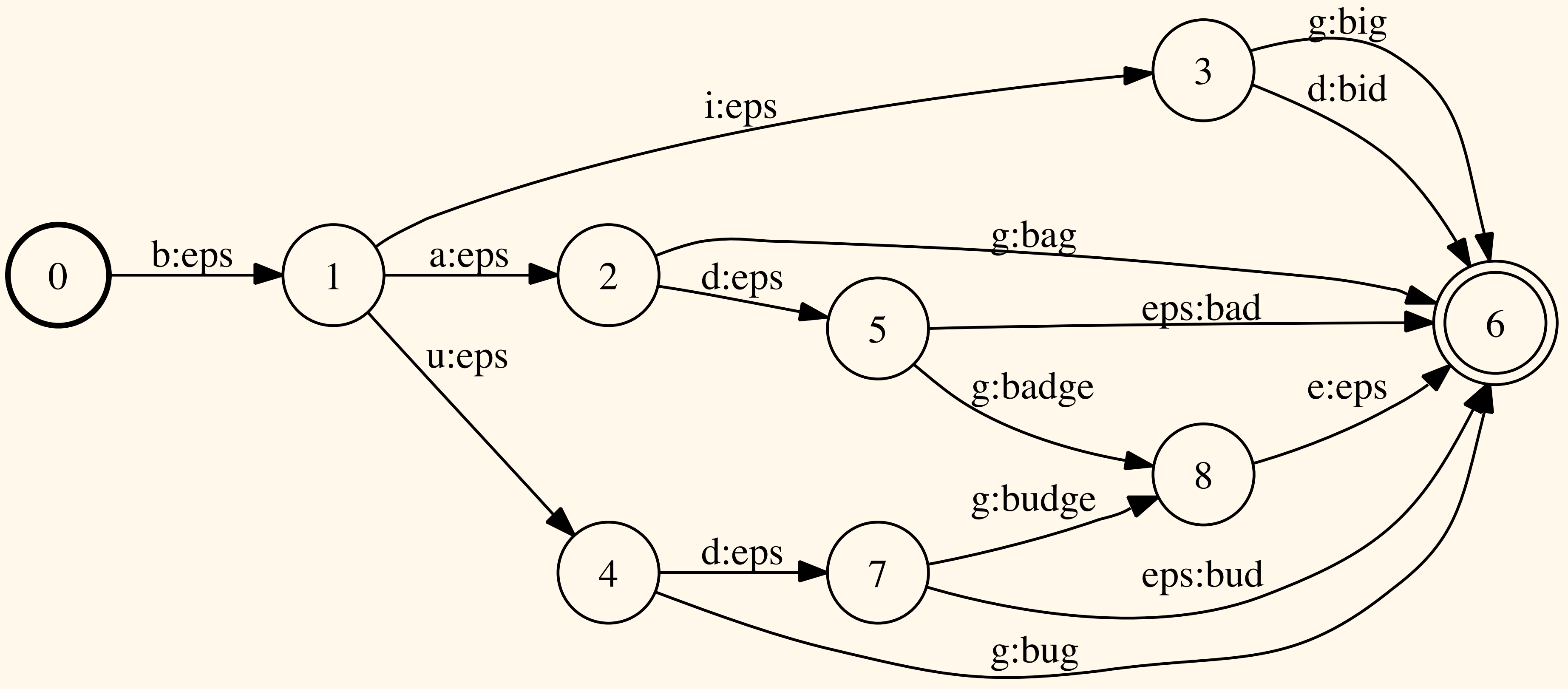
Finite State Transducers:
Background & Theory



Steven Bedrick
CS/EE 5/655, 10/6/14

http://www.openfst.org/twiki/pub/FST/FstHltTutorial/tutorial_part3.pdf

Plan for the day:

1. Quick review of last time
2. Finite-state transducers
3. Weights
4. Monoids and Semirings
5. FST Operations
6. Demo?

Formal definition:

An FSA is defined by:

$Q = q_0, q_1, q_2 ... q_{n-1}$	A finite state of *n* states
Σ	A finite input alphabet of symbols
q_0	A start state
F	Set of final states $F \subseteq Q$
$\delta(q, i)$	Transition matrix between states

Conceptual operation:

There exists an “input tape” made up of a linear sequence of symbols...

Starting at q0, we read the first input symbol...

If it matches one of our transition arcs, we move to the destination state and advance the input tape...

If we hit a final state before we run out of input, we “succeed”; otherwise, we “fail.”

Regular languages, by definition, can be recognized by an FSA.

$$\begin{tikzcd}
q0 \arrow[r, "b"] & q1 \arrow[r, "a"] & q2 \arrow[r, "a"] & q3 \arrow[r, "!"] \arrow[loop above, "a"] & q4
\end{tikzcd}$$

baaa!   baaaaaaaa!   baaaa   ba!

FSAs can *match* strings from a language…

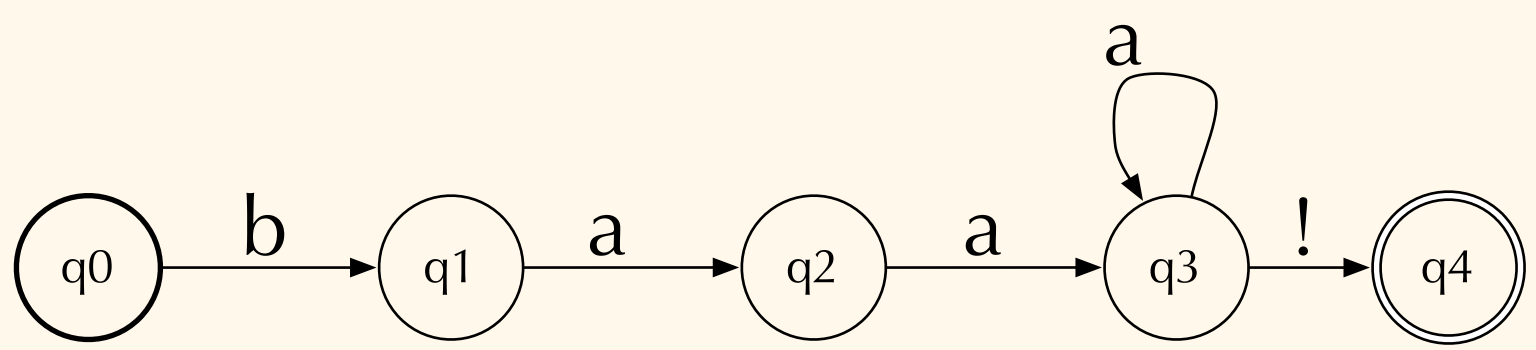
… and they can *generate* strings in a language…

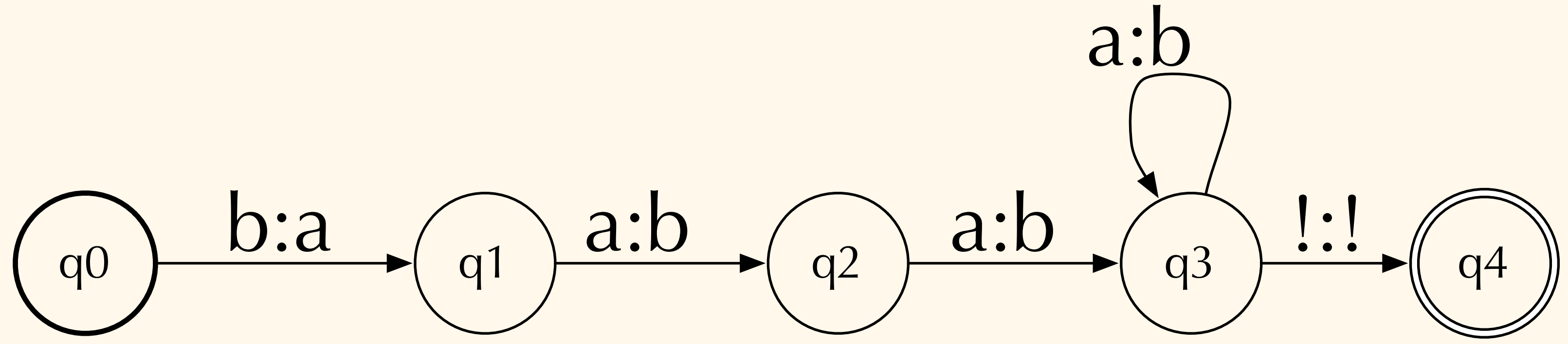
… what if we need more?

Finite State *Transducers* are FSAs that *map between sets of symbols*.

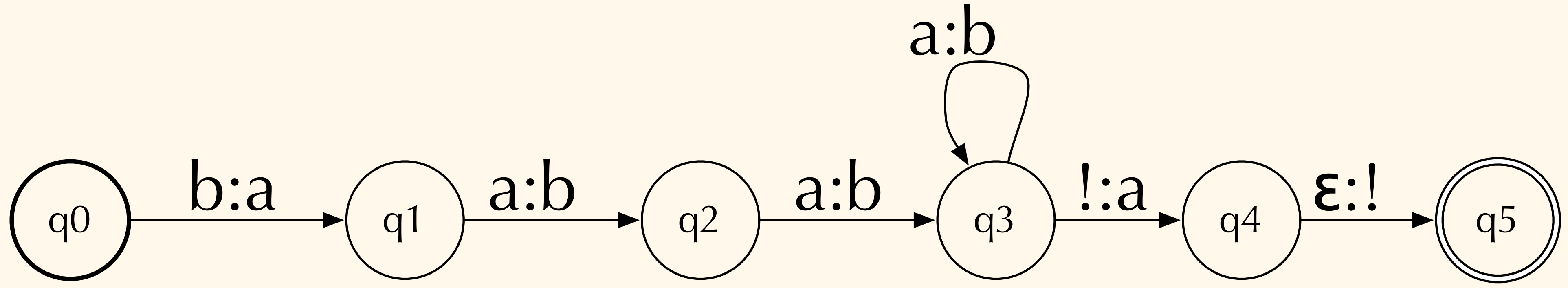
In other words, an FSA with both input *and* output tapes.

$Q = q_0, q_1, q_2 \ldots q_{n-1}$	A finite state of *n* states
Σ	A finite input alphabet of symbols
Δ	Alphabet of output symbols
q_0	A start state
F	Set of final states $F \subseteq Q$
$\delta(q, w)$	Transition matrix/function between states and inputs
$\sigma(q, w)$	Output function giving possible output symbols for each state and input.





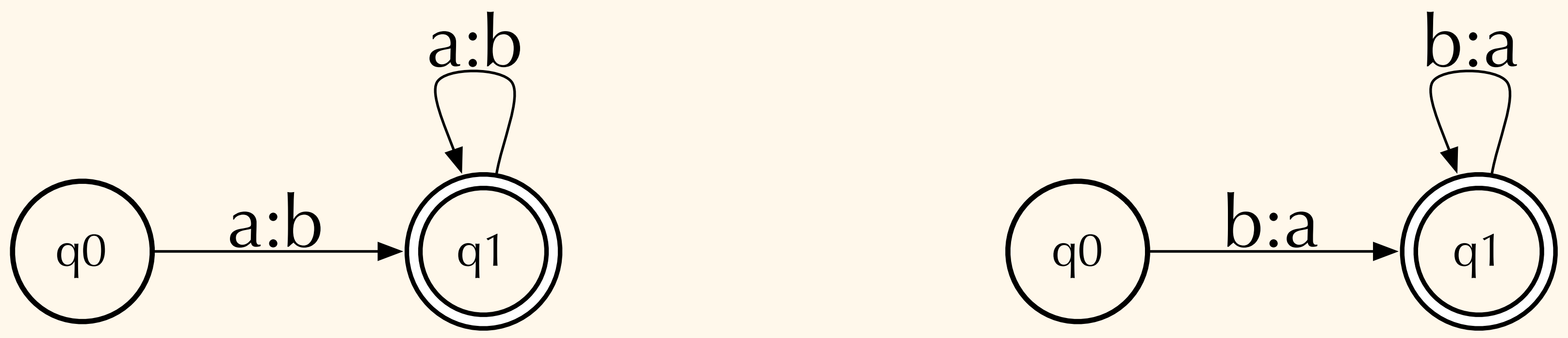
baaa! abbb!



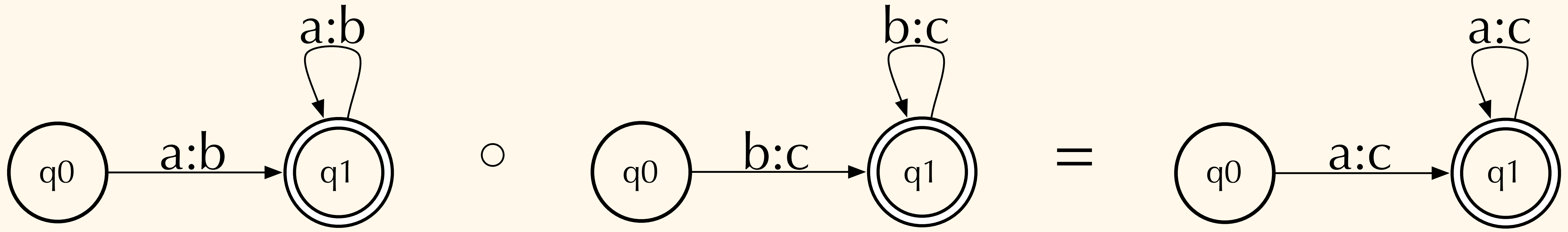
baaa! abbba!

FSTs have many of the same operations as FSAs (union, concatenation, etc.) as well as several others:

Inversion swaps input and output labels...



Composition maps the output labels from one FST to the input labels of another:



Like FSAs, FSTs can be non-deterministic...

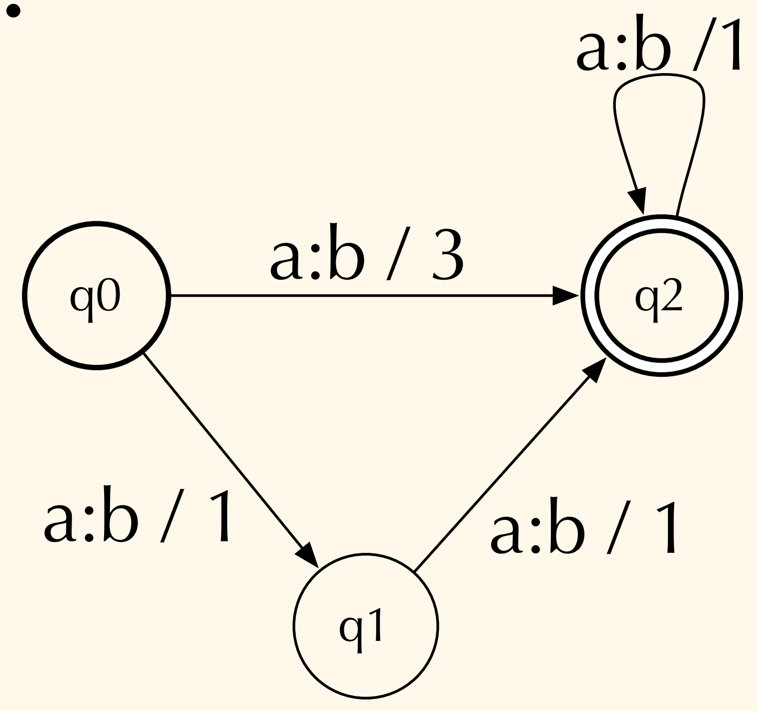
... but *unlike* FSAs, not *all* FSTs can be determinized.

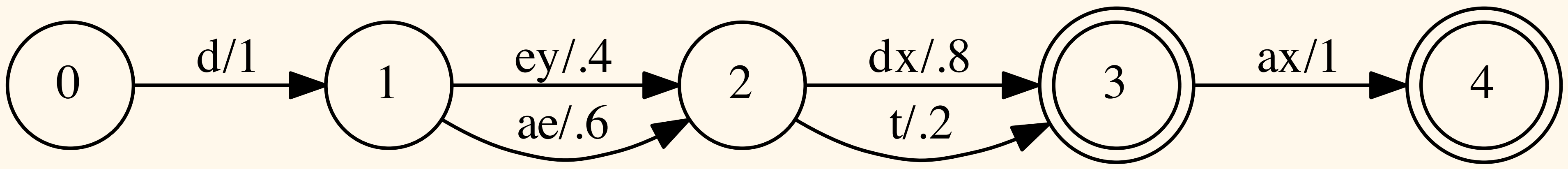
(in practice, though, most of the ones we care about can be determinized)

An extension:

wFSTs (and wFSAs) allow *weights* on their edges...

... to help decide how to travel through a non-deterministic FST.





/d ae dx ax/ $1 \times 0.6 \times 0.8 \times 1 = 0.48$

/d ey t ax/ $1 \times 0.4 \times 0.2 \times 1 = 0.08$

wFSAs, formally defined:

$Q = q_0, q_1, q_2 ... q_{n-1}$	A finite state of *n* states
Σ	A finite input alphabet of symbols
q_0	A start state
F	Set of final states
$\sigma : Q \times \Sigma^* \rightarrow K$	Function assigning a weight to paths between pairs of states
λ, ρ	Initial and final output functions (assigns weight to entering & leaving the network)

The wFST is the natural extension to the wFSA, and works as you would expect.

Just one thing is missing…

$$\begin{tikzcd}
0 \arrow[r, "d/1"] & 1 \arrow[r, "ey/.4"] \arrow[r, bend right, "ae/.6"'] & 2 \arrow[r, "dx/.8"] \arrow[r, bend right, "t/.2"'] & 3 \arrow[r, "ax/1"] & 4
\end{tikzcd}$$

/d ae dx ax/ $1 \times 0.6 \times 0.8 \times 1 = 0.48$

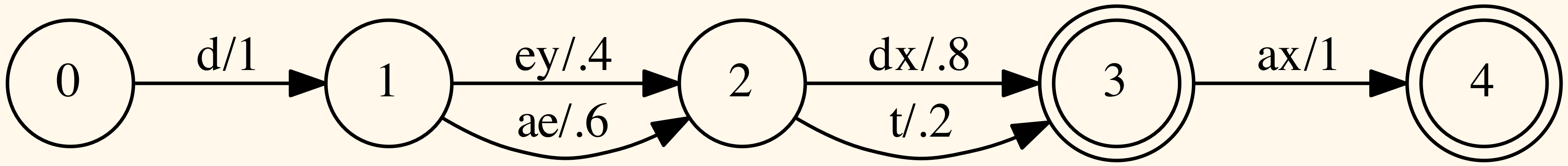
/d ey t ax/ $1 \times 0.4 \times 0.2 \times 1 = 0.08$

Why multiply?

The weights on an wFS[AT] can represent many different things...

... the actual operations will need to vary.

For example, in our toy example:



The weights represent real-space probabilities...

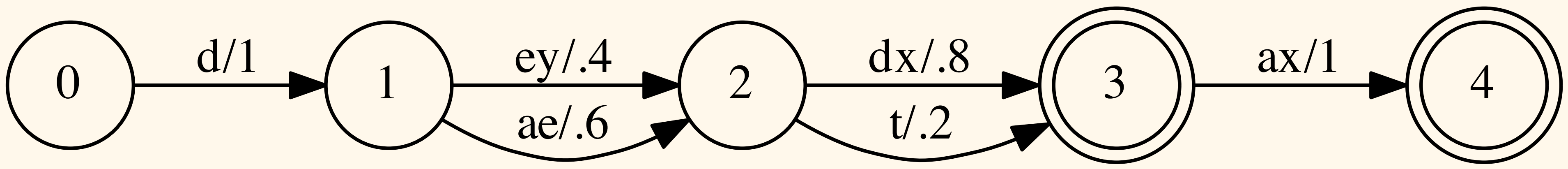
... which won't work with larger models.

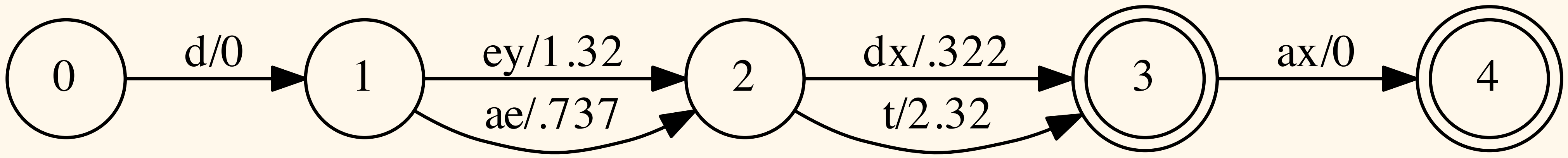
For real applications, the probabilities involved are usually *very* small...

... which means that we run the risk of numerical underflow when we multiply them!

```
>>> 2 ** -1000
9.332636185032189e-302
>>> 2 ** -2000
0.0
```

Solution: represent small probabilities as negative log-probabilities.





/d ae dx ax/ $0 + 0.737 + 0.322 + 0 = 1.059$

$$2^{-1.059} = 0.48$$

So: we need some way to say which operation to use when computing weights.

Luckily, there are mathematical constructs designed for just this purpose.

We will discuss two: *monoids*, and *semirings*.

A *monoid* is a pair (M, ⊕) where:

M is a *set*, and

⊕ is a binary operation on M obeying three rules:

1. Closure: for all a, b in M, $a \oplus b$ is also in M.

2. Identity: there exists an element e in M s.t.:

$$\forall a \in M, a \oplus e = e \oplus a = a$$

3. Associativity: ⊕ is associative:

$$\forall a, b, c \in M, (a \oplus b) \oplus c = a \oplus (b \oplus c)$$

For example, real numbers and regular addition form a monoid:

1. Closure: any real number plus any other real number is still a real number;

2. Identity: 0 is the additive identity:

$$\forall a \in M, a \oplus e = e \oplus a = a$$

$$2 + 0 = 0 + 2 = 2$$

3. Associativity: addition is associative:

$$(2 + 4) + 3 = 2 + (4 + 3)$$

A *semiring* is a monoid with an additional operator and more constraints:

A semiring is a triple (K, ⊕, ⊗) where:

1. (K, ⊕) form a monoid (whose neutral element we'll call 0)
2. (K, ⊗) form a monoid (whose neutral element we'll call 1)

3. ⊗ distributes with respect to ⊕:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$

4. $\forall a \in K, a \otimes 0 = 0 \otimes a = 0$

If those constraints are met, *any* binary two operators can be used!

SEMIRING	SET	$\oplus$	$\otimes$	$\overline{0}$	$\overline{1}$
Boolean	$\{0, 1\}$	$\vee$	$\wedge$	0	1
Probability	$\mathbb{R}_+$	$+$	$\times$	0	1
Log	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\oplus_{\log}$	$+$	$+\infty$	0
Tropical	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\min$	$+$	$+\infty$	0
String	$\Sigma^* \cup \{\infty\}$	$\wedge$	$\cdot$	∞	ϵ

For our purposes, the Probability (a.k.a. "Real") and Tropical semirings are most useful.

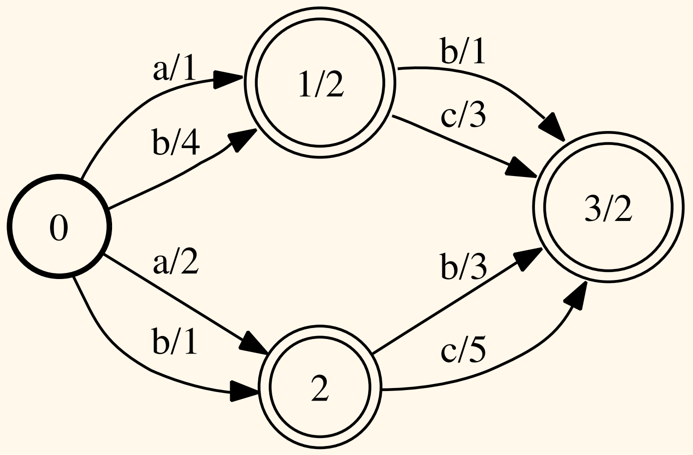
This and some other figures from: http://www.openfst.org/twiki/pub/FST/FstHltTutorial/tutorial_part1.pdf

Back to wFSAs: there are two main mathematical operations we need to do to work with weights:

Product: to compute the weight of a given path;

Sum: to compute the weight of a given sequence (which might have multiple paths)

What is the "cost" of the string "ab"?



Probability ($\mathbb{R}_+$,+,x,0,1)	Tropical ($\mathbb{R}_+$,min,+,∞,0)
[[A]](ab) = 14	[[A]](ab) = 4
(1 x 1 x 2 + 2 x 3 x 2) = 14	min(1+1+2, 3+2+2)

Applications for FSTs:

Tagging (transform sequence of words into sequence of tags)

Morphological analysis

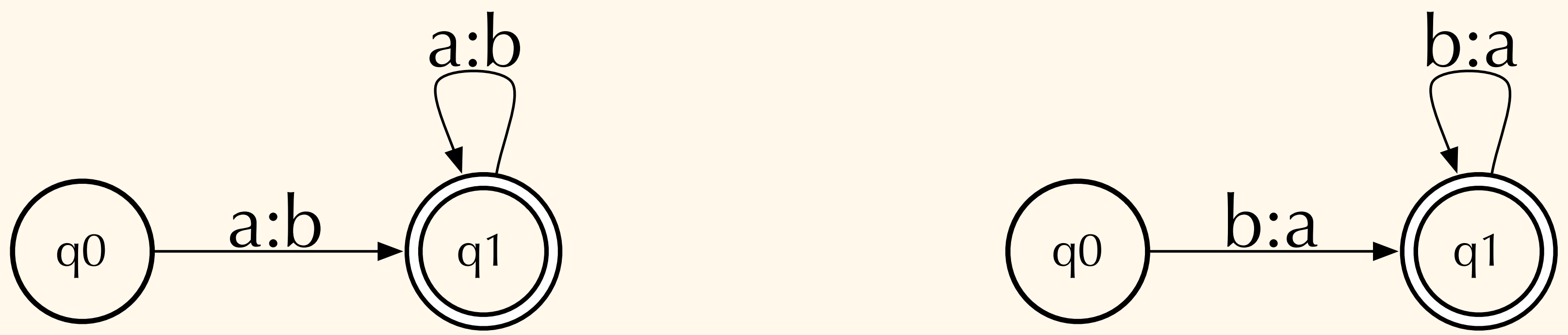
Spelling correction (transform mis-spelled word into lattice of possible words)

ASR (transform acoustic signal into phonemes, phonemes into words, etc.)

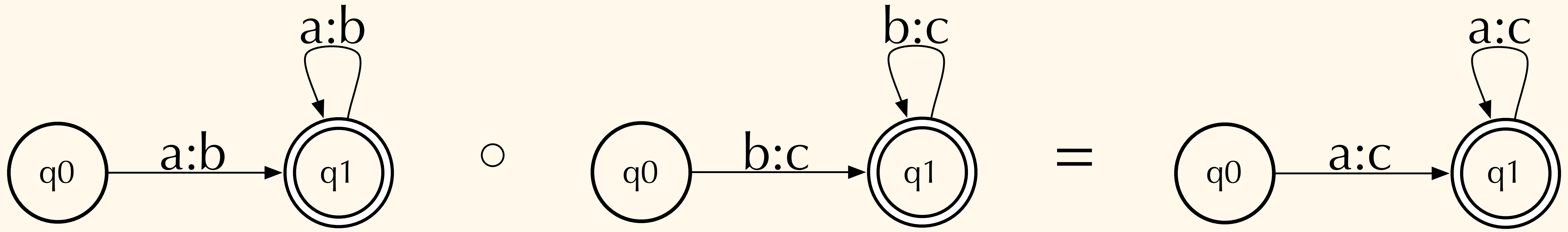
Etc....

FSTs have many of the same operations as FSAs (union, concatenation, etc.) as well as several others:

Inversion swaps input and output labels...

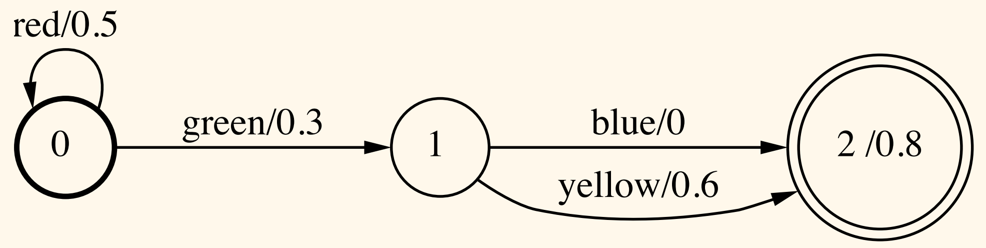


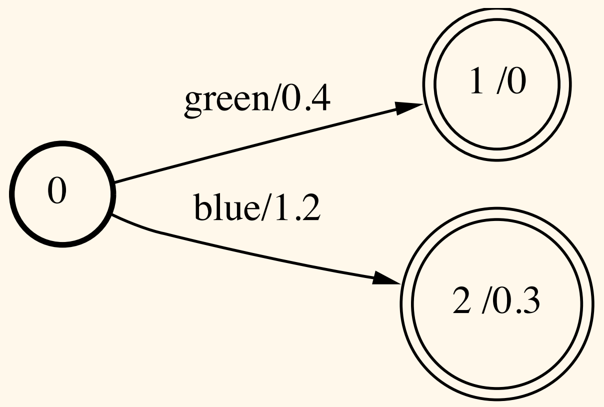
Composition maps the output labels from one FST to the input labels of another:

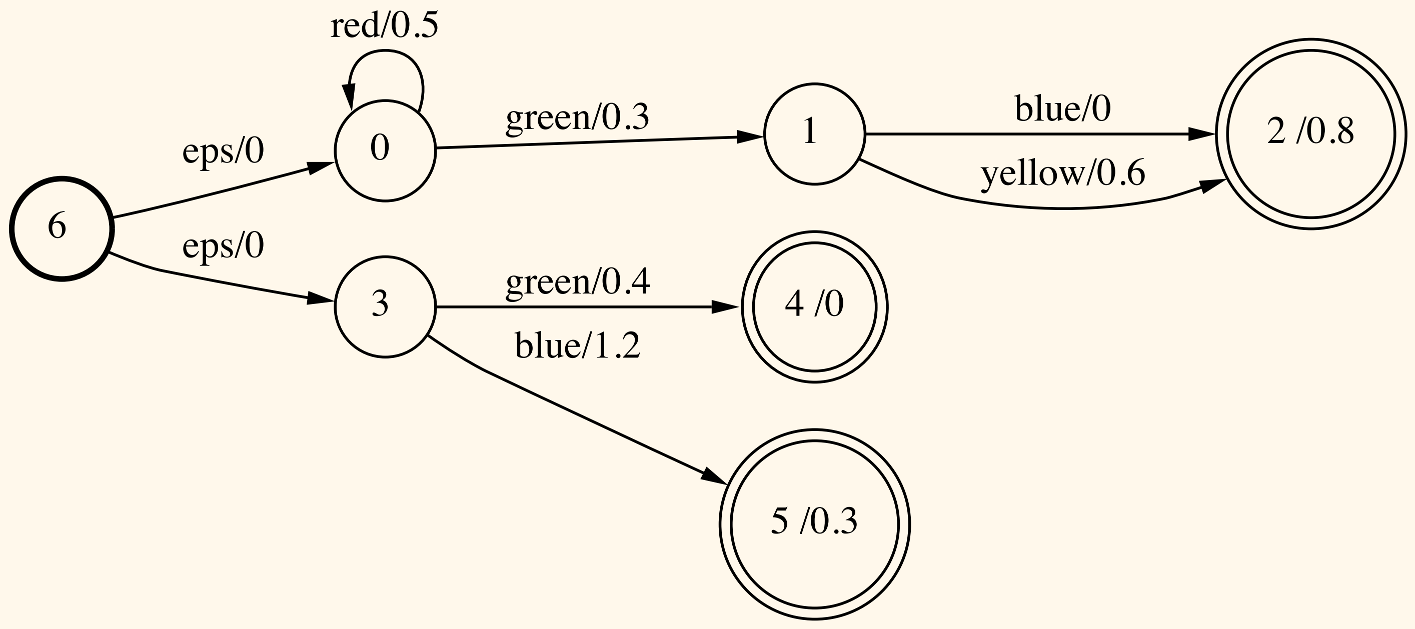


wFS[AT] Operations, in more detail:

Union:

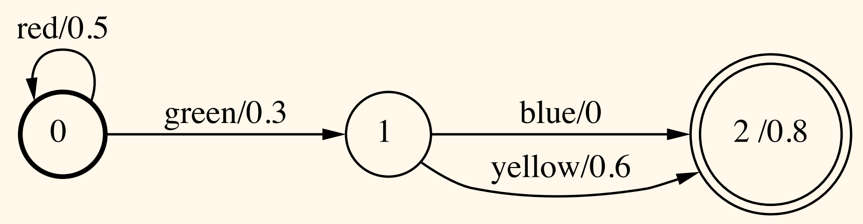


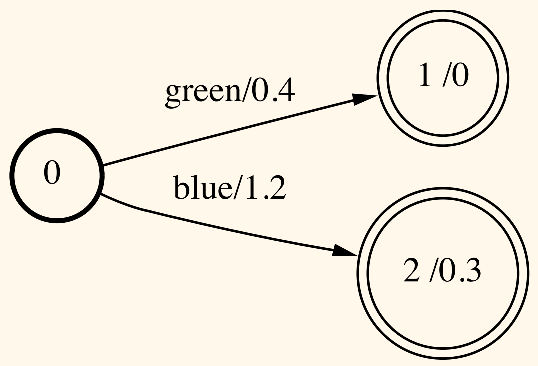


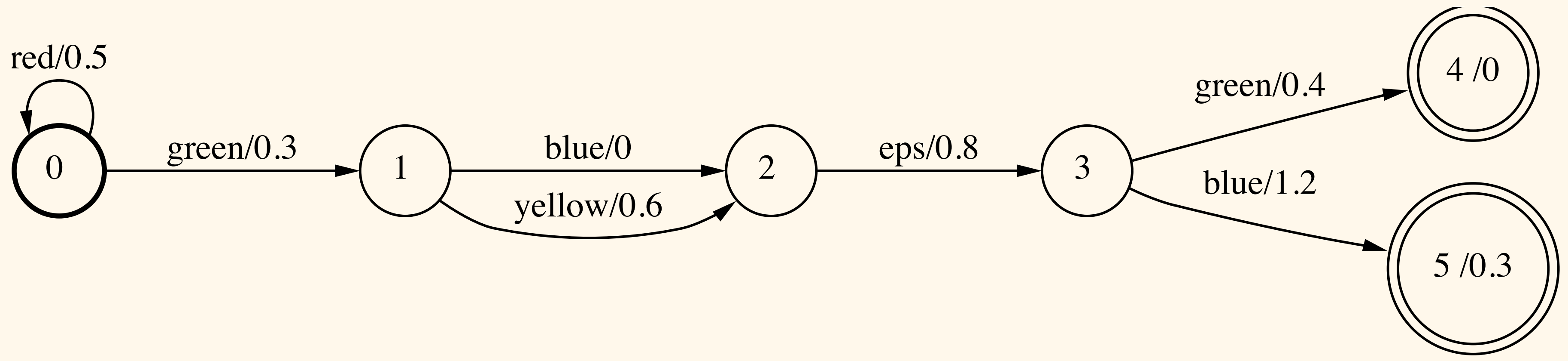


wFS[AT] Operations, in more detail:

Concatenation:

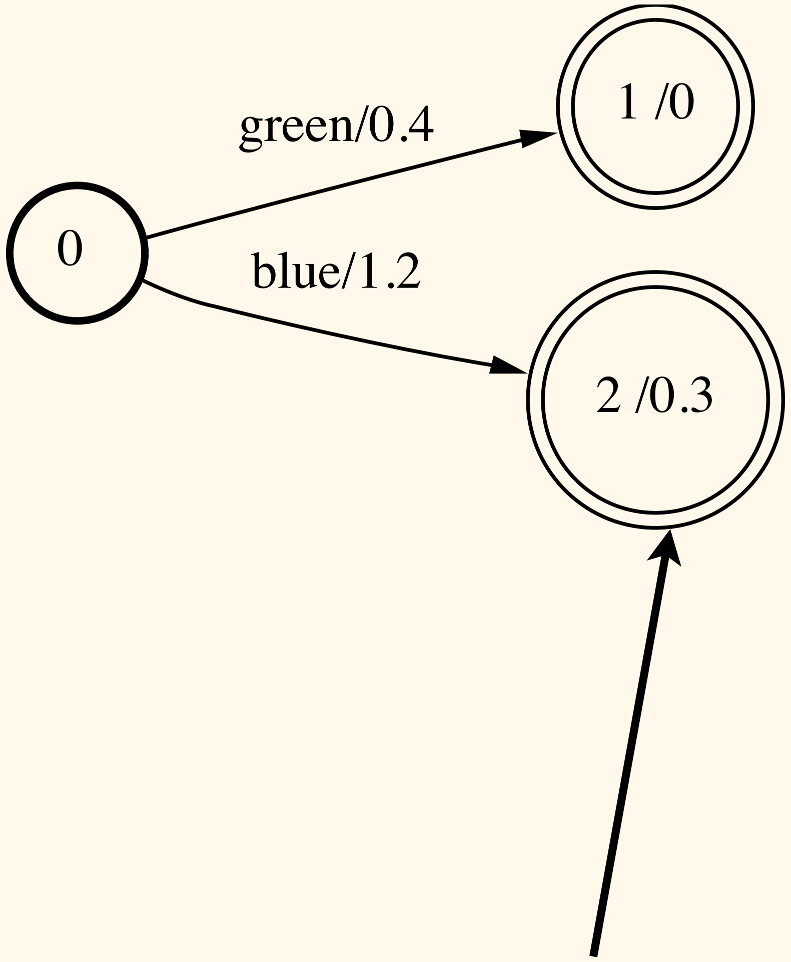


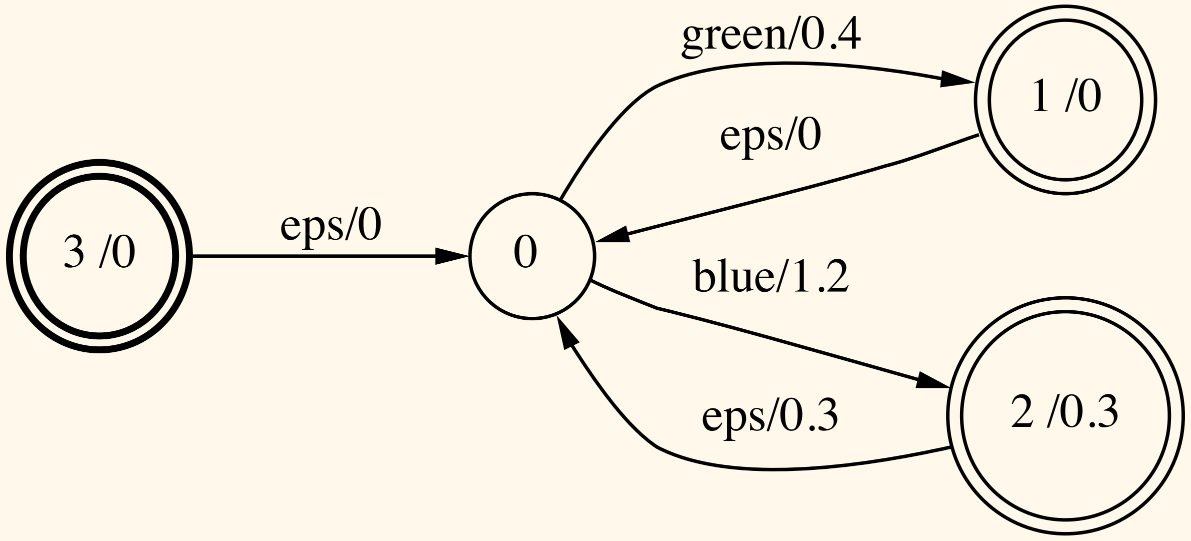




wFS[AT] Operations, in more detail:

Closure (Kleene*):

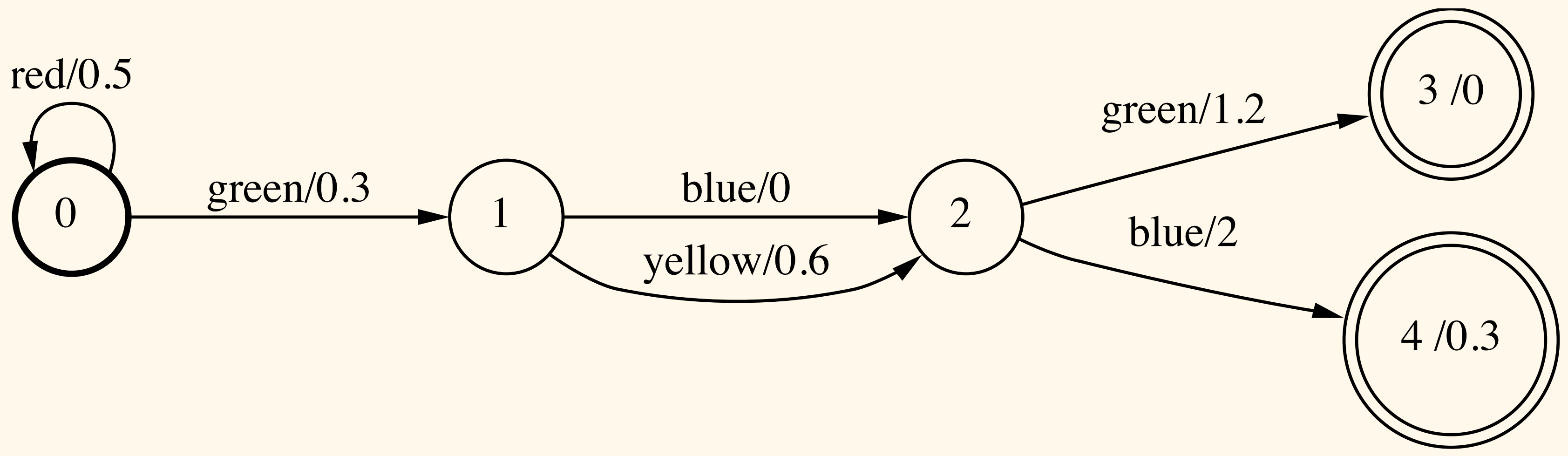


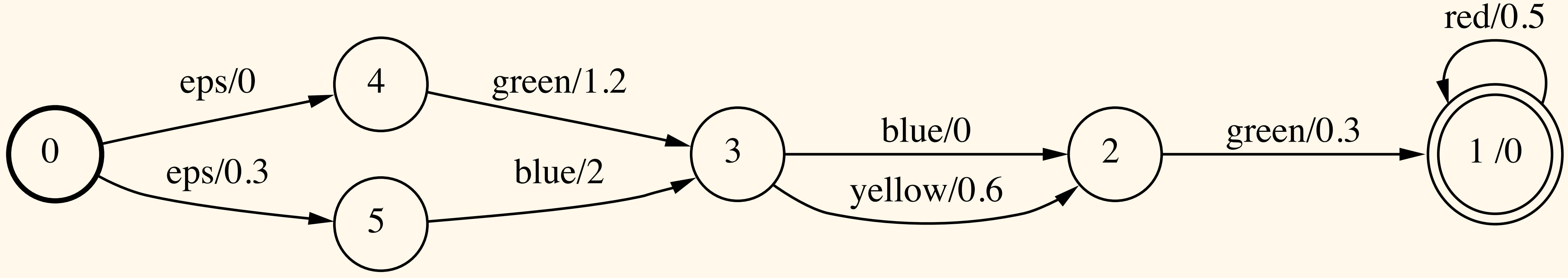


Note the exit weight...

Unary wFS[AT] Operations, in more detail:

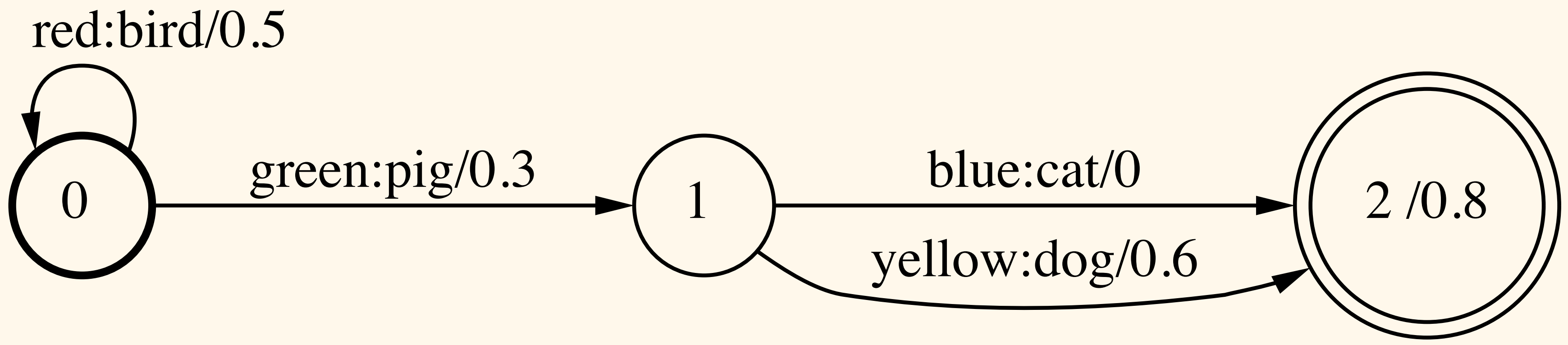
Reversal:

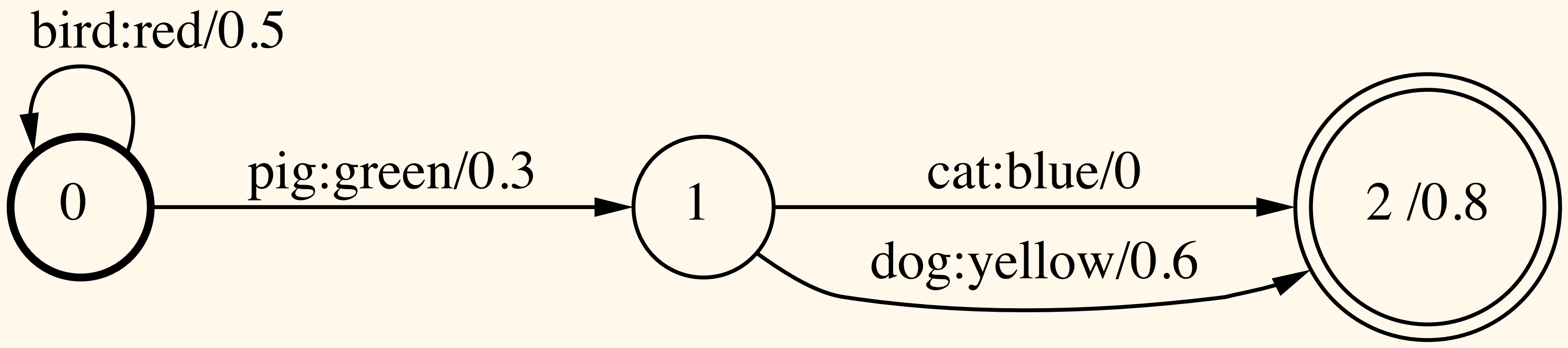




Unary wFS[AT] Operations, in more detail:

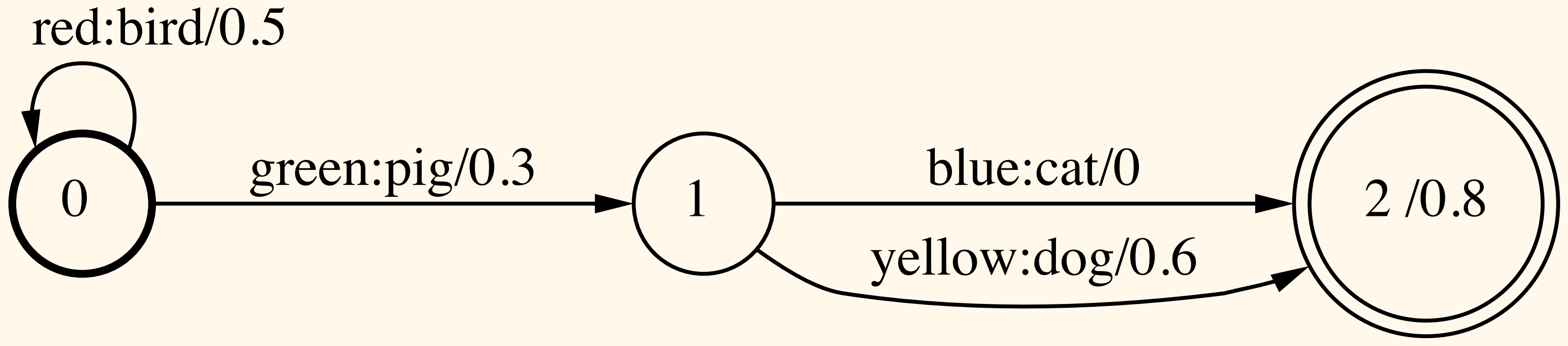
Inversion:

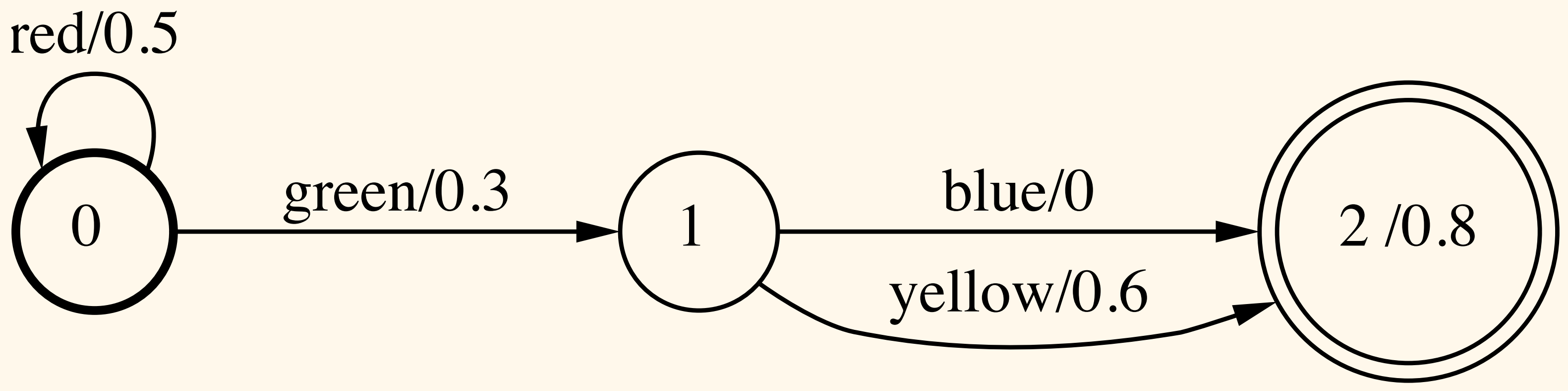




Unary wFS[AT] Operations, in more detail:

Projection:





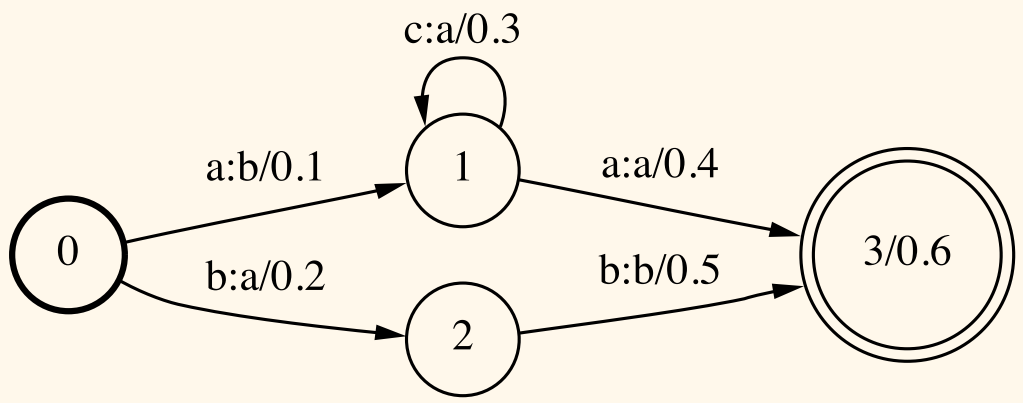
These operations can all be done in linear time with respect to the number of states and arcs...

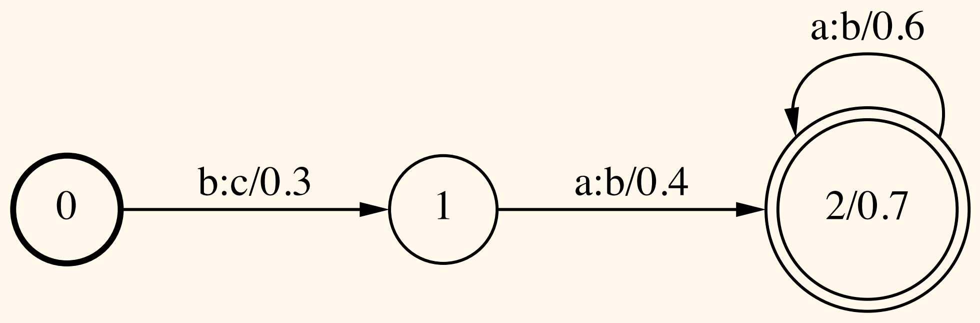
The *binary* operations are more involved.

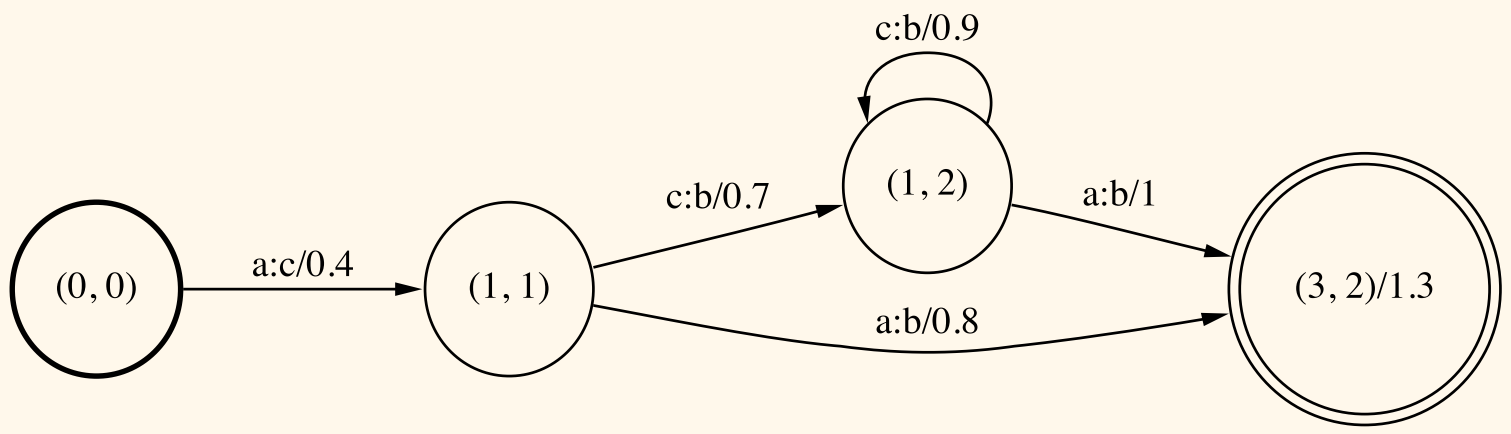
Also, some of them impose additional constraints.

Binary wFS[AT] Operations, in more detail:

Composition:



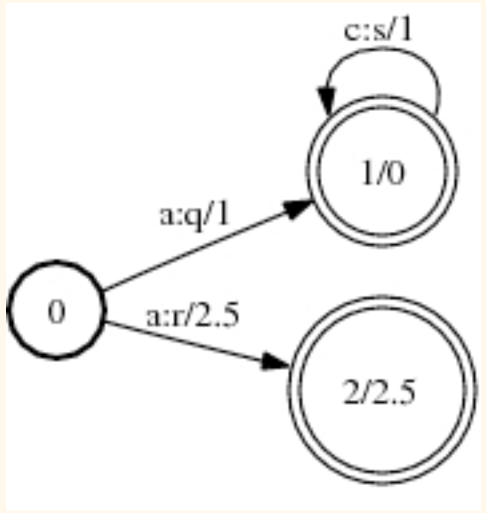


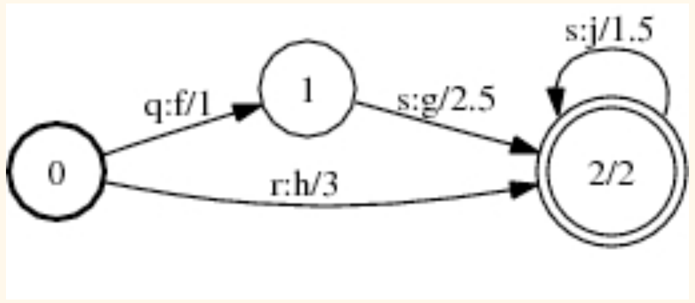


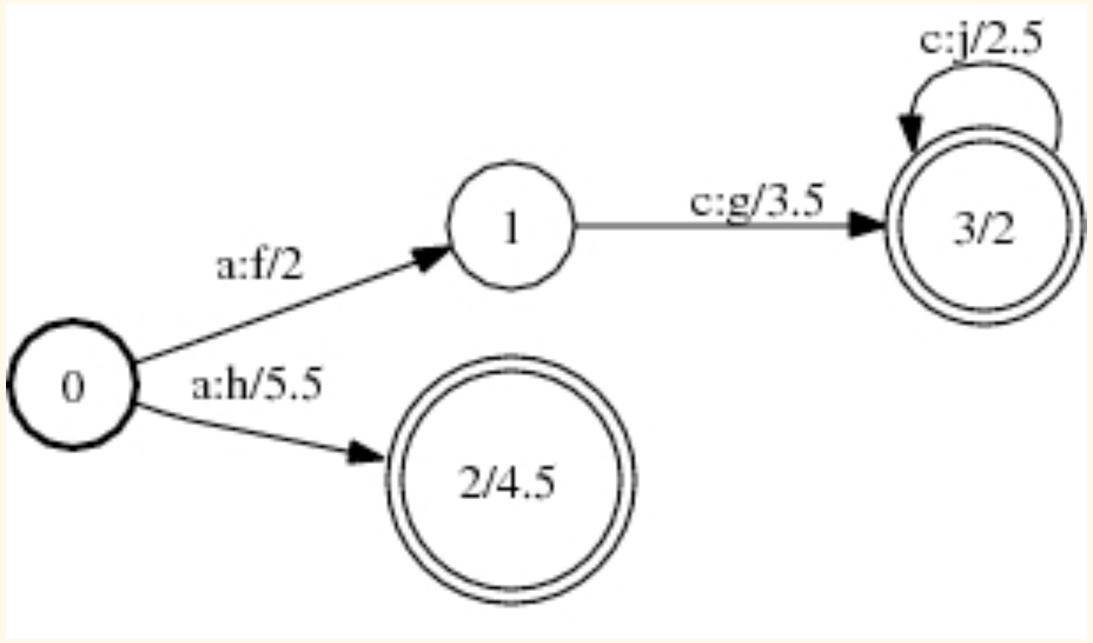
Note that composition uses the ⊗ operator to combine weights!

Binary wFS[AT] Operations, in more detail:

Composition:







http://www.openfst.org/twiki/bin/view/FST/ComposeDoc

Binary wFS[AT] Operations, in more detail:

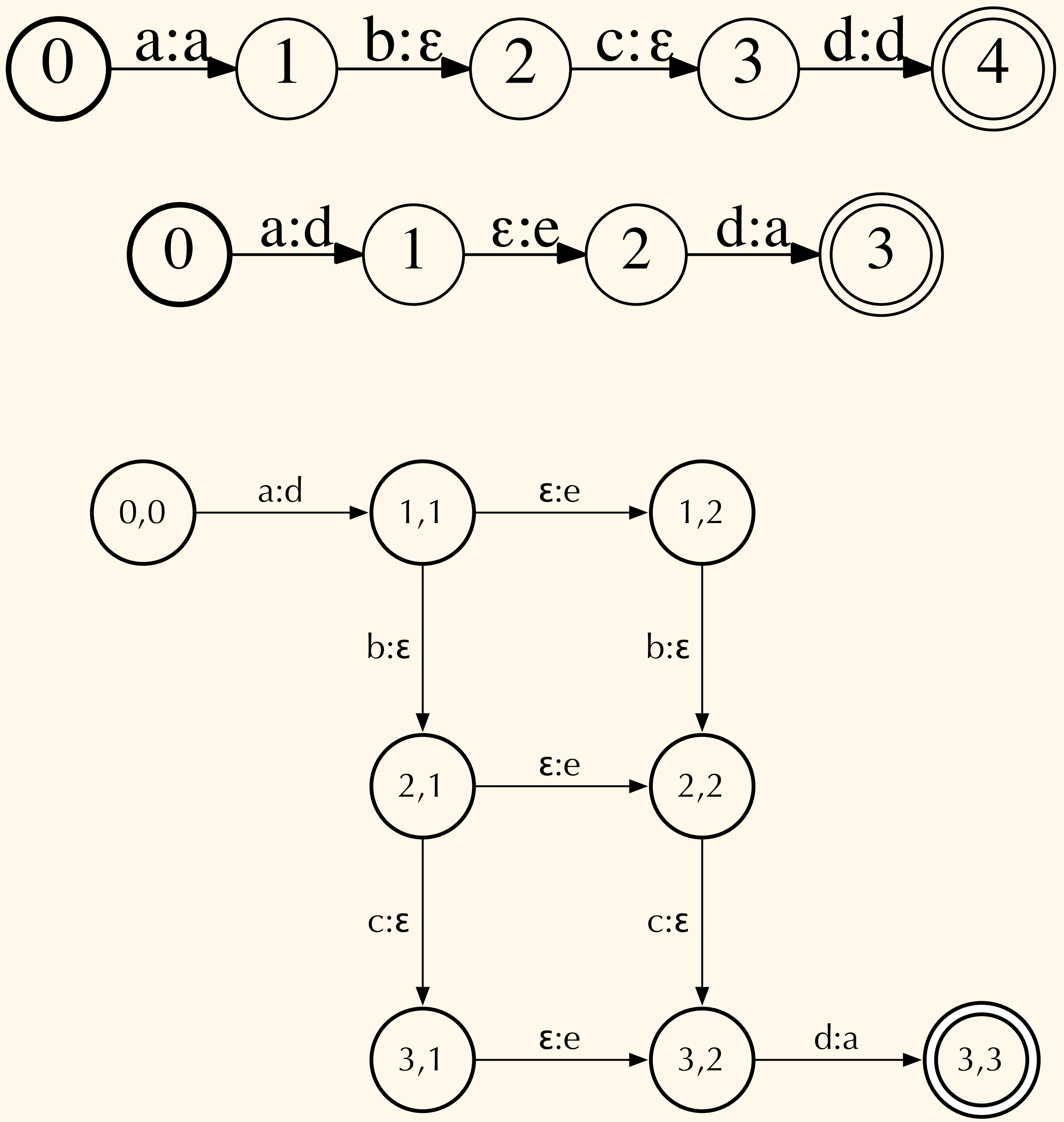
Composition Notes:

The composition algorithm is quadratic:

$$O((|E_1| + |Q_1|)(|E_2| + |Q_2|))$$

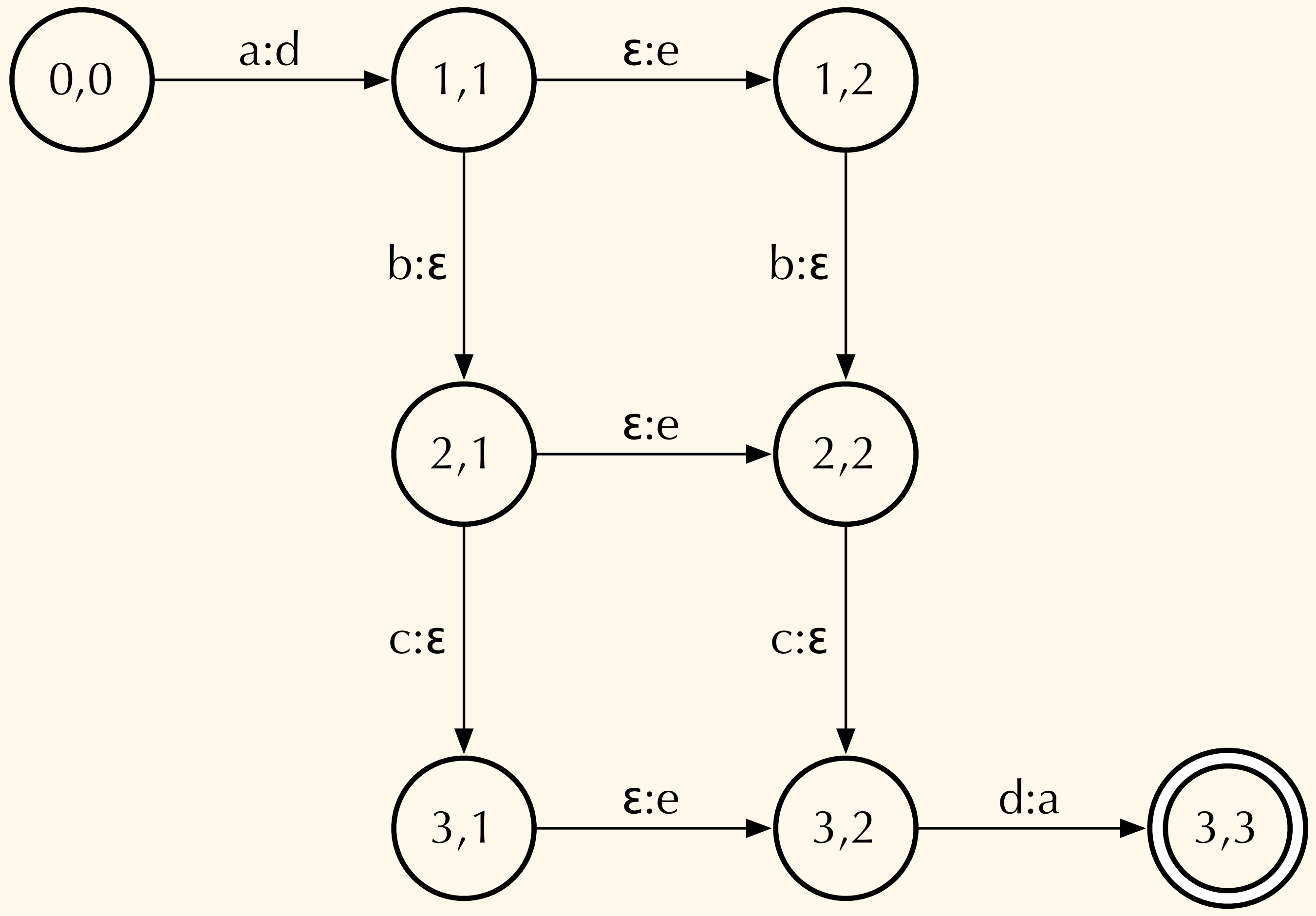
For it to work at all efficiently, outgoing arcs from each state must be stored in some sorted order!

Epsilon arcs cause serious headaches!



Note the redundant paths!

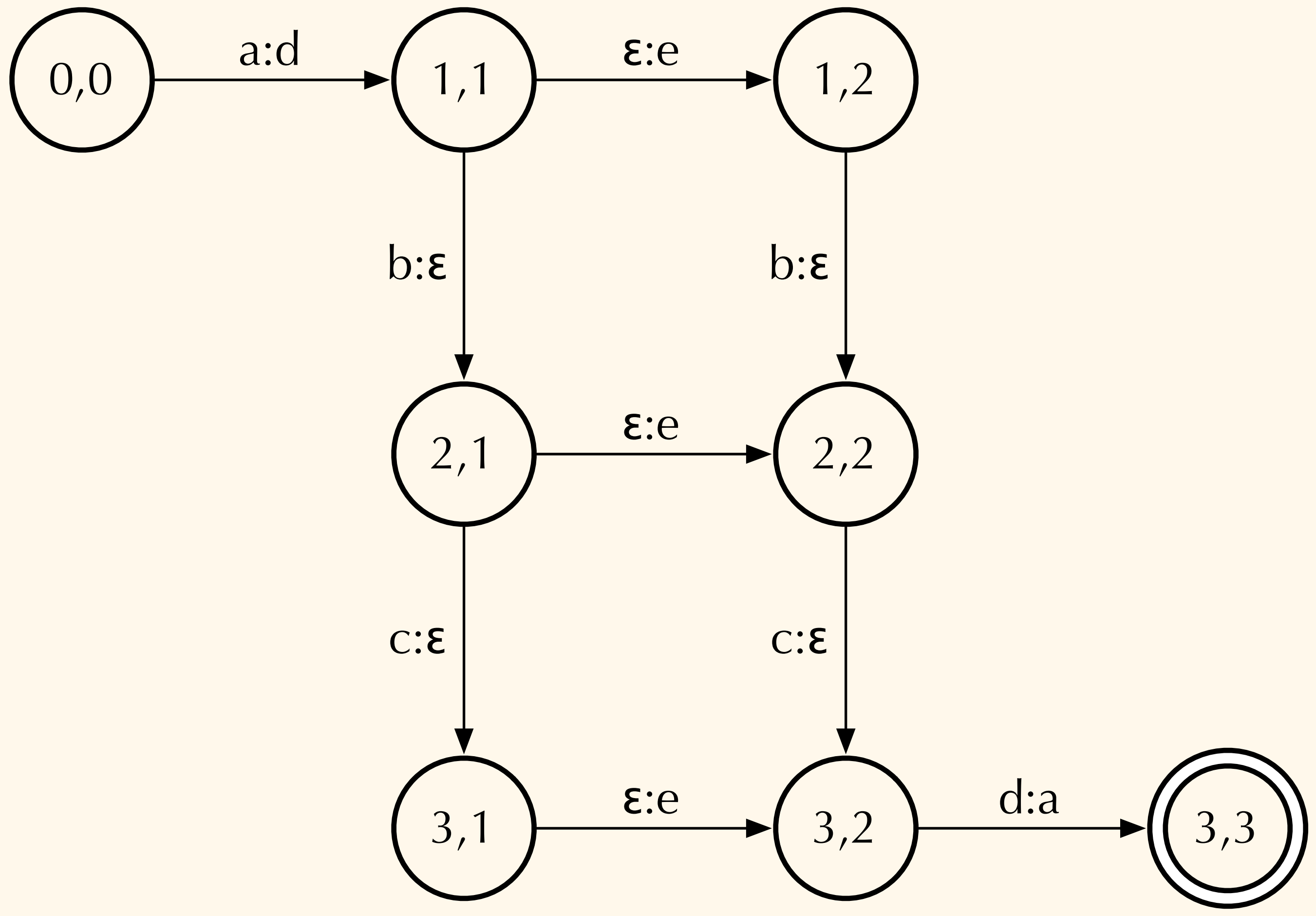
Example from Pereira & Riley's chapter in Roche & Schabes's "Finite State Language Processing"



Not only does this waste space...

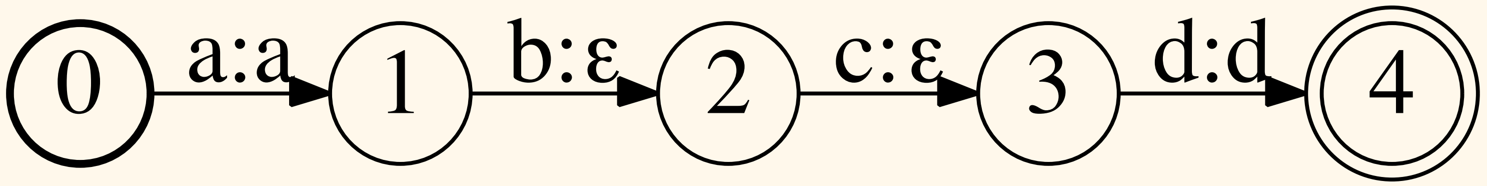
... and make future computations more complex...

... but, if there are weights, can lead to incorrect weights for paths through the machine!

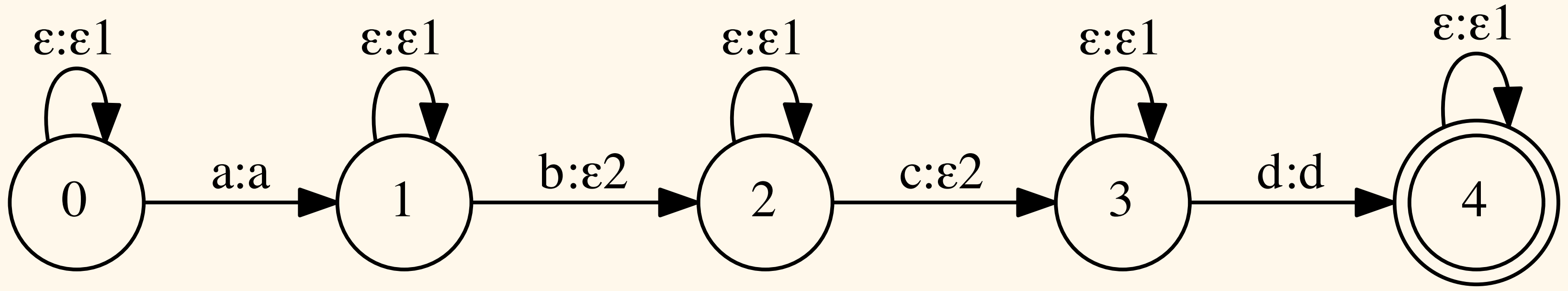


A clever technique called epsilon-filtering can help!

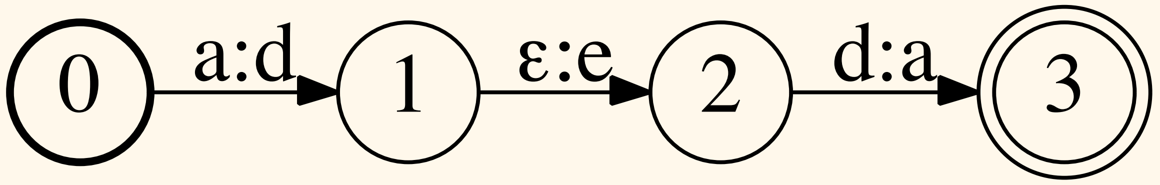
A:



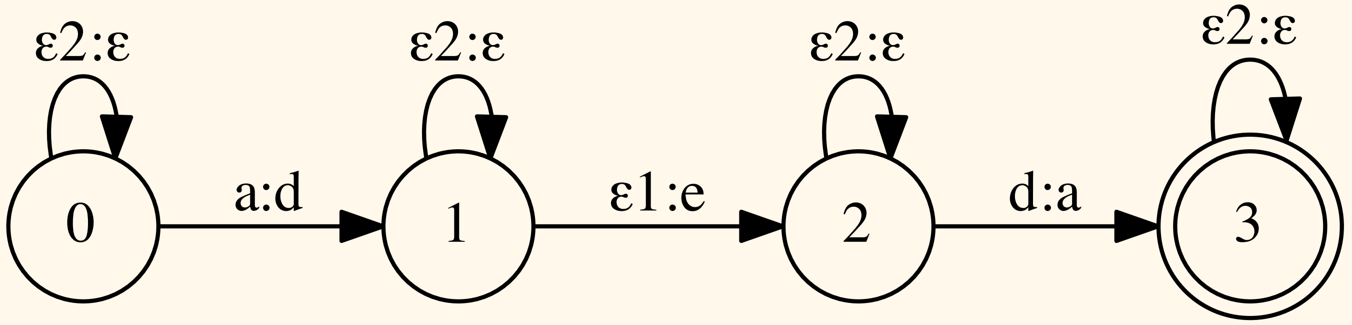
A′:



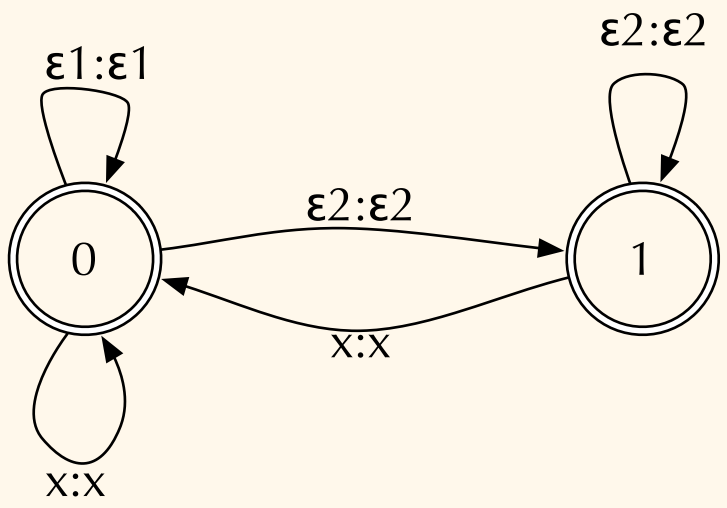
B:



B′:

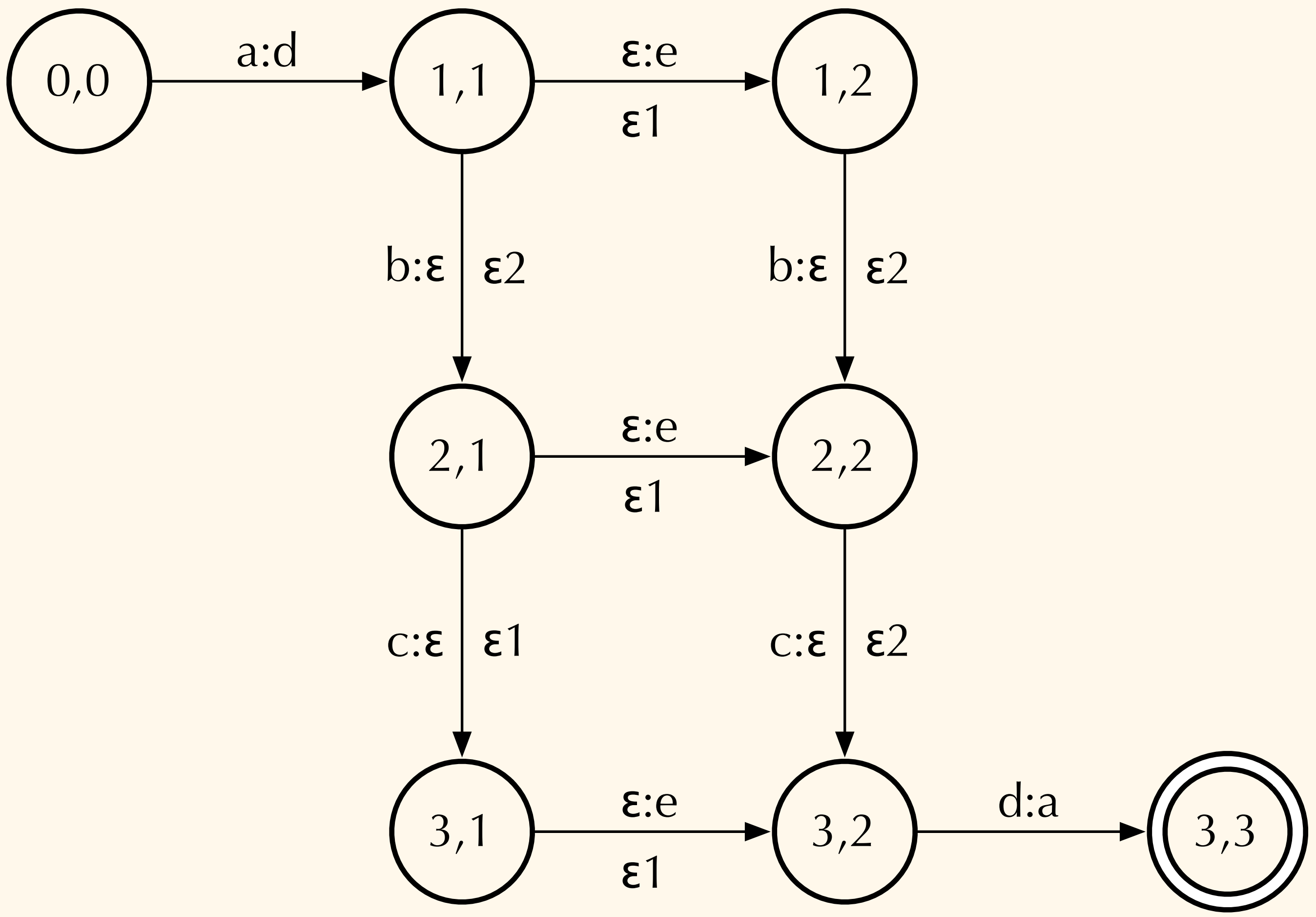


F:



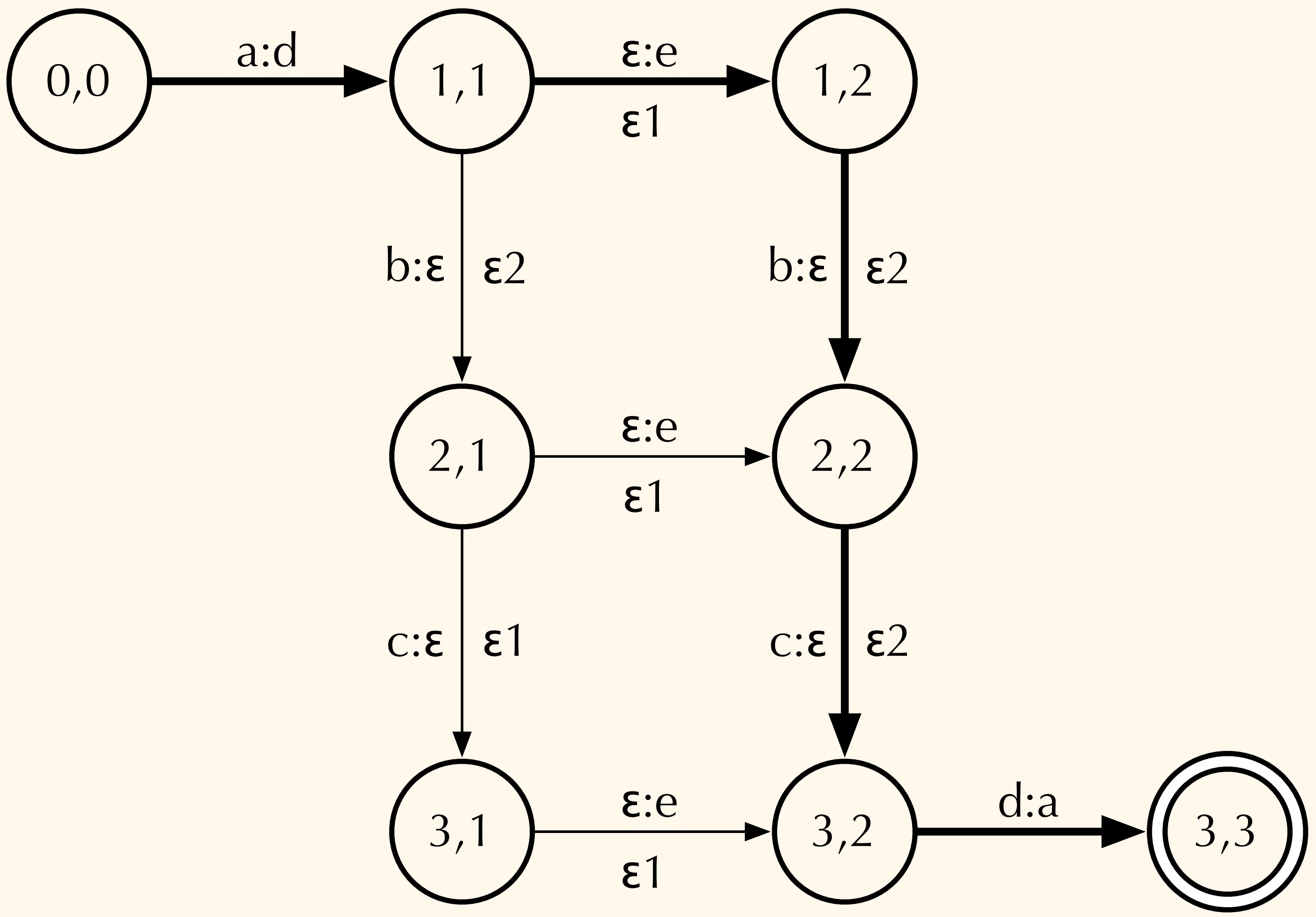
Example from Pereira & Riley's chapter in Roche & Schabes's "Finite State Language Processing"

$A' \circ F \circ B'$:



Example from Pereira & Riley's chapter in Roche & Schabes's "Finite State Language Processing"

A′ ∘ F ∘ B′:

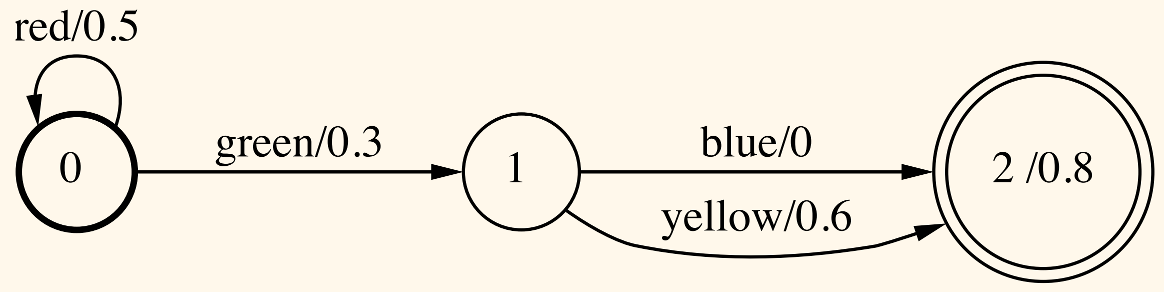


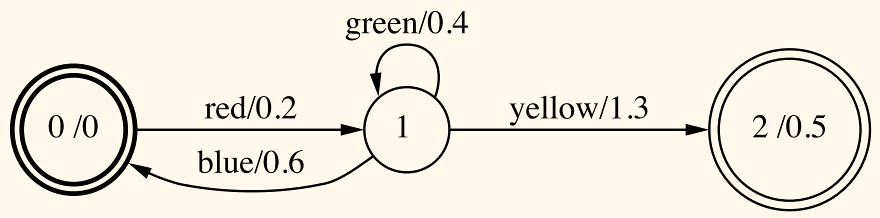
In practice, most implementations of composition do this for us...

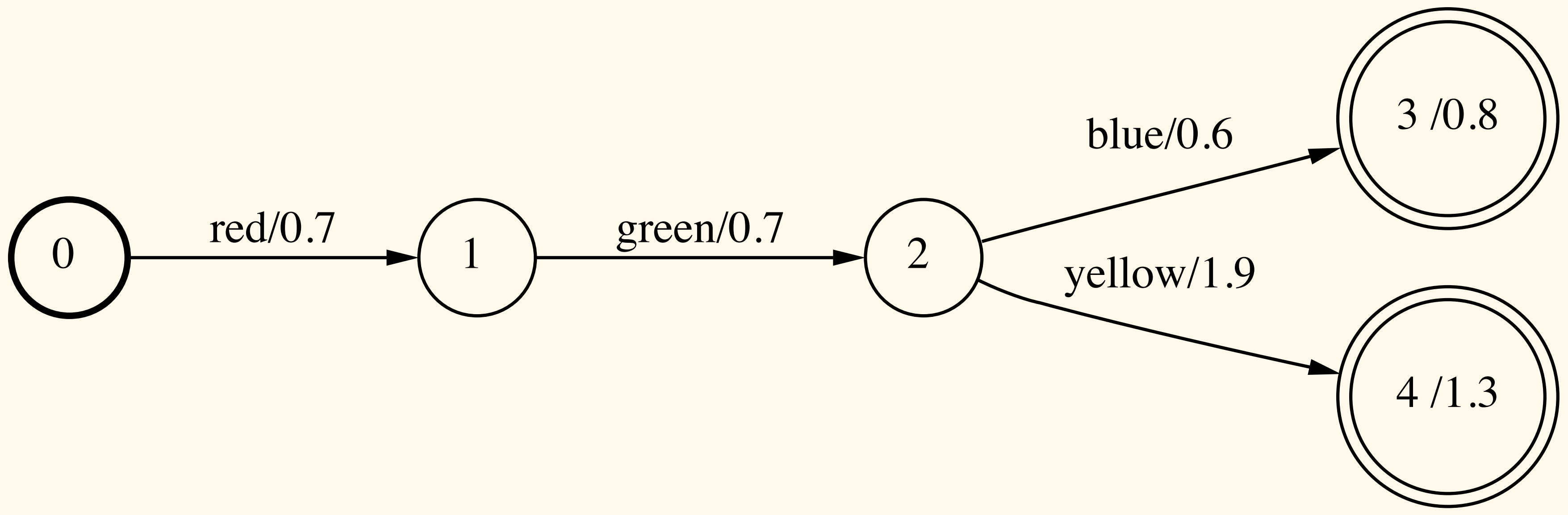
Example from Pereira & Riley's chapter in Roche & Schabes's "Finite State Language Processing"

Binary wFS[AT] Operations, in more detail:

Intersection:

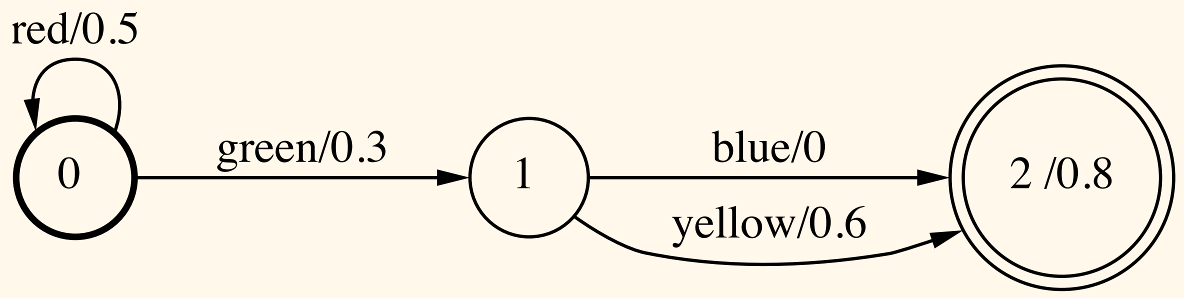


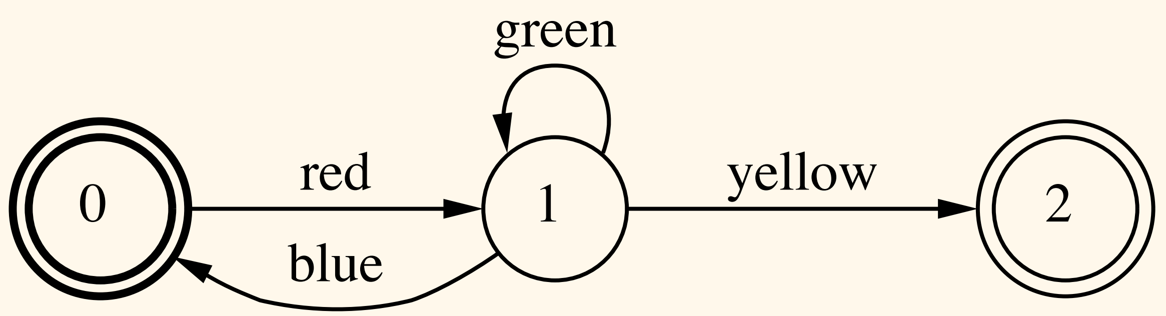


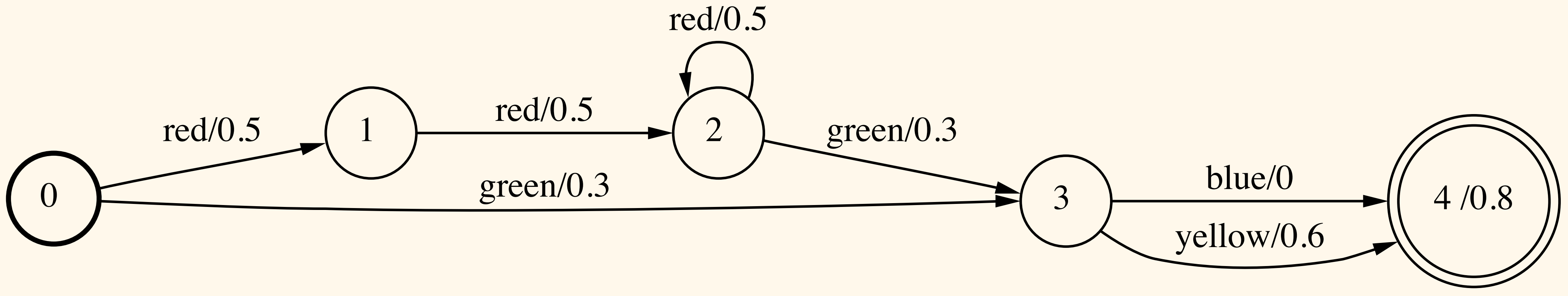


Binary wFS[AT] Operations, in more detail:

Difference:



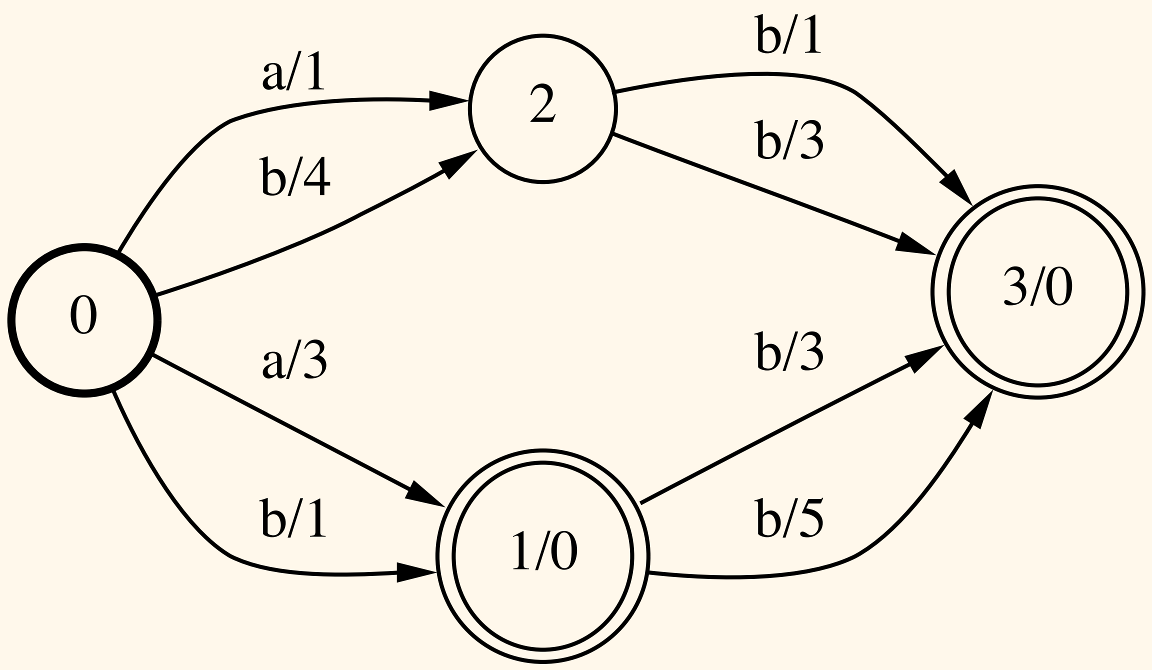


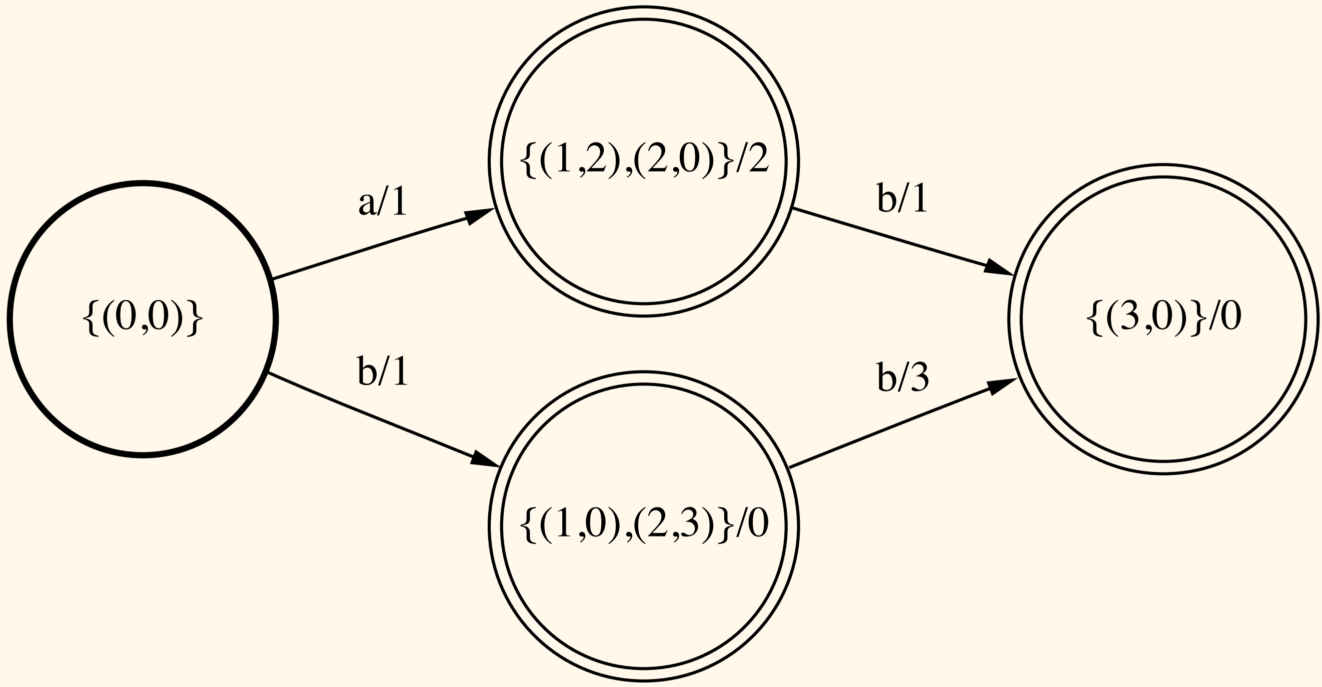


Other important operations:

ε-removal;

Determinization;





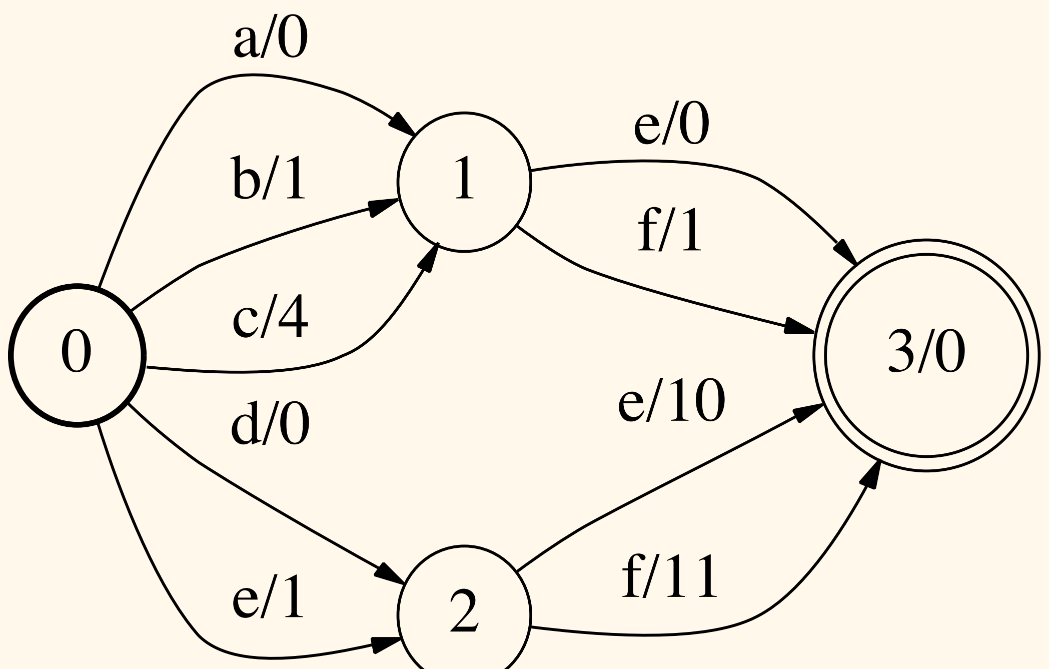
Note: Not all transducers are determinizable!

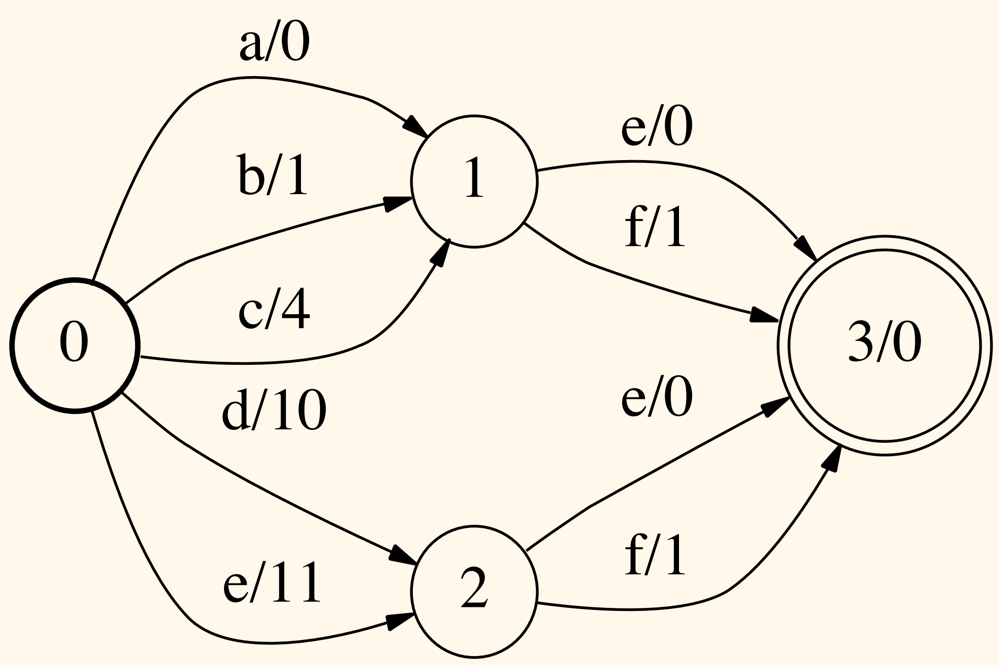
Other important operations:

ε-removal;

Determinization;

Pushing (either of weights or labels);



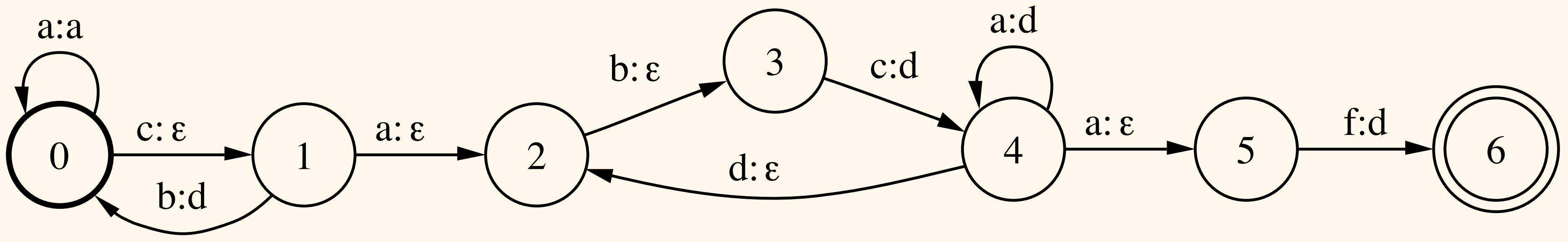


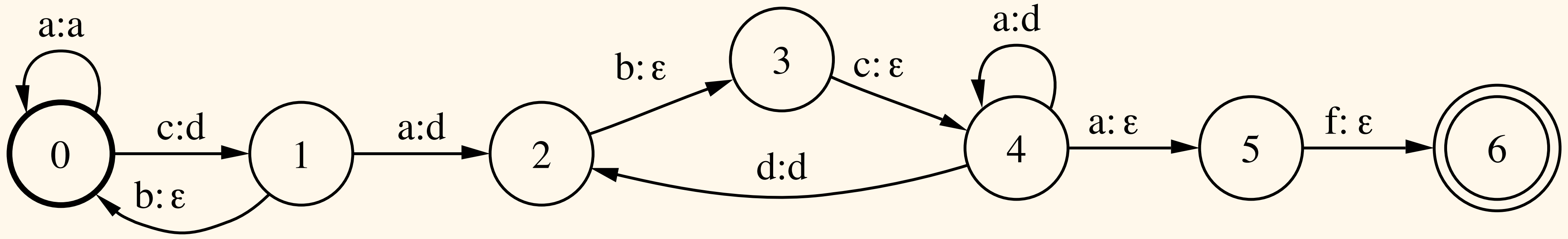
Other important operations:

ε-removal;

Determinization;

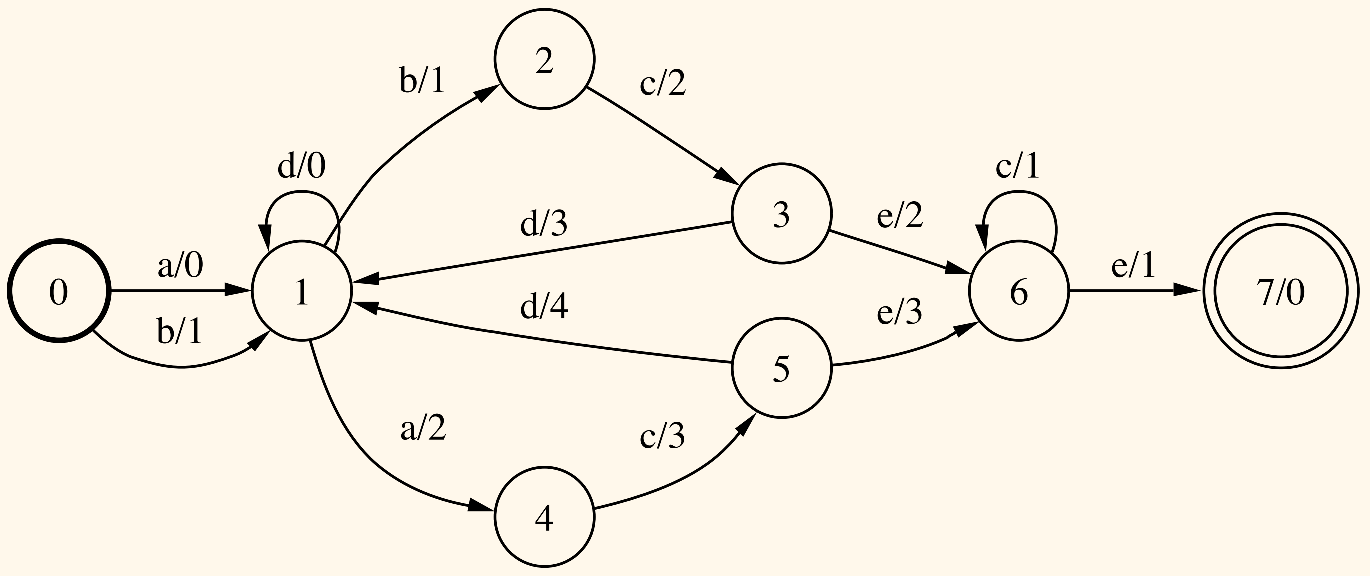
Pushing (either of weights or labels);

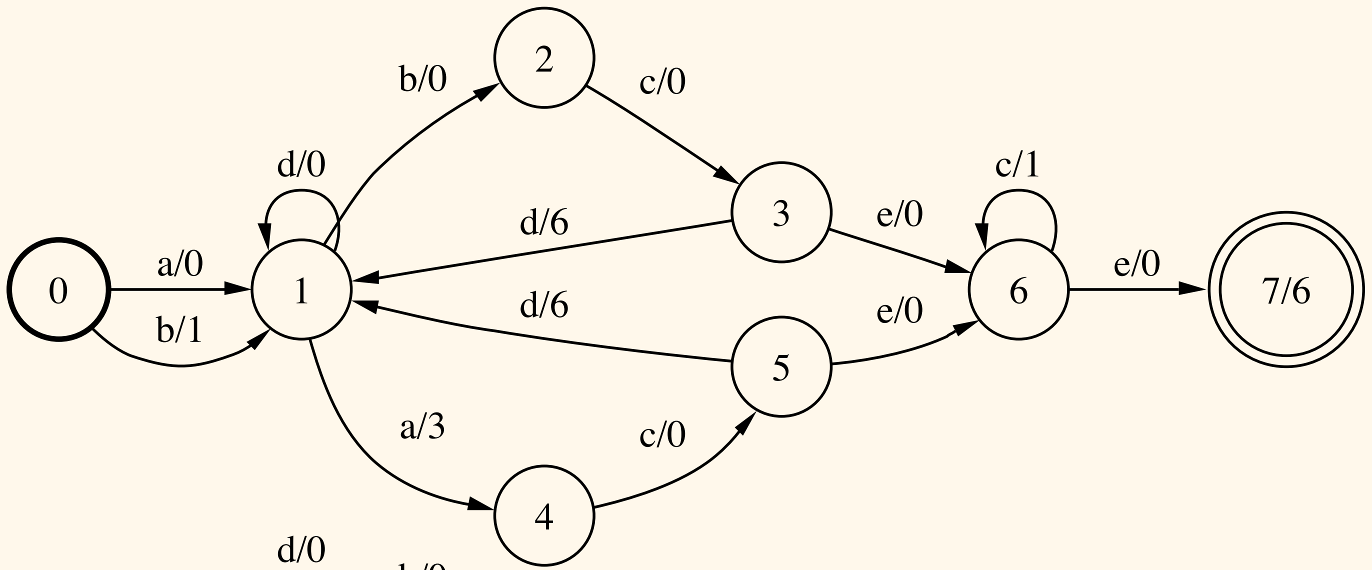


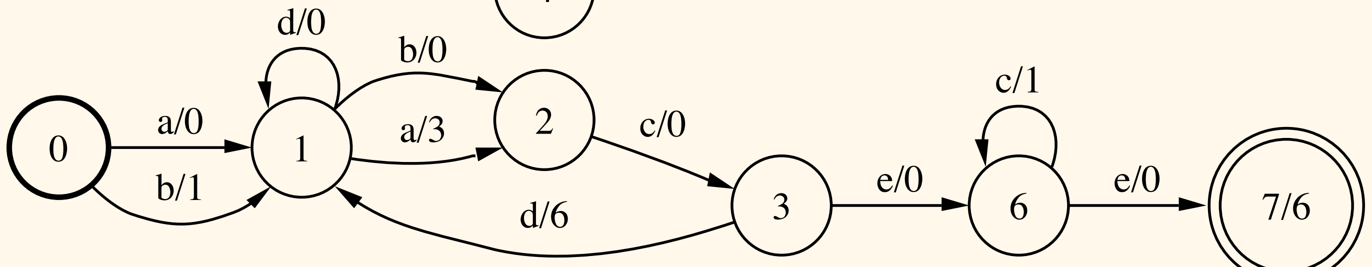


Other important operations:

Minimization:

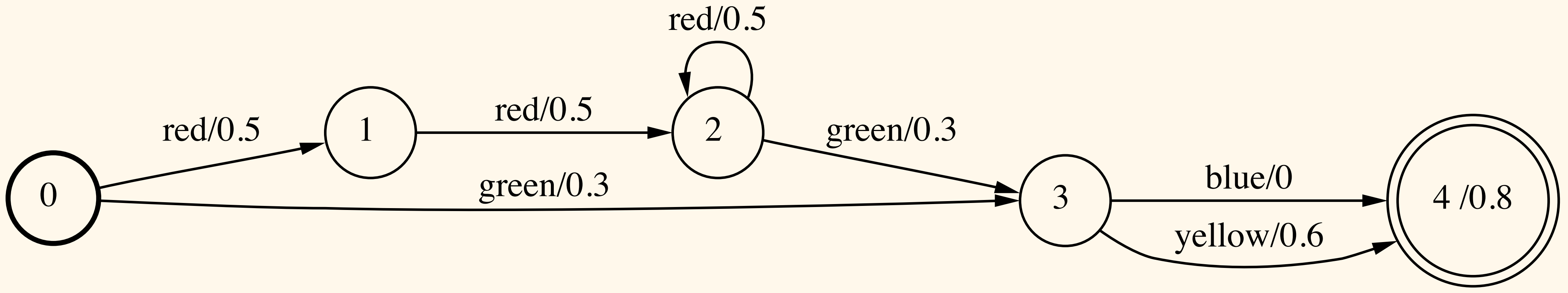


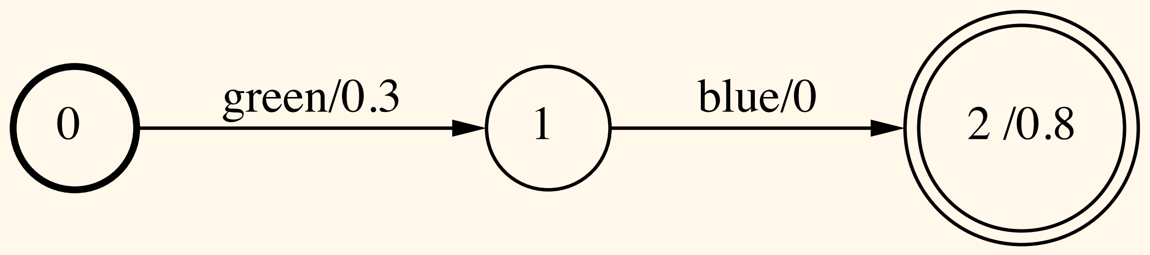




Other important operations:

Shortest path/distance:





(Semiring must have path property: $a \oplus b \in \{a, b\}$)

Do we have time for a demonstration?

1. Installing OpenFST

2. Solving shortest-path problems