FSTs: Beyond acceptance

Computational Linguistics (LING 455)

Rutgers University Oct 29 & Nov 2, 2021

Mini-project: a quick reminder

Homework	**75%**
Final mini-project	**20%**
▷ Proposal	5%
▷ Presentation	5%
▷ Project	10%
Class involvement	**5%**

Two options for the final mini-project:

- A program written in Haskell that builds on one of the techniques that we learned in the class.
- A research paper of $\geqslant$5 pages on some issue in computational linguistics, either theoretical or applied.

Mini-project

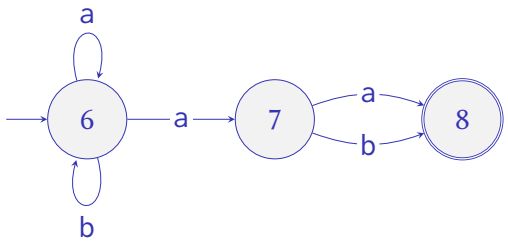
The mini-project is not supposed to be huge: roughly the 'size' of two homework assignments is a reasonable goal to aim for.

The mini-project has 3 parts: a proposal of 250–500 words (Week 11), a short presentation of 5–10 minutes (Weeks 14 & 15), and the project itself (end of the course).

- The proposal is due **Nov 12** (2 weeks from today)
- I will post suggestions for possible topics over the weekend

FSTs

Review: FSAs




```
step :: Eq a => [Transition a] -> a -> State -> [State]
step delta x q = [ r' | (r,y,r') <- delta, q==r, y==x ]
```

```
walk :: Eq a => [Transition a] -> [a] -> State -> [State]
walk delta str q = case str of
  []   -> [q]
  x:xs -> let oneStep = step delta x q
          in  oneStep >>= \q' -> walk delta xs q'
```

```
parseFSA :: Eq a => FSA a -> [a] -> Bool
parseFSA (_,_,i,f,delta) str = isAccepting walkFromi
  where walkFromi      = i >>= \q' -> walk delta str q'
        isAccepting qs = or [ elem qn f | qn <- qs ]
```

Review: FSAs

Finite state **acceptors**:

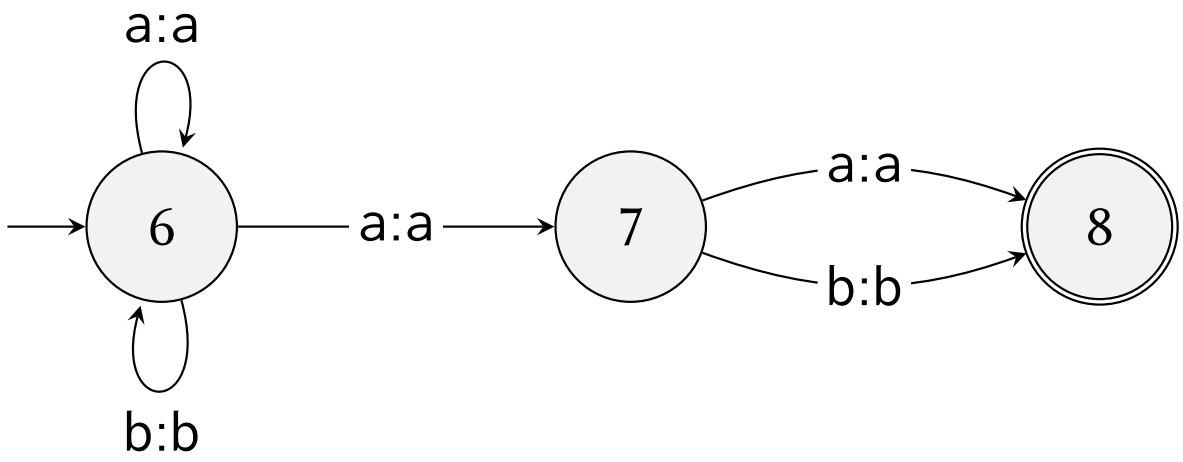
- Tell us whether a string of symbols is accepted or not. Yes or no.
- . . . And nothing else. We don't receive any output, or know anything about *how* a string was parsed (maybe there are multiple ways to an accepting state).

Other kinds of information can be useful. We might wish. . .

- For output giving info on the path walked
- To change a string into another string
- To calculate probabilities
- To minimize costs

FSTs over Σ, M

Transition arrows have labels $x{:}u$, where $x \in \Sigma$ and $u \in M^*$: reading a symbol x produces some output u.

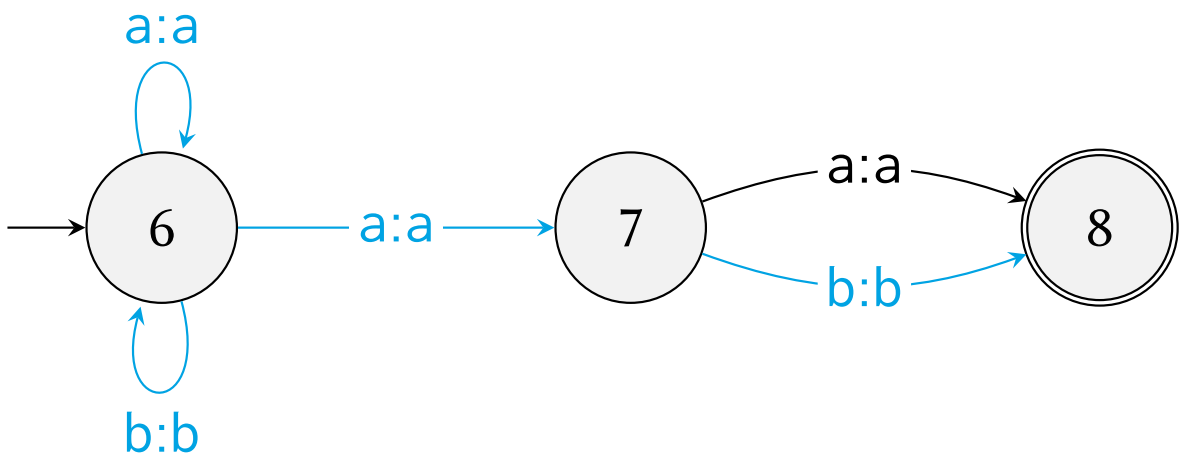


How does this machine parse abab?

Here, Σ and M are both $\{\mathsf{a}, \mathsf{b}\}$, but they can be different alphabets (or Haskell types).

FSTs over Σ, M

Transition arrows have labels $x{:}u$, where $x \in \Sigma$ and $u \in M^*$: reading a symbol x produces some output u.

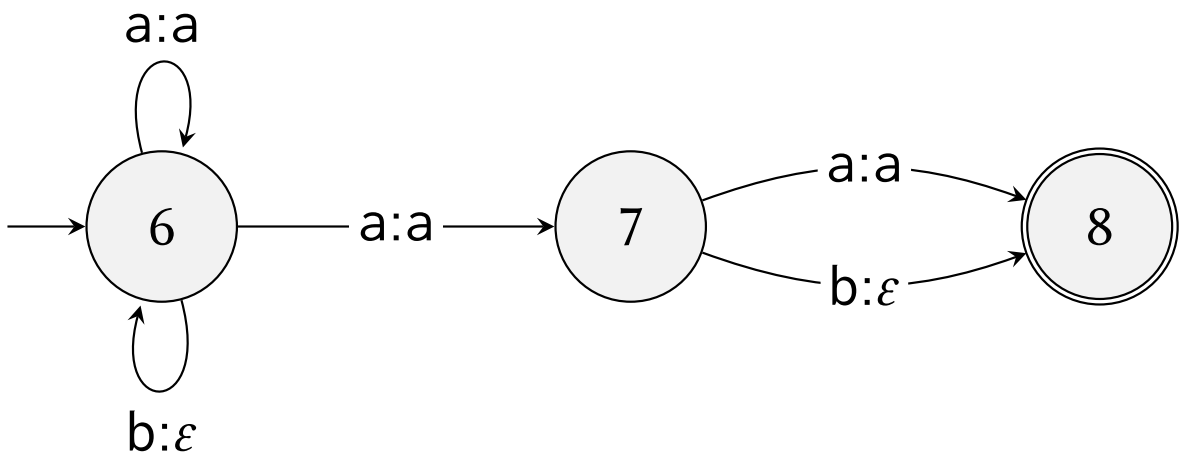


How does this machine parse abab? abab

Here, Σ and M are both $\{\mathsf{a}, \mathsf{b}\}$, but they can be different alphabets (or Haskell types).

FSTs over Σ, M

Transition arrows have labels $x{:}u$, where $x \in \Sigma$ and $u \in M^*$: reading a symbol x produces some output u.

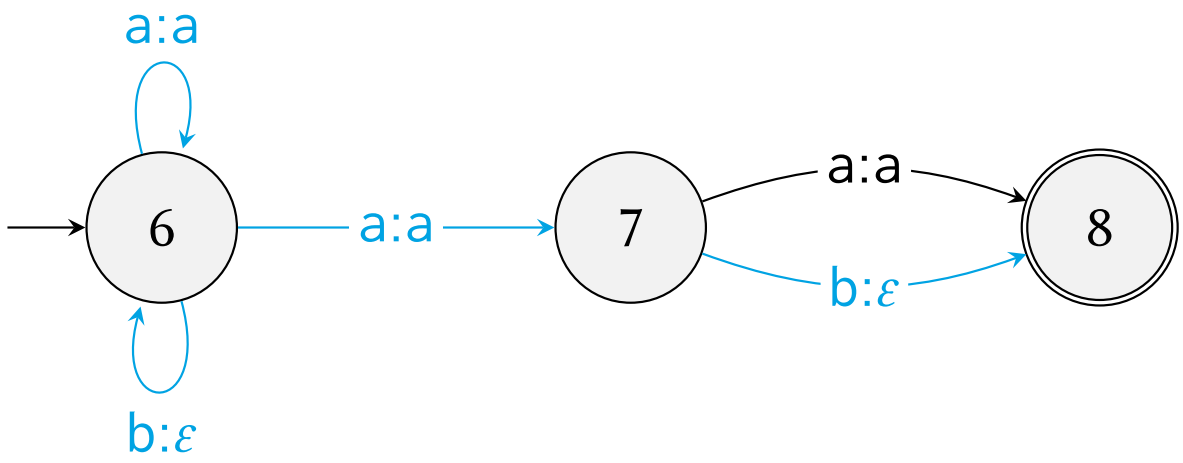


How does this machine parse abab?

Here, Σ and M are both $\{\mathsf{a}, \mathsf{b}\}$, but they can be different alphabets (or Haskell types).

FSTs over Σ, M

Transition arrows have labels $x{:}u$, where $x \in \Sigma$ and $u \in M^*$: reading a symbol x produces some output u.

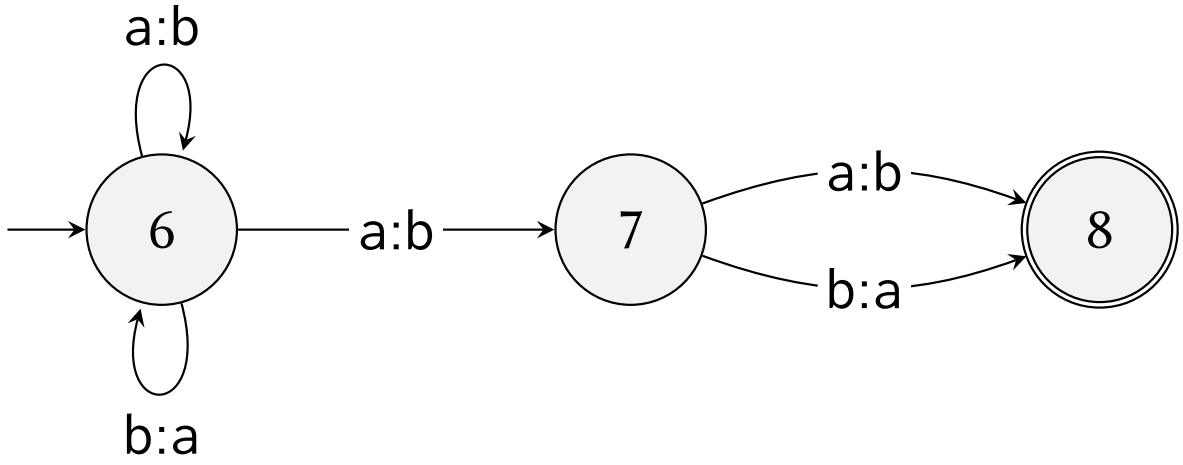


How does this machine parse abab? aa

Here, Σ and M are both $\{\mathsf{a}, \mathsf{b}\}$, but they can be different alphabets (or Haskell types).

FSTs over Σ, M

Transition arrows have labels $x{:}u$, where $x \in \Sigma$ and $u \in M^*$: reading a symbol x produces some output u.

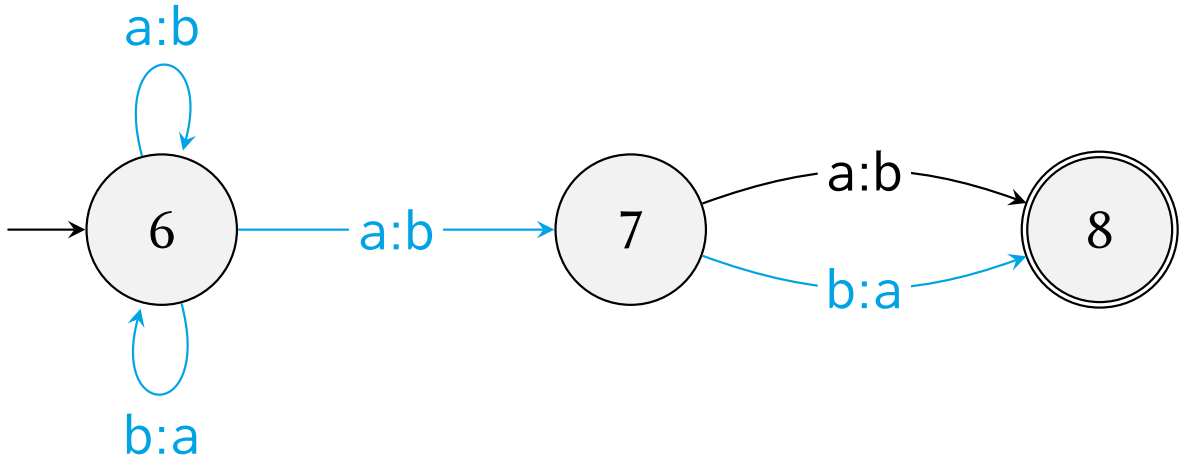


How does this machine parse abab?

Here, Σ and M are both $\{\mathsf{a}, \mathsf{b}\}$, but they can be different alphabets (or Haskell types).

FSTs over Σ, M

Transition arrows have labels $x{:}u$, where $x \in \Sigma$ and $u \in M^*$: reading a symbol x produces some output u.



How does this machine parse abab? baba

Here, Σ and M are both $\{\mathsf{a}, \mathsf{b}\}$, but they can be different alphabets (or Haskell types).

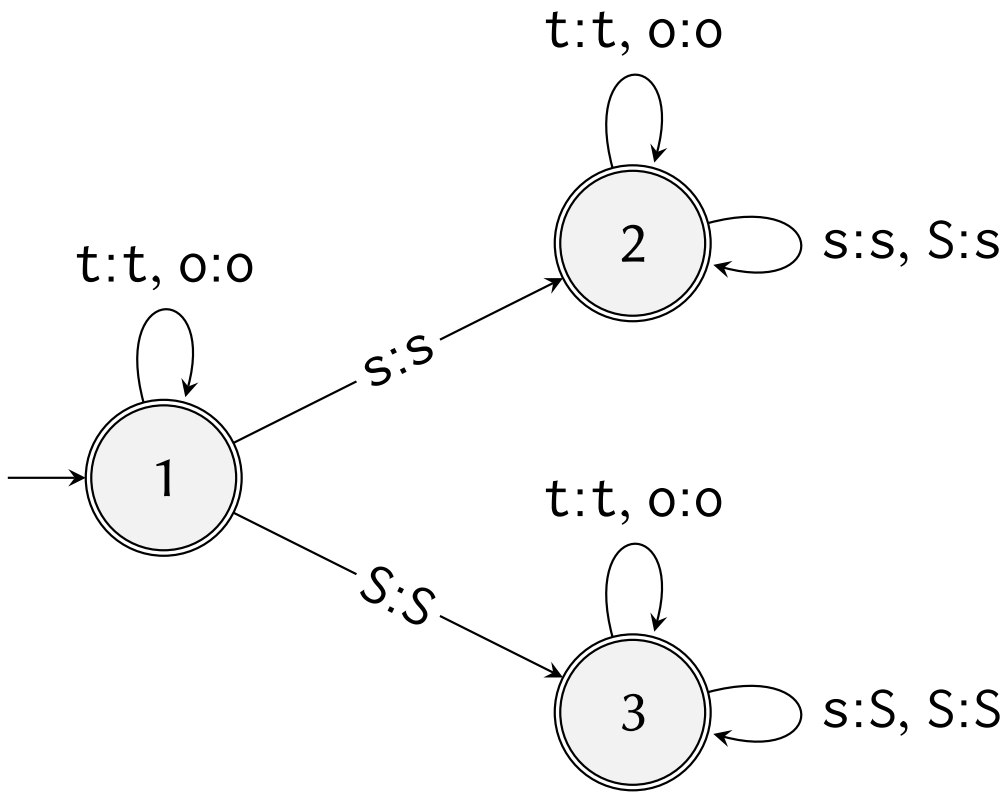
String alterations

In both theoretical and computational linguistic applications, we often want to change strings into other strings.

tacks	/tæk+z/ → [tæks]
dogs	/dɔg+z/ → [dɔgz]
passes	/pæs+z/ → [pæsəz]

In general, **phonological rules** of the sort you've studied in other ling courses are alterations in strings of phonological symbols.

Example: progressive sibilant harmony (Heinz 2017)

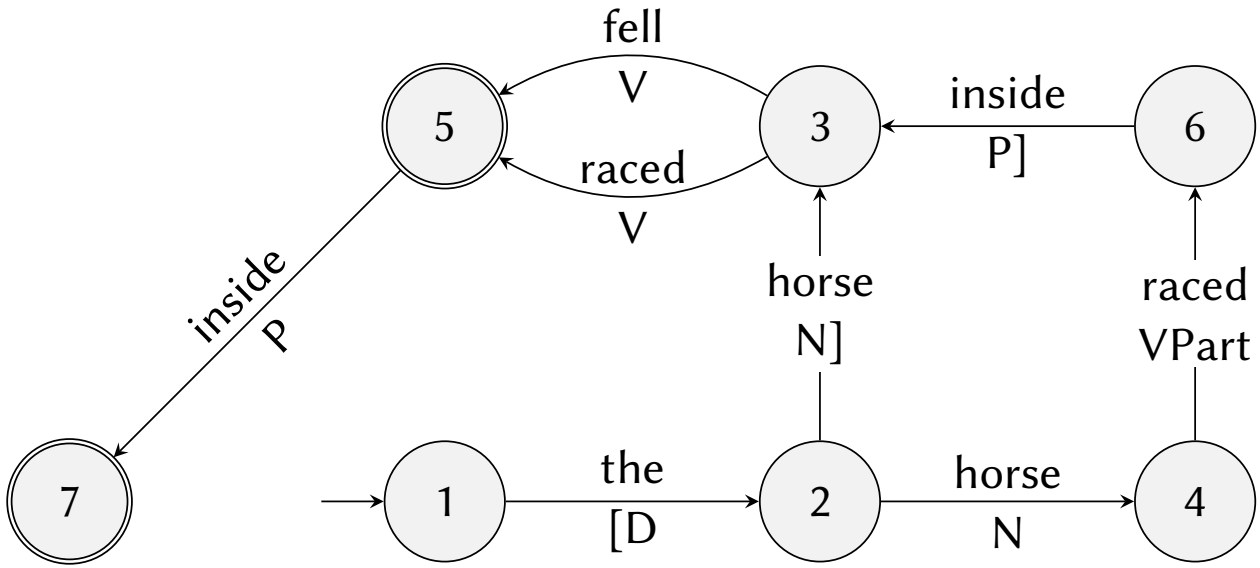


Other examples

Transducers may be used for:

- Part of speech tagging. What is Σ? What is M?
- Morphological parsing. E.g., taking cats to CAT-PL.
- Some syntactic parsing (recalling the limitations of finite state machines with respect to natural language syntax).

Syntactic parsing



- the horse raced inside → [D N] V P
- the horse raced inside fell → [D N VPart P] V

FSTs in Haskell

FST types

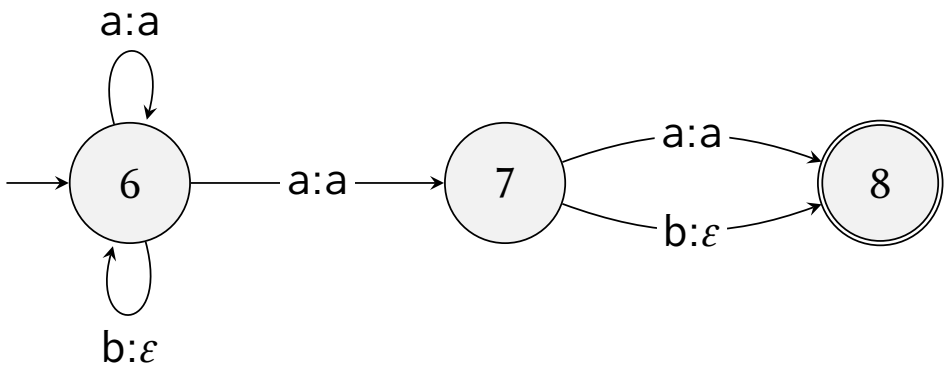
Technically, FSTs are represented as $(Q, \Sigma, M, I, F, \Delta)$. This is all as with FSAs, except for the following:

- M is the output alphabet
- Δ is now a transition arrow with two labels: the $x \in \Sigma$ being read by the machine, and the $u \in M^*$ yielded as an output.

We work with a simplified representation of FSTs, leaving Q implicit, and using `m` to indicate M^* directly.

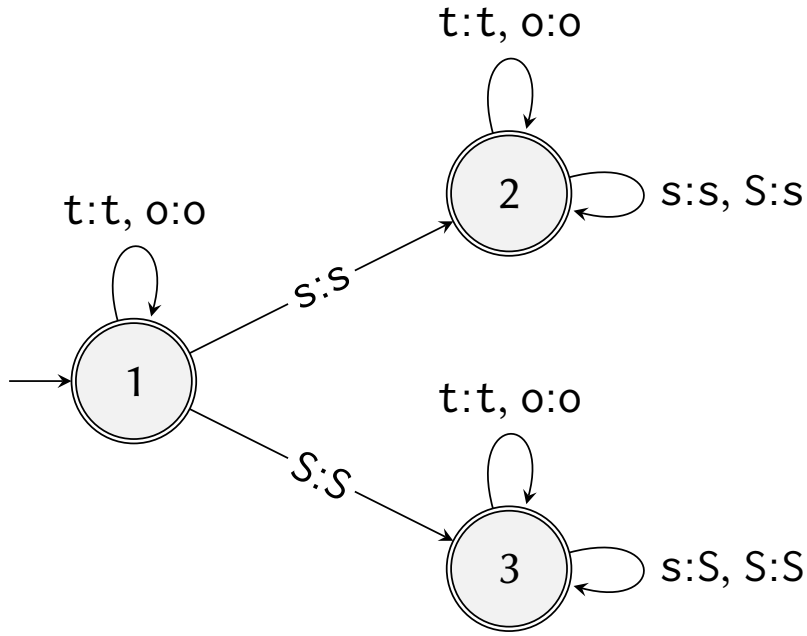
```
type State = Int
type FST s m = ( [State]         -- I
               , [State]         -- F
               , [Arrow s m] ) -- Δ
type Arrow s m = (State, s, m, State)
```

FST example



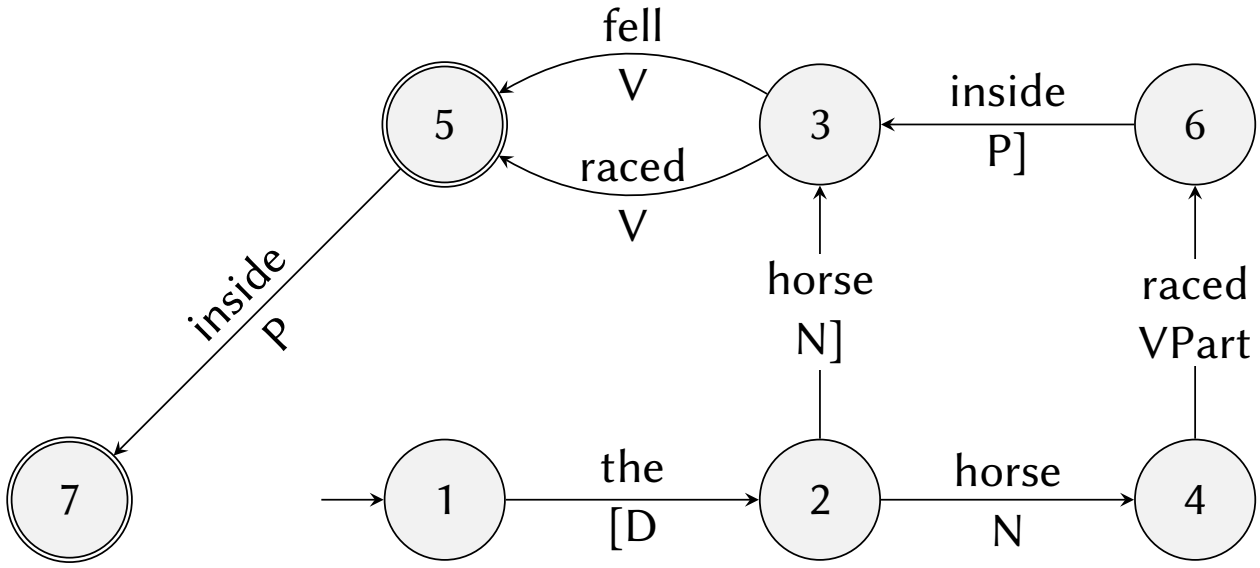

```
fstPenAWriteAs :: FST Char String
fstPenAWriteAs =
  ( [6], [8], -- I, F
    [ (6,'a',"a",6), (6,'b',"",6), (6,'a',"a",7),
      (7,'a',"a",8), (7,'b',"",8) ] )
```

Another example




```
fstSibHarm :: FST Char String
fstSibHarm =
  ( [1], [1,2,3],
    [ (1,'t',"t",1), (1,'o',"o",1), (1,'s',"s",2),
      (1,'S',"S",3), (2,'t',"t",2), (2,'o',"o",2),
      (2,'s',"s",2), (2,'S',"s",2), (3,'t',"t",3),
      (3,'o',"o",3), (3,'s',"S",3), (3,'S',"S",3) ] )
```

One more




```
fstGardenPath :: FST String String
fstGardenPath =
 ( [1], [5,7],
   [ (1,"the","[D",2),       (2,"horse","N",4),
     (4,"raced","VPart",6), (2,"horse","N]",3),
     (6,"inside","P]",3),    (3,"fell","V",5),
     (3,"raced","V",5),      (5,"inside","P",7) ] )
```

Every journey begins with one step

Let's consider how to change the `step` function to deal with FSTs...

```
step :: Eq c => [Transition c] -> c -> State -> [State]
step delta x q = [ r' | (r,y,r') <- delta, q==r, y==x ]
```

At any stage in FST parsing, what do we know?

Every journey begins with one step

Let's consider how to change the `step` function to deal with FSTs...

```
step :: Eq c => [Transition c] -> c -> State -> [State]
step delta x q = [ r' | (r,y,r') <- delta, q==r, y==x ]
```

At any stage in FST parsing, what do we know?

- The possible transitions, each associated with some new output
- The next character to parse
- The current state, plus the output accumulated so far

As with FSAs, we must find arrows whose source and symbol match the current source and symbol. New: that step comes with some new output, which we **tack onto** the output accumulated thusfar.

FST steps

FSA steps: next states based on the transition matrix, current state, and next character...

```
step :: Eq c => [Transition c] -> c -> State -> [State]
step delta x q = [ r' | (r,y,r') <- delta, q==r, y==x ]
```

FST steps: next states based on transition matrix, current state, and next character; accumulated output `m` combined with new output `n`...

```
step :: Eq c => [Arrow c String] -> c ->
                (State, String) -> [(State, String)]
step delta x (q,m) = [ (r',m<>n) | (r,y,n,r') <- delta,
                                   q==r, y==x ]
```

I use `<>` as a synonym for `++`. The type's provisional as well. We'll see why on Friday.

From steps to walks

Here is our old `walk` function. What about it needs to change?

```
walk delta str p = case str of
  []   -> [p]
  x:xs -> let oneStep = step delta x p
          in  oneStep >>= \p' -> walk delta xs p'
```

From steps to walks

Here is our old `walk` function. What about it needs to change?

```
walk delta str p = case str of
  []   -> [p]
  x:xs -> let oneStep = step delta x p
          in  oneStep >>= \p' -> walk delta xs p'
```

Nothing! This function is generic enough to work for FSTs just the same as it works for FSAs. Provisionally, its type is now:

```
-- walk :: Eq c => [Arrow c String] -> [c] ->
--                 (State, String) -> [(State, String)]
```

Transduction

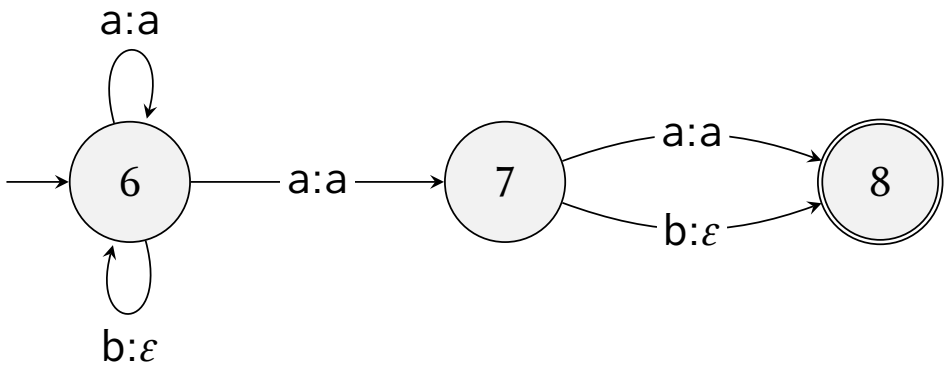
```haskell
transduce :: Eq c => FST c String -> [s] -> [String]
transduce (i, f, delta) str = outputs
  where
    forward = i >>= \q -> walk delta str (q, mempty)
    outputs = [ t | (qn, t) <- forward, elem qn f ]
```

Transduction means starting from from an initial state, with no accumulated output, and talking a walk to parse your `str`.

- Once parsed forward, you'll have some outputs. Keep only those that are associated with a final state.

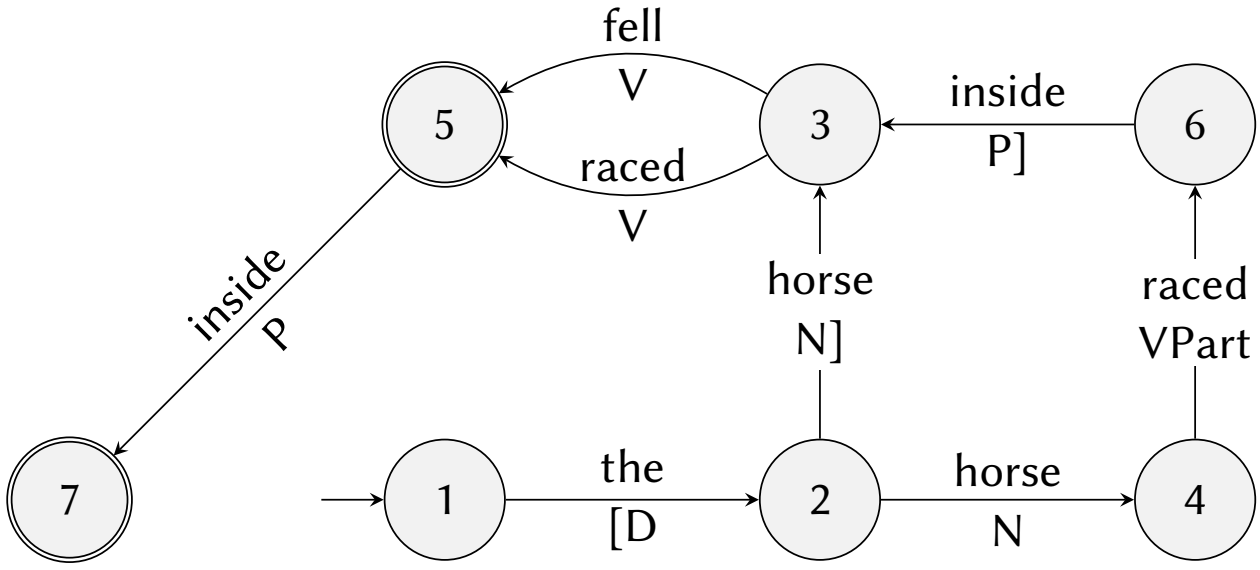
`mempty` in this context functions as a synonym for `""`. Again, stay tuned.

A toy transduction



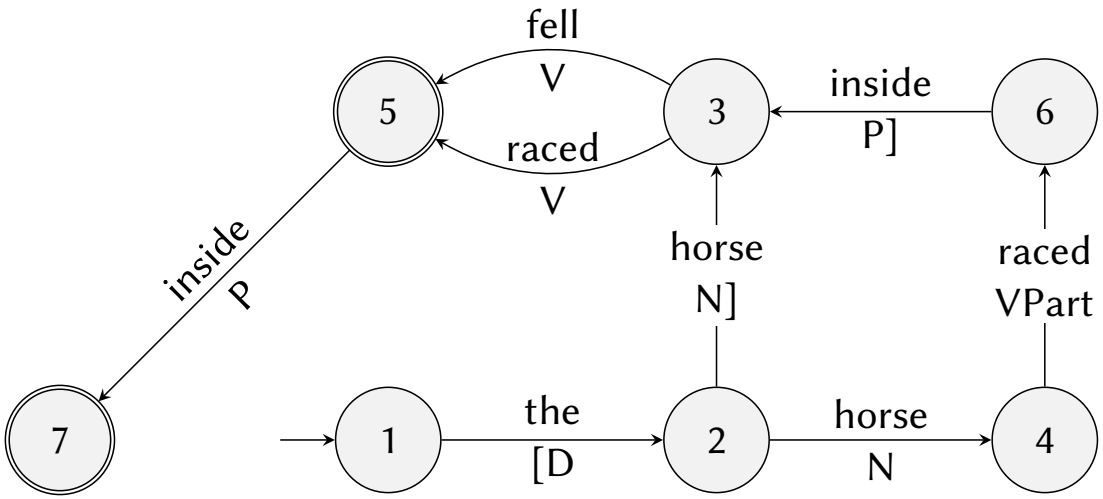

```
*W9> transduce fstPenAWriteAs "abbababab"
["aaaa"]
*W9> transduce fstPenAWriteAs "abbababbb"
[]
```

A syntactic transduction




```
*W9> s1 = words "the horse raced inside"
*W9> transduce fstGardenPath s1
["[DN]VP"]
*W9> s2 = words "the horse raced inside fell"
*W9> transduce fstGardenPath s2
["[DNVPartP]V"]
```

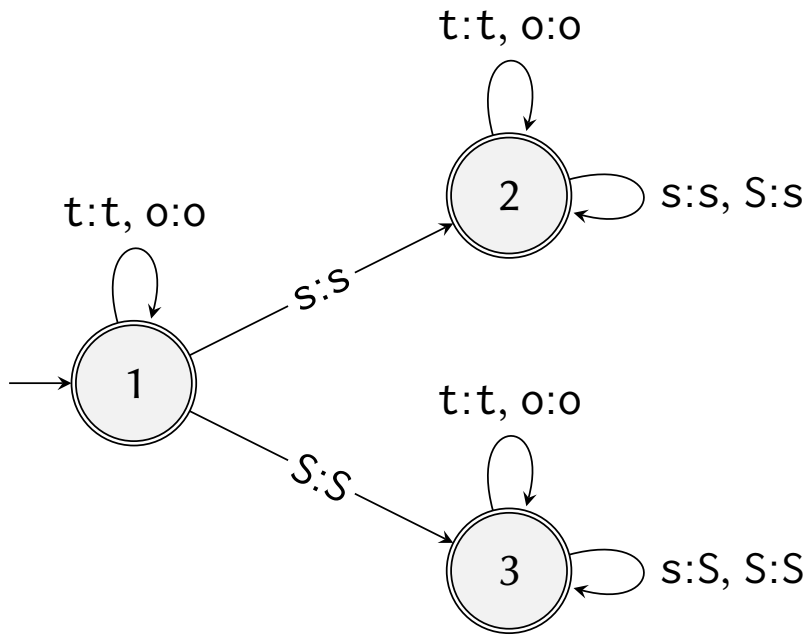
Incremental ambiguity resolution




```
*W9> (i,_,delta) = fstGardenPath
*W9> inc = words "the horse raced inside"
*W9> i >>= \q -> walk delta inc (q,"")
[(3,"[DNVPartP]"),(7,"[DN]VP")]
```

The ambiguity is eventually resolved in our examples. How might we write a FST for ambiguous strings like “I saw an elk with a telescope”.

Progressive sibilant harmony




```
*W9> transduce fstSibHarm "sototoSoS"
["sototosos"]
*W9> transduce fstSibHarm "otSottosososo"
["otSottoSoSoSo"]
```

Regressive sib. harmony

More often, later sibilants condition the realization of earlier ones. Here is data from Samala Chumash (Applegate 1972, via Heinz 2017).

/ha-s-xintila-waʃ/	[haʃxintilawaʃ]	‘his former Indian name’
/k-su-kili-mekeken-ʃ/	[kʃuk’ilimekeketʃ]	‘I straighten myself up’
/k-su-al-puj-ʊn-ʃaʃi/	[kʃalpujatʃɨʃi]	‘I get myself wet’
/s-taja-nowon-waʃ/	[ʃtojowonowaʃ]	‘it stood upright’

How can we account for this?

Regressive sib. harmony

More often, later sibilants condition the realization of earlier ones. Here is data from Samala Chumash (Applegate 1972, via Heinz 2017).

/ha-s-xintila-waʃ/	[haʃxintilawaʃ]	‘his former Indian name’
/k-su-kili-mekeken-ʃ/	[kʃuk’ilimekeketʃ]	‘I straighten myself up’
/k-su-al-puj-ʊn-ʃaʃi/	[kʃalpujatʃiʃi]	‘I get myself wet’
/s-taja-nowon-waʃ/	[ʃtojowonowaʃ]	‘it stood upright’

How can we account for this? By processing strings **right-to-left**:

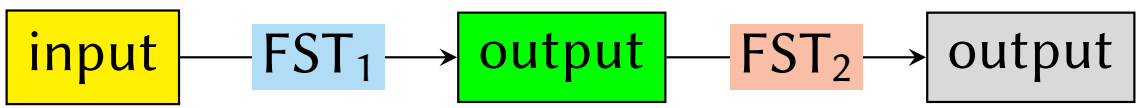
```
-- transduceRL :: Eq c => FST c String -> [s] -> [String]
transduceRL t w = map reverse (transduce t (reverse w))
```

```
*W9> transduceRL fstSibHarm "sosotoSo"
["SoSotoSo"]
```

Composition

Outputs and inputs

Unlike simple FSAs, FSTs yield **output** — typically a new (transduced) string of symbols. This output could be acted upon by another FST!



It is natural to use individual transducers to **decompose** complex processes into components.

Phonological rules

In phonological theory, it is common to think of rule **chaining**: a given rule R_1 applies, and another rule R_2 applies to the output of R_1.

Cad. English, like Am. English, has a rule of **intervocalic flapping**:

[bæɾəɹ] 'batter' [bæɾəɹ] 'badder'
[tɹeɾəɹ] 'traitor' [tɹeɾəɹ] 'trader'

$$\left.\begin{array}{ll} \text{[bæɾəɹ] 'batter'} & \text{[bæɾəɹ] 'badder'} \\ \text{[tɹeɾəɹ] 'traitor'} & \text{[tɹeɾəɹ] 'trader'} \end{array}\right\} \text{t,d} \rightarrow \text{ɾ} \;/\; V\_V$$

Phonological rules

In phonological theory, it is common to think of rule **chaining**: a given rule R_1 applies, and another rule R_2 applies to the output of R_1.

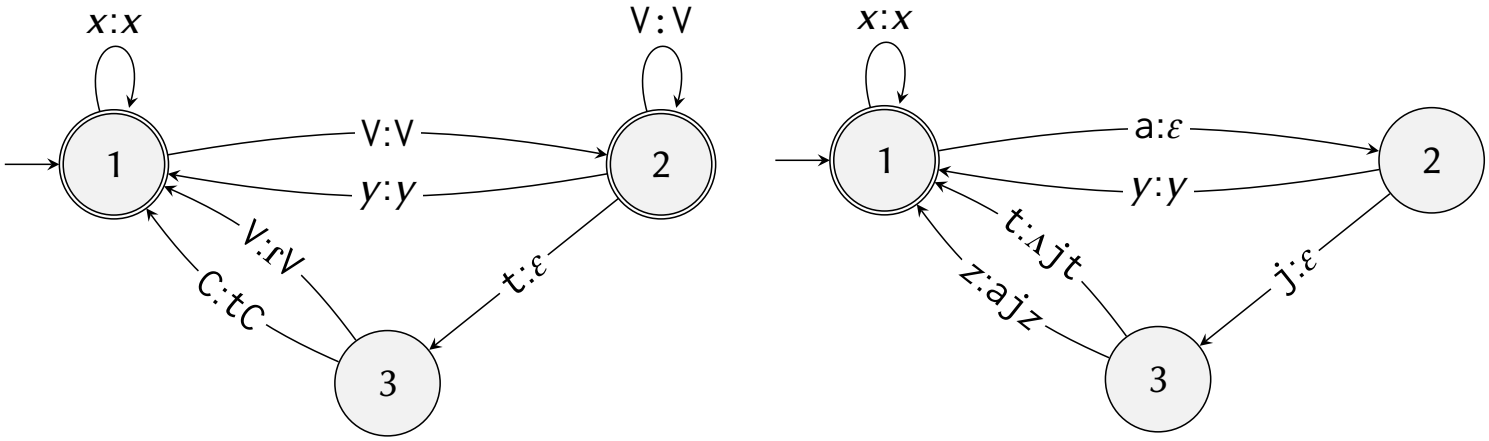
Cad. English, like Am. English, has a rule of **intervocalic flapping**:

[bæɾəɹ] 'batter'	[bæɾəɹ] 'badder'	t,d → ɾ / *V*_*V*
[tɹeɾəɹ] 'traitor'	[tɹeɾəɹ] 'trader'	

Cad. English also has a rule of **vowel raising** before voiceless C's:

[hawz] 'house (V)'	[hʌws] 'house (N)'	aj → ʌj / _*C*[−voice]
[hajd] 'hide'	[hʌjt] 'height'	aw → ʌw / _*C*[−voice]

Some (very simplified) transducers



[ɹajɾəɹ] for ‘rider’ and [ɹʌjɾəɹ] for ‘writer’ suggest that vowel raising happens **before** flapping neutralizes the distinction between [t], [d].

- Complex grammars can be built from simple processes, chained.

It is more accurate to say that this is what happens in one dialect (Joos 1942).

Composing transductions

Recall the type of a FST transduction in Haskell:

```
-- transduce :: Eq c => FST c String -> [s] -> [String]
```

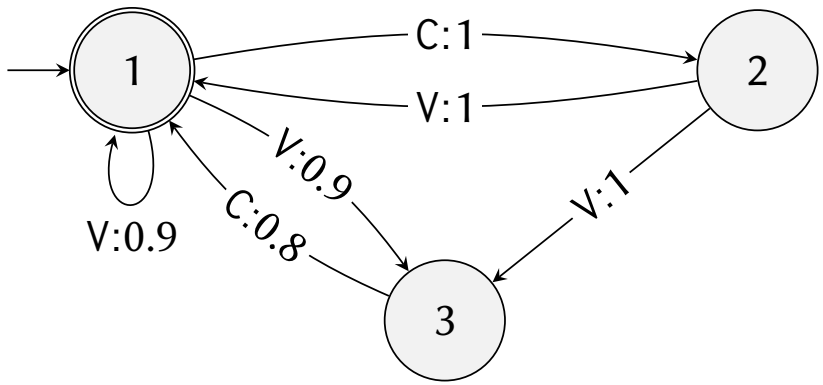
This means that we can chain together FST transductions, by feeding the outputs of one as inputs to the other:

```
composeFST t u s = transduce t s >>= transduce u
--                 [String]        String -> [String]
--                 ------------------------------------
--                             [String]
```

Sometimes the order of `t` and `u` won't matter. Sometimes it will.

Other kinds of output

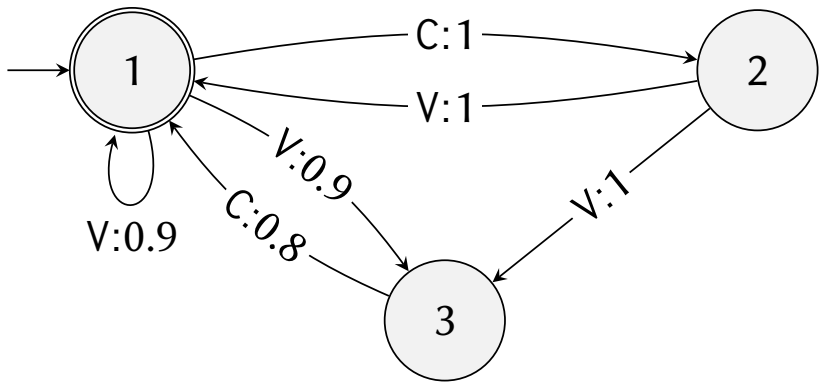
A weighted FSA



Here is the linguistic idea being implemented by this machine:

- State 1 represents a syllable boundary
- The 0.8 transition penalizes…

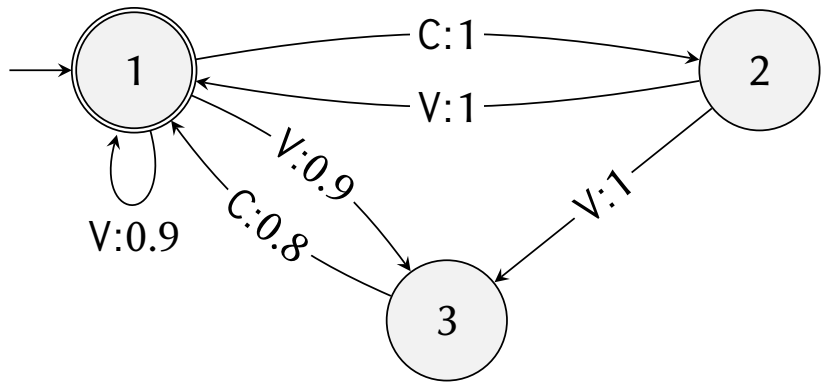
A weighted FSA



Here is the linguistic idea being implemented by this machine:

- State 1 represents a syllable boundary
- The 0.8 transition penalizes…syllables that have a coda
- The 0.9 transitions penalize…

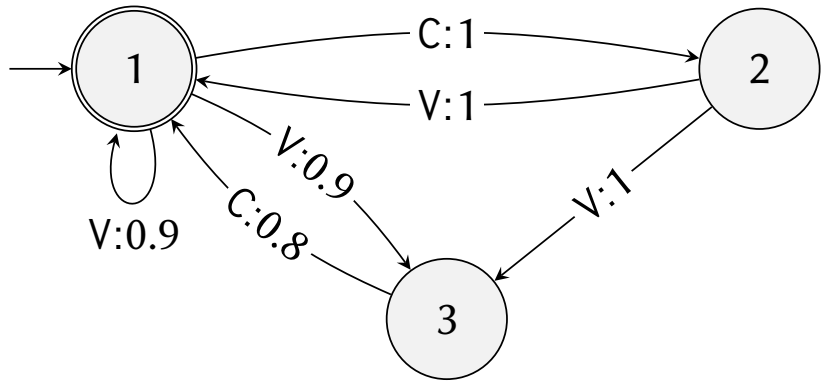
A weighted FSA



Here is the linguistic idea being implemented by this machine:

- State 1 represents a syllable boundary
- The 0.8 transition penalizes…syllables that have a coda
- The 0.9 transitions penalize…syllables that don't have an onset

Parsing with weights



Parsing involves **multiplying** weights along a path:

- CV $\Longrightarrow 1 \times 1$
- VC $\Longrightarrow 0.9 \times 0.8$
- CVC $\Longrightarrow 1 \times 1 \times 0.8$
- CVV $\Longrightarrow 1 \times 1 \times 0.9$
- VVC $\Longrightarrow 0.9 \times 0.9 \times 0.8$

Outputs vs weights

The logic of parsing with weighted FSAs seems remarkably similar to the logic of parsing with FSTs.

- Outputs become weights
- Concatenation of outputs becomes multiplication of weights

What do we need to change in our FST functions (`walk` is unchanged from FSAs) to allow us to handle weights instead of output?

```
step delta x (q,m) = [ (r',m<>n) | (r,y,n,r') <- delta,
                                   q==r, y==x ]

transduce (i, f, delta) str = outputs
  where
    forward = i >>= \q -> walk delta str (q, mempty)
    outputs = [ t | (qn, t) <- forward, elem qn f ]
```

Generic

As it turns out, *nothing!* `<>` and `mempty` are generic enough to work for both outputs (`String`'s) and weights (`Double`'s).

- `<>` expresses **accumulation** (aka `mappend`)
- `mempty` is the **identity element** for a given `<>`

Any `<>` and `mempty` should satisfy the following laws.

- **Associativity:** `(a <> b) <> c == a <> (b <> c)`
- **Identity:** `a <> mempty == a == mempty <> a`

When we have `<>` and `mempty` operations for a given type meeting these specifications, we say that that type is `Monoid`.

Monoids for outputs and weights

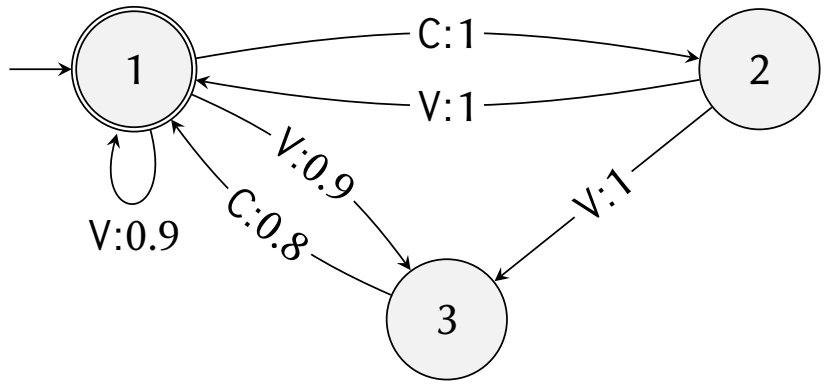
```haskell
-- predefined in Prelude
instance Monoid String where
  mempty  = ""
  mappend = (++)

instance Semigroup String where -- boilerplate
  (<>) = mappend
```

```haskell
instance Monoid Double where
  mempty  = 1.0
  mappend = (*)

instance Semigroup Double where -- boilerplate
  (<>) = mappend
```

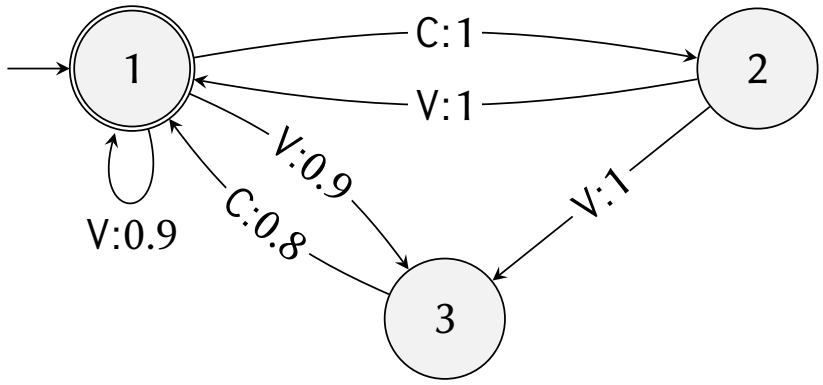
Our weighted FS machine




```
data CV = C | V deriving (Eq,Show)

fstCVWeights :: FST CV Double
fstCVWeights = ( [1], [1],
 [ (1,V,0.9,1), (1,C,1.0,2), (2,V,1.0,1),
   (2,V,1.0,3), (1,V,0.9,3), (3,C,0.8,1) ] )
```

Some transductions




```
*W9> transduce fstCVWeights [C,V]
[1.0]
*W9> transduce fstCVWeights [V,C]
[0.72]
*W9> transduce fstCVWeights [C,V,C]
[0.8]
*W9> transduce fstCVWeights [C,V,V]
[0.9]
*W9> transduce fstCVWeights [V,V,C]
[0.648]
```

Our final types

```
step
  :: (Eq c, Monoid m) =>
     [Arrow c m] -> c -> (State, m) -> [(State, m)]
step delta x (q,m) = [ (r',m<>n) | (r,y,n,r') <- delta,
                                   q==r, y==x ]

transduce
  :: (Eq c, Monoid m) =>
     FST c m -> [c] -> [m]
transduce (i, f, delta) str = outputs
  where
    forward = i >>= \q -> walk delta str (q, mempty)
    outputs = [ t | (qn, t) <- forward, elem qn f ]
```

Some food for thought

How about FSTs that assign **costs**, which add up over a path?

- What's the relevant monoid?

What about **ambiguity**? (Consider: VCV, CVCV).

- How should we deal? Can we report a summary value?
- Do different monoids imply different ways of summarizing?