Automata Theory for Language

Name: Thomas Graf
Email: mathcamp@thomasgraf.net
Course Website: mathcamp.thomasgraf.net
Personal Website: thomasgraf.net

1 Language as a mathematical problem

Every language follows certain rules of grammar. I do not mean the usual grammar-nazi malarkey like "Do not split infinitives! Do not strand prepositions!". Rather, there are words and sentences that are correct, and others that have something wrong with them.

Example 1

Consider the English word *denaturalization*. It is built up from discrete parts:

1. nature
2. nature + al = natural
3. natural + ize = naturalize
4. de + naturalize = denaturalize
5. denaturalize + ation = denaturalization

No other order of these parts produces a good word of English:
denaturizalation, *ationalizenaturde*, *naturalizedeation*, ...

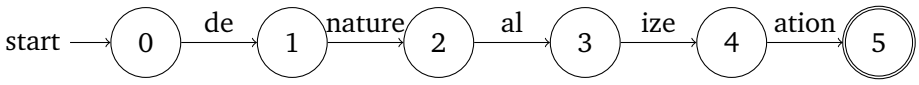
[Exercise 1] How many combinations are mathematically possible?

We see, then, that there must be a very strict system of rules that determines the order in which parts of a word may appear. What can we say about this rule system? Can we study it just like, say, the laws that determine how atoms combine into molecules?

One strategy would be to look at the human brain. After all, language must involve some kind of brain mechanisms that allow us to produce correct forms while avoiding incorrect ones. But research in *neurolinguistics* has shown that this approach won't get us very far — the brain is just too complicated to be studied this way. Instead, we need a more abstract model of these brain mechanisms. And math allows us to do just that!

2 Graphs for language: Finite-state automata

Our mathematical model are *finite-state automata* (FSA), which are a special case of labeled graphs. Here is a simple example of an automaton representing *denaturalization*:



- The circles are the vertices of the graph, which are called *states*.
- The *arcs* connecting the states all must have a label.
- The *start* arrow indicates the beginning point of the automaton, called the *initial state*.
- The doubly circled state is a *final state*. It marks the end point of the automaton.

An FSA can be used to succinctly describe words and sentences. Every path through the automaton that takes you from an initial state to a final state represents a well-formed structure. In the automaton above, there is only one such path, which takes us from 0 to 1, then to 2, 3, 4, and finally 5. But automata can have multiple paths through them.

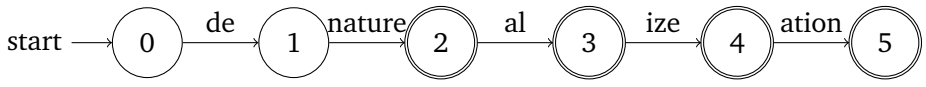
Example 2

Besides *denaturalization*, English also has the words

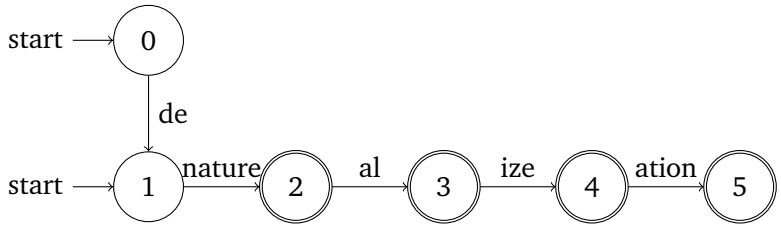
- *nature*,
- *natural*,
- *naturalize*,
- *denaturalize*,
- *naturalization*.

We can modify the previous automaton so that it also allows for these other forms. Let's do this step by step.

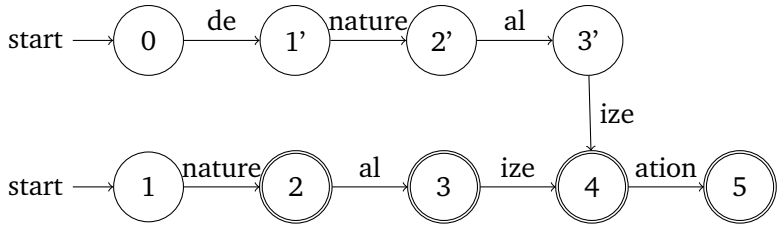
First we want to allow words to also end with *nature*, *al*, and *ize*, so we make the states that are reached through those arcs final.



But every path must still start with *de*, so the automaton does not allow the patterns *nature*, *natural*, or *naturalize*. To fix this, we make 1 an initial state, too.



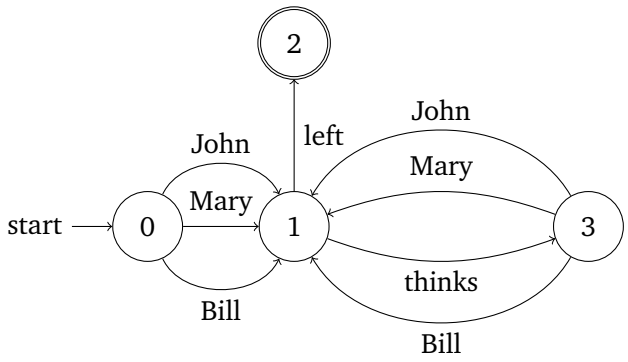
But this automaton is too general, it also allows patterns like *denature* or *denatural*. This is a little trickier to fix: we add a few more states to keep track of the fact that after *de*, the first final state can only be one reached by *ize*.



When you now try all possible paths from an initial state to a final state, you'll see that they are exactly

- *nature*,
- *natural*,
- *naturalize*,
- *denaturalize*,
- *naturalization*,
- *denaturalization*.

FSAs can be much more complex than this. As long as the number of states is finite, pretty much any graph you can imagine is an FSA. You can have multiple arcs leaving a node, multiple arcs leading to the same node, or arcs that lead back to a previous node, possibly even the one they came from (*loops*). That is to say, you can have convoluted automata like the one below:



[Exercise 2] What collection of sentences is described by the automaton above?

3 Some curious limits of language

Linguists put the parts that make up *denaturalization* into three distinct groups:

stem the base form, which can appear by itself; *nature*

prefix a part that can only appear before a stem; *de-*

suffix a part that can only appear after a stem; *-al*, *-ize*, *-ation*

[Exercise 3] Prefixes and suffixes can sometimes be iterated. Your grandfather's father is your *great grandfather*, whose father is your *great great grandfather*, whose father is your *great great great grandfather*, and so on. Try to write an FSA that produces *grandfather*, *great grandfather*, and so on. You may analyze *grandfather* as a stem.

[Exercise 4] Of course we also have *great grandmother*, *great great grandmother*, and so on. How could the automaton from the previous example be modified to also allow these words?

Those are not the only classes — among other things there are also *circumfixes*. An example of a circumfix is German *ge- -t*, which is used to form the participle form of a verb. So *rauf-en* 'brawl-to' can be turned into *ge-rauf-t* '(has) brawled'.

[Exercise 5] German actually has a split with respect to this circumfix. For some verbs it is *ge- -t*, for others it is *ge- -en*. And for some verbs, the form is irregular.

Verb stem	Infinitival form	Past participle form
lauf 'run'	laufen	gelaufen
rauf 'brawl'	raufen	gerauft
sauf 'guzzle'	saufen	gesoffen
tauf 'baptize'	taufen	getauft

Write an FSA that produces all the correct infinitival and past participle forms, and only those. Try to keep the number of states as small as possible.

One would expect that languages can freely choose whether they use prefixes, suffixes or circumfixes. But this does not seem to be the case. What we find is that across languages, circumfixes do not seem to be iterable.

Example 3 Saying 'the day after tomorrow' in German and Ilocano

German has a much simpler way than English to say 'the day after tomorrow'. You just take *morgen* (the word for 'tomorrow') and add the prefix *über* 'over'. What makes this even nicer is that *über* can be iterated.

1. morgen 'tomorrow'
2. über-morgen 'the day after tomorrow'
3. über-über-morgen 'the day after the day after tomorrow'
4. ...

Ilocano (an Austronesian language spoken in the Philippines) also has a single word for 'the day after tomorrow', but it is built with the circumfix *ka- -an* instead of a prefix.

1. bigat 'tomorrow'
2. ka-bigat-an 'the day after tomorrow'

In contrast to German *über-*, however, Ilocano *ka- -an* cannot be freely iterated. So you cannot say something like *ka-ka-bigat-an-an*.

[Exercise 6] It is very easy to give an FSA for the German pattern — it's just a variation of the *great* ⋯ *great grandfather* pattern we saw before for English. Similarly, an automaton for the Ilocano pattern is easy to come up with. But what if Ilocano were also unbounded, so that we could also have words like *ka-ka-ka-bigat-an-an-an*, i.e. words with the same number of *ka-* prefixes and *-an* suffixes. Can you write an automaton for this?

The difference between German *über-* and Ilocano *ka- -an* is an interesting puzzle, and it is part of a much larger observation: prefixes and suffixes can be freely iterated, but circumfixes cannot. Why should languages work this way? FSAs provide an answer.

4 The limits of finite-state automata (aka the hard part)

FSAs are fairly powerful devices, and they have found many applications in the real world. They control elevators, help biologists with genome sequencing, give you word suggestions when you write a text message, and are even involved in the automatic creation of subtitles for Youtube videos. Tons of technology uses FSAs, including language technology.

But FSAs cannot do everything. There is a very strict bound on the complexity of the patterns they can handle. And what we will see now is that even though FSAs can iterate prefixes and suffixes (as you've already shown yourself with the FSA for *great* ⋯ *great grandfather*), they cannot correctly iterate circumfixes.

Before we state the theorem, let us think about how FSAs actually work. In particular, what do the states stand for? Suppose you have taken a path that leads to state 1 in the convoluted automaton from the previous section. How can you continue to get a well-formed pattern? Well, by taking any path that leads you from 1 to a final state. There may be multiple paths to choose from, but anyone will do. It does not

matter how you actually got into 1. Whether you came from 0 to 1 via *John*, *Mary*, or *Bill* is completely irrelevant, the only thing that matters is that you are in state 1 now and may take any path that gets you from 1 to 2.

Crucial Insight 1 If you have paths that take you from an initial state to the same state, then those paths can be continued in exactly the same fashion towards a final state.

FSAs have only finitely many states. Suppose we have some pattern, and we compare all paths that could possibly arise in that pattern. We put two paths in the same bin if they can be continued in exactly the same fashion. Then the pattern can be handled by an FSA only if we end up with a finite number of distinct bins.

Example 4 Bins for the *great* ⋯ *great grandfather* Pattern

Consider the pattern *grandfather*, *great grandfather*, *great great grandfather*, and so on. If the pattern starts with *grandfather*, then there is nothing else that can be added, so there are no continuation paths at all. For *great*, one possible continuation path is *grandfather*, producing *great grandfather*. But we could also continue with *great grandfather*, *great great grandfather*, and so on. Here's a table:

Incoming Path	Continuation Paths
grandfather	—
great	grandfather, great grandfather, …
great great	grandfather, great grandfather, …
great great great	grandfather, great grandfather, …
⋮	

So there's only two bins for paths. One is for *grandfather*, which cannot be continued. The other one is for any path that starts with *great*, as they can all be continued by an arbitrary number of *great*s followed by *grandfather*. Since we only have two bins, the pattern can be handled by an FSA.

The bins are essentially the states in the automaton: two strings are in the same bin because they take you to the same state in the automaton, and from there you can proceed in a specific manner irrespective of how you got there. But a finite-state automaton can only have a finite number of states, so it can only describe patterns where the number of distinct bins is finite.

Crucial Insight 2 A pattern can be captured by an FSA only if its paths can be grouped into finitely many bins.

This insight allows us to show that unbounded circumfixation is impossible with FSAs. The proof is a joint class exercise.

5 Wrapping up

The iterability difference between prefixes and suffixes on the one hand and circumfixes on the other is really puzzling from non-mathematical perspective. But once we look

at them from a mathematical perspective, we see that there is a huge complexity difference: FSAs can iterate prefixes and suffixes, but not circumfixes. So if the mechanisms our brain uses to handle word structure are similar to FSAs, then it is not surprising that circumfixes are never iterated in any languages — the relevant mechanisms simply cannot do it!

We have accomplished something that few people would consider possible: we have studied language from a mathematical perspective, and we were able to explain properties that are shared by all human languages by purely mathematical means.

The Take Home Message. Mathematics explains language!

[Exercise 7] Word structure indeed seems to be fully captured by FSAs. But there are ways to show that the structure of English sentences is more complicated then what FSAs can handle. Do you have an idea what the argument might be?

Hint: The sentences below play a crucial role.

- A man left.
- A man that a woman saw left.
- A man that a woman that a woman saw saw left.
- A man that a woman that a woman that a woman saw saw saw left.

6 I wanna learn more!

Cool, here's a few pointers. I am also working on an interactive learning platform that teaches mathematical techniques for linguistics, and **I am looking for volunteers** to test it and provide feedback. If you are interested, or if you have any questions on today's material, shoot me a line: mail@thomasgraf.net

- **Automata Theory**
  - *Introduction to the Theory of Computation*
    undergraduate course at Stony Brook, Computer Science
  - Michael Sipser. 2012. 3rd edition.
    Introduction to the Theory of Computation. (yep, that's the same name)

    Pro tip: The 2nd edition is almost the same and used copies are much cheaper. A decent scan can easily be found on the first Google page, I'm not sure about its legality.
- **Computational Linguistics**
  - *North American Computational Linguistics Olympiad*
    we have training sessions at Stony Brook
  - *Language and Technology*
    undergraduate course at Stony Brook, Linguistics
  - *Computational Linguistics*
    undergraduate course at Stony Brook, Linguistics

  - *Natural Language Processing*
    undergraduate course at Stony Brook, Computer Science
  - Markus Dickinson, Chris Brew, and Detmar Meurers. 2012.
    Language and Computers.

    Used copies are available for 10 bucks on Amazon.

- **Language from a Formal and Cognitive Perspective**
  - Check out my website, teaching materials can be found under *Students*
  - Steven Pinker. *The Language Instinct: How the Mind Creates Language*.

- **The Human Mind as a Computer**
  - Douglas Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*.

Addendum: Automata as matrix multiplication

We probably won't have time to cover this in class, but I figured it's too cool a thing not to include on the handout.

Background: Matrix multiplication

You all probably know how to calculate the dot product of two vectors. First you multiply their components to get a single vector, and then you calculate its magnitude by summing. For example:

$$\begin{pmatrix}0\\2\\3\end{pmatrix}\cdot\begin{pmatrix}8\\5\\2\end{pmatrix}=\left|\begin{pmatrix}0\cdot 8\\2\cdot 5\\3\cdot 2\end{pmatrix}\right|=\left|\begin{pmatrix}0\\10\\6\end{pmatrix}\right|=0+10+6=16$$

Matrices are a generalization of vectors where we can have multiple columns in addition to multiple rows.

$$\begin{pmatrix}5&2\\3&1\\0&2\end{pmatrix}$$

Like vectors, matrices can also be multiplied, but things are slightly different. First, the number of columns of the first matrix must match the number of rows of the second matrix.

$$\begin{pmatrix}5&2\\3&1\\0&2\end{pmatrix}\cdot\begin{pmatrix}1&2&0&2\\3&0&1&3\end{pmatrix}=\text{alright, let's do this!}$$

$$\begin{pmatrix}5&2\\3&1\\0&2\end{pmatrix}\cdot\begin{pmatrix}1&2&1\\3&0&2\\5&1&0\end{pmatrix}=\text{sorry, no can do!}$$

As long as that is the case, we can multiply two matrices as follows:

1. Write down the **first row of the first matrix** as a vector (top-down rather than left-to-right).
2. Calculate the dot product of this vector and the **first column of the second matrix**. Write it down.
3. Do the same with the **second column of the second matrix**, and put the result next to the one you already wrote down.
4. Do the same with all other rows of the second matrix.
5. First cycle done. Now repeat the same steps with the **second row of the first matrix**.
6. Continue until all rows are done. You've got yourself a shiny new matrix.

Example 5 Matrix multiplication

Let's try the following matrix multiplication:

$$\begin{pmatrix} 5 & 2 \\ 3 & 1 \\ 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 0 & 2 \\ 3 & 0 & 1 & 3 \end{pmatrix} = \textbf{???}$$

We write down the first row of the first matrix as a vector.

$$\begin{pmatrix} 5 \\ 2 \end{pmatrix}$$

Next we have to calculate the dot products for each column of the second matrix.

$$\begin{pmatrix} 5 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \end{pmatrix} = 11 \qquad \begin{pmatrix} 5 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} = 10 \qquad \begin{pmatrix} 5 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 2 \qquad \begin{pmatrix} 5 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \end{pmatrix} = 16$$

So the first row for the new matrix is (11 10 2 16). Repeating the same procedure with the second and third row, we can solve the full equation.

$$\begin{pmatrix} 5 & 2 \\ 3 & 1 \\ 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 0 & 2 \\ 3 & 0 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 11 & 10 & 2 & 16 \\ 6 & 6 & 1 & 9 \\ 6 & 0 & 2 & 6 \end{pmatrix}$$

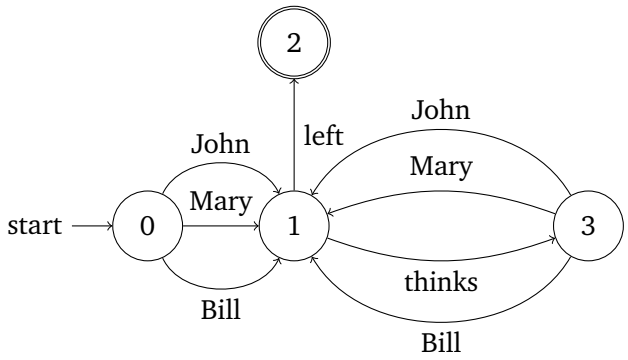
Boolean matrix multiplication for FSAs

Boolean matrices are a special case where all matrices can only have 0 and 1 as their components. So *Boolean* is just another term for *binary*. Boolean matrix multiplication works almost exactly like normal matrix multiplication, except that the value of a dot product can never exceed 1:

$$\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = 1 \quad \text{(not 2)}$$

Every FSA can be represented as a collection of matrices. This is best explained with an example.

Here's an automaton we've seen before.



We already know that it accepts *John thinks Mary left*, but not *John thinks Mary left Bill*. We can get the very same result through matrix multiplication.

First, every one of the symbols *John*, *Mary*, *Bill*, *thinks*, and *left* is given associated with a matrix that encodes whether one can go from some state to another state with this symbol. This works as follows. Imagine that we identify each state with a row and with a column as in the template below.

$$\begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} \end{array}$$

We put a 1 in a cell in row x and column y iff it is possible to go from state x to state y with the symbol we're building the matrix for.

Here's how this works for John. With *John* we cannot go from state 0 to state 0, so we put a 0 in the very first cell. But we can go from 0 to state 1, so the next cell gets a 1. *John* by itself cannot take us from state 0 to state 2 or 3, so the next two cells are also 0.

$$\begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ & & & \\ & & & \\ & & & \end{pmatrix} \end{array}$$

We could in fill the other rows step by step, too, but there's a smarter way. Looking at the automaton, we can see immediately that *John* can only take us from 0 to 1 and from 3 to 1. The first one we've already kept track of in the first row, leaving only the move from 3 to 1. To capture this, we just put a 1 in the second column of the last row. All other cells are filled with 0.

$$\begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Repeating the same procedure for the other symbols, we get the following matrices:

$$\begin{array}{ccc} \textbf{John/Mary/Bill} & \textbf{thinks} & \textbf{left} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

We're almost done, only two more matrices are needed. One records which states we can go to from the start.

$$\begin{array}{c} \\ \textbf{start} \end{array} \begin{array}{c} \begin{array}{cccc} \textbf{0} & \textbf{1} & \textbf{2} & \textbf{3} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

And the other records from which states we can move to the exit.

$$\begin{array}{c} \\ \textbf{0} \\ \textbf{1} \\ \textbf{2} \\ \textbf{3} \end{array} \begin{array}{c} \textbf{exit} \\ \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \end{array}$$

Now we're finally able to compute for any given string whether it is accepted by the automaton. Let's do the calculation for the accepted *John thinks Mary left*.

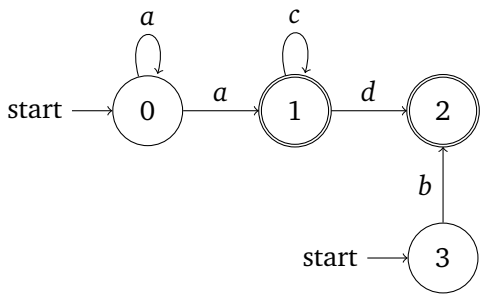
$$\begin{aligned} &\textbf{start} \cdot \textbf{John} \cdot \textbf{thinks} \cdot \textbf{Mary} \cdot \textbf{left} \cdot \textbf{exit} \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ &= 1 \end{aligned}$$

We got a 1. If you do the calculation with the bad sentence *John thinks Mary*, instead, you'll get a 0. So good sentences always result in a 1, bad sentences in a 0.

[Exercise 8] Calculate the values for the following strings:

1. John
2. John thinks Mary left Bill.
3. John thinks Bill thinks Mary left.

[Exercise 9] Consider the automaton below.



Represent it with Boolean matrices and the calculate the values of the following strings:

1. a
2. b
3. aa
4. ad
5. bd
6. $accccc$

Bottom-line: FSAs $\approx$ specific kind of matrix multiplication