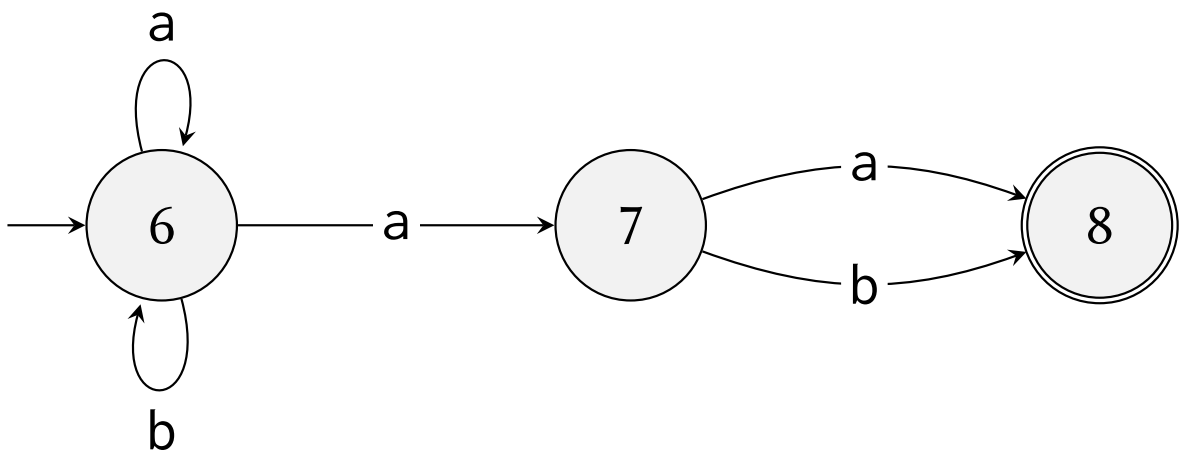
Parsing with FSAs

Computational Linguistics (LING 455)

Rutgers University

October 22, 2021

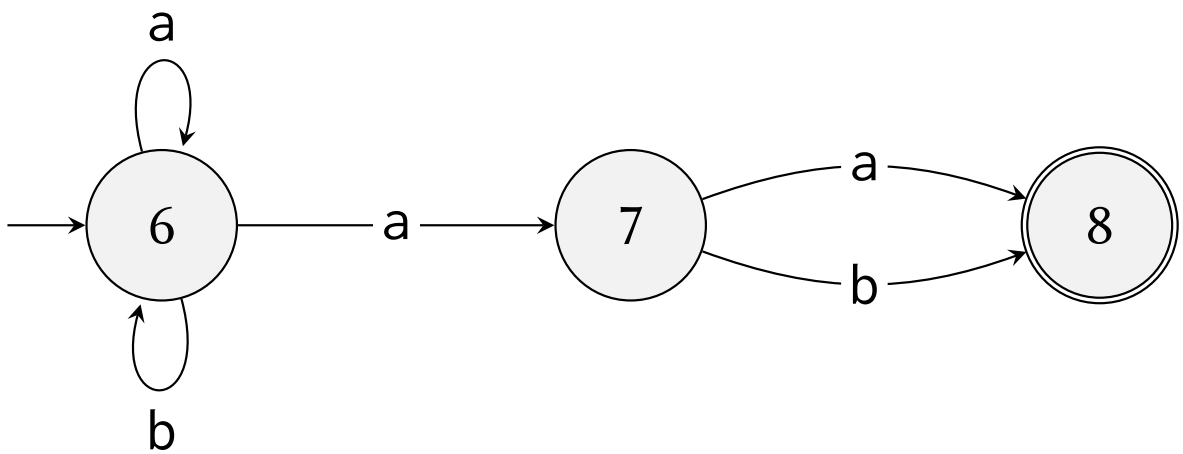
Week 7 review: FSA practice



What strings does this FSA recognize?

What is an RE that characterizes this pattern?

Week 7 review: FSA practice

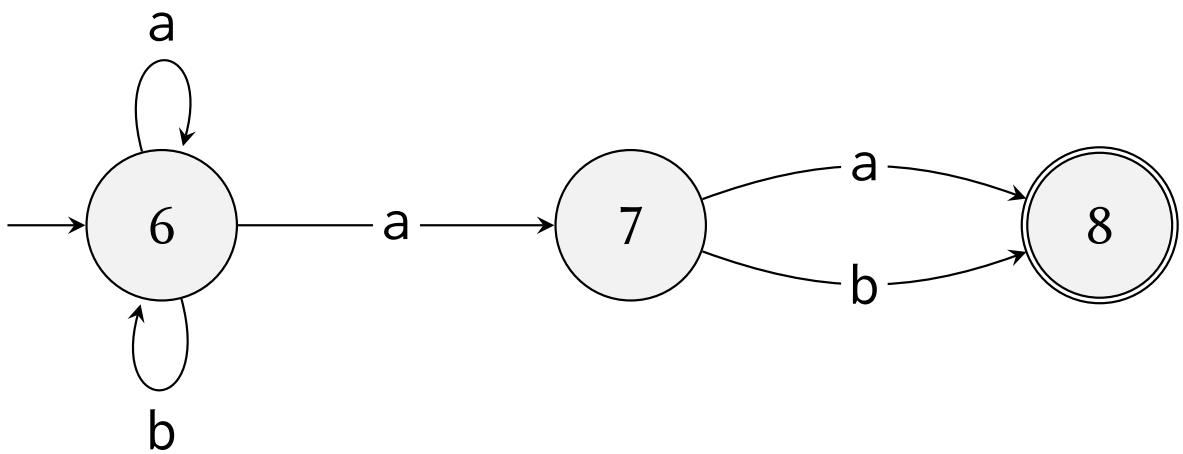


What strings does this FSA recognize?

- Strings over $\Sigma = \{a, b\}$ whose 2nd-to-last symbol is a

What is an RE that characterizes this pattern?

Week 7 review: FSA practice



What strings does this FSA recognize?

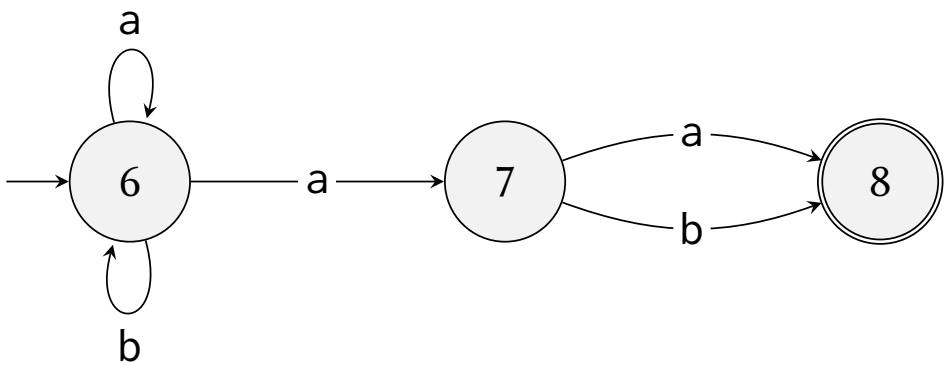
- Strings over $\Sigma = \{a, b\}$ whose 2nd-to-last symbol is a

What is an RE that characterizes this pattern?

- $\Sigma^* \cdot a \cdot (a \mid b)$

Week 7 review: FSA's in Haskell

```haskell
type State = Int
type Transition a = (State,a,State)
type FSA sym = ( [State]                -- Q (states)
               , [sym]                  -- Σ (alphabet)
               , [State]                -- I (initial)
               , [State]                -- F (final)
               , [Transition sym] )     -- Δ (arrows)
```

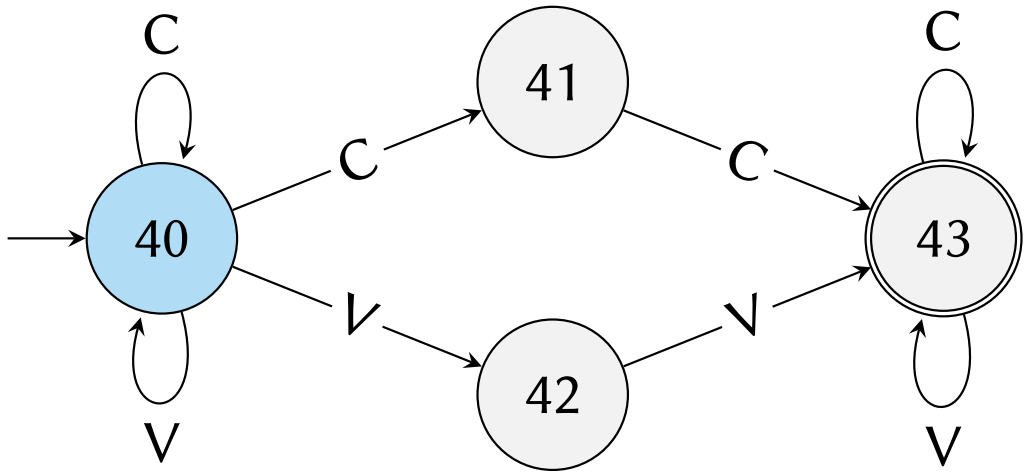



```haskell
fsaPenultA :: FSA Char
fsaPenultA = ( [6,7,8], ['a','b'], [6], [8],
  [(6,'a',6),(6,'b',6),(6,'a',7),(7,'a',8),(7,'b',8)] )
```

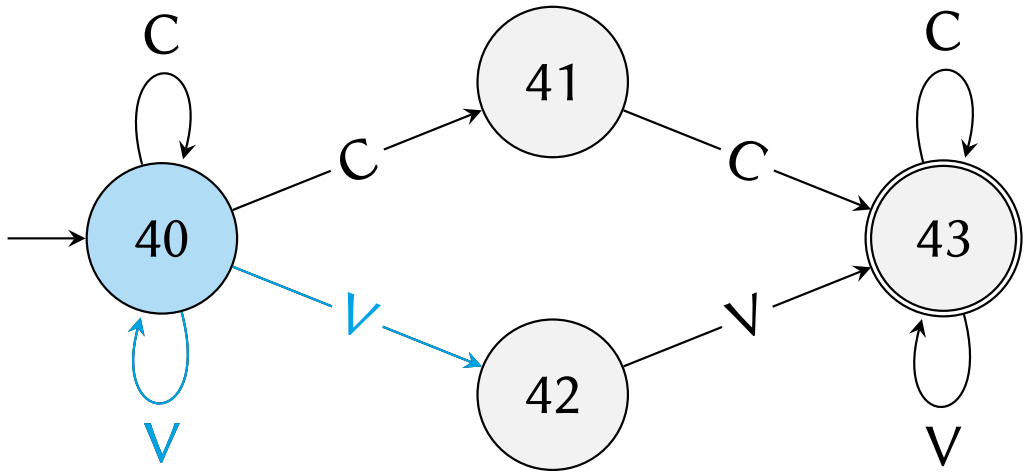
This week

- Parsing: automating FSA recognition
- Forward sets, and their significance
- A bit on different ways to construct FSAs
- The Myhill-Nerode theorem

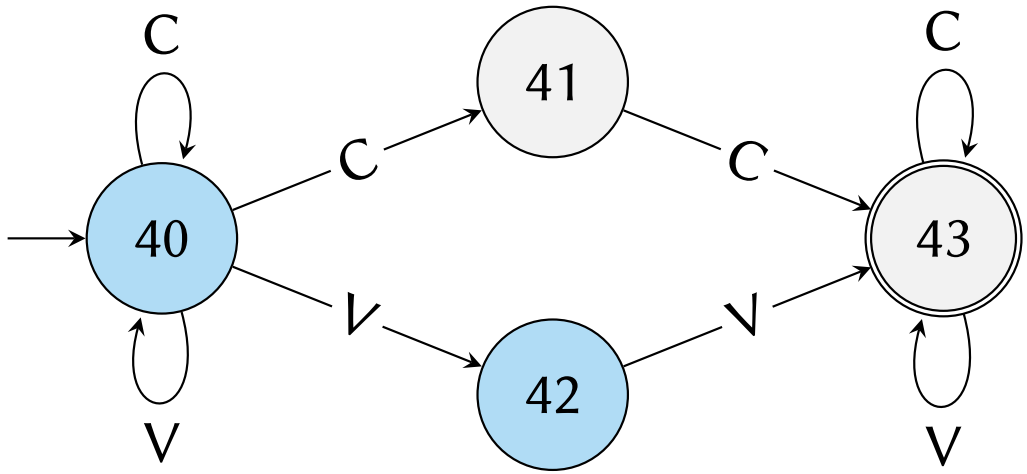
Taking one step



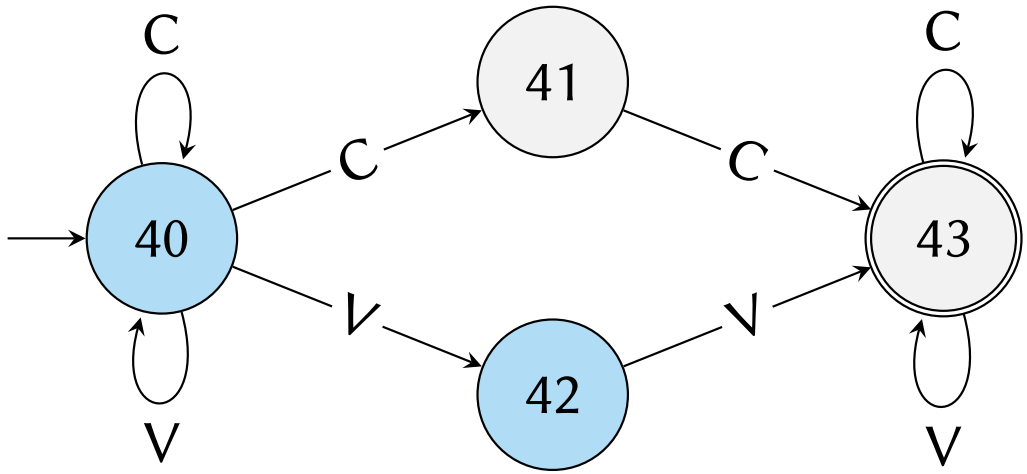
Taking one step



Taking one step



Taking one step




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

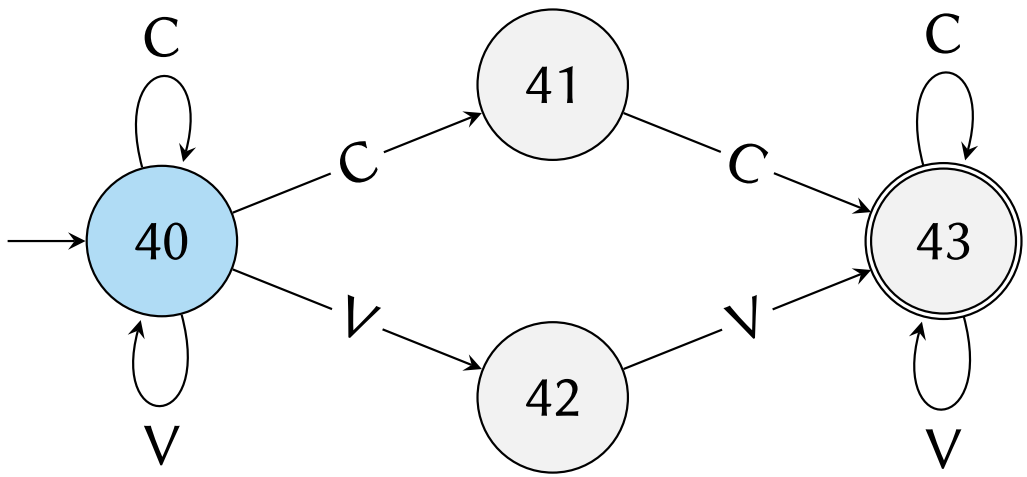
```
*W8> (_,_,_,_,delta) = fsaCCVV
*W8> step delta V 40
[40,42]
```

`Eq a => ...` is a **type context**, a reflection that this definition can only handle Σ's whose symbols can be checked for equality. This is due to `y==x`.

From steps to walks: VV

This tells us how to traverse a FSA with a string:

- Take a step with the first symbol
- Keep stepping from each possible next state
- Do it till you run out of string, then rest

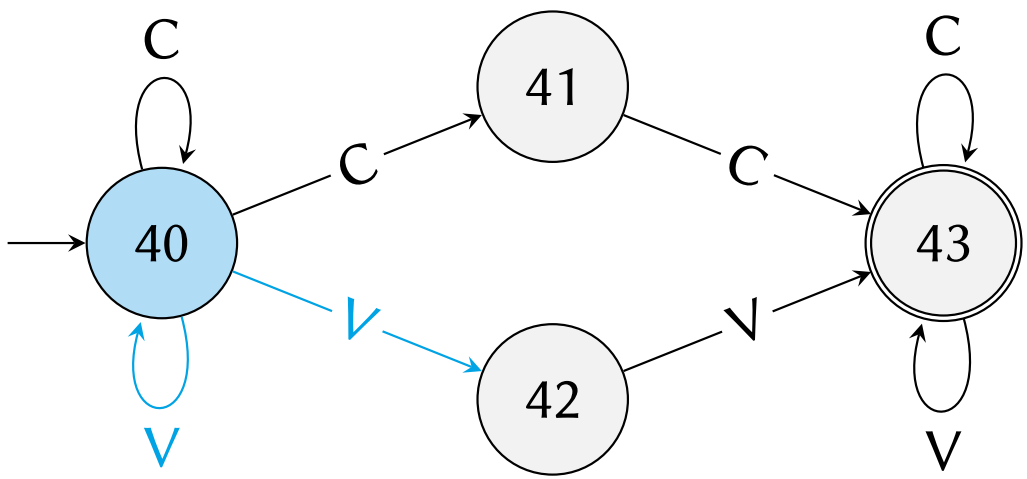


3 ways to parse VV: 40⟶40⟶40, 40⟶40⟶42, 40⟶42⟶43

From steps to walks: VV

This tells us how to traverse a FSA with a string:

- Take a step with the first symbol
- Keep stepping from each possible next state
- Do it till you run out of string, then rest

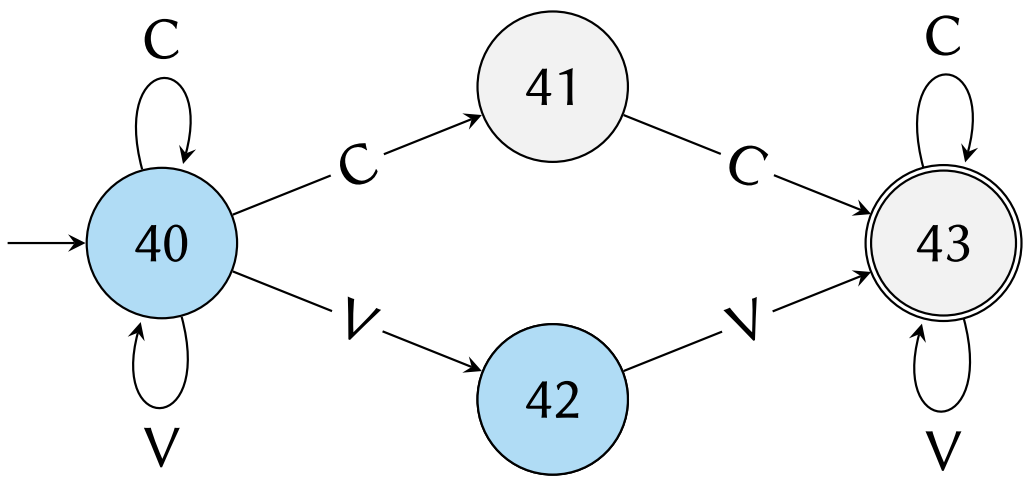


3 ways to parse VV: 40⟶40⟶40, 40⟶40⟶42, 40⟶42⟶43

From steps to walks: VV

This tells us how to traverse a FSA with a string:

- Take a step with the first symbol
- Keep stepping from each possible next state
- Do it till you run out of string, then rest

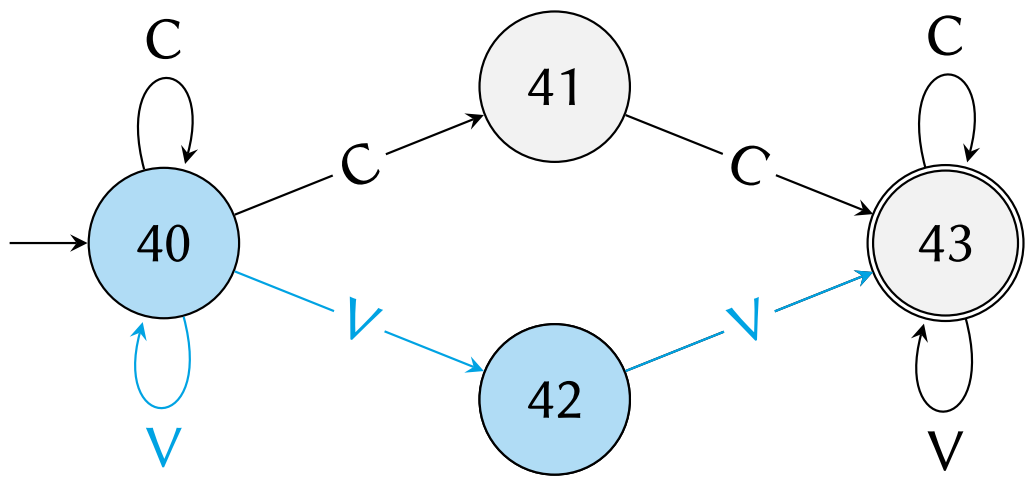


3 ways to parse VV: 40⟶40⟶40, 40⟶40⟶42, 40⟶42⟶43

From steps to walks: VV

This tells us how to traverse a FSA with a string:

- Take a step with the first symbol
- Keep stepping from each possible next state
- Do it till you run out of string, then rest

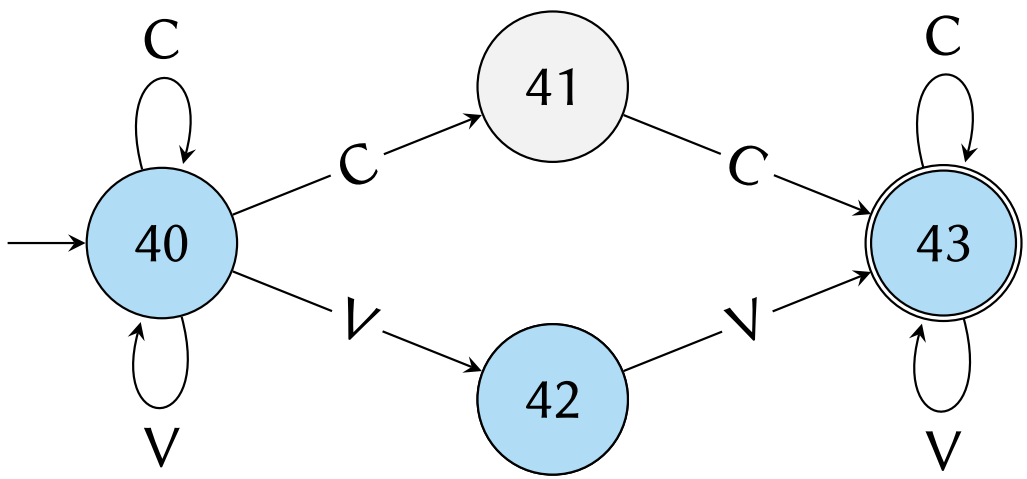


3 ways to parse VV: 40⟶40⟶40, 40⟶40⟶42, 40⟶42⟶43

From steps to walks: VV

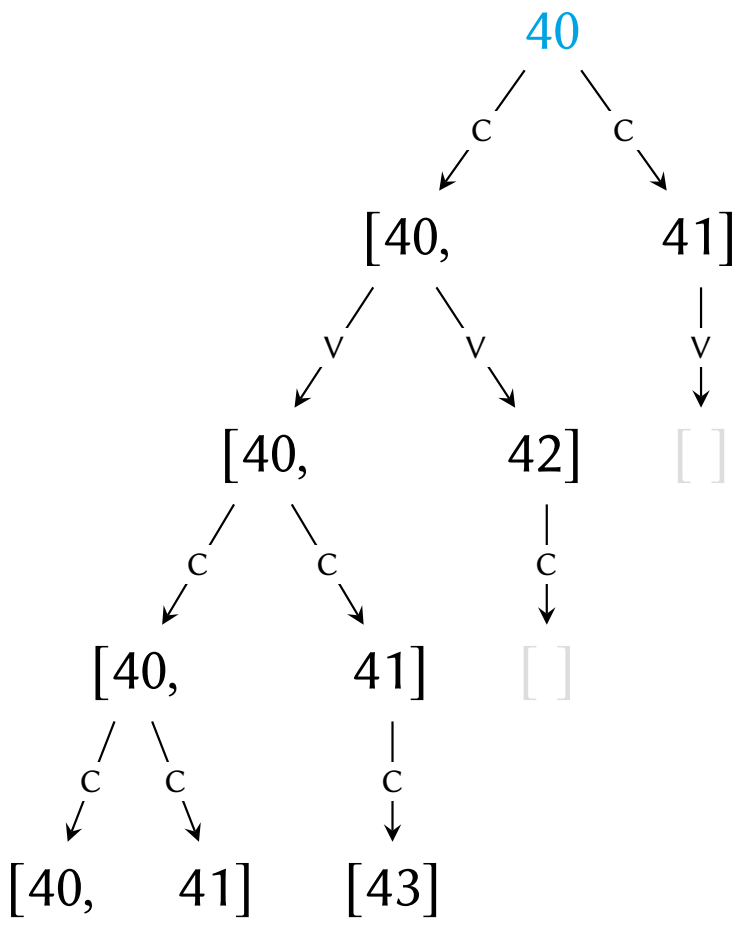
This tells us how to traverse a FSA with a string:

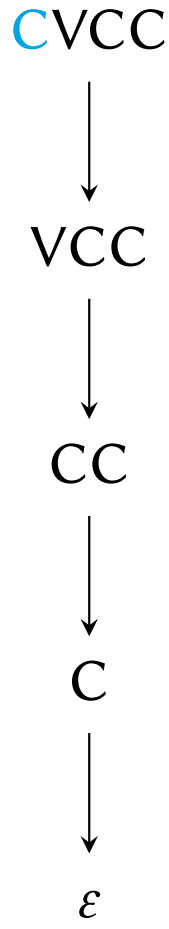
- Take a step with the first symbol
- Keep stepping from each possible next state
- Do it till you run out of string, then rest

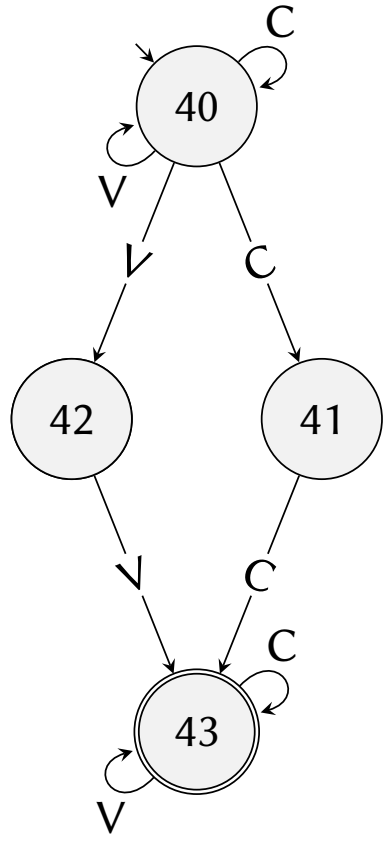


3 ways to parse VV: 40⟶40⟶40, 40⟶40⟶42, 40⟶42⟶43

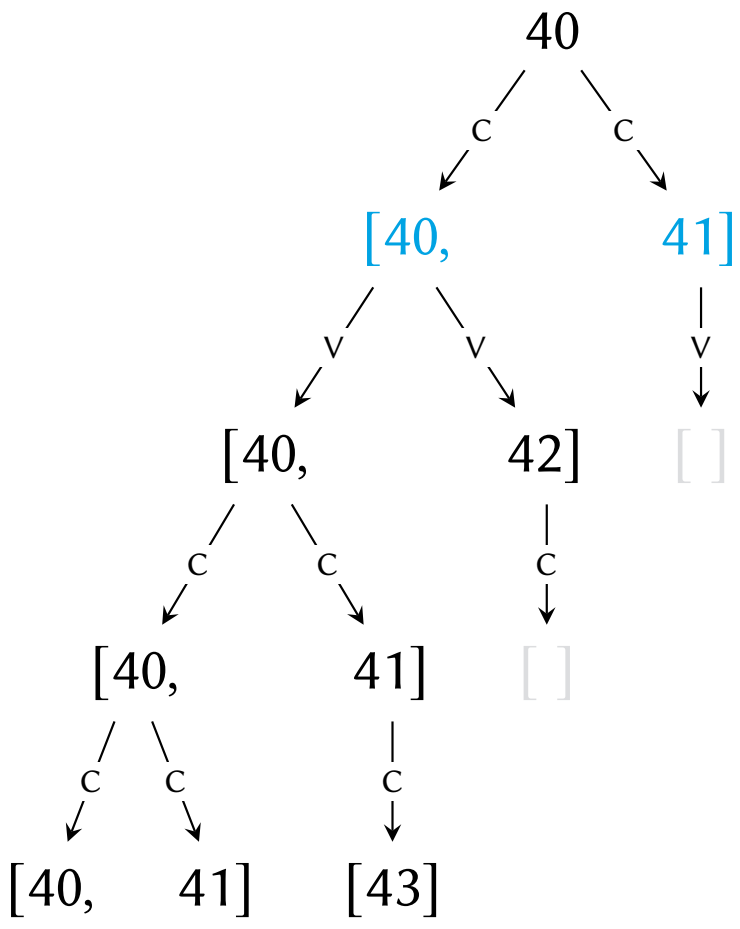
Take steps till you run out of string

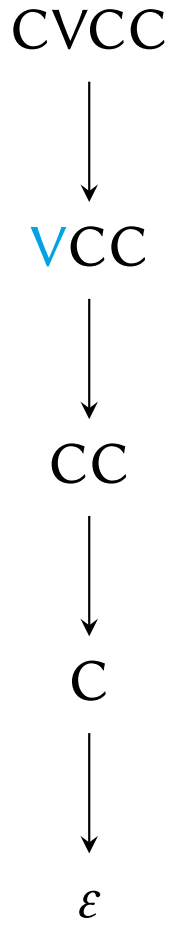


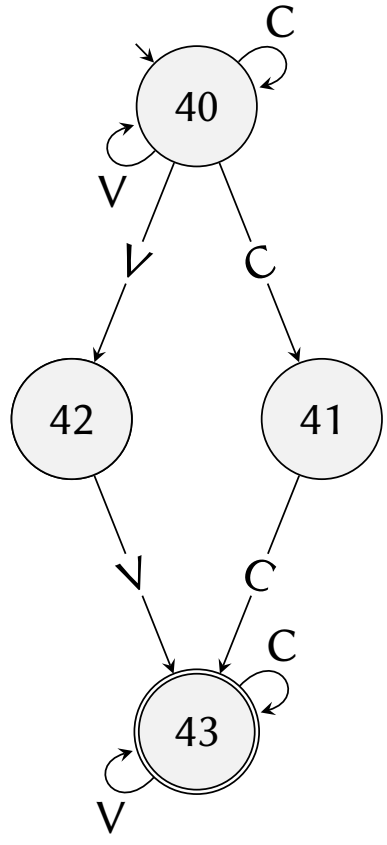




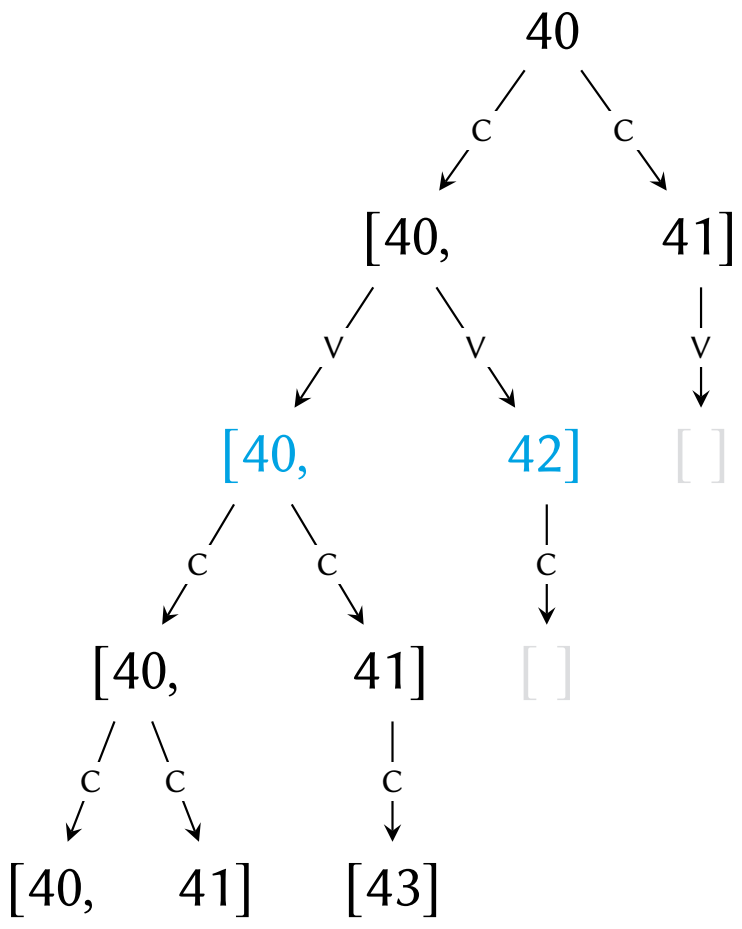
Take steps till you run out of string

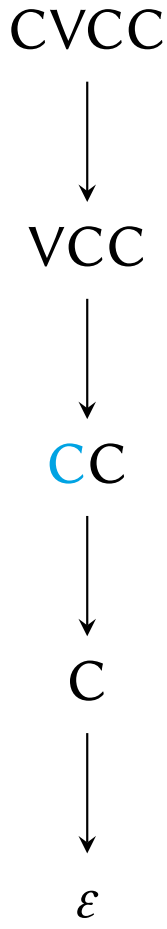


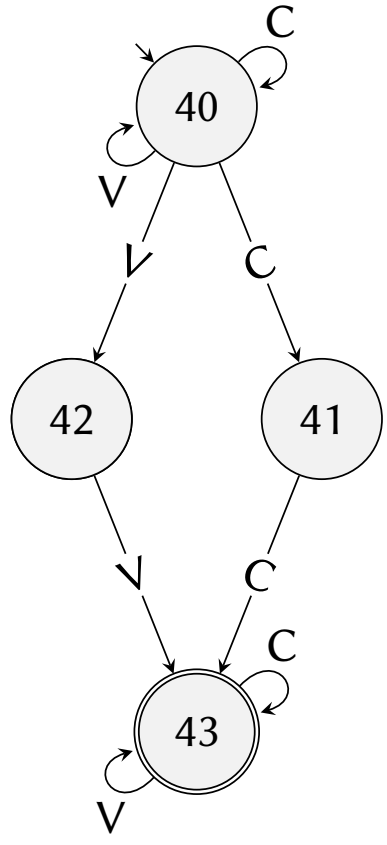




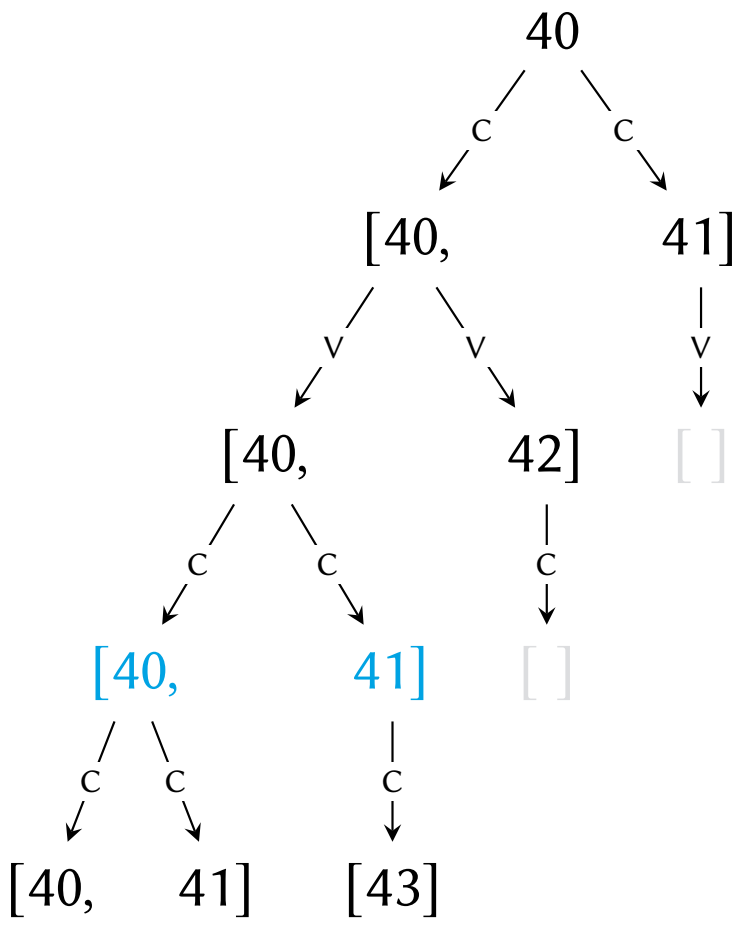
Take steps till you run out of string

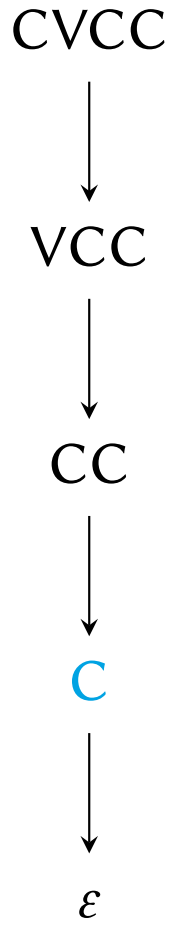


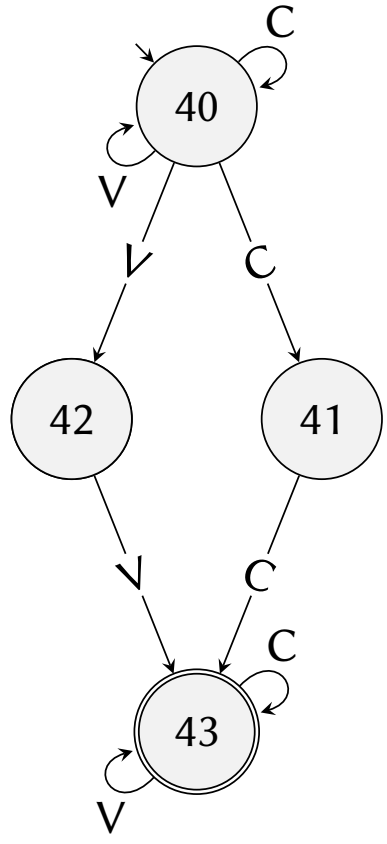




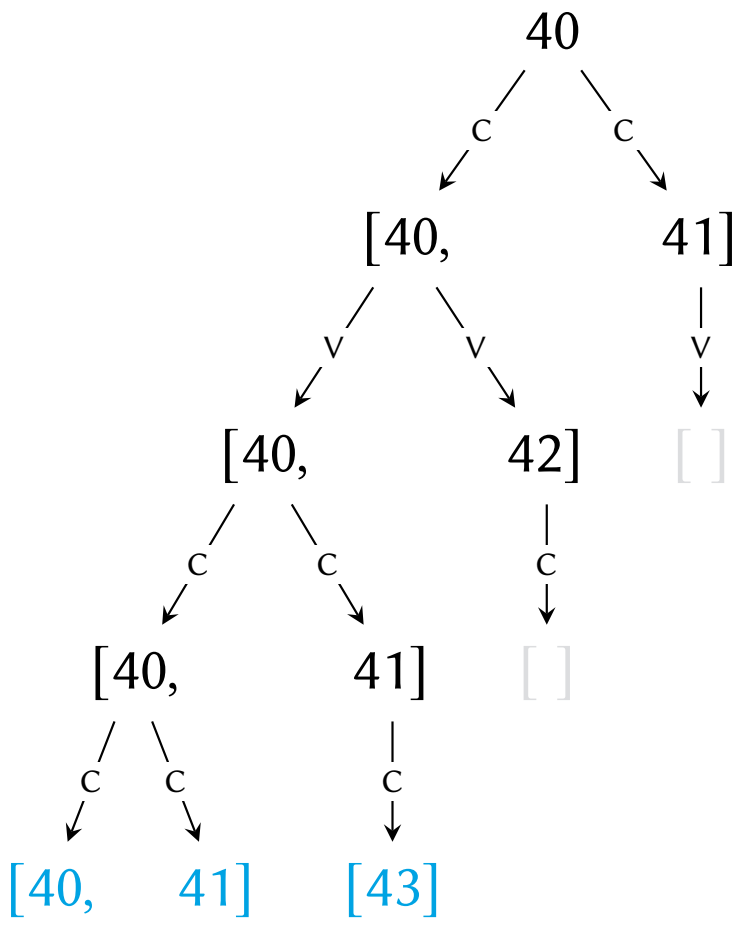
Take steps till you run out of string

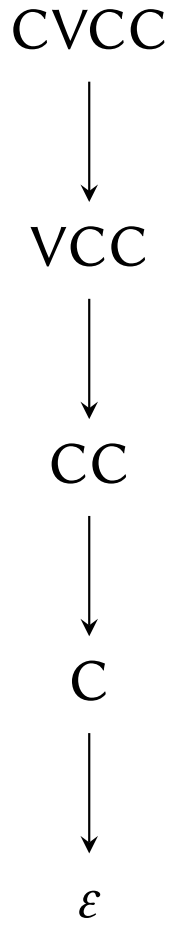


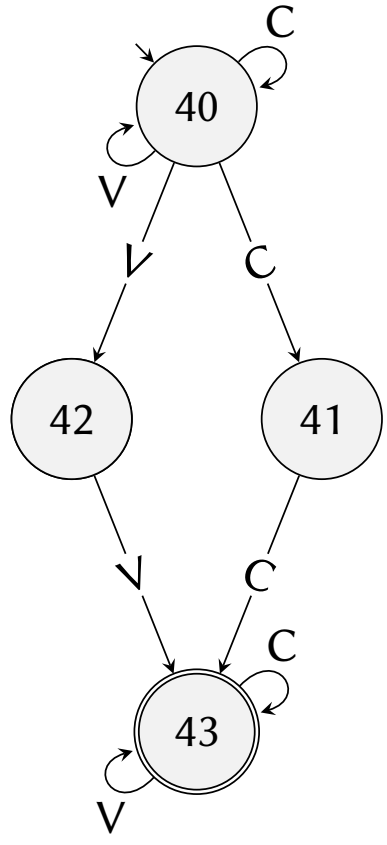




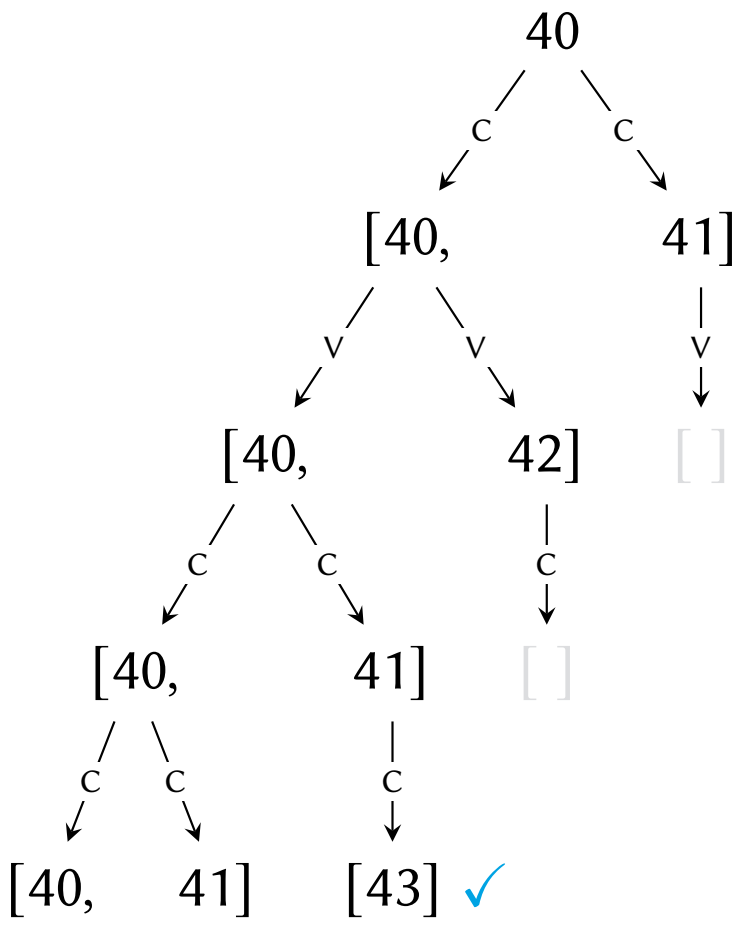
Take steps till you run out of string

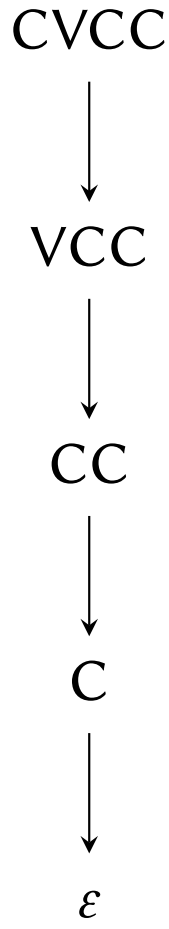


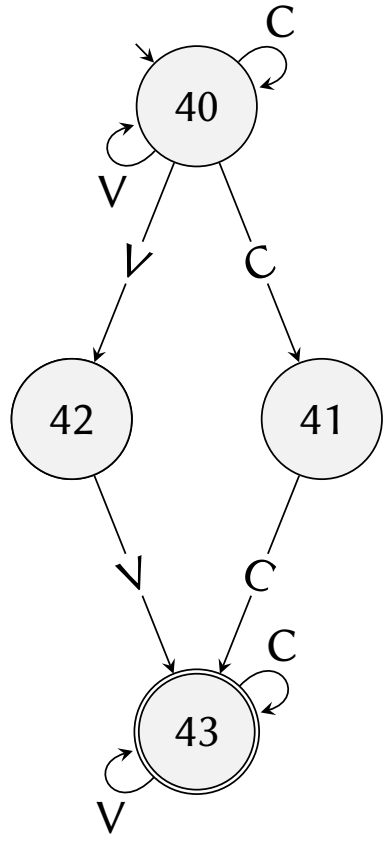




Take steps till you run out of string







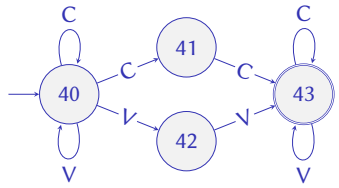
Putting it all together

```haskell
walk :: Eq a => [Transition a] -> [a] -> State -> [State]
walk delta str q =
  case str of
    []   -> [q] -- you consumed the string, rest here!
    x:xs -> let oneStep  = step delta x q -- [S]
                walkTail = walk delta xs  -- S -> [S]
            in  oneStep >>= walkTail
```

We use a handy bit of Haskell infix notation for `concatMap`:

```haskell
(>>=) :: [State] -> (State -> [State]) -> [State]
xs >>= f = concatMap f xs
```

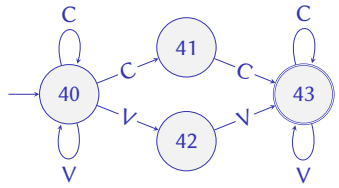
Example




```
walk delta str q =
  case str of
    []   -> [q] -- you consumed the string, rest here!
    x:xs -> let oneStep  = step delta x q -- [S]
                walkTail = walk delta xs  -- S -> [S]
            in  oneStep >>= walkTail
```

```
walk delta [V,V] 40
  ==> oneStep >>= walkTail
  ==> [40,42] >>= walk delta [V]
  ==> concat [walk delta [V] 40, walk delta [V] 42]
  ...
  ==> concat [[40,42],[43]]
  ==> [40,42,43]
```

Example




```
walk delta str q =
  case str of
    []   -> [q] -- you consumed the string, rest here!
    x:xs -> let oneStep  = step delta x q -- [S]
                walkTail = walk delta xs  -- S -> [S]
            in  oneStep >>= walkTail
```

```
walk delta [V,C] 40
  ==> oneStep >>= walkTail
  ==> [40,42] >>= walk delta [C]
  ==> concat [walk delta [C] 40, walk delta [C] 42]
  ...
  ==> concat [[40,41],[]]
  ==> [40,41]
```

accept-ance

The only remaining thing is to ensure that we:

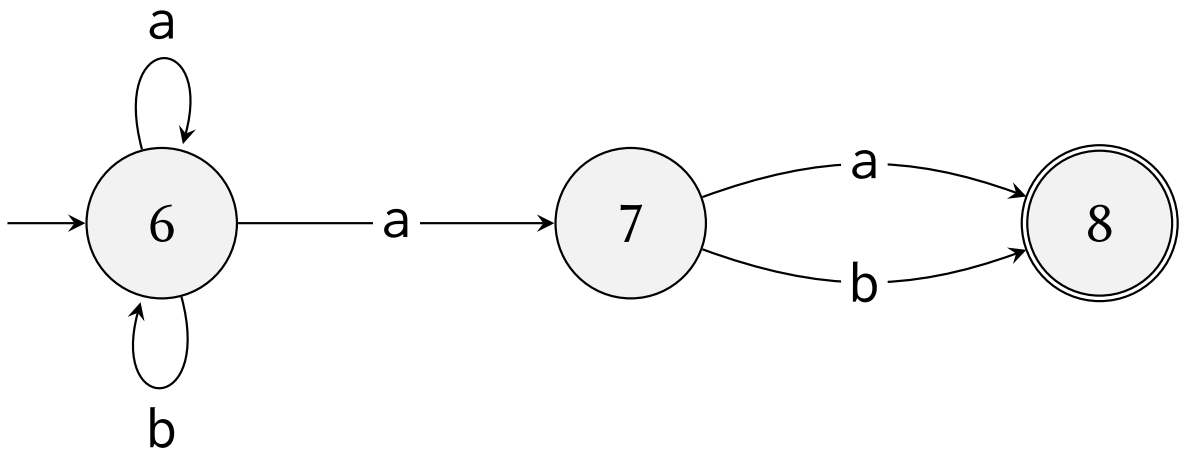
- Start walking from the initial states
- End up with at least one accepting state

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

```
*W8> parseFSA fsaCCVV [V,V]
True
*W8> parseFSA fsaCCVV [V,C]
False
*W8> parseFSA fsaCCVV [V,C,C,V,C,V]
True
```

Forward sets: `walk`-ing from `i`

`walk`-ing some strings from `i`




```
*W8> (_,_,i,_,delta) = fsaPenultA
*W8> i >>= walk delta "aa"
[6,7,8]
*W8> i >>= walk delta "aab"
[6,8]
*W8> i >>= walk delta "aabb"
[6]
```

Forward sets

A **forward set** (for a string, given a FSA) is the possible states you could be in after parsing the string from an initial state.

What do we know when two strings have **the same forward set**?

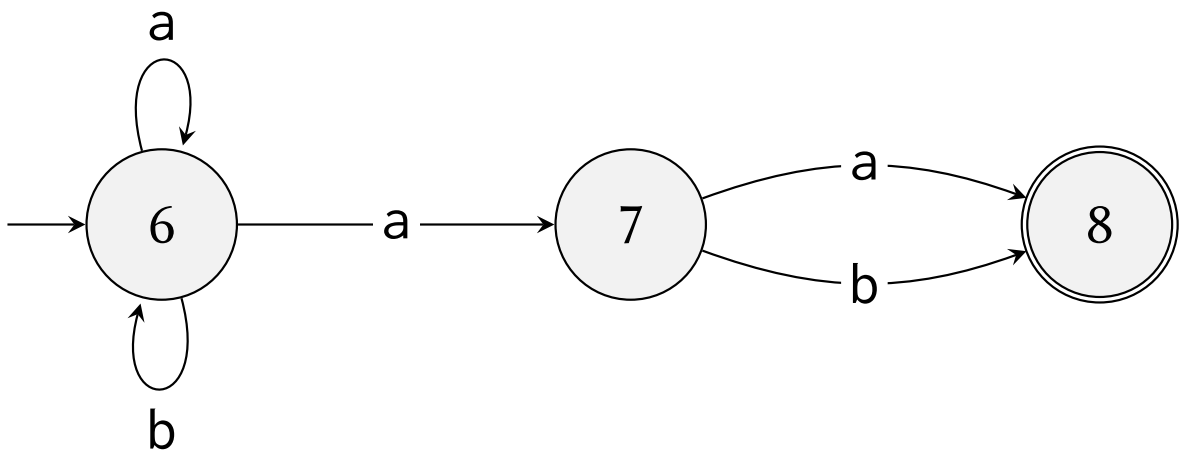
Forward sets

A **forward set** (for a string, given a FSA) is the possible states you could be in after parsing the string from an initial state.

What do we know when two strings have **the same forward set**?

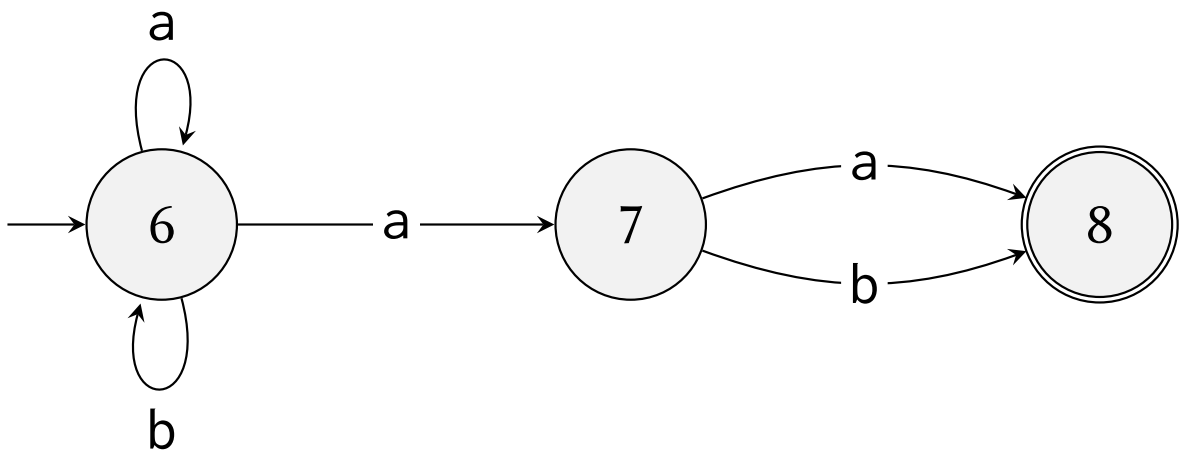
- Whatever you can('t) do after one, you can('t) do after the other.
- The FSA classifies the strings as **equivalent prefixes**.

An example



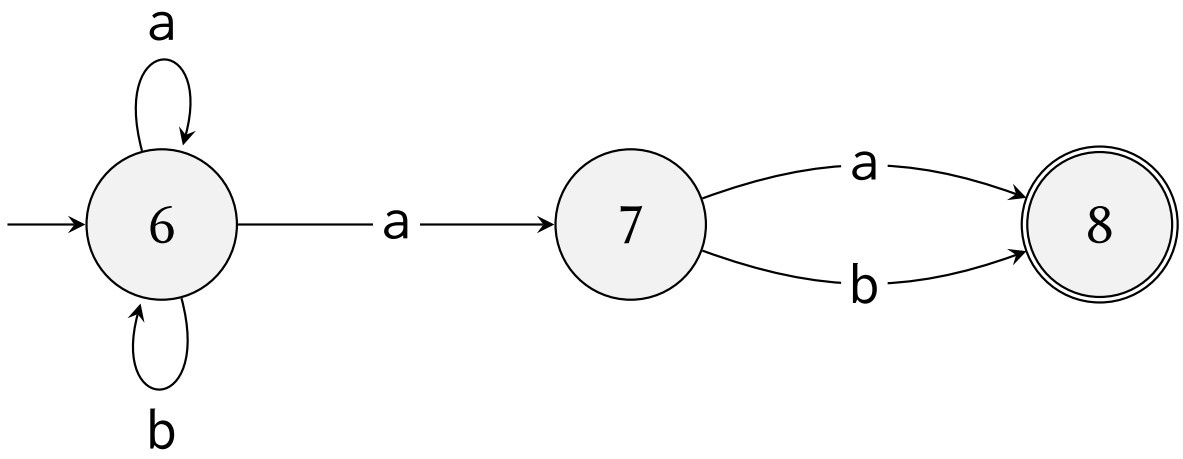
	penult isn't a	penult is a
last isn't a	FWD SET: $\{6\}$ EX. STRINGS: ε, b, bb, abb	FWD SET: $\{6, 8\}$ EX. STRINGS: ab aab, bab
last is a	FWD SET: $\{6, 7\}$ EX. STRINGS: a, ba, aba, bba	FWD SET: $\{6, 7, 8\}$ EX. STRINGS: aa aaa, baa

An example



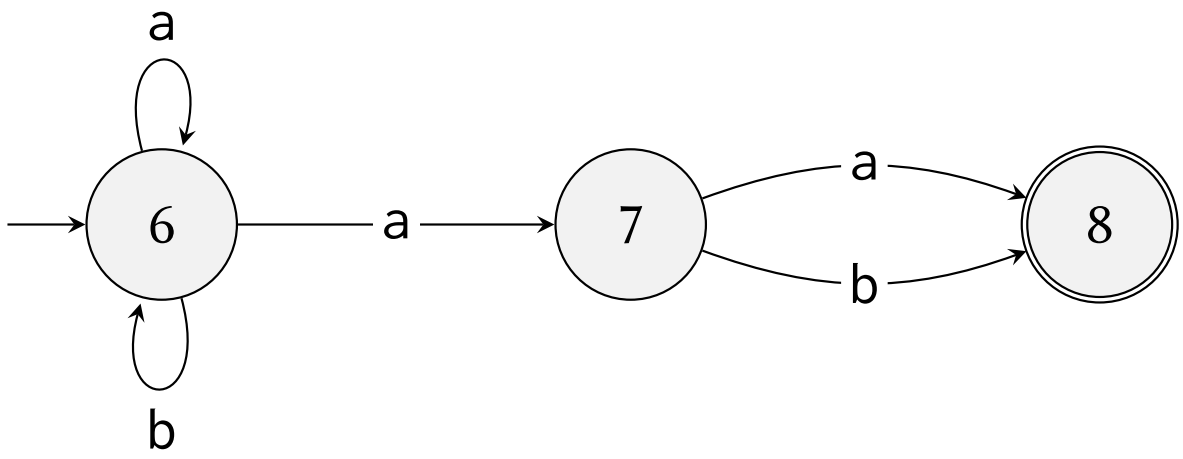
	penult isn't a	penult is a
last isn't a	FWD SET: $\{6\}$ EX. STRINGS: ε, b, bb, abb	FWD SET: $\{6, 8\}$ EX. STRINGS: ab aab, bab
last is a	FWD SET: $\{6, 7\}$ EX. STRINGS: a, ba, aba, bba	FWD SET: $\{6, 7, 8\}$ EX. STRINGS: aa aaa, baa

An example



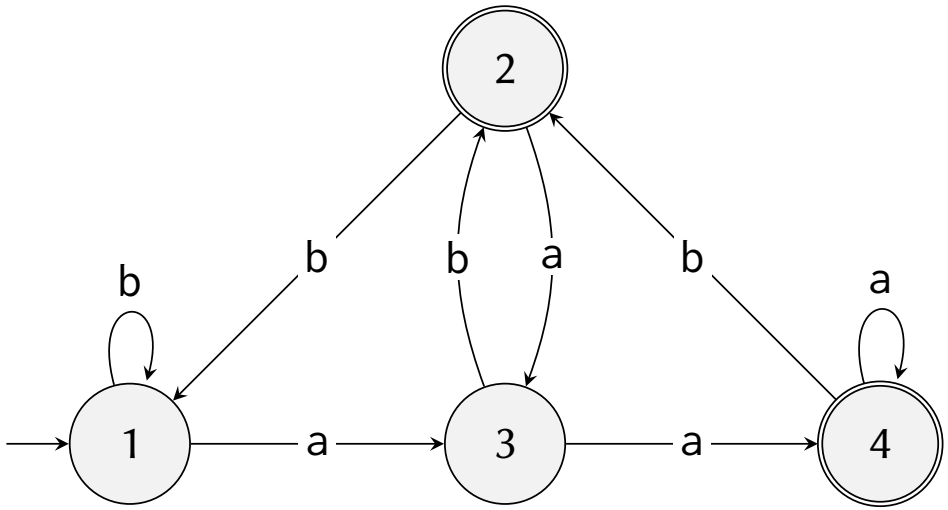
	penult isn't a	penult is a
last isn't a	FWD SET: $\{6\}$ EX. STRINGS: ε, b, bb, abb	FWD SET: $\{6, 8\}$ EX. STRINGS: ab aab, bab
last is a	FWD SET: $\{6, 7\}$ EX. STRINGS: a, ba, aba, bba	FWD SET: $\{6, 7, 8\}$ EX. STRINGS: aa aaa, baa

An example



	penult isn't a	penult is a
last isn't a	FWD SET: $\{6\}$ EX. STRINGS: ε, b, bb, abb	FWD SET: $\{6, 8\}$ EX. STRINGS: ab aab, bab
last is a	FWD SET: $\{6, 7\}$ EX. STRINGS: a, ba, aba, bba	FWD SET: $\{6, 7, 8\}$ EX. STRINGS: aa aaa, baa

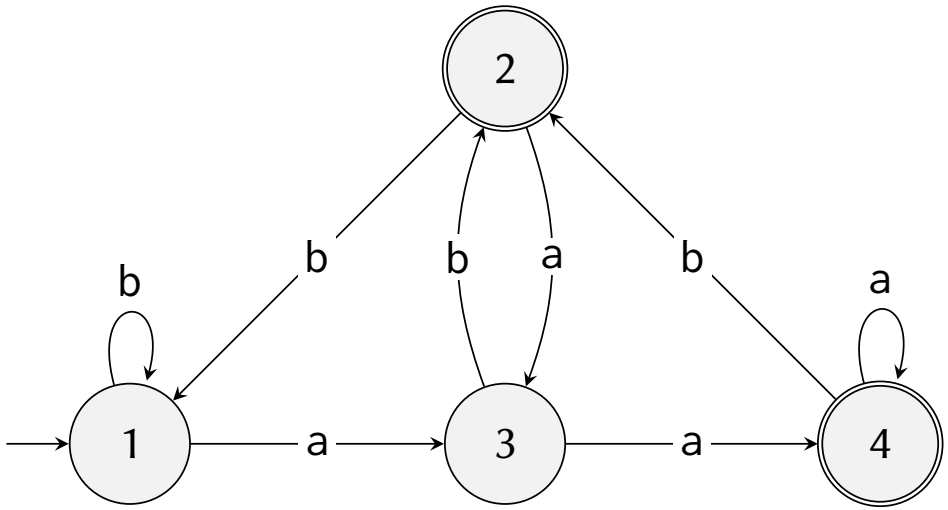
An equivalent FSA where states **are** fwd sets



	penult isn't a	penult is a
last isn't a	STATE: 1 EX. STRINGS: ε, b, bb, abb	STATE: 2 EX. STRINGS: ab aab, bab
last is a	STATE: 3 EX. STRINGS: a, ba, aba, bba	STATE: 4 EX. STRINGS: aa aaa, baa

Always possible! This machine is a **minimal deterministic finite automaton**.

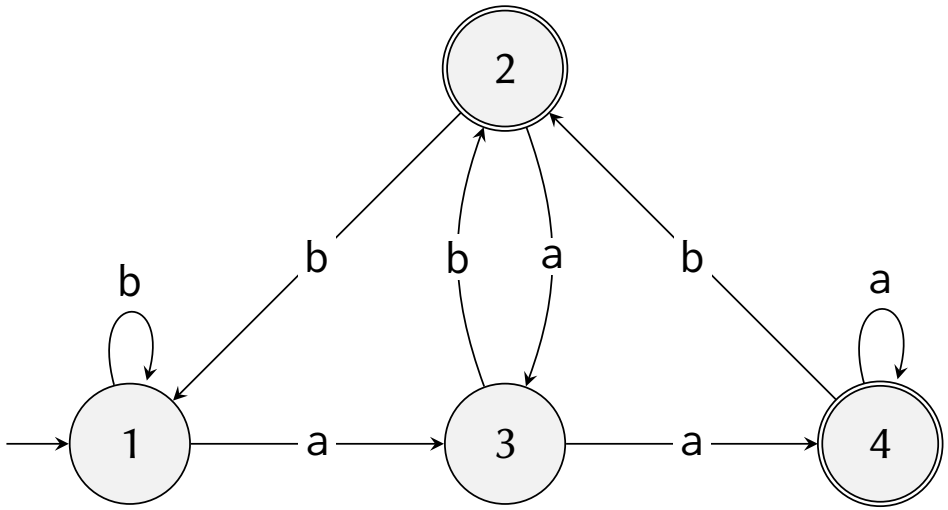
An equivalent FSA where states **are** fwd sets



	penult isn't a	penult is a
last isn't a	STATE: 1 EX. STRINGS: ε, b, bb, abb	STATE: 2 EX. STRINGS: ab aab, bab
last is a	STATE: 3 EX. STRINGS: a, ba, aba, bba	STATE: 4 EX. STRINGS: aa aaa, baa

Always possible! This machine is a **minimal deterministic finite automaton**.

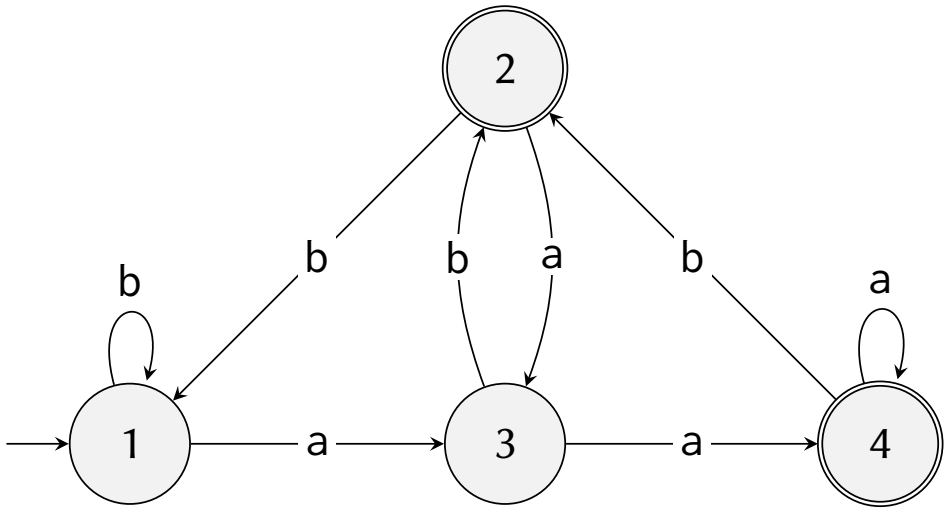
An equivalent FSA where states **are** fwd sets



	penult isn't a	penult is a
last isn't a	STATE: 1 EX. STRINGS: ε, b, bb, abb	STATE: 2 EX. STRINGS: ab aab, bab
last is a	STATE: 3 EX. STRINGS: a, ba, aba, bba	STATE: 4 EX. STRINGS: aa aaa, baa

Always possible! This machine is a **minimal deterministic finite automaton**.

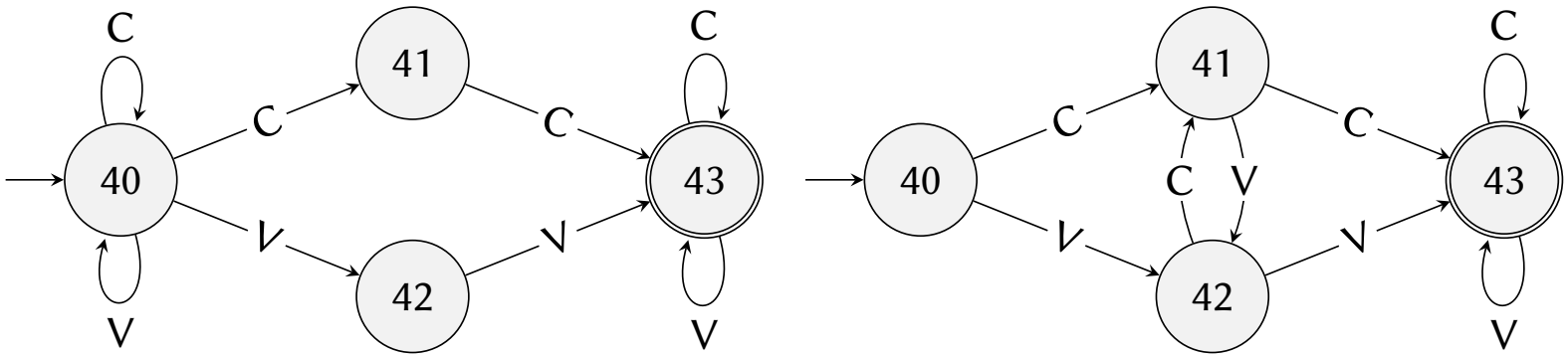
An equivalent FSA where states **are** fwd sets



	penult isn't a	penult is a
last isn't a	STATE: 1 EX. STRINGS: ε, b, bb, abb	STATE: 2 EX. STRINGS: ab aab, bab
last is a	STATE: 3 EX. STRINGS: a, ba, aba, bba	STATE: 4 EX. STRINGS: aa aaa, baa

Always possible! This machine is a **minimal deterministic finite automaton**.

$\Sigma^* \cdot ((C \cdot C) \mid (V \cdot V)) \cdot \Sigma^*$, two ways



These machines are equivalent (in that they recognize the same language). But their forward sets work different:

- In both, equiv fwd sets means equiv prefixes (as always)
- In the latter only, equiv prefixes means equiv fwd set too

Compare the fwd sets for (e.g.) CC and VV in the two machines.

Why do forward sets matter?

Forward sets show how FSA grammars care about some distinctions, and don't care about others.

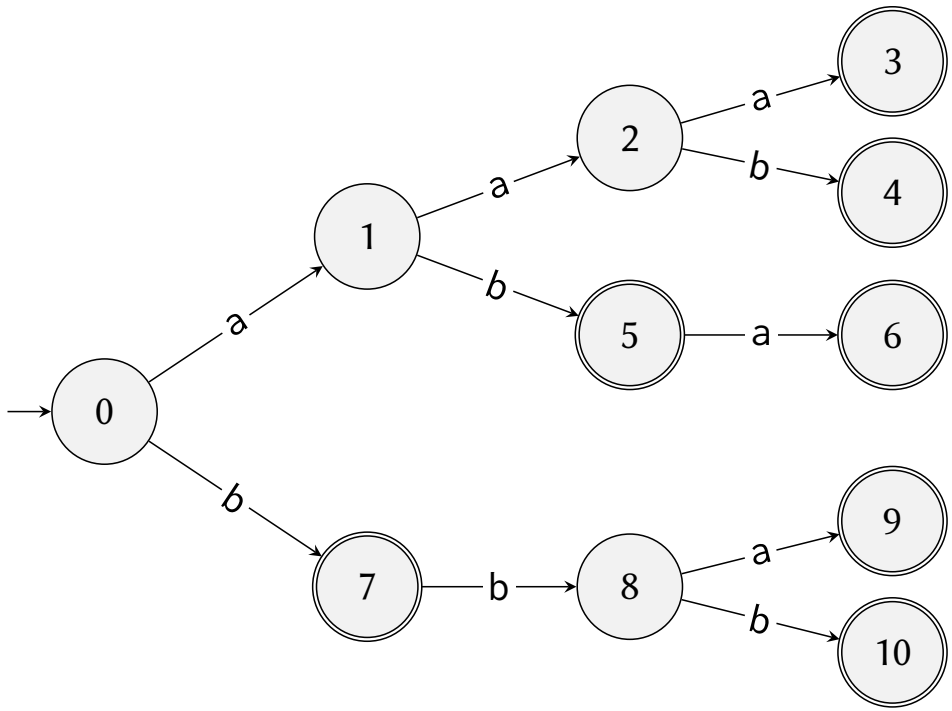
- Two different strings having the same forward set tells us that the grammar doesn't care about the differences between them.
- The options for continuing the strings are the same.

There is something intuitive about this: knowing a language means caring about certain distinctions (e.g., *cat* vs *devour*), and not caring about others (e.g., *cat* vs *dog*).

- Forward sets are like FSA versions of **categories**

A boring FSA for a finite language: everything matters

Here's an FSA for a finite $\mathcal{L} = \{\text{aaa, aab, ab, aba, b, bba, bbb}\}$:



The forward sets for the strings in $\mathcal{L}$ are all distinct. No two strings are "similar". Every distinction matters.

FSA "buckets"

If an FSA's Q has n states, it can have as many as 2^n forward sets:

- Forward states are subsets of Q, and any set S has $2^{|S|}$ subsets
- This is an **upper** limit. Generally there will be fewer fwd sets

It follows that any FSA will have a **finite** number of forward sets (since FSA's are by definition finite).

- Any FSA has a finite number of ways to distinguish prefixes
- This is known at the **Myhill-Nerode theorem**

$a^n \cdot b^n$

Previously we saw that no RE could generate $\mathcal{L} = \{a^n \cdot b^n \mid n \geqslant 1\}$.

- We used the pumping lemma for REs: no part of any $s \in \mathcal{L}$ can be pumped (repeated) without taking us out of $\mathcal{L}$.
- In general, any sufficiently long string in an infinite regular language can be pumped: infinity requires $*$.

Forward sets show that the same limitation holds of FSAs:

- Any a^n, a^{n+1} will have different forward sets. Why?

$a^n \cdot b^n$

Previously we saw that no RE could generate $\mathcal{L} = \{a^n \cdot b^n \mid n \geqslant 1\}$.

- We used the pumping lemma for REs: no part of any $s \in \mathcal{L}$ can be pumped (repeated) without taking us out of $\mathcal{L}$.
- In general, any sufficiently long string in an infinite regular language can be pumped: infinity requires $*$.

Forward sets show that the same limitation holds of FSAs:

- Any a^n, a^{n+1} will have different forward sets. Why? They can be continued in different ways. E.g., $a^n \cdot b^n$ vs $a^{n+1} \cdot b^{n+1}$.
- But there's an infinite number of such a^n's to contend with, each with a different forward set.
- We'd need an infinite number of forward sets, hence an infinite number of states. No FSA can do this.

Center embedding

The cat ran.

Center embedding

The cat ran.
NP VP

Center embedding

The cat	the dog	chased	ran.
NP	NP	VP	VP

Center embedding

The cat	the dog	Sam	owns	chased	ran.
NP	NP	NP	VP	VP	VP

Center embedding

The cat	the dog	Sam	owns	chased	ran.
NP	NP	NP	VP	VP	VP

* The cat the dog Sam owns ran.
NP NP NP VP VP

REs ~ FSAs

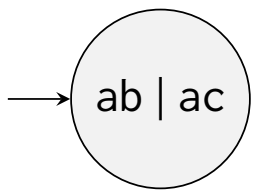
This identical limitation of REs and FSAs is no coincidence: REs and FSAs are capable of generating the exact same languages!

We will not show this rigorously, but I will sketch methods for translating from a RE to a FSA, and vice versa.

RE $\longrightarrow$ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels $\longrightarrow$ numbers).

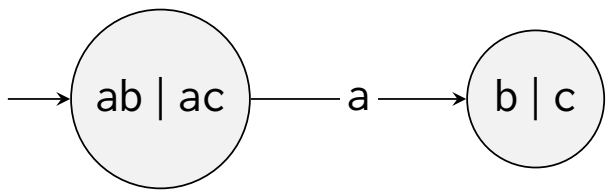


This method is due to Brzozowski 1964.

RE $\longrightarrow$ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels $\longrightarrow$ numbers).

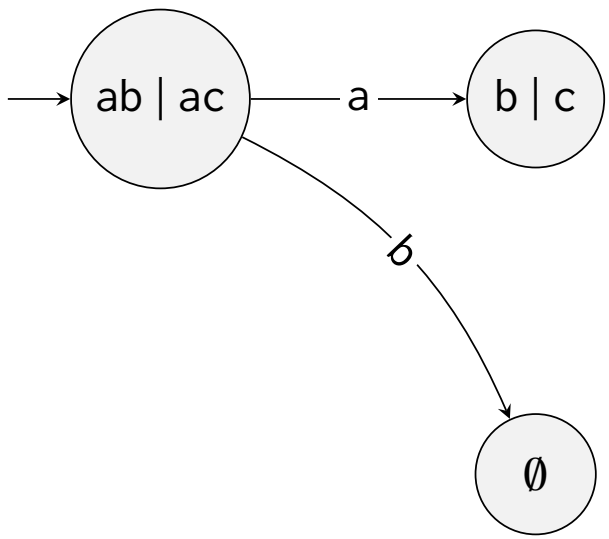


This method is due to Brzozowski 1964.

RE ⟶ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels ⟶ numbers).

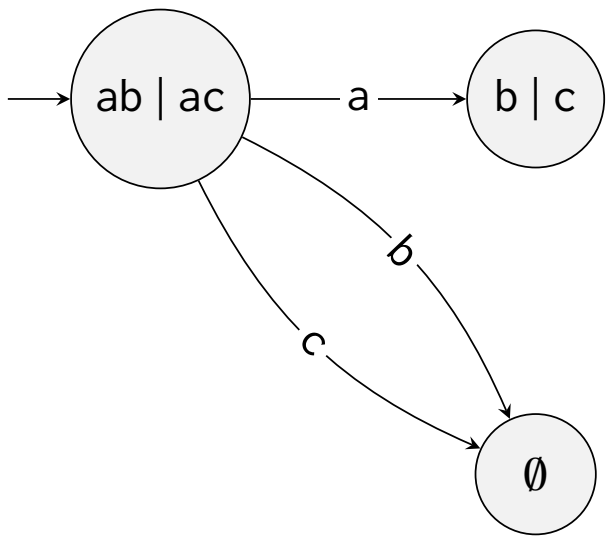


This method is due to Brzozowski 1964.

RE ⟶ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels ⟶ numbers).

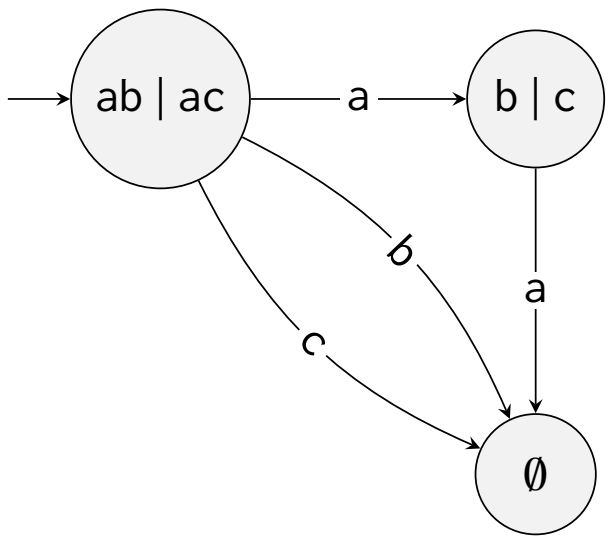


This method is due to Brzozowski 1964.

RE ⟶ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels ⟶ numbers).

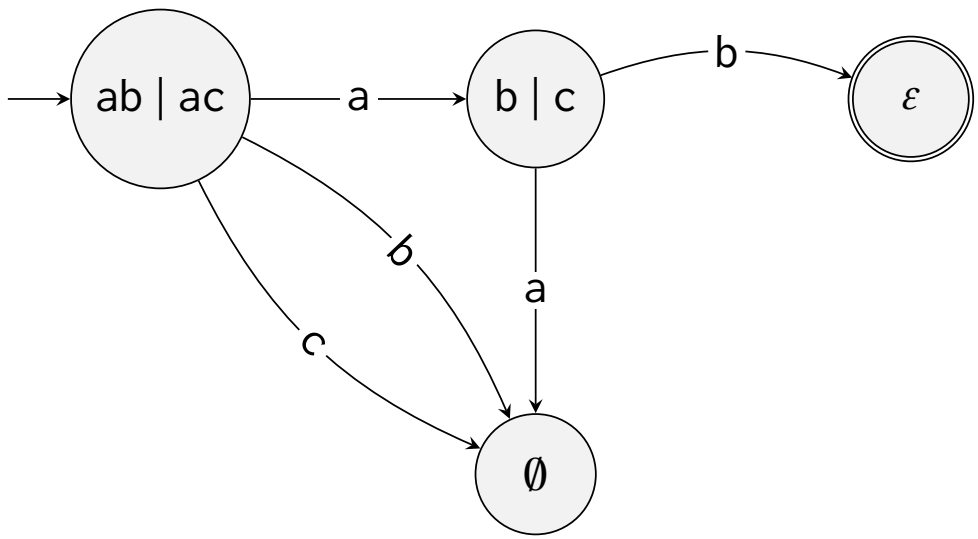


This method is due to Brzozowski 1964.

RE ⟶ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels ⟶ numbers).

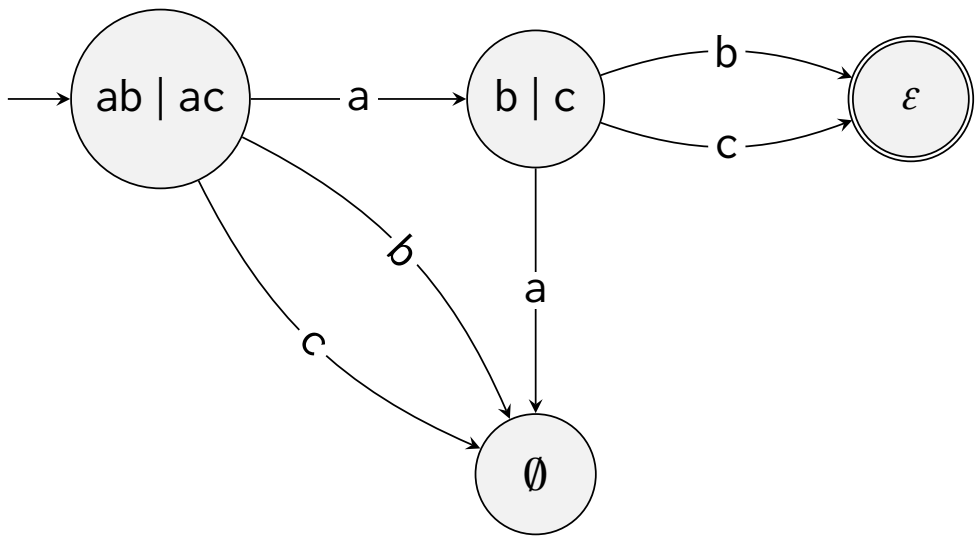


This method is due to Brzozowski 1964.

RE ⟶ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels ⟶ numbers).

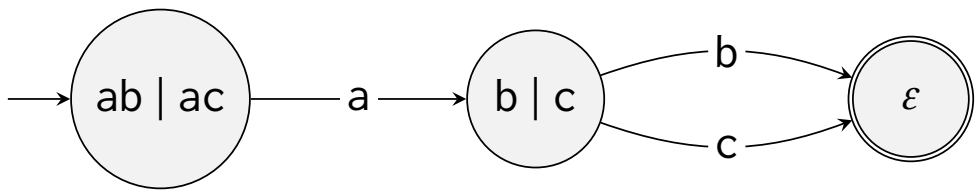


This method is due to Brzozowski 1964.

RE ⟶ FSA

To turn an RE r into an FSA, make an intial state labeled with r, then:

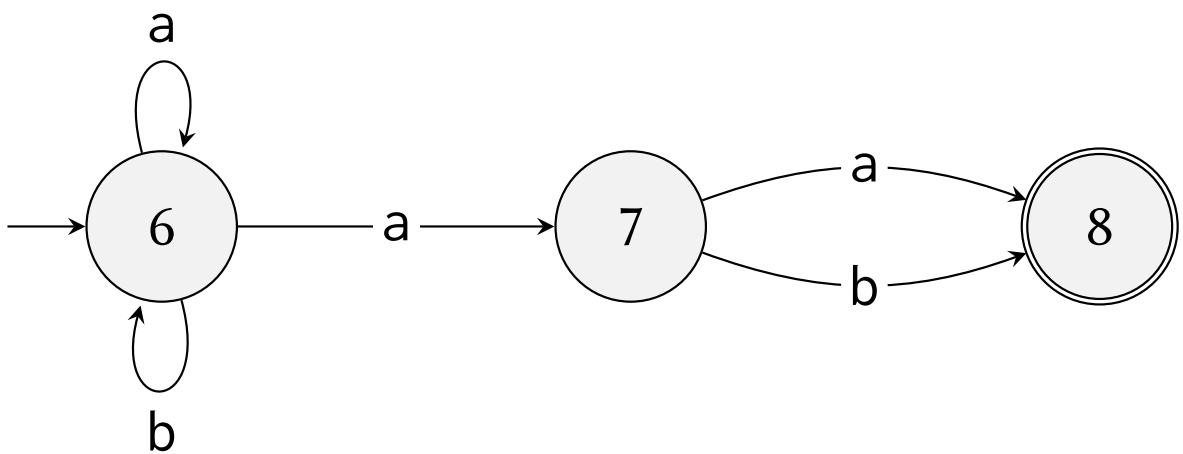
- Take the **derivative** of r with respect to each $\sigma \in \Sigma$.
- Repeat until you end up with ε. This is your accepting state.
- Optionally: erase the dead end $\emptyset$ (and/or labels ⟶ numbers).



This method is due to Brzozowski 1964.

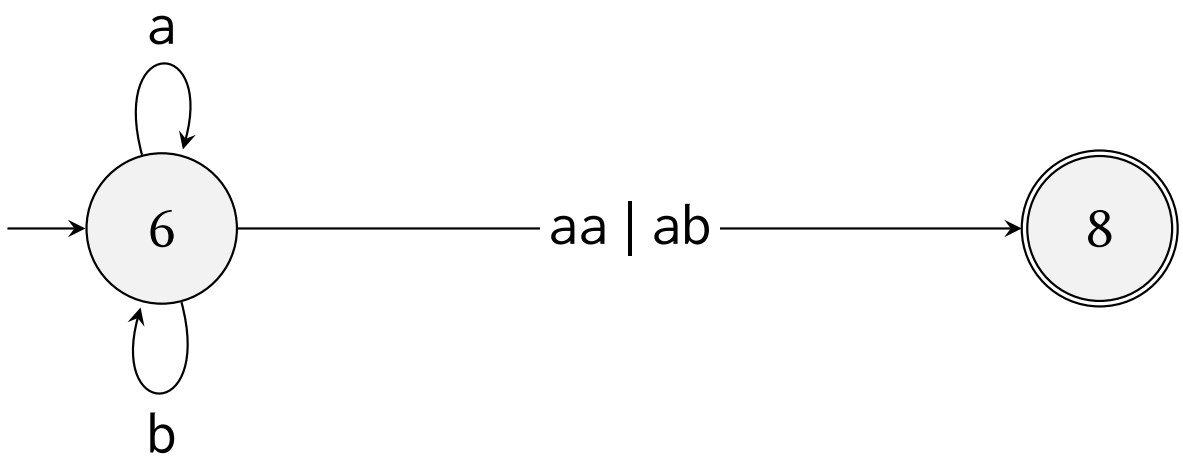
FSA $\longrightarrow$ RE

To turn an FSA into an RE, erase nodes, preserving the information you have removed in the transition arrows:



FSA ⟶ RE

To turn an FSA into an RE, erase nodes, preserving the information you have removed in the transition arrows:



FSA $\longrightarrow$ RE

To turn an FSA into an RE, erase nodes, preserving the information you have removed in the transition arrows: