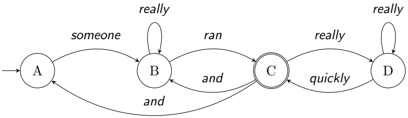
Introducing finite state automata

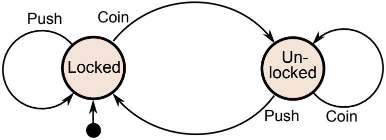
Computational Linguistics (LING 455)

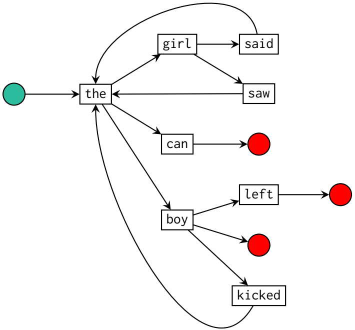
Rutgers University October 15, 2021

Finite state machines

Some FSAs we've encountered







Automata and FSAs

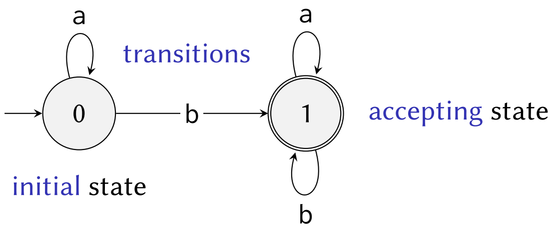
Automaton (pl. **automata**): *abstract* machine that represents…

- the memory a computation requires
- the complexity of a computation

Finite state acceptor (FSA): parses a string, **accepts** or **rejects** it

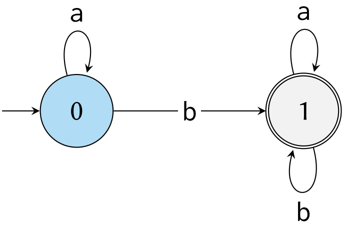
- equivalent to regular expressions!

Walking paths in an FSA



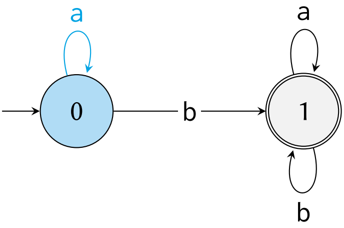
Walking paths in an FSA

abaab



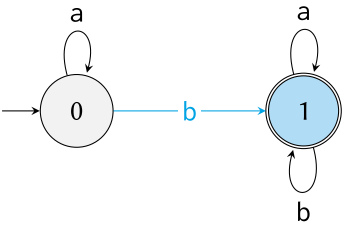
Walking paths in an FSA

abaab



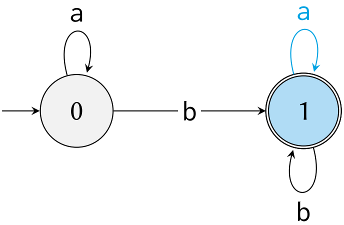
Walking paths in an FSA

abaab



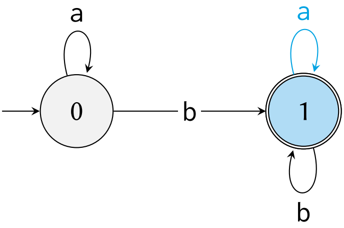
Walking paths in an FSA

abaab



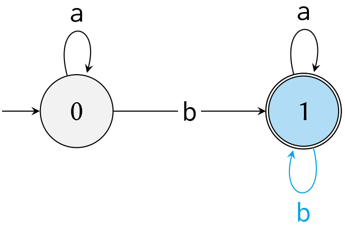
Walking paths in an FSA

abaab



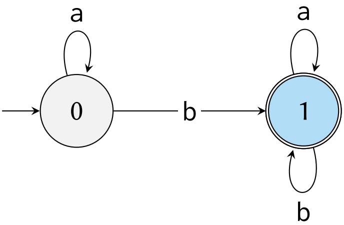
Walking paths in an FSA

abaab



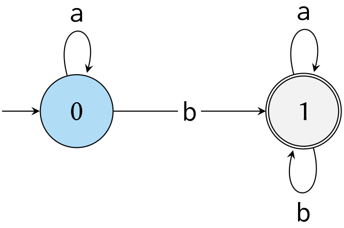
Walking paths in an FSA

abaab ✓



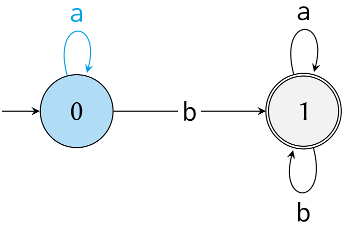
Walking paths in an FSA

abaab ✓
aaaaa



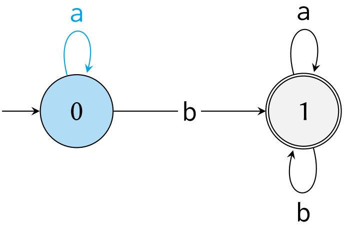
Walking paths in an FSA

abaab ✓
aaaaa



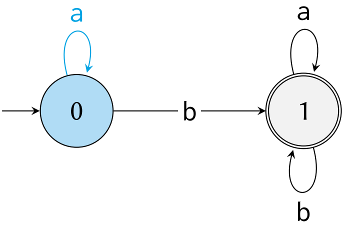
Walking paths in an FSA

abaab ✓
aaaaa



Walking paths in an FSA

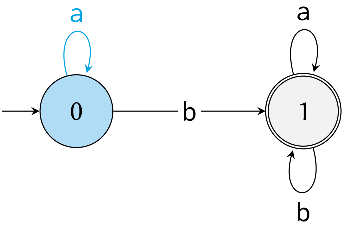
abaab ✓
aaaaa



Walking paths in an FSA

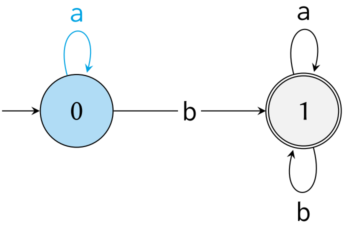
abaab ✓

aaaaa

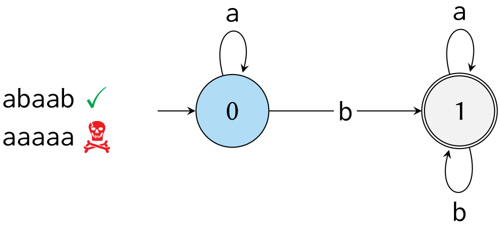


Walking paths in an FSA

abaab ✓
aaaaa



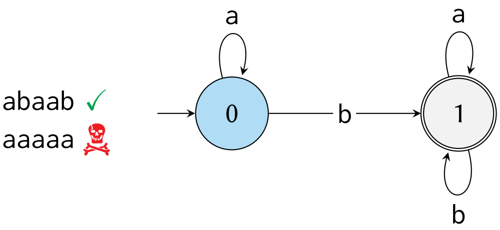
Walking paths in an FSA



What strings does this FSA recognize?

What is an RE that characterizes this pattern?

Walking paths in an FSA

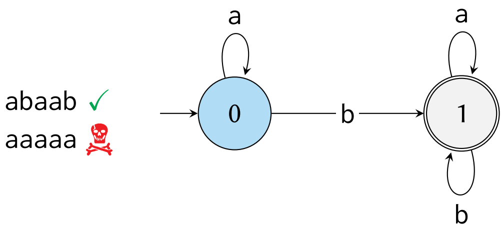


What strings does this FSA recognize?

- Strings over $\Sigma = \{a, b\}$ with at least one b

What is an RE that characterizes this pattern?

Walking paths in an FSA



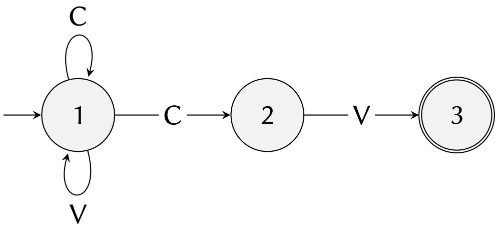
What strings does this FSA recognize?

- Strings over $\Sigma = \{a, b\}$ with at least one b

What is an RE that characterizes this pattern?

- $\Sigma^* \cdot b \cdot \Sigma^*$ [or, $a^* \cdot b \cdot \Sigma^*$]

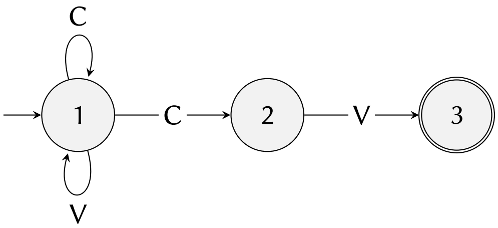
FSA practice



What strings does this FSA recognize?

What is an RE that characterizes this pattern?

FSA practice

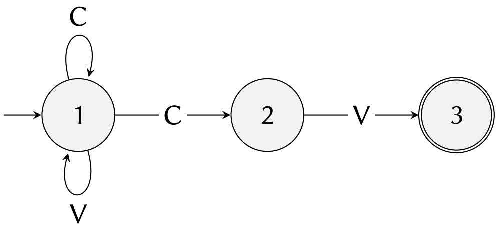


What strings does this FSA recognize?

- Strings over $\Sigma = \{C, V\}$ ending with CV

What is an RE that characterizes this pattern?

FSA practice



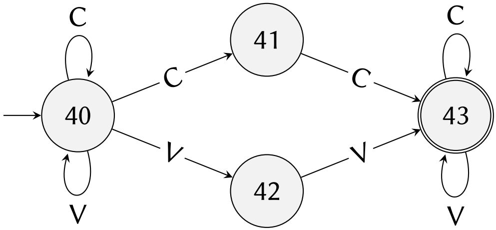
What strings does this FSA recognize?

- Strings over $\Sigma = \{C, V\}$ ending with CV

What is an RE that characterizes this pattern?

- $\Sigma^* \cdot C \cdot V$

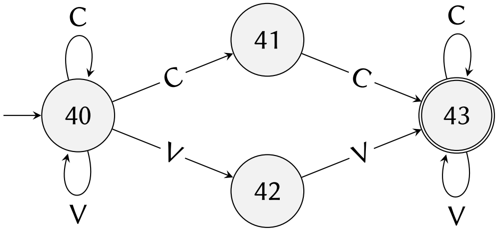
More FSA practice



What strings does this FSA recognize?

What is an RE that characterizes this pattern?

More FSA practice

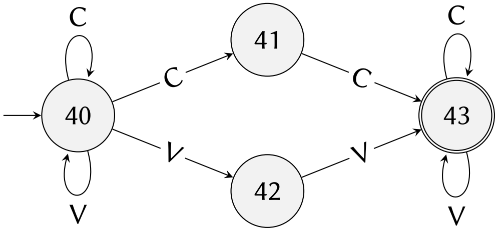


What strings does this FSA recognize?

- Strings over $\Sigma = \{C, V\}$ with 2 adjacent C's, or 2 adjacent V's

What is an RE that characterizes this pattern?

More FSA practice



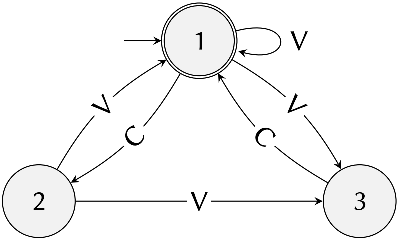
What strings does this FSA recognize?

- Strings over $\Sigma = \{C, V\}$ with 2 adjacent C's, or 2 adjacent V's

What is an RE that characterizes this pattern?

- $\Sigma^* \cdot ((C \cdot C) \mid (V \cdot V)) \cdot \Sigma^*$

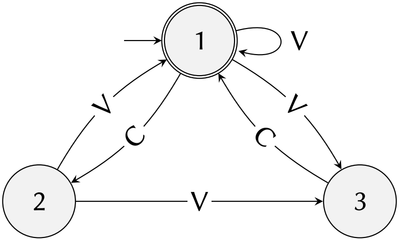
Even more FSA practice



What strings does this FSA recognize?

What is an RE that characterizes this pattern?

Even more FSA practice

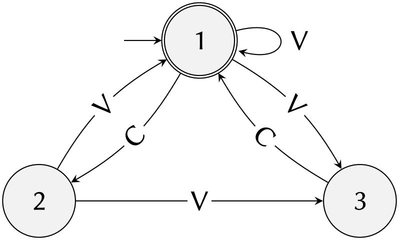


What strings does this FSA recognize?

- Strings over $\Sigma = \{C, V\}$ built from repetitions of (C)V(C)

What is an RE that characterizes this pattern?

Even more FSA practice



What strings does this FSA recognize?

- Strings over $\Sigma = \{C, V\}$ built from repetitions of (C)V(C)

What is an RE that characterizes this pattern?

- $((C \mid \varepsilon) \cdot V \cdot (C \mid \varepsilon))^*$

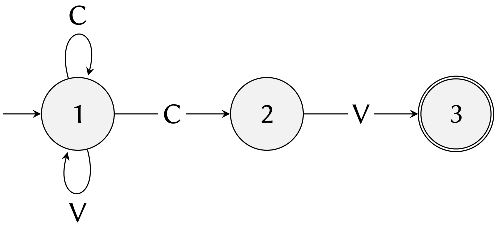
FSAs formally

Official definition

An FSA is a 5-tuple $(Q, \Sigma, I, F, \Delta)$ where:

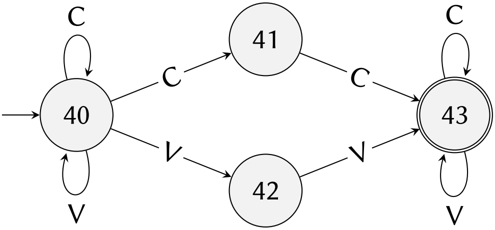
- Q is a set of states — we'll use numbers to name states
- Σ is an alphabet of symbols — same as REs
- I is the states in Q that are initial — entering arrow
- F is the states in Q that are final — circled 2x
- Δ is the set of transitions — labeled arrows
  - Transitions are triples (q, σ, q') of a starting state q, an ending state q', and a letter σ

Example: the data in an FSA graph



- $Q = \{1, 2, 3\}$
- $\Sigma = \{C, V\}$
- $I = \{1\}$
- $F = \{3\}$
- $\Delta = \{(1, C, 1), (1, V, 1), (1, C, 2), (2, V, 3)\}$

Another example



- $Q = \{40, 41, 42, 43\}$
- $\Sigma = \{\mathrm{C}, \mathrm{V}\}$
- $I = \{40\}$
- $F = \{43\}$
- $\Delta = \{(40, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43),$
  $(40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, 43)\}$

On 'isomorphism'

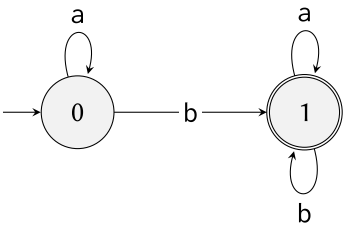
You should convince yourself that the two representations of FSAs convey equivalent information:

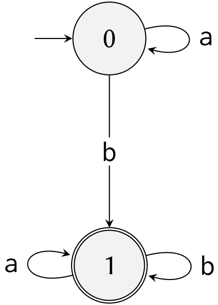
- Given $(Q, \Sigma, I, F, \Delta)$ you could draw the picture
- Given the picture you could determine $(Q, \Sigma, I, F, \Delta)$

Moreover, you could always convert one representation to another, and then convert it back. You wouldn't ever lose anything by doing so. This is just what is meant by **isomorphism**.

It is true that the 'flat' representation leaves out certain pictorial details, like the orientation of the graph. But all such details are unimportant to how the FSA actually works.

Two equivalent FSA graphs





- $Q = \{0, 1\}$
- $\Sigma = \{a, b\}$
- $I = \{0\}$
- $F = \{1\}$
- $\Delta = \{(0, a, 0), (0, b, 1), (1, a, 1), (1, b, 1)\}$

When does an FSA accept a string?

If we have a string of 3 symbols, xyz, $(Q, \Sigma, I, F, \Delta)$ generates it if there are 4 states q_0, q_1, q_2, q_3 in Q such that:

- $q_0 \in I$ — q_0 is initial
- $(q_0, x, q_1) \in \Delta$ — $q_0 \xrightarrow{x} q_1$
- $(q_1, y, q_2) \in \Delta$ — $q_1 \xrightarrow{y} q_2$
- $(q_2, z, q_3) \in \Delta$ — $q_2 \xrightarrow{z} q_3$
- $q_3 \in F$ — q_3 is final

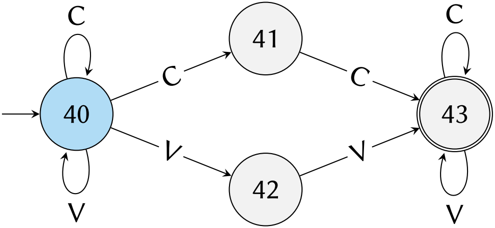
Less formally: if you can walk a path in the FSA starting with an initial state, ending with the final state, labeled with x, y, and z.

More generally

$(Q, \Sigma, I, F, \Delta)$ generates $x_1 \ldots x_n$ if there are states $q_0 \ldots q_n$ such that:

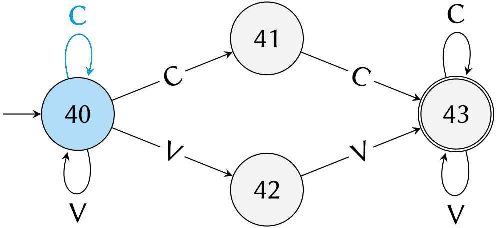
- $q_0 \in I$ — q_0 is initial
- $(q_0, x_1, q_1) \in \Delta$ — $q_0 \xrightarrow{x_1} q_1$
- $(q_1, x_2, q_2) \in \Delta$ — $q_1 \xrightarrow{x_2} q_2$
- $\ldots$ — $\ldots$
- $(q_{n-1}, x_n, q_n) \in \Delta$ — $q_{n-1} \xrightarrow{x_n} q_n$
- $q_n \in F$ — q_n is final

Example: CVVCV



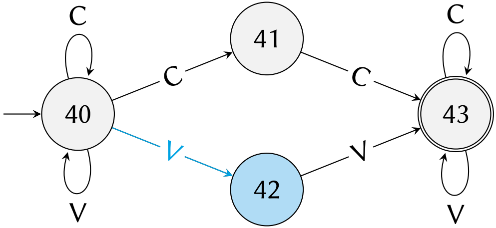
$$\Delta = \{(\overset{I}{40}, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43), (40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, \underset{F}{43})\}$$

Example: CVVCV



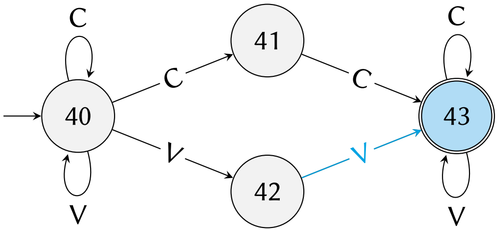
$$\Delta = \{\overset{I}{(40}, C, 40), (40, V, 40), (40, C, 41), (41, C, 43),$$
$$(40, V, 42), (42, V, 43), (43, C, 43), (43, V, \underset{F}{43)}\}$$

Example: CVVCV



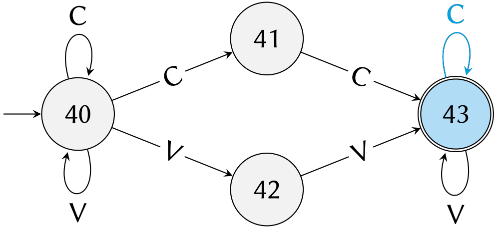
$$\Delta = \{(\overset{I}{40}, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43),$$
$$(40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, \underset{F}{43})\}$$

Example: CVVCV



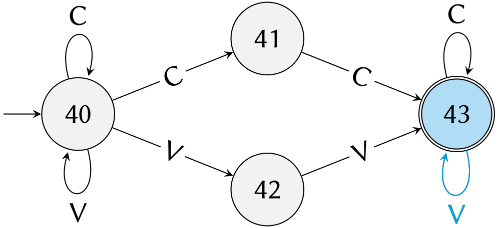
$$\Delta = \{(\overset{I}{40}, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43),$$
$$(40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, \underset{F}{43})\}$$

Example: CVVCV



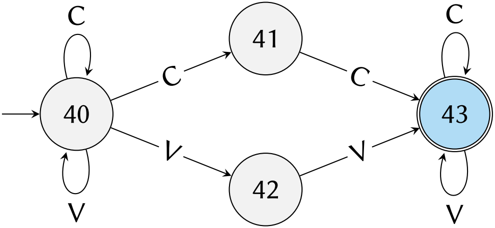
$$\begin{aligned}\Delta = \{&(\overset{I}{40}, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43),\\ &(40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, \underset{F}{43})\}\end{aligned}$$

Example: CVVCV



$$\Delta = \{(\overset{I}{40}, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43),\\ (40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, \underset{F}{43})\}$$

Example: CVVCV ✓



$$
\begin{aligned}
\Delta = \{&(\overset{I}{40}, \mathrm{C}, 40), (40, \mathrm{V}, 40), (40, \mathrm{C}, 41), (41, \mathrm{C}, 43),\\
&(40, \mathrm{V}, 42), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, \underset{F}{43})\}
\end{aligned}
$$

FSAs in Haskell

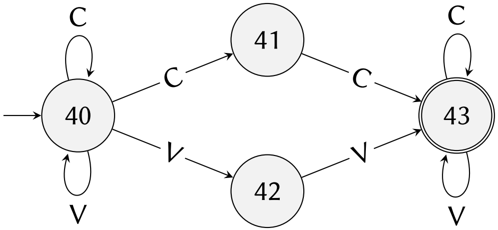
The type of FSA's

Recalling: an FSA is a 5-tuple $(Q, \Sigma, I, F, \Delta)$ where:

- Q is a set of states (we'll use numbers)
- Σ is an alphabet (same as REs)
- I is the states in Q that are initial
- F is the states in Q that are final
- Δ is the set of transitions

```
type State = Int
type FSA sym = ( [State]                -- Q
               , [sym]                  -- Σ
               , [State]                -- I
               , [State]                -- F
               , [Transition sym] )     -- Δ
type Transition sym = (State, sym, State)
```

Example FSA




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

fsaCCVV :: FSA CV
fsaCCVV = ( [40,41,42,43], [C,V], [40], [43],
            [(40,C,40),(40,V,40),(40,C,41),(41,C,43),
             (40,V,42),(42,V,43),(43,C,43),(43,V,43)] )
```

Checking that an FSA is legal

```
validFSA :: Eq a => FSA a -> Bool
validFSA (states, syms, i, f, ds) =
  let validState q          = elem q states
      validTrans (q,sym,r) = validState q  &&
                             elem sym syms &&
                             validState r  in
  all validState i && -- all odd  [1,3,5] == True
  all validState f && -- all even [2,3,4] == False
  all validTrans ds
```

```
*W7> validFSA fsaCCVV
True
```

Parsing

The logic of parsing

$(Q, \Sigma, I, F, \Delta)$ generates $x_1 \ldots x_n$ if there are states $q_0 \ldots q_n$ such that:

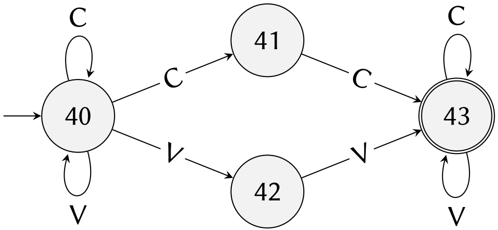
- $q_0 \in I$ — q_0 is initial
- $(q_0, x_1, q_1) \in \Delta$ — $q_0 \xrightarrow{x_1} q_1$
- $(q_1, x_2, q_2) \in \Delta$ — $q_1 \xrightarrow{x_2} q_2$
- … — …
- $(q_{n-1}, x_n, q_n) \in \Delta$ — $q_{n-1} \xrightarrow{x_n} q_n$
- $q_n \in F$ — q_n is final

Informally this is very simple, and suggests a natural algorithm:

- Start at an initial state
- Given x_1, go wherever Δ allows
- Repeat in the new state for x_2
- Keep going
- Once no letters are left, succeed if you are in a final state

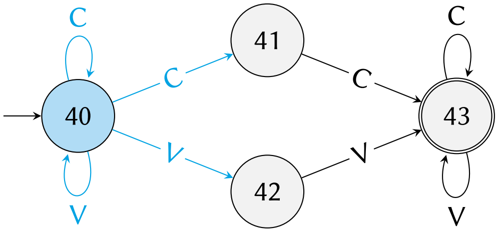
Zooming in

We'll start with a simpler task. If we are currently in a given state q, we know that the only relevant transitions are those with source q:



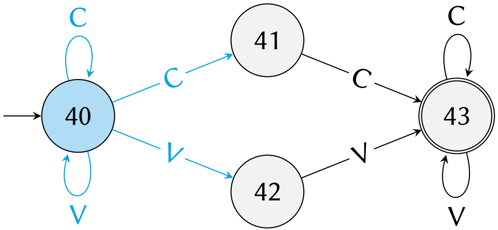
Zooming in

We'll start with a simpler task. If we are currently in a given state q, we know that the only relevant transitions are those with source q:



Zooming in

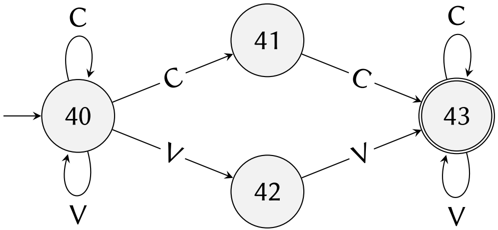
We'll start with a simpler task. If we are currently in a given state q, we know that the only relevant transitions are those with source q:




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

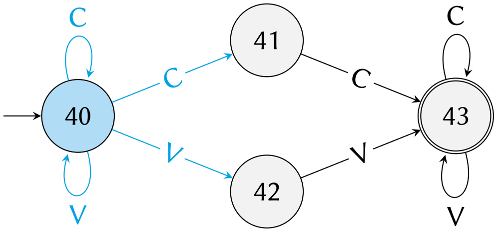
Taking one step

Similarly, given the current symbol in the string, we know which states we can potentially move to next:



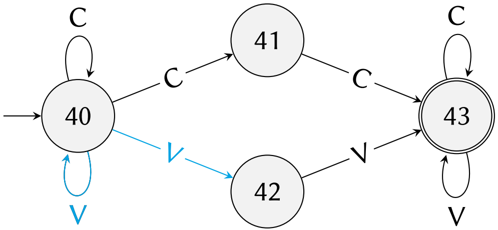
Taking one step

Similarly, given the current symbol in the string, we know which states we can potentially move to next:



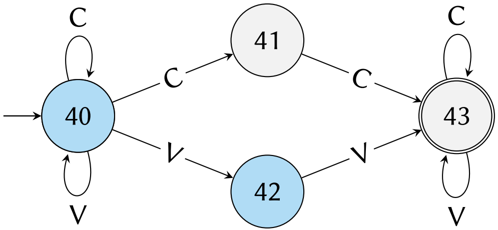
Taking one step

Similarly, given the current symbol in the string, we know which states we can potentially move to next:



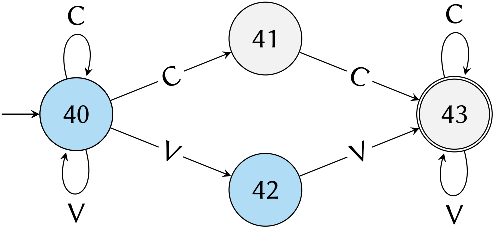
Taking one step

Similarly, given the current symbol in the string, we know which states we can potentially move to next:



Taking one step

Similarly, given the current symbol in the string, we know which states we can potentially move to next:

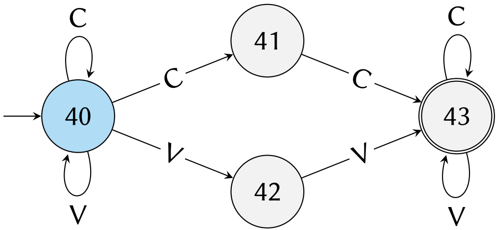



```
step :: Eq a => [Transition a] -> a -> State -> [State]
step delta x q = [ r' | (r,y,r') <- focus delta q, y==x ]
```

From steps to walks: VV

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

- Take a step with the first symbol
- For each possible next state, keep taking steps
- Do it till you run out of string, then rest

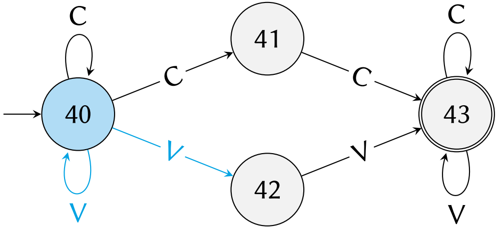


3 ways to parse VV: 40→40→40, 40→40→42, 40→42→43

From steps to walks: VV

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

- Take a step with the first symbol
- For each possible next state, keep taking steps
- Do it till you run out of string, then rest

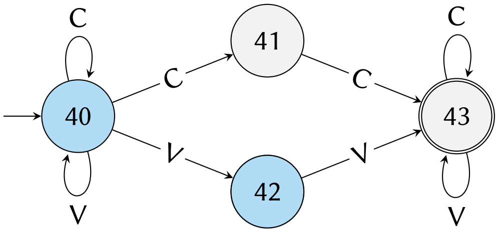


3 ways to parse VV: 40→40→40, 40→40→42, 40→42→43

From steps to walks: VV

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

- Take a step with the first symbol
- For each possible next state, keep taking steps
- Do it till you run out of string, then rest

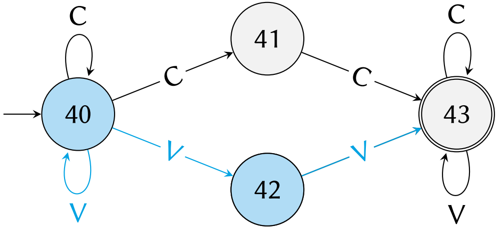


3 ways to parse VV: 40→40→40, 40→40→42, 40→42→43

From steps to walks: VV

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

- Take a step with the first symbol
- For each possible next state, keep taking steps
- Do it till you run out of string, then rest

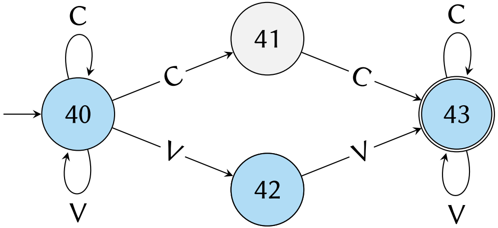


3 ways to parse VV: 40→40→40, 40→40→42, 40→42→43

From steps to walks: VV

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

- Take a step with the first symbol
- For each possible next state, keep taking steps
- Do it till you run out of string, then rest



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

How to use `step` to `walk`

You are in some state `q` with a string `x:xs`…

- Taking one `x`-step opens up new options for where to travel next:

```
nexts = step delta x q
-- :: [State]
```

- By repeatedly taking `step`'s — `walk`-ing — you can construct a path to some final `dests` using the tail `xs`:

```
dests q' = walk delta xs q'
-- :: State -> [State]
```

- Once the string is consumed, your journey is done.

Where we're at

```haskell
walk delta (x:xs) q =
-- ...
  let nexts = step delta x q
--    nexts :: [State]
      dests q' = walk delta xs q' in
--    dests :: State -> [State]
-- ...
```

If we `map dests` over `nexts`, we construct a list of final `dests` **for each** potential next state, type: `[[State]]`. Flatten with `concat`.

- Or simply `concatMap dests nexts`!

Putting it all together

```haskell
walk :: Eq a => [Transition a] -> [a] -> State -> [State]
walk delta str q =
  case str of
    x:xs -> let nexts = step delta x q
--                  nexts :: [State]
                dests q' = walk delta xs q' in
--                  dests :: State -> [State]
                concatMap dests nexts
    []   -> [q]
--  good job, you consumed the string you can rest!
```

`accept`-ance

The only remaining thing is to ensure that we begin in an initial state, and conclude in a final (i.e., accepting) state:

```
accepts :: Eq a => FSA a -> [a] -> Bool
accepts (states, syms, i, f, delta) str =
  or [ elem qn f | q0 <- i, qn <- walk delta str q0 ]
```

- Start at some `q0 <- i`
- Walk `delta`-paths from `q0`, parsing `str` letter by letter
- Test that the destinations `qn` are in `f`
- If at least one is (the job of `or`), the FSA accepts `str`