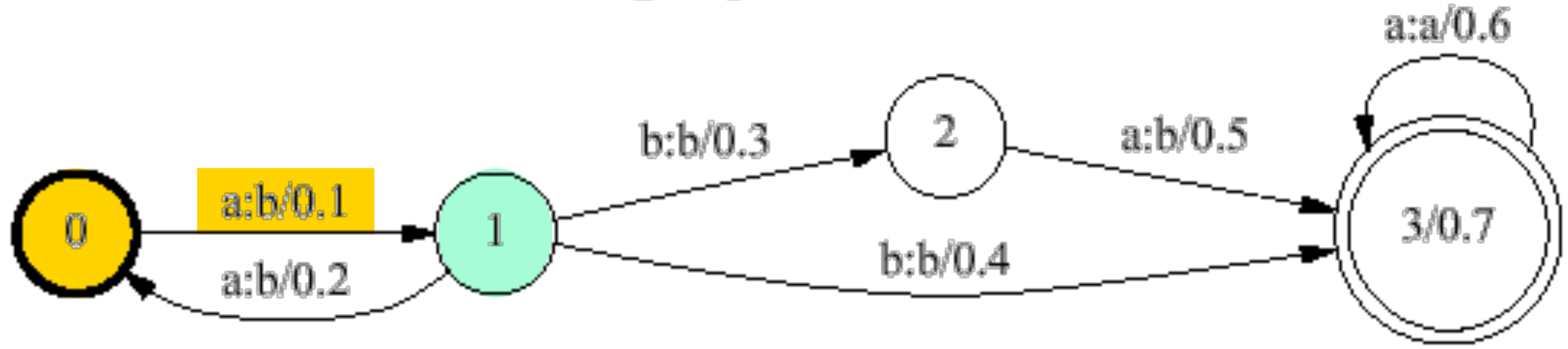


Composition

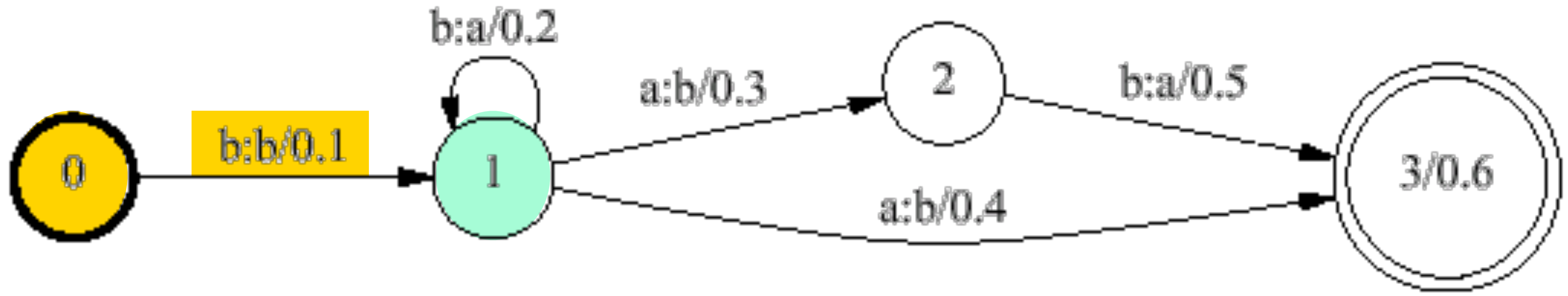
example courtesy of M. Mohri



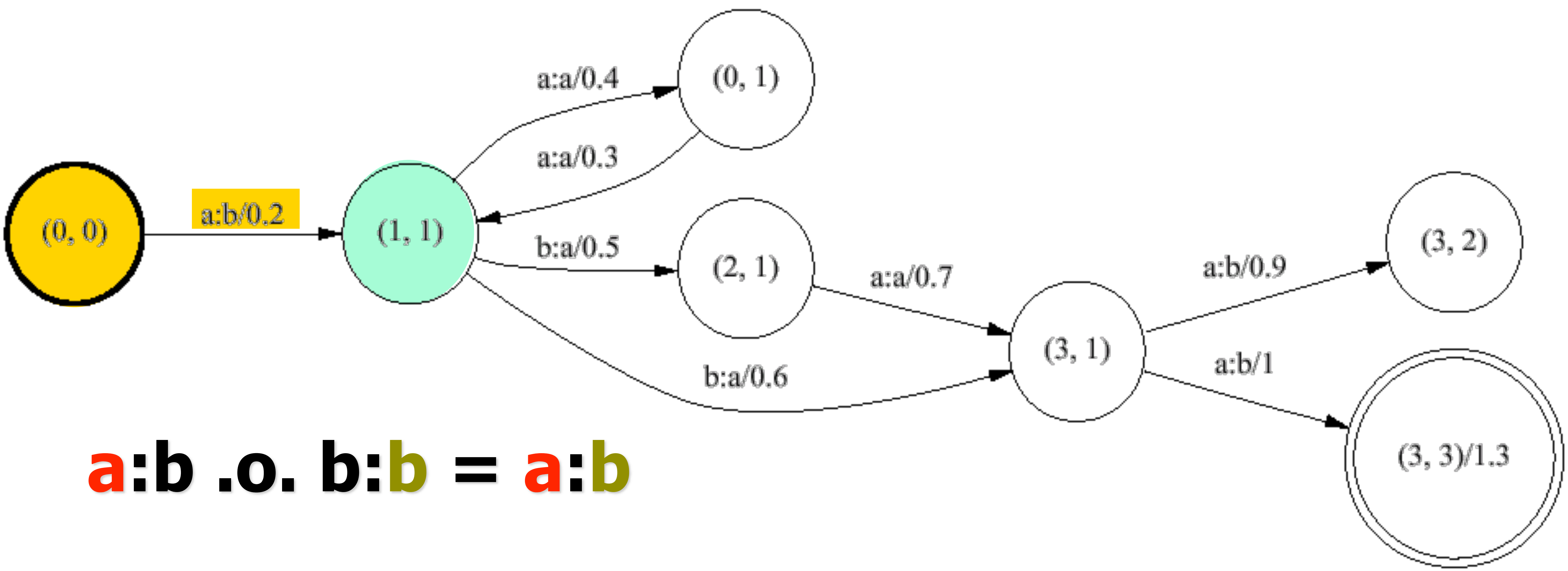
Composition



.o.

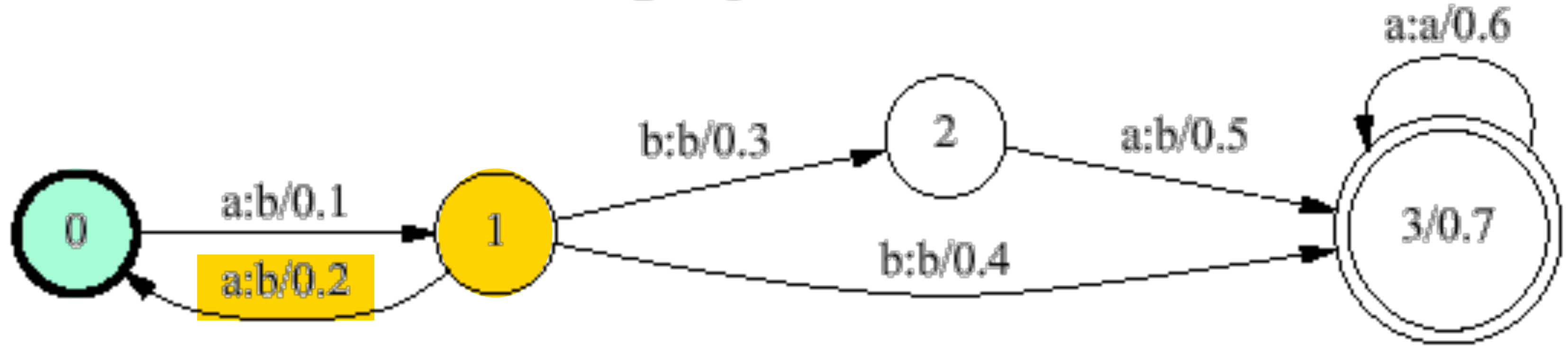


=

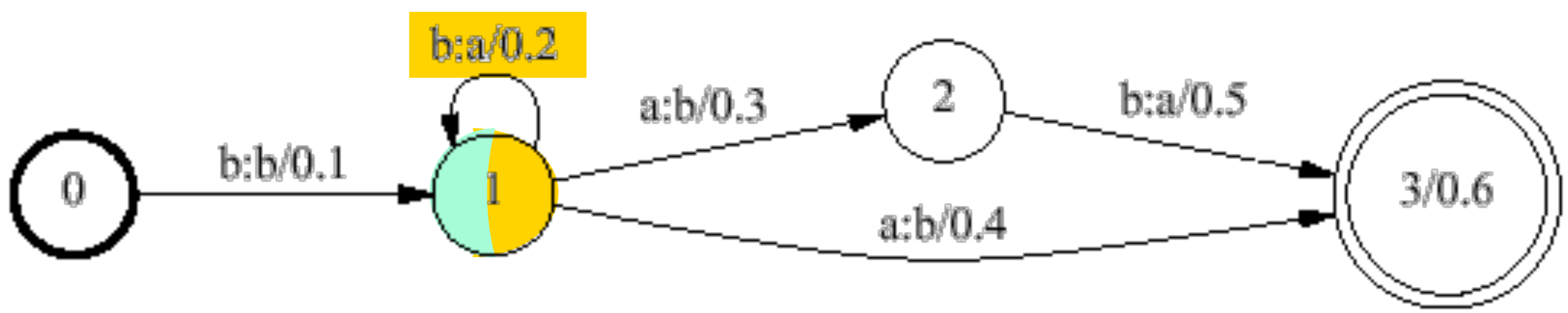


a:b .o. b:b = a:b

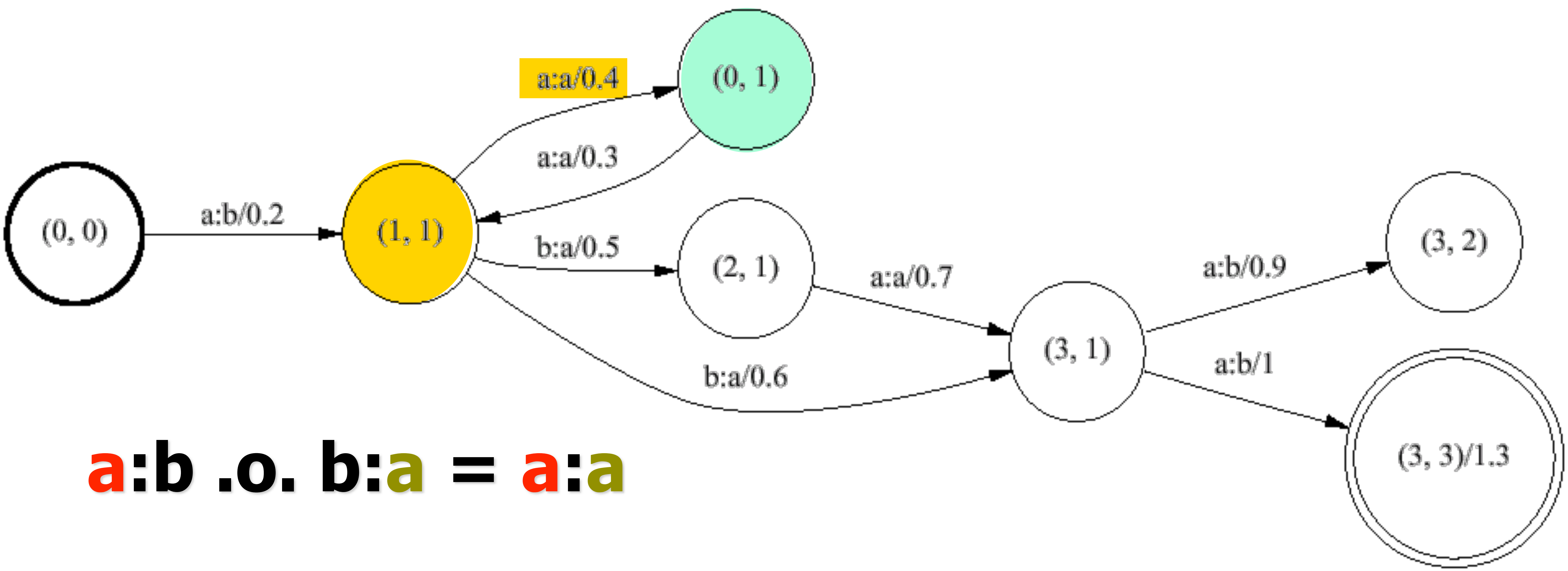
Composition



.o.

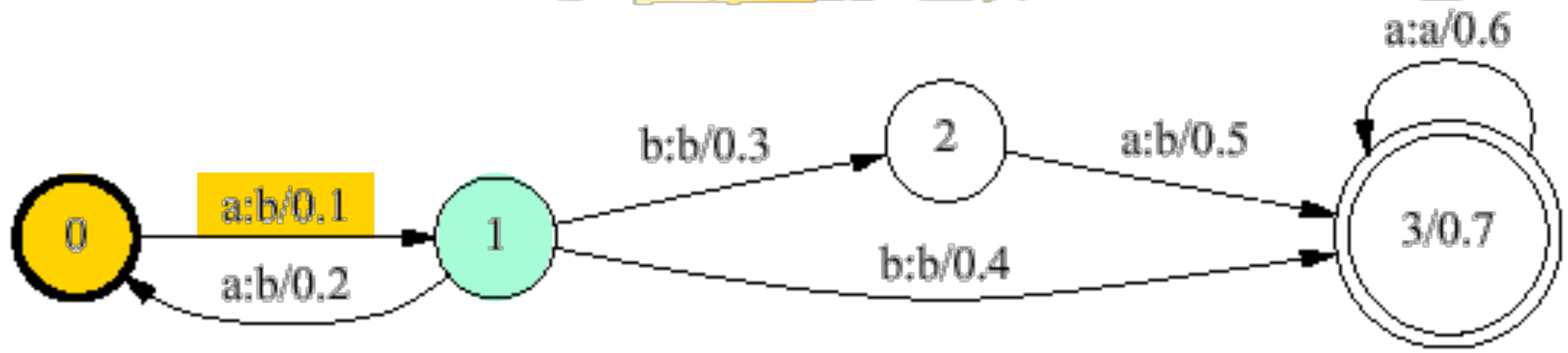


=

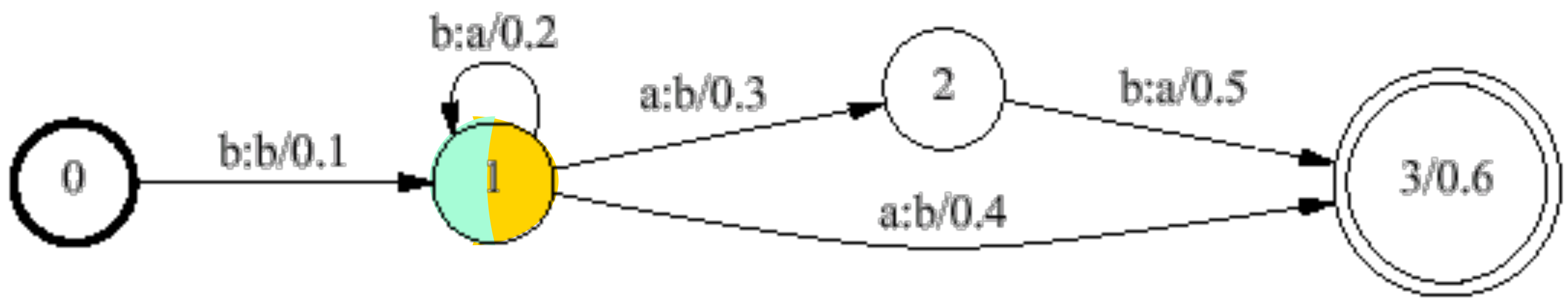


a:b .o. b:a = a:a

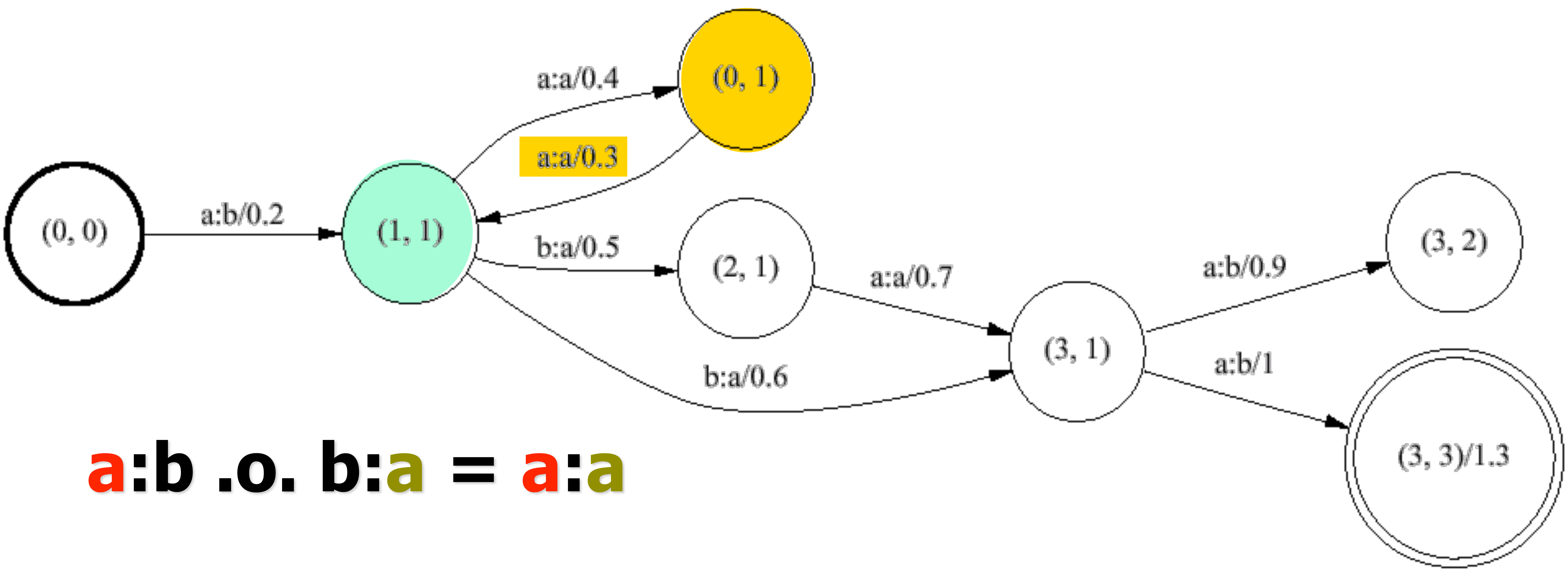
Composition



.o.

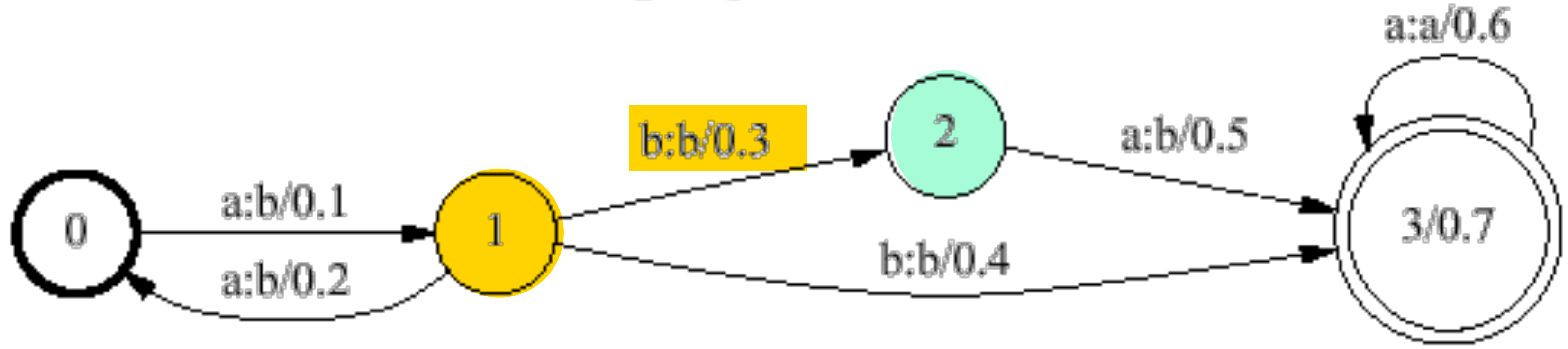


=

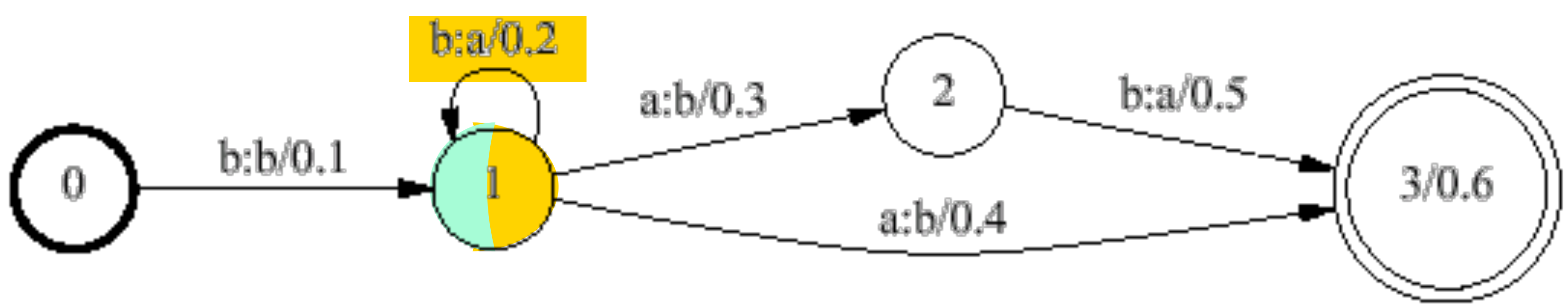


a:b .o. b:a = a:a

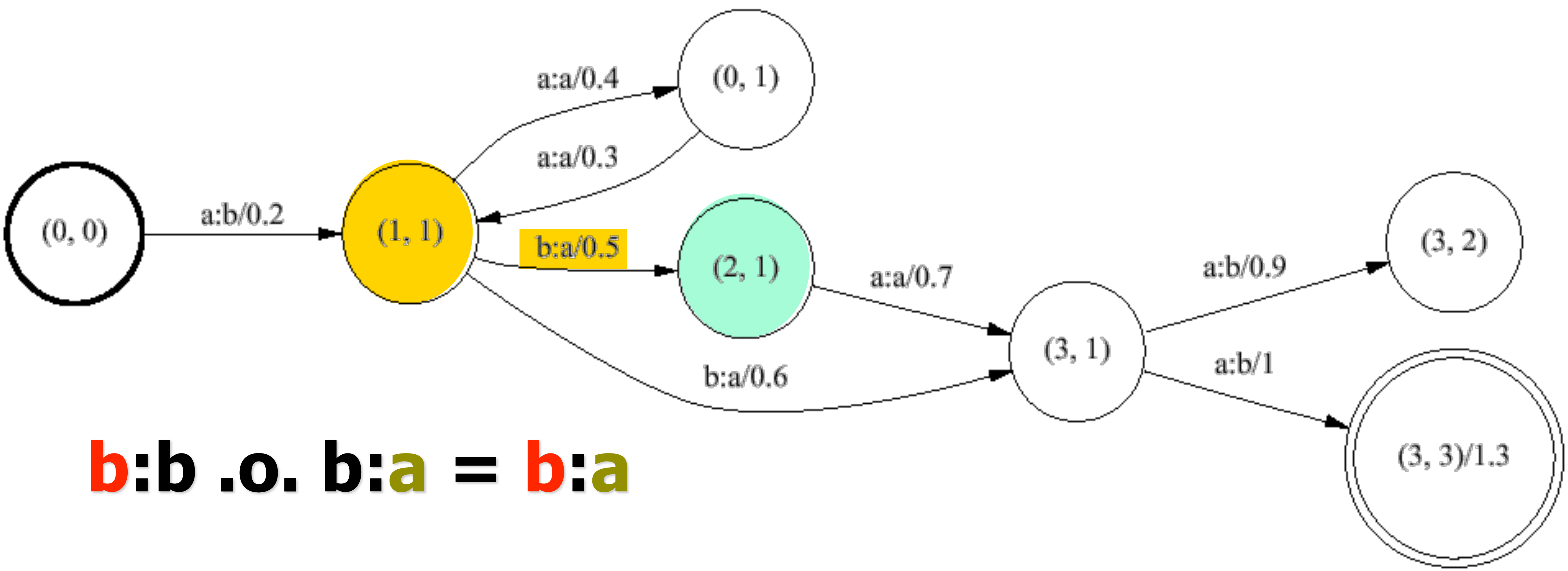
Composition



.o.

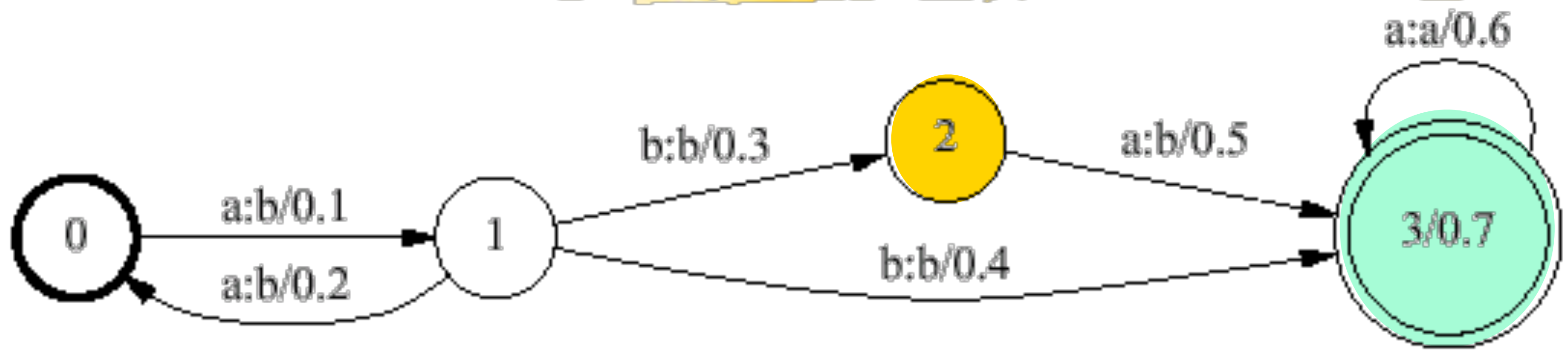


=

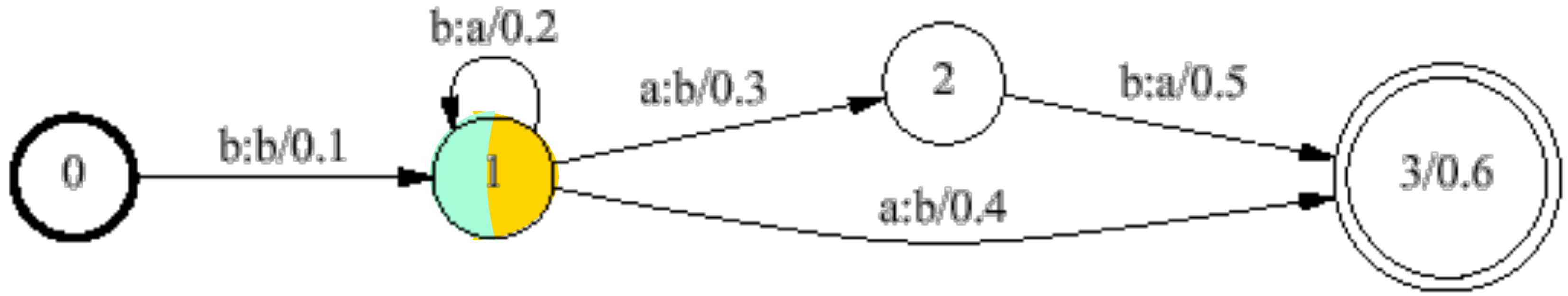


b:b .o. b:a = b:a

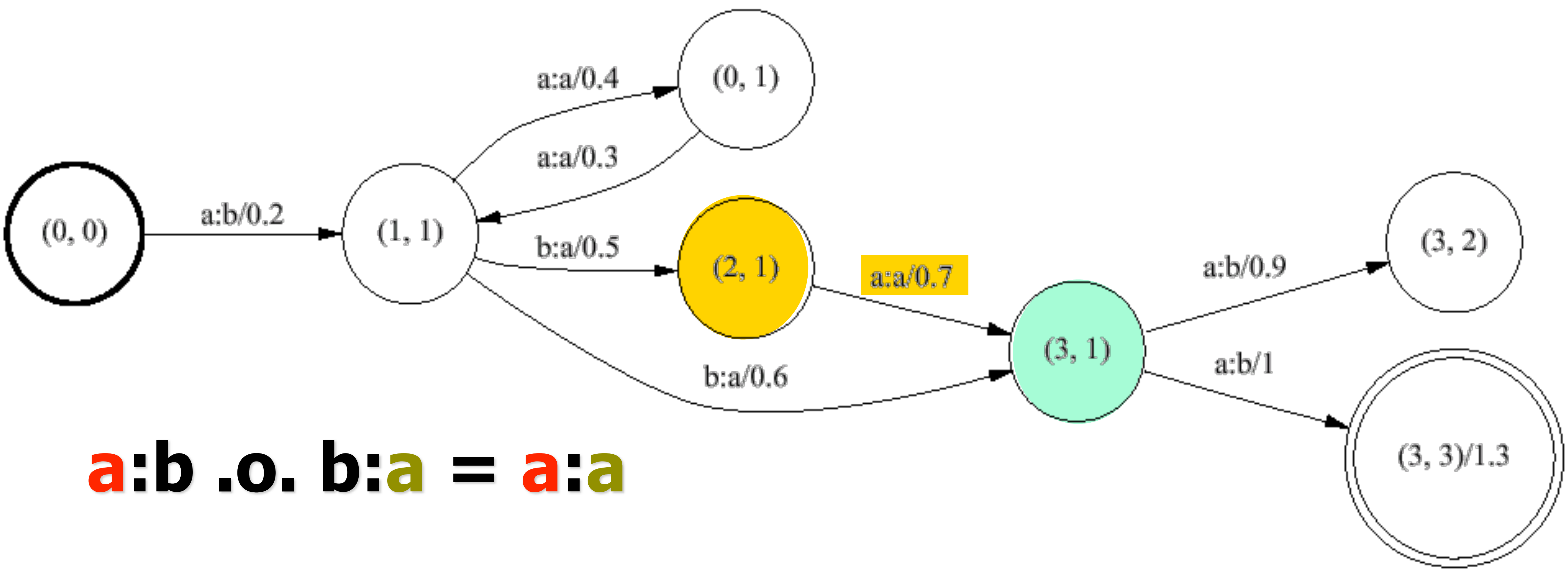
Composition



.o.

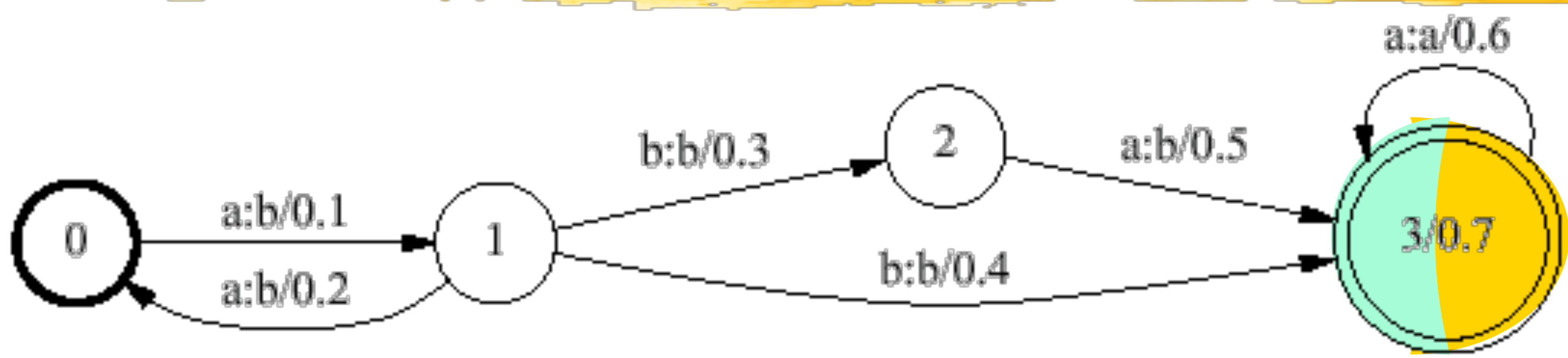


=

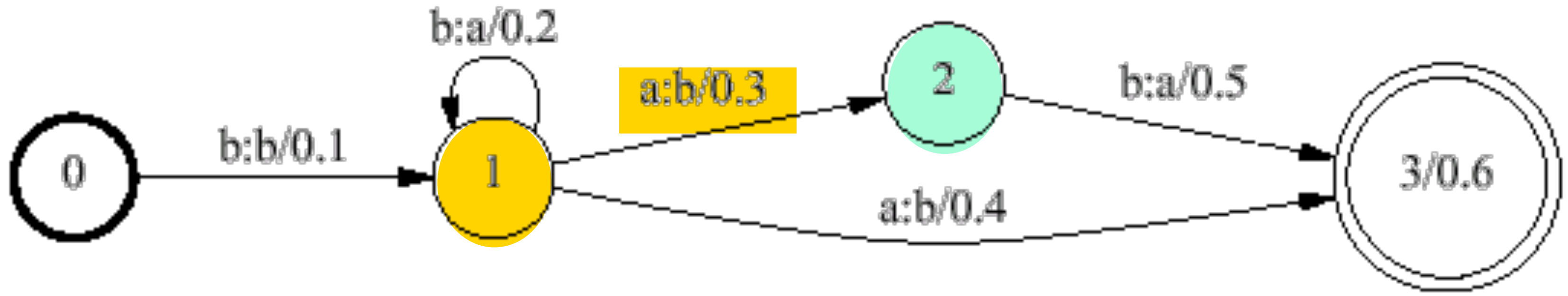


a:b .o. b:a = a:a

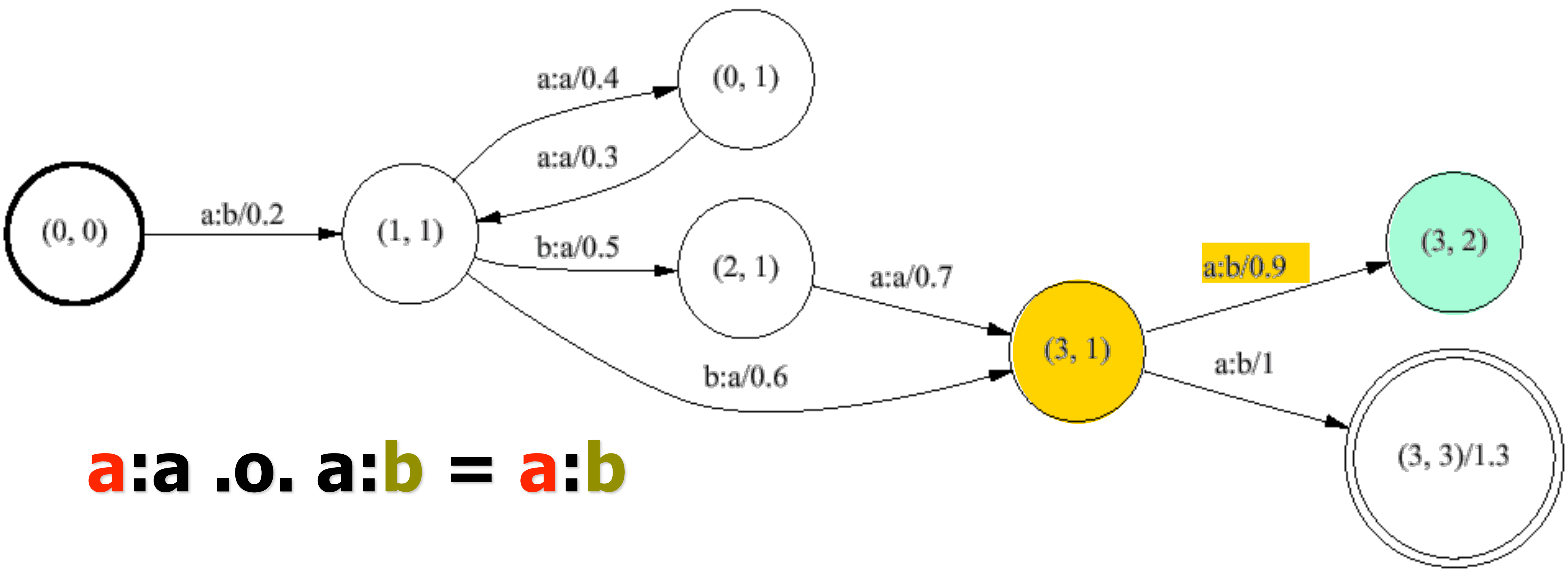
Composition



.o.

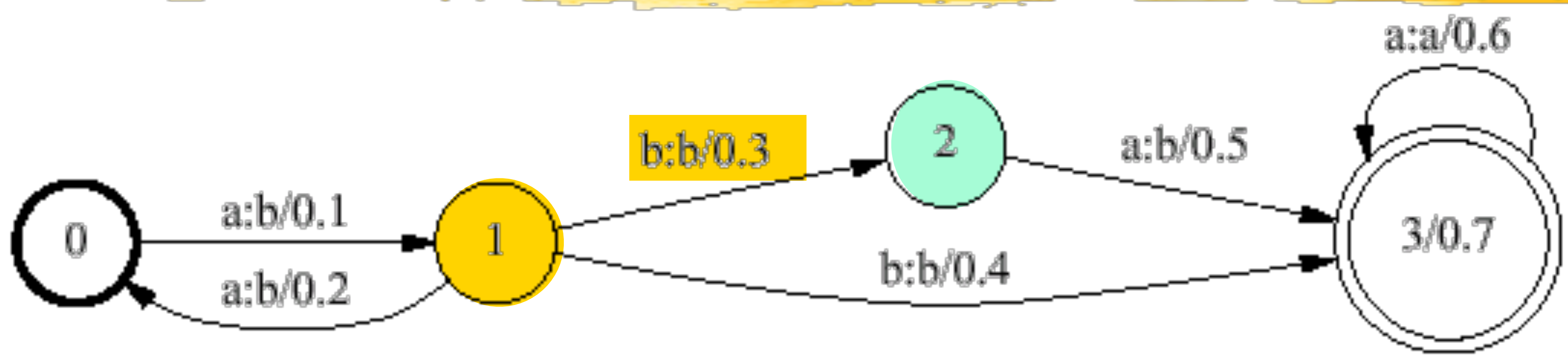


=

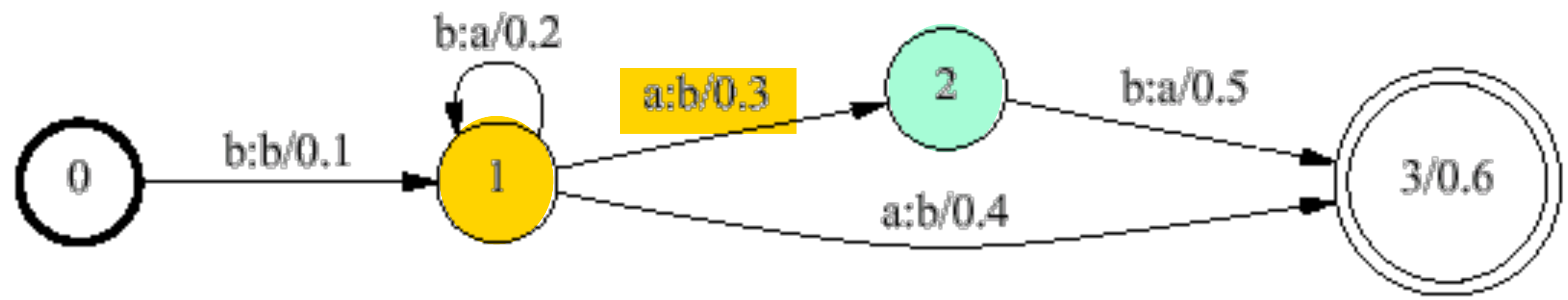


a:a .o. a:b = a:b

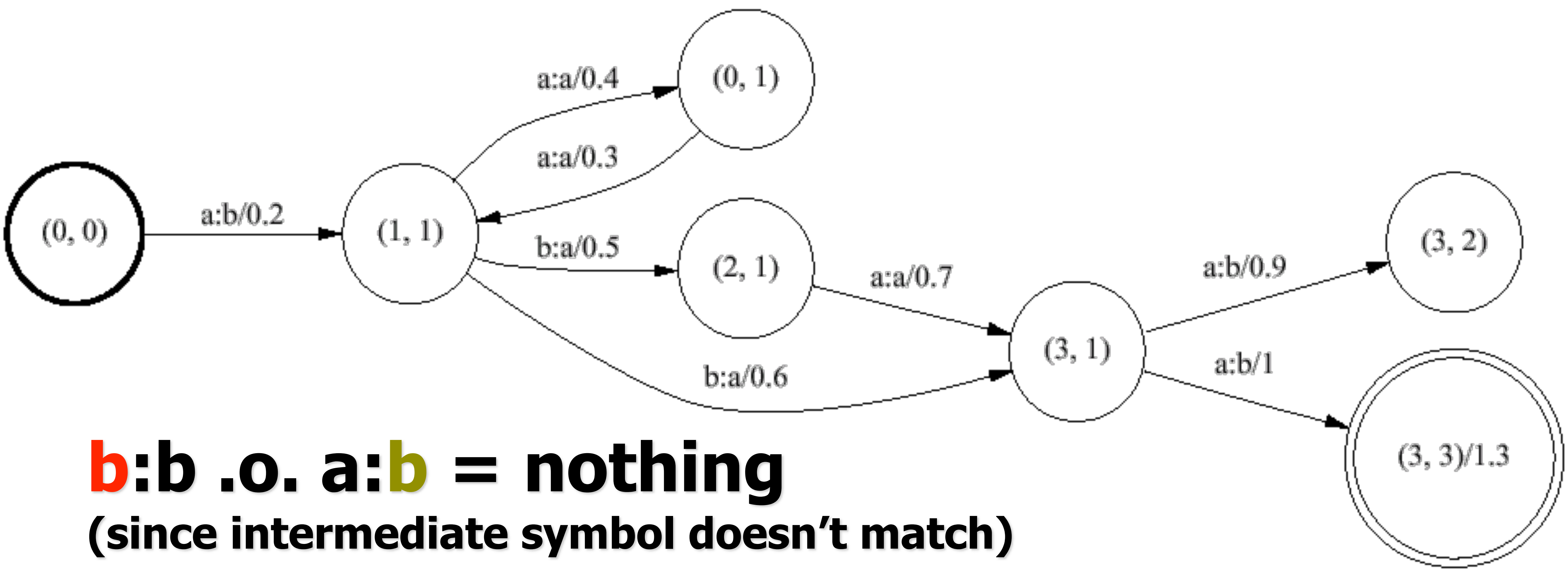
Composition



.o.



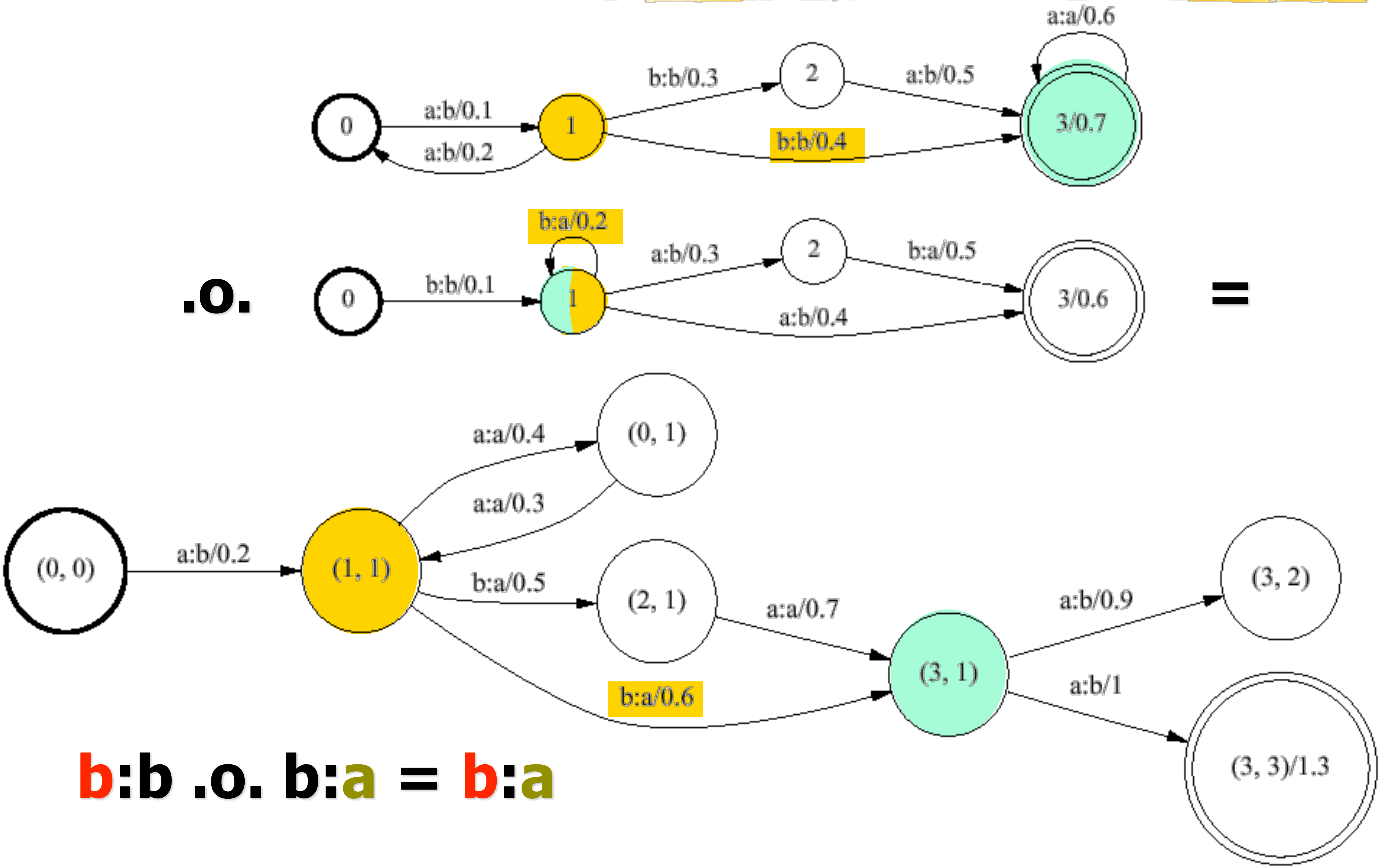
=



b:b .o. a:b = nothing

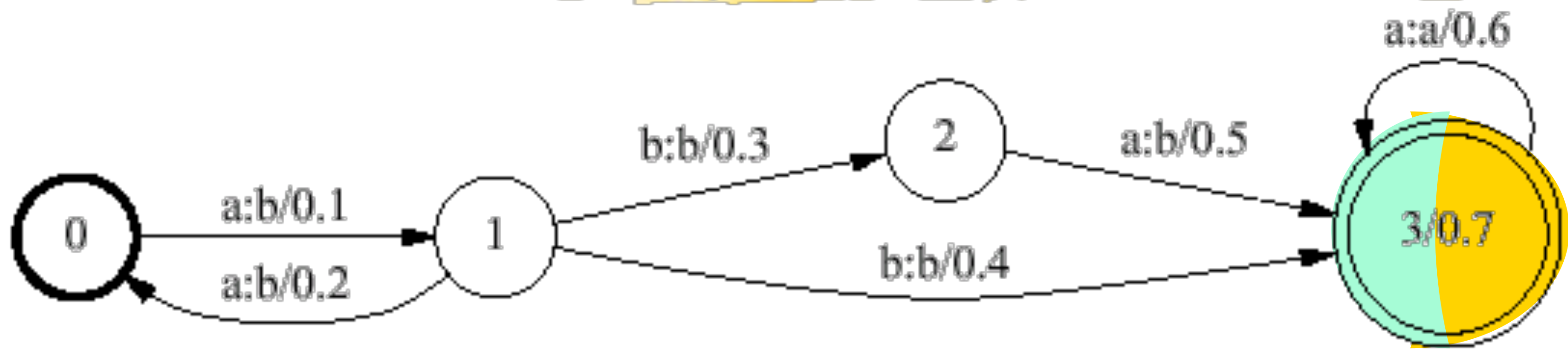
(since intermediate symbol doesn't match)

Composition

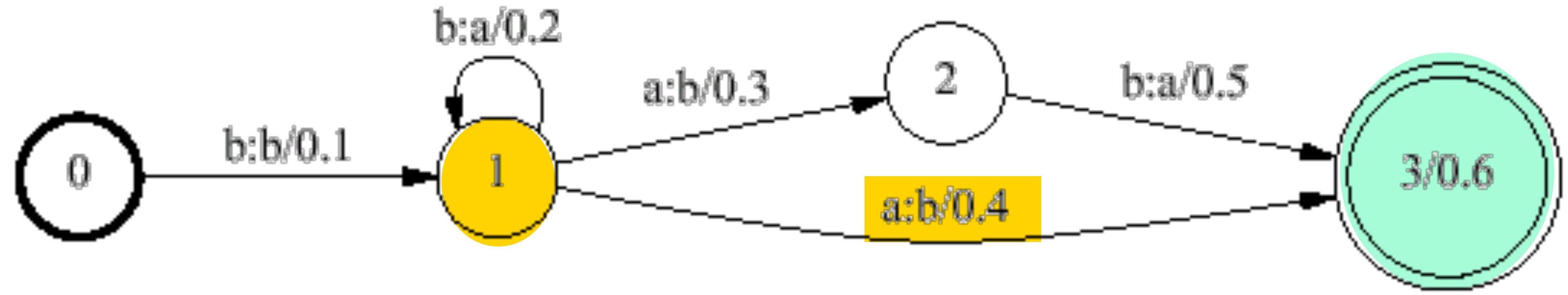


b:b .o. b:a = b:a

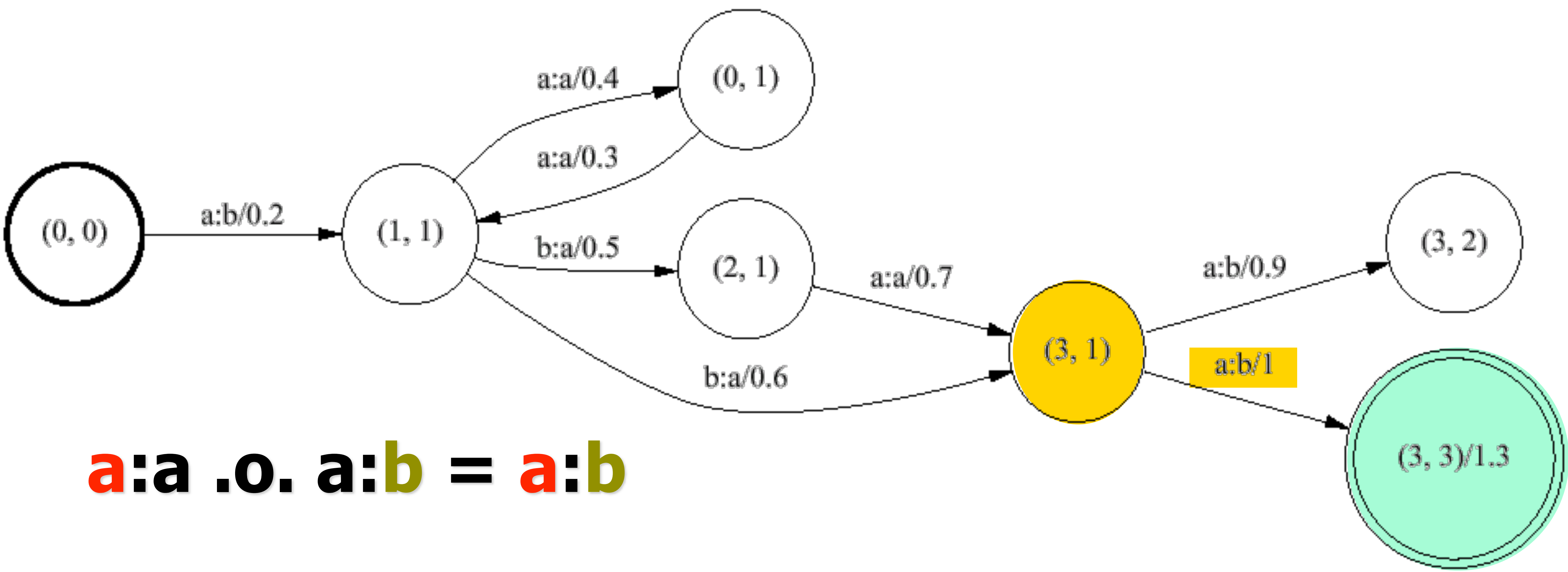
Composition



.o.



=



a:a .o. a:b = a:b

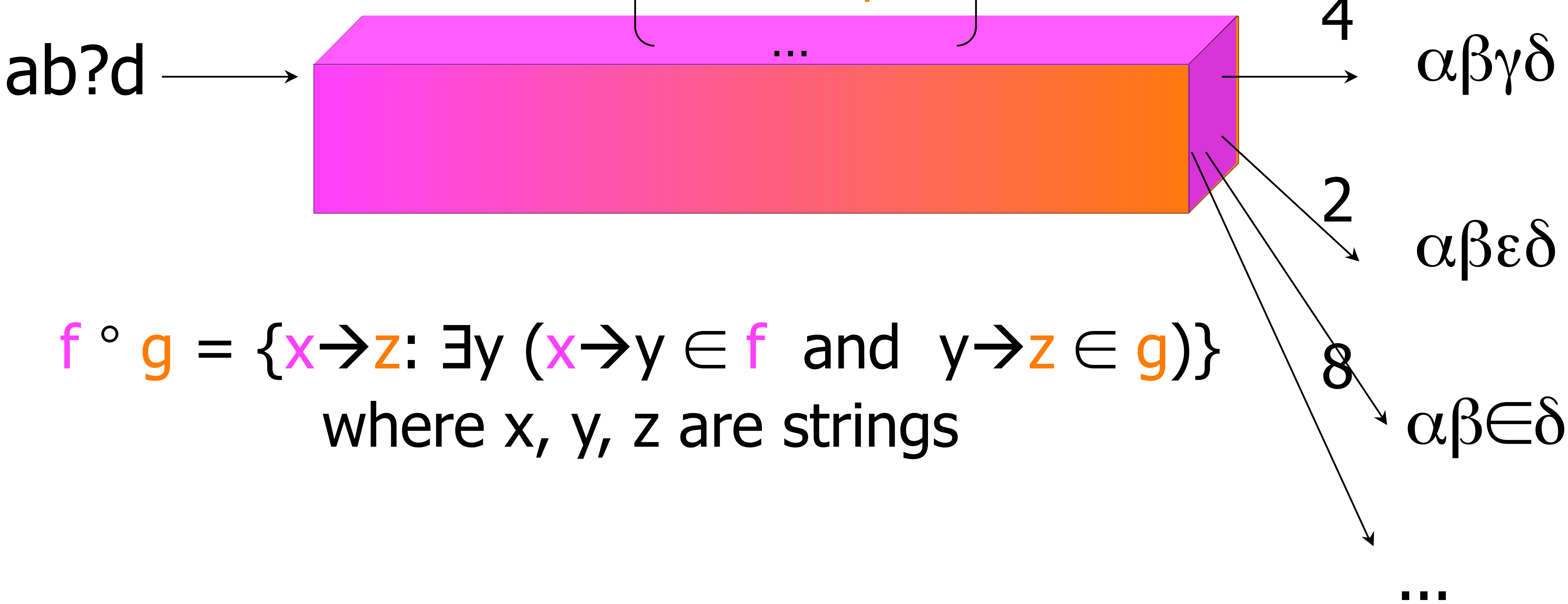
Relation = set of pairs

$$\left\{\begin{array}{l} \text{ab?d} \rightarrow \text{abcd} \\ \text{ab?d} \rightarrow \text{abed} \\ \text{ab?d} \rightarrow \text{abjd} \\ \quad\ldots \end{array}\right\}$$

$$\left\{\begin{array}{l} \text{abcd} \rightarrow \alpha\beta\gamma\delta \\ \text{abed} \rightarrow \alpha\beta\varepsilon\delta \\ \text{abed} \rightarrow \alpha\beta\in\delta \\ \quad\ldots \end{array}\right\}$$

$f \circ g$

$$\left\{\begin{array}{l} \text{ab?d} \rightarrow \alpha\beta\gamma\delta \\ \text{ab?d} \rightarrow \alpha\beta\varepsilon\delta \\ \text{ab?d} \rightarrow \alpha\beta\in\delta \\ \quad\ldots \end{array}\right\}$$



$f \circ g = \{x \rightarrow z\text{: } \exists y\ (x \rightarrow y \in f \text{ and } y \rightarrow z \in g)\}$

where x, y, z are strings

Composition with Sets

Composition with Sets

- We've defined A .o. B where both are FSTs

Composition with Sets

- We’ve defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA

Composition with Sets

- We've defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):

$$A \circ B = \{x \rightarrow z : \exists y \, (x \rightarrow y \in A \text{ and } y \rightarrow z \in B)\}$$

Composition with Sets

- We've defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):

  $$A \circ B = \{x \rightarrow z : \exists y\ (x \rightarrow y \in A \text{ and } y \rightarrow z \in B)\}$$

- Set and relation:

Composition with Sets

- We've defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):

$$A \circ B = \{x \rightarrow z : \exists y\ (x \rightarrow y \in A \text{ and } y \rightarrow z \in B)\}$$

- Set and relation:

$$A \circ B = \{x \rightarrow z : \qquad x \in A \text{ and } x \rightarrow z \in B\}$$

Composition with Sets

- We've defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):

$$A \circ B = \{x \rightarrow z : \exists y\ (x \rightarrow y \in A \ \text{ and } \ y \rightarrow z \in B)\}$$

- Set and relation:

$$A \circ B = \{x \rightarrow z : \quad x \in A \ \text{ and } \ x \rightarrow z \in B\}$$

- Relation and set:

$$A \circ B = \{x \rightarrow z : \quad x \rightarrow z \in A \ \text{ and } \ z \in B\}$$

Composition with Sets

- We've defined A .o. B where both are FSTs
- Now extend definition to allow one to be a FSA
- Two relations (FSTs):

  $A \circ B = \{x \rightarrow z : \exists y\ (x \rightarrow y \in A \text{ and } y \rightarrow z \in B)\}$

- Set and relation:

  $A \circ B = \{x \rightarrow z : x \in A \text{ and } x \rightarrow z \in B\}$

- Relation and set:

  $A \circ B = \{x \rightarrow z : x \rightarrow z \in A \text{ and } z \in B\}$

- Two sets (acceptors) – same as intersection:

  $A \circ B = \{x : x \in A \text{ and } x \in B\}$

Composition and Coercion

- Really just treats a set as identity relation on set

  $\{abc, pqr, \ldots\} = \{abc \rightarrow abc, pqr \rightarrow pqr, \ldots\}$

- Two relations (FSTs):

  $A \circ B = \{x \rightarrow z: \exists y\ (x \rightarrow y \in A \text{ and } y \rightarrow z \in B)\}$

- Set and relation is now special case (if $\exists y$ then $y=x$):

  $A \circ B = \{x \rightarrow z: \quad x \rightarrow x \in A \text{ and } x \rightarrow z \in B\}$

- Relation and set is now special case (if $\exists y$ then $y=z$):

- $A \circ B = \{x \rightarrow z: \quad x \rightarrow z \in A \text{ and } z \rightarrow z \in B\}$

- Two sets (acceptors) is now special case:

  $A \circ B = \{x \rightarrow x: \quad x \rightarrow x \in A \text{ and } x \rightarrow x \in B\}$

3 Uses of Set Composition:

3 Uses of Set Composition:

- Feed string into Greek transducer:

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - [{abed} .o. Greek].l = {αβεδ, αβ∈δ}

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - [{abed} .o. Greek].l = {αβεδ, αβ∈δ}
- **Feed several strings in parallel:**

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - [{abed} .o. Greek].l = {αβεδ, αβ∈δ}
- **Feed several strings in parallel:**
  - {abcd, abed} .o. Greek
    = {abcd→αβγδ, abed→αβεδ, abed→αβ∈δ}

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - [{abed} .o. Greek].l = {αβεδ, αβ∈δ}
- **Feed several strings in parallel:**
  - {abcd, abed} .o. Greek
    = {abcd→αβγδ, abed→αβεδ, abed→αβ∈δ}
  - [{abcd,abed} .o. Greek].l = {αβγδ, αβεδ, αβ∈δ}

3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - [{abed} .o. Greek].l = {αβεδ, αβ∈δ}
- **Feed several strings in parallel:**
  - {abcd, abed} .o. Greek = {abcd→αβγδ, abed→αβεδ, abed→αβ∈δ}
  - [{abcd,abed} .o. Greek].l = {αβγδ, αβεδ, αβ∈δ}
- **Filter result via No**ε = {αβγδ, αβ∈δ, …}

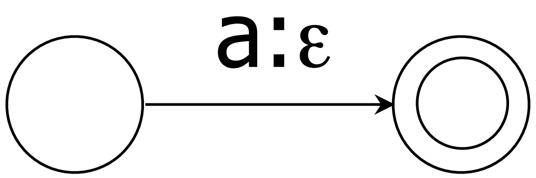
3 Uses of Set Composition:

- **Feed string into Greek transducer:**
  - {abed→abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - {abed} .o. Greek = {abed→αβεδ, abed→αβ∈δ}
  - [{abed} .o. Greek].l = {αβεδ, αβ∈δ}
- **Feed several strings in parallel:**
  - {abcd, abed} .o. Greek
    = {abcd→αβγδ, abed→αβεδ, abed→αβ∈δ}
  - [{abcd,abed} .o. Greek].l = {αβγδ, αβεδ, αβ∈δ}
- **Filter result via Noε = {αβγδ, αβ∈δ, …}**
  - {abcd,abed} .o. Greek .o. Noε
    = {abcd→αβγδ, abed→αβ∈δ}

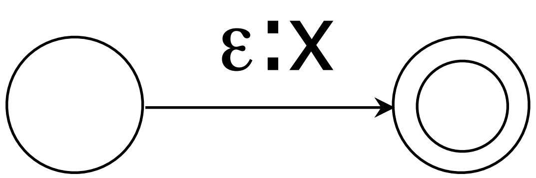
What are the “basic” transducers?

- The operations on the previous slides combine transducers into bigger ones
- But where do we start?

- $a{:}\varepsilon$ for $a \in \Sigma$



- $\varepsilon{:}x$ for $x \in \Delta$



- Q: Do we also need a:x? How about $\varepsilon{:}\varepsilon$?

Reading

- Okan Kolak, William Byrne, and Philip Resnik. A Generative Probabilistic OCR Model for NLP Applications. In *HLT-NAACL*, 2003.

  http://aclweb.org/anthology-new/N/N03/N03-1018.pdf