CS3102 Theory of Computation

11011 ✓

$\varepsilon = \varepsilon\ \varepsilon$

$\{0,1\}^* - XOR$

Warm up:

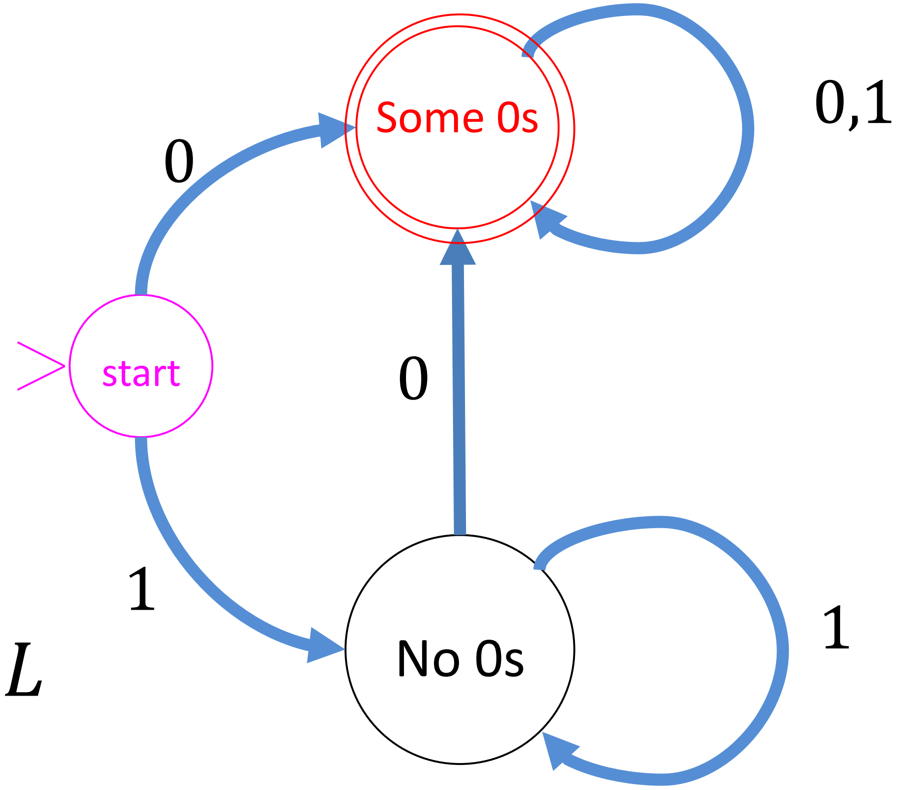
$$XOR = \{x \in \{0,1\}^* \mid x \text{ has an odd number of 1s}\}$$

Write a regex for XOR^c (i.e. $\overline{XOR}$, i.e. the complement of XOR)

$$\underbrace{0^*}_{\varepsilon}\underbrace{(\boxed{1}\,0^*\,\boxed{1}\,0^*)^{\boxed{*}}}_{11011}$$

AND to NAND

- AND:
  - $Q = \{start, No0s, Some0s\}$
  - $q_0 = start$
  - $F = \{start, No0s\}$
  - δ defined as the arrows
- NAND:
  - Q, q_0, δ don't change
  - $F = Q - F$
- In general, If we can compute a language L with a FSA, we can compute L^c as well



Logistics

- Homework 4 due Tonight and Friday @ 11:59
- You'll have an assignment due the Friday you return from the break (no early deadline)
- Quiz due Tuesday 1774

Last Time

- Regular Expressions
  - Equivalent to FSAs (but we haven't shown that yet)

1) closure
2) non-determinism
   - ↳ several things at once
   - ↳ modelling parallel computing
   - ↳ quantum

Regular Expressions

Name	Decision Problem	Function	Language
Regex	Does this string match this pattern?	$f(b) = \begin{cases} 0 & \text{the string matches} \\ 1 & \text{the string doesn't} \end{cases}$	$\{b \in \Sigma^* \mid b \text{ matches the pattern}\}$

- A way of describing a language
- Give a "pattern" of the strings, every string matching that pattern is in the language
- Examples:
  - $(a|b)c$ matches : ac and bc
  - $(a|b)^*c$ matches : $c, ac, bc, aac, abc, bac, bbc, \dots$

FSA = Regex

- Finite state Automata and Regular Expressions are equivalent models of computing
- Any language I can represent as a FSA I can also represent as a Regex (and vice versa)
- How would I show this?

Showing FSA $\leq$ Regex

- Show how to convert any FSA into a Regex for the same language
- We're going to skip this:
  - It's tedious, and people virtually never go this direction in practice, but you can do it (see textbook theorem 9.12)

Showing Regex ≤ FSA

- Show how to convert any regex into a FSA for the same language
- Idea: show how to build each “piece” of a regex using FSA

"Pieces" of a Regex

Structural Induction

- Empty String:
  - Matches just the string of length 0
  - Notation: ε or ""

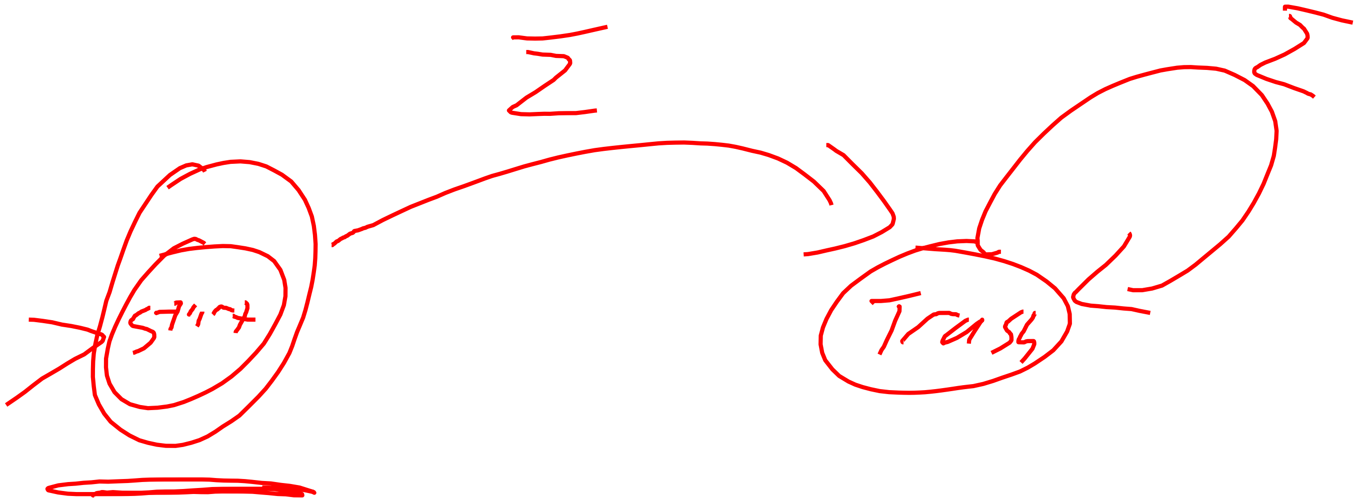
FSA

- Literal Character
  - Matches a specific string of length 1
  - Example: the regex a will match just the string a

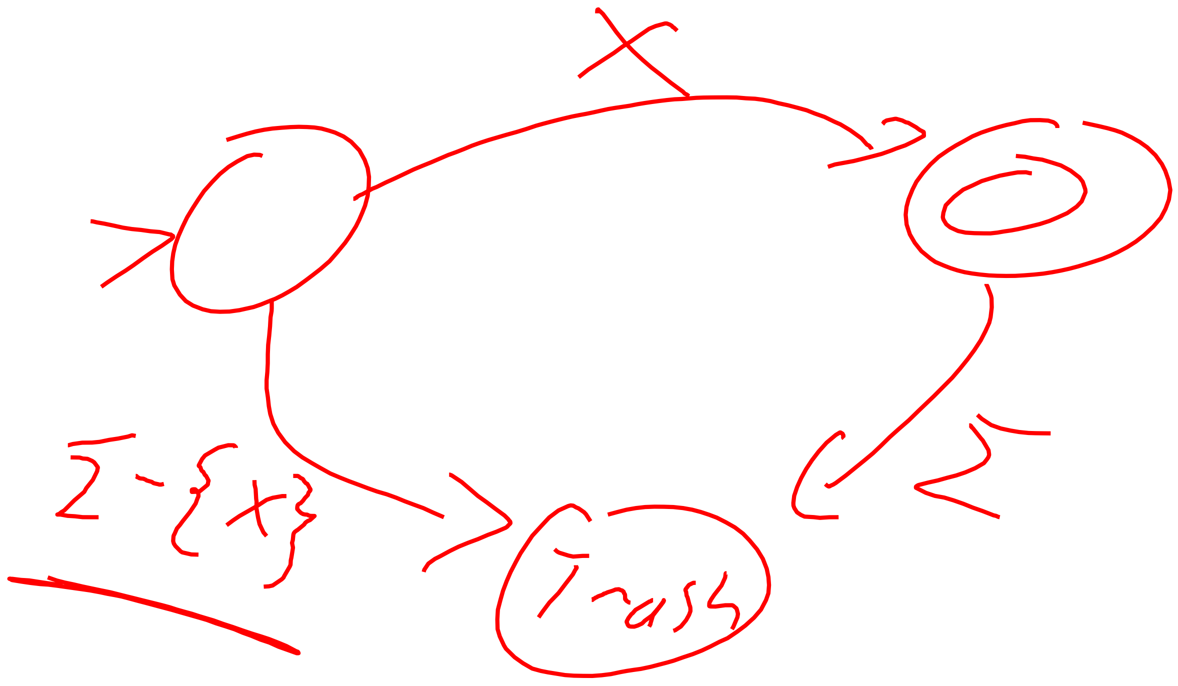
FSA

- Alternation/Union
  - Matches strings that match at least one of the two parts
  - Example: the regex $a|b$ will match a and b
- Concatenation
  - Matches strings that can be dividing into 2 parts to match the things concatenated
  - Example: the regex $(a|b)c$ will match the strings ac and bc
- Kleene Star
  - Matches strings that are 0 or more copies of the thing starred
  - Example: $(a|b)c^*$ will match a, b, or either followed by any number of c's

FSA for the empty string



FSA for a literal character = x



FSA for Alternation/Union

- Tricky…
- What does it need to do?

is computability with FSA closed under union?

$L_1 \cup L_2$

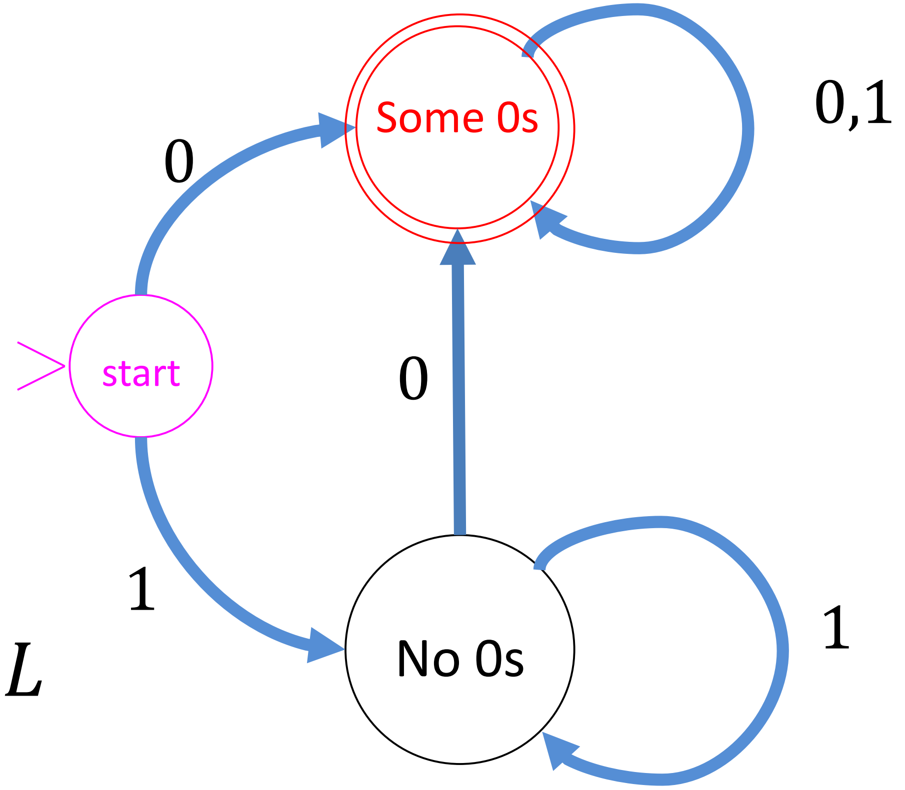
M_1 M_2

$M_\cup$ must return 1 for any string that M_1 or M_2 returns 1 on

we can read the input once

Recall: AND to NAND

- AND:
  - $Q = \{start, No0s, Some0s\}$
  - $q_0 = start$
  - $F = \{start, No0s\}$
  - δ defined as the arrows
- NAND:
  - Q, q_0, δ don't change
  - $F = Q - F$
- In general, If we can compute a language L with a FSA, we can compute L^c as well



Computing Complement

- If FSA $M = (Q, \Sigma, \delta, q_0, F)$ computes $\boxed{L}$
- Then FSA $M' = (Q, \Sigma, \delta, q_0, Q - F)$ computes $\boxed{\overline{L}}$

opposite

- Why?
  - Consider string $w \in \Sigma^*$
  - $w \in L$ means it ends at some state $f \in F$, which will be non-final in M' and therefore it will return False
  - $w \notin L$ means it ends at some state $q \notin F$, which will be final in M' and therefore it will return True

Computing Union

- Let FSA $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ compute L_1
- Let $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ compute $L_{\cancel{1}2}$
- Will there always be some automaton $M_\cup$ to compute $L_1 \cup L_2$
- What must $M_\cup$ do?
  - Somehow end up in a final state if either M_1 or M_2 did
  - Idea: build $M_\cup$ to "simulate" both M_1 and M_2

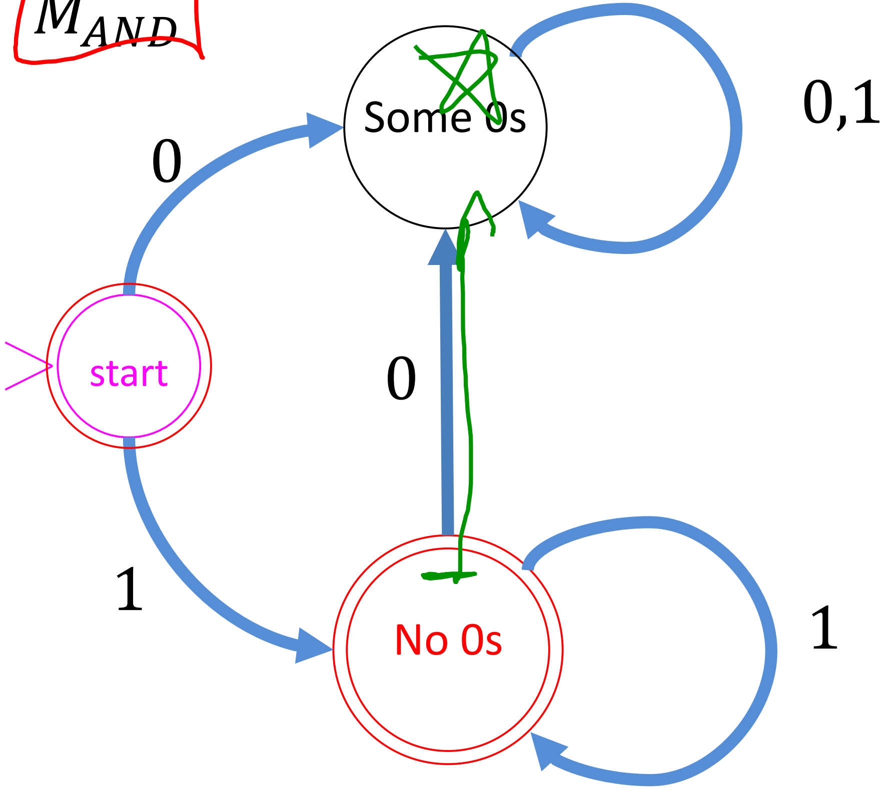
Example

- $AND \cup XOR$
  - What is the resulting language?

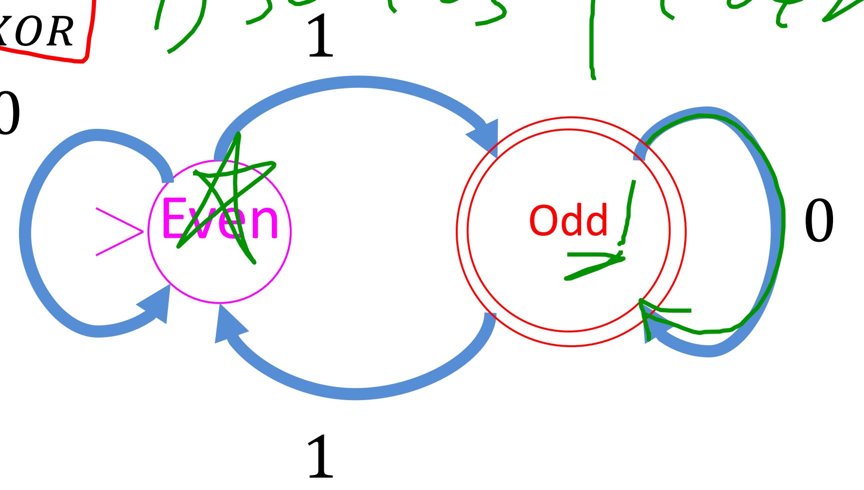
~~ε~~ ~~0~~ ~~0~~ 1 ✓

M_{AND}	M_{XOR}
start	even
1) No 0s	odd
0) Some 0s	odd
0) Some 0s	odd
1) Some 0s	even

M_{AND}



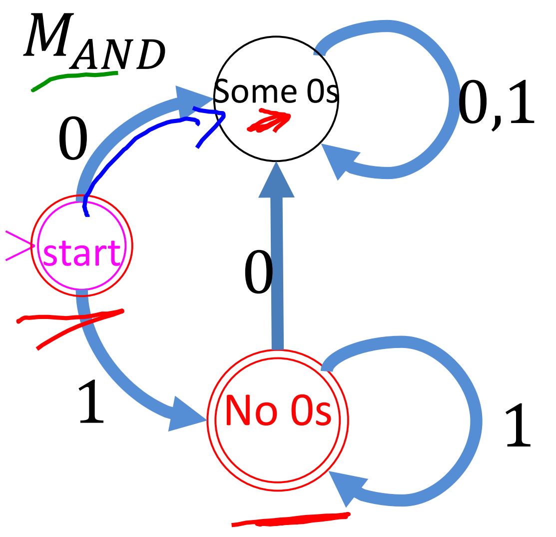
M_{XOR}

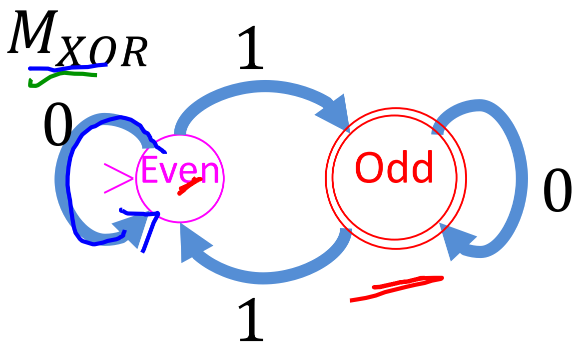


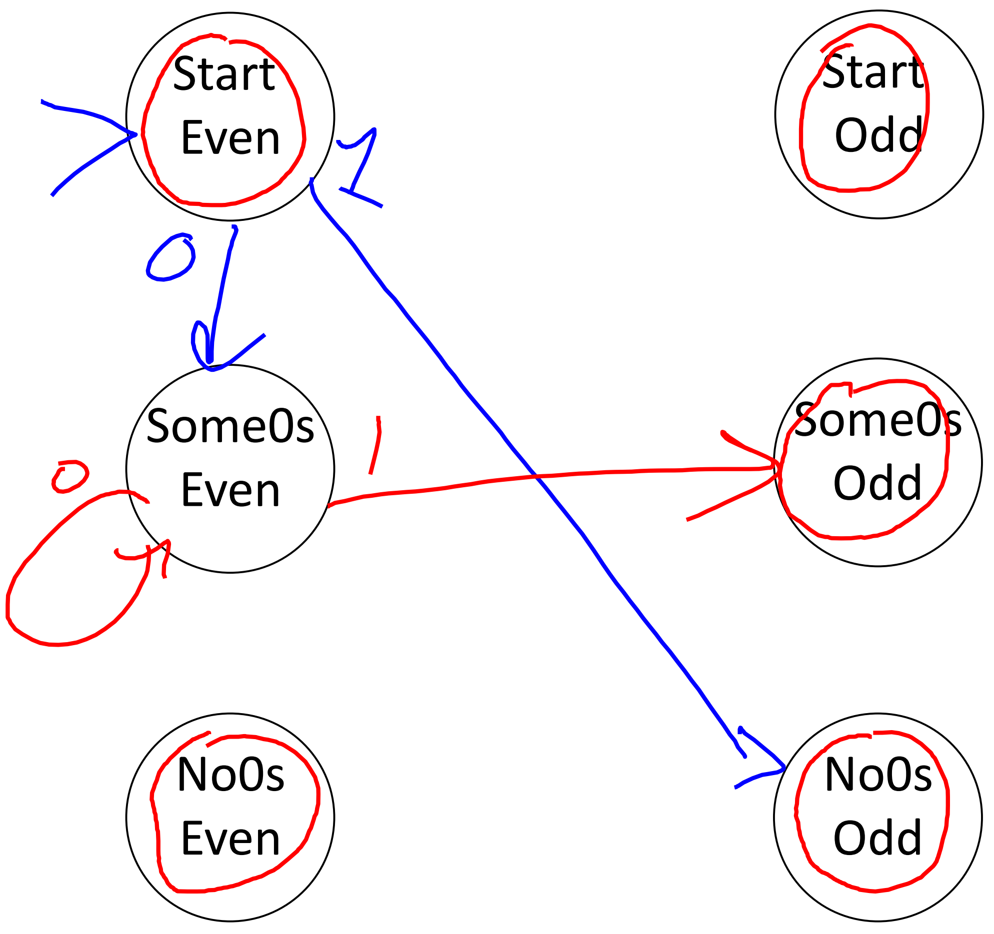
Cross-Product Construction

M_u

- 2 machines at once!

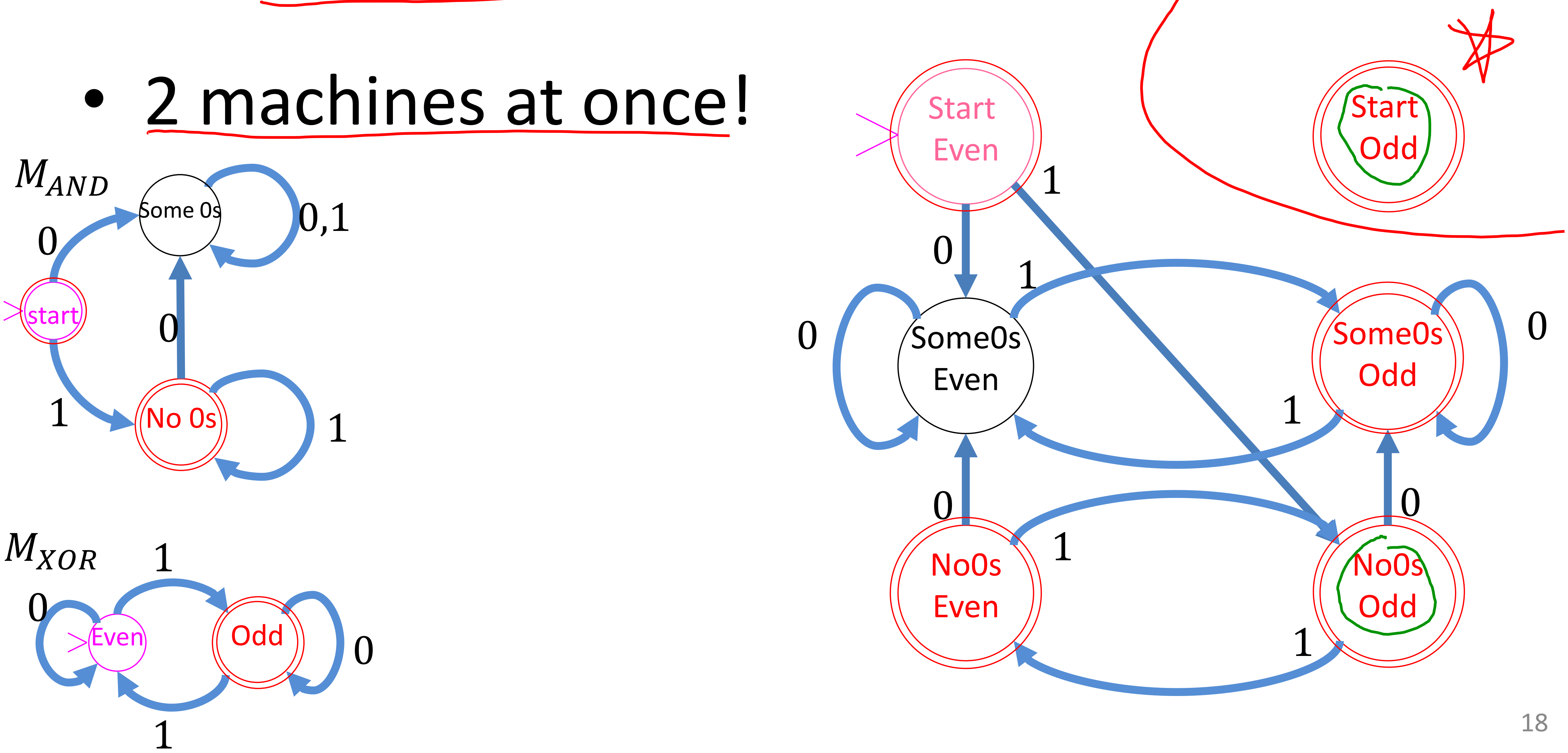






Cross-Product Construction

- 2 machines at once!



Cross Product Construction

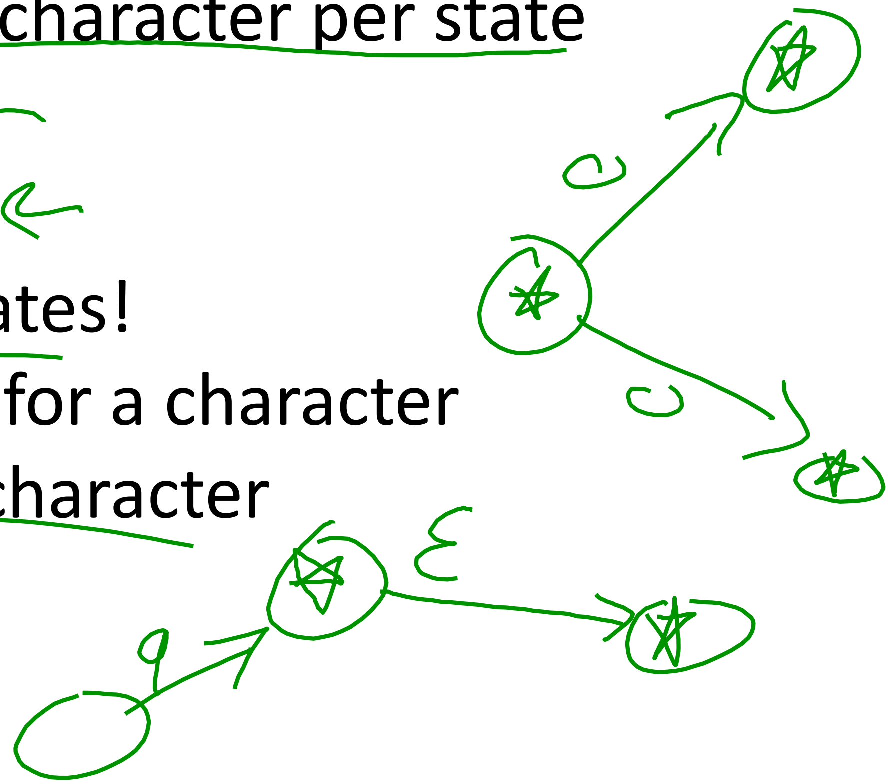
state alph tr: start finals

- Let FSA $M_1 = \boxed{(Q_1, \Sigma, \delta_1, q_{01}, F_1)}$ compute L_1
- Let $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ compute L_1
- $\boxed{M_\cup} = (Q_1 \times Q_2, \Sigma, \delta_\cup, (q_{01}, q_{02}), F_\cup)$ computes $L_1 \cup L_2$
  - $\delta_\cup((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$
  - $F_\cup = \{(q_1, q_2) \in Q_1 \times Q_2 | q_1 \in F_1 \text{ or } q_2 \in F_2\}$
- How could we do intersection?

$L_1 - L_2$

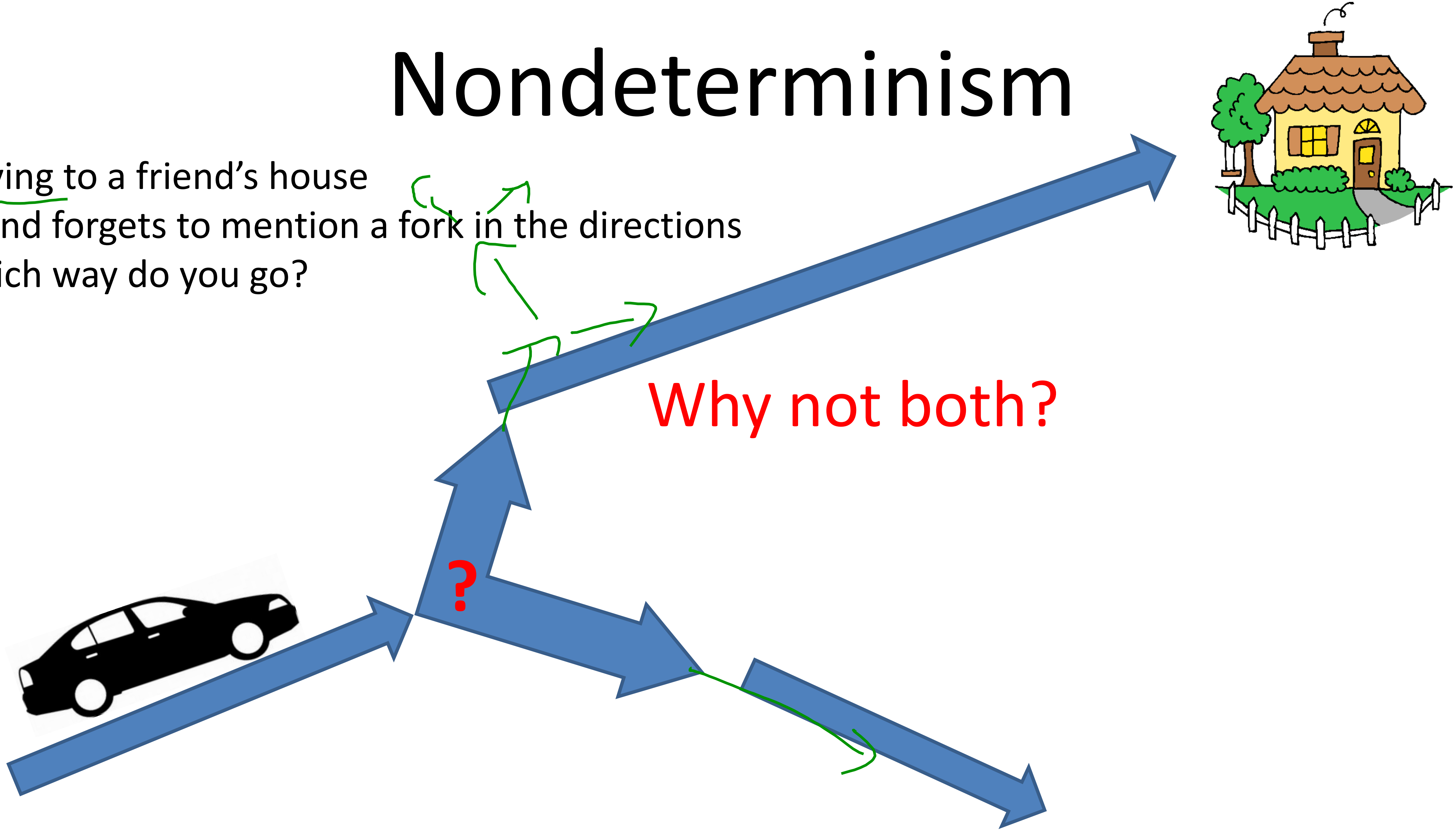
Non-determinism

- Things could get easier if we "relax" our automata
- So far:
  - Must have exactly one transition per character per state
  - Can only be in one state at a time
- Non-deterministic Finite Automata:
  - Allowed to be in multiple (or zero) states!
  - Can have multiple or zero transitions for a character
  - Can take transitions without using a character
  - Models parallel computing



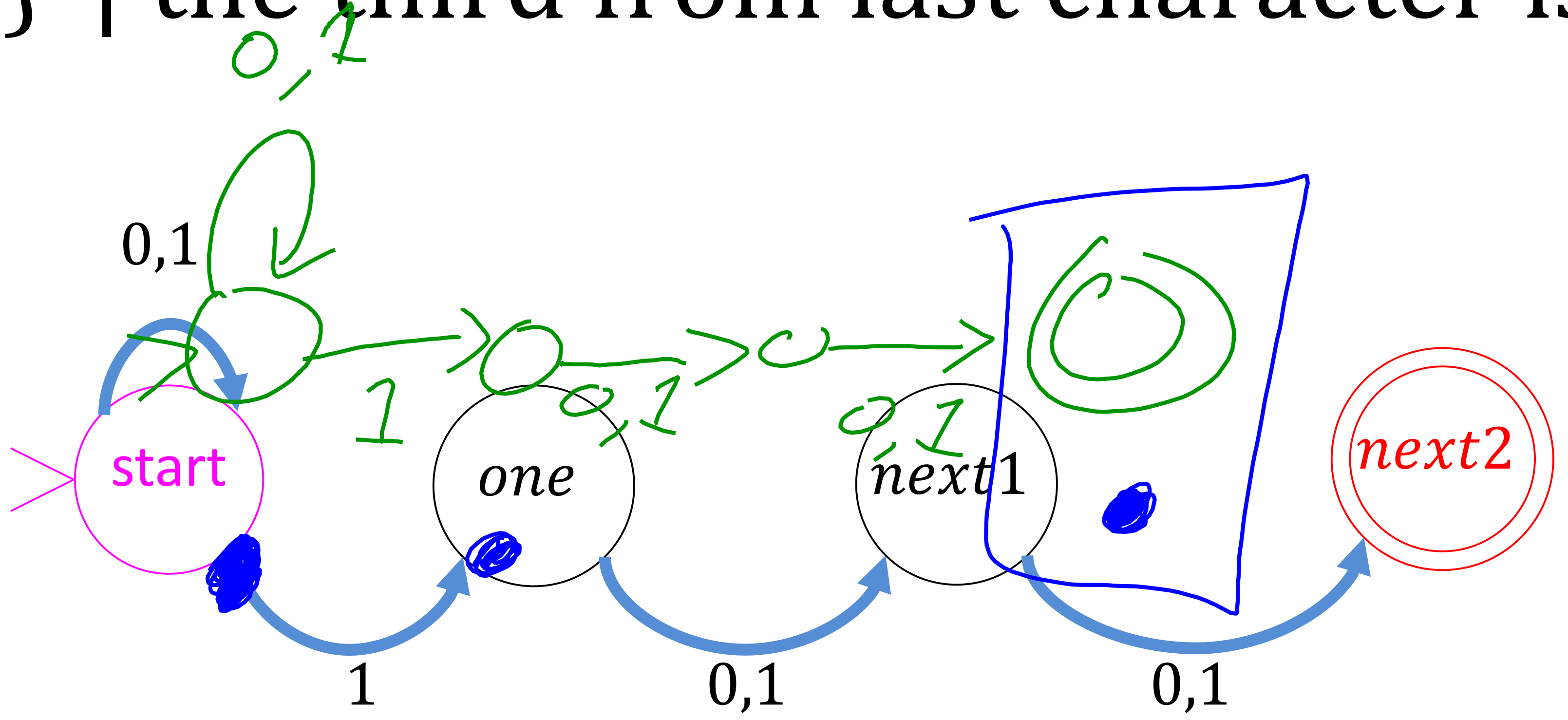
Nondeterminism

Driving to a friend's house
Friend forgets to mention a fork in the directions
Which way do you go?

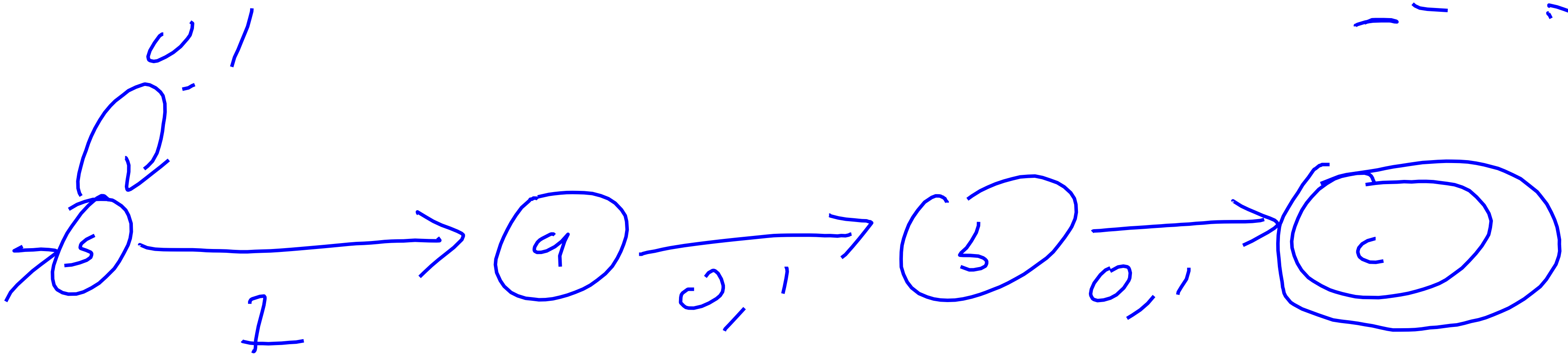


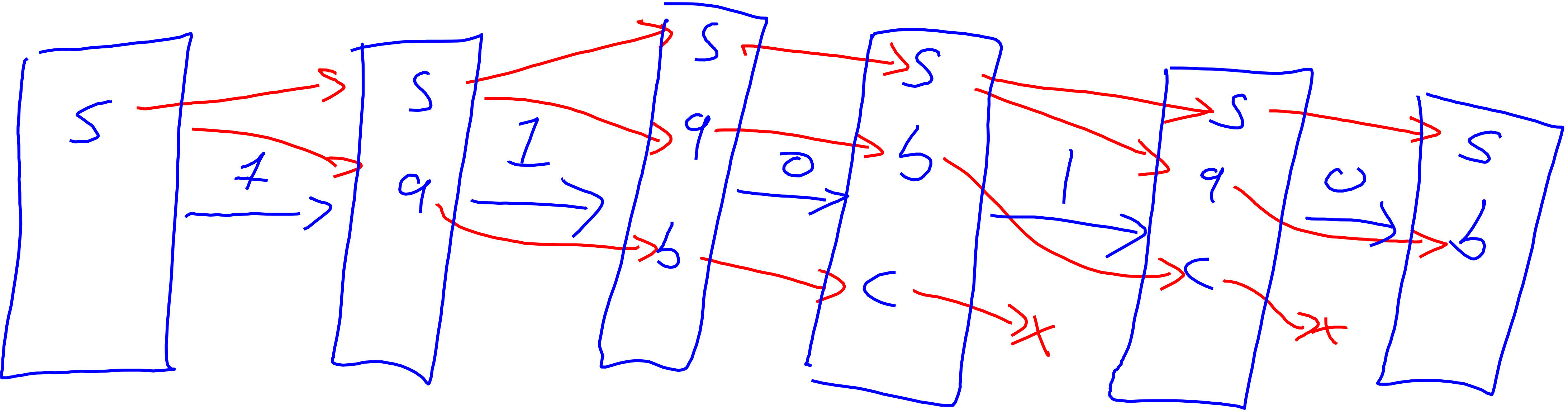
Example Non-deterministic Finite Automaton

- $ThirdLast1 = \{w \in \{0,1\}^* \mid \text{the third from last character is a 1}\}$



11010

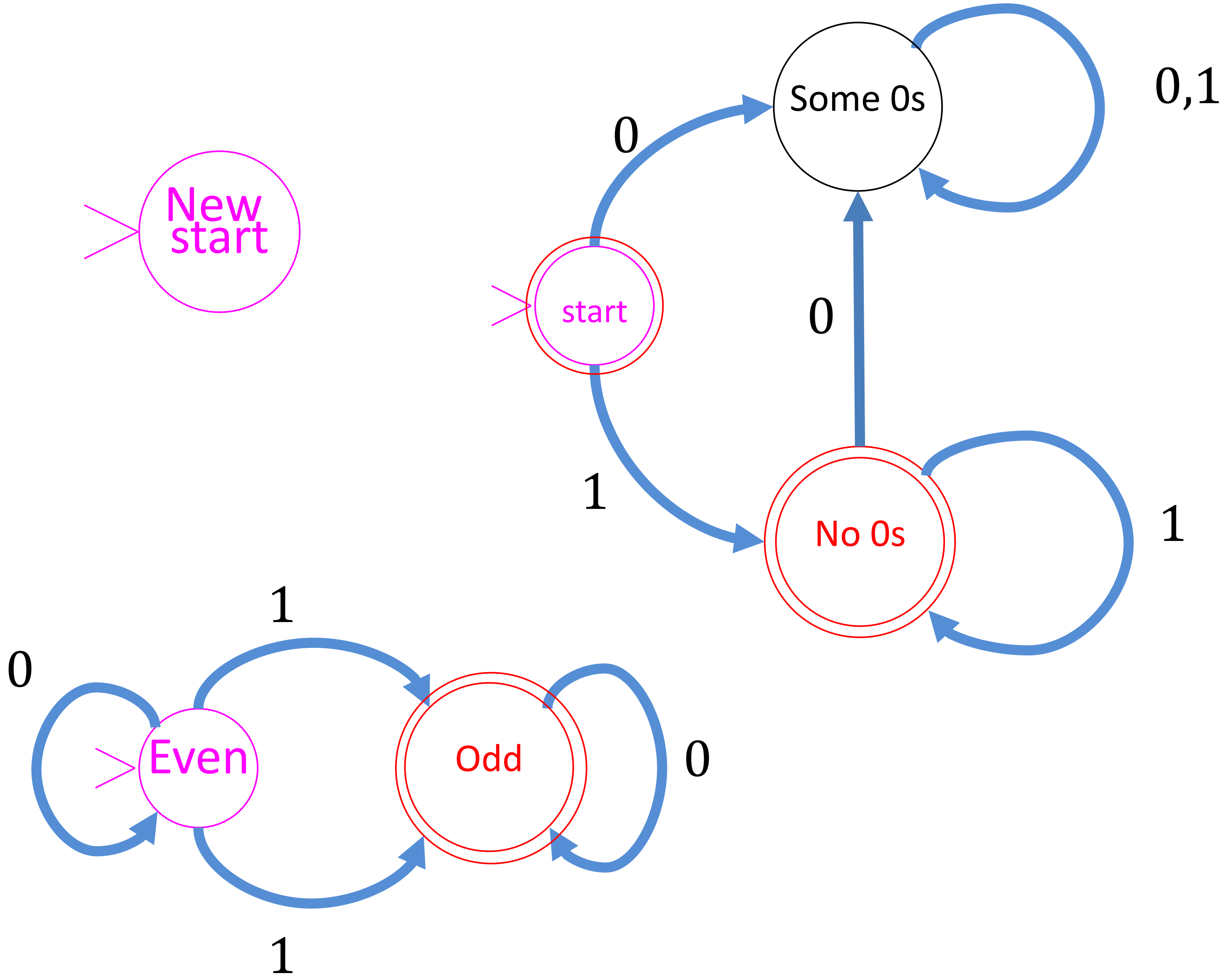




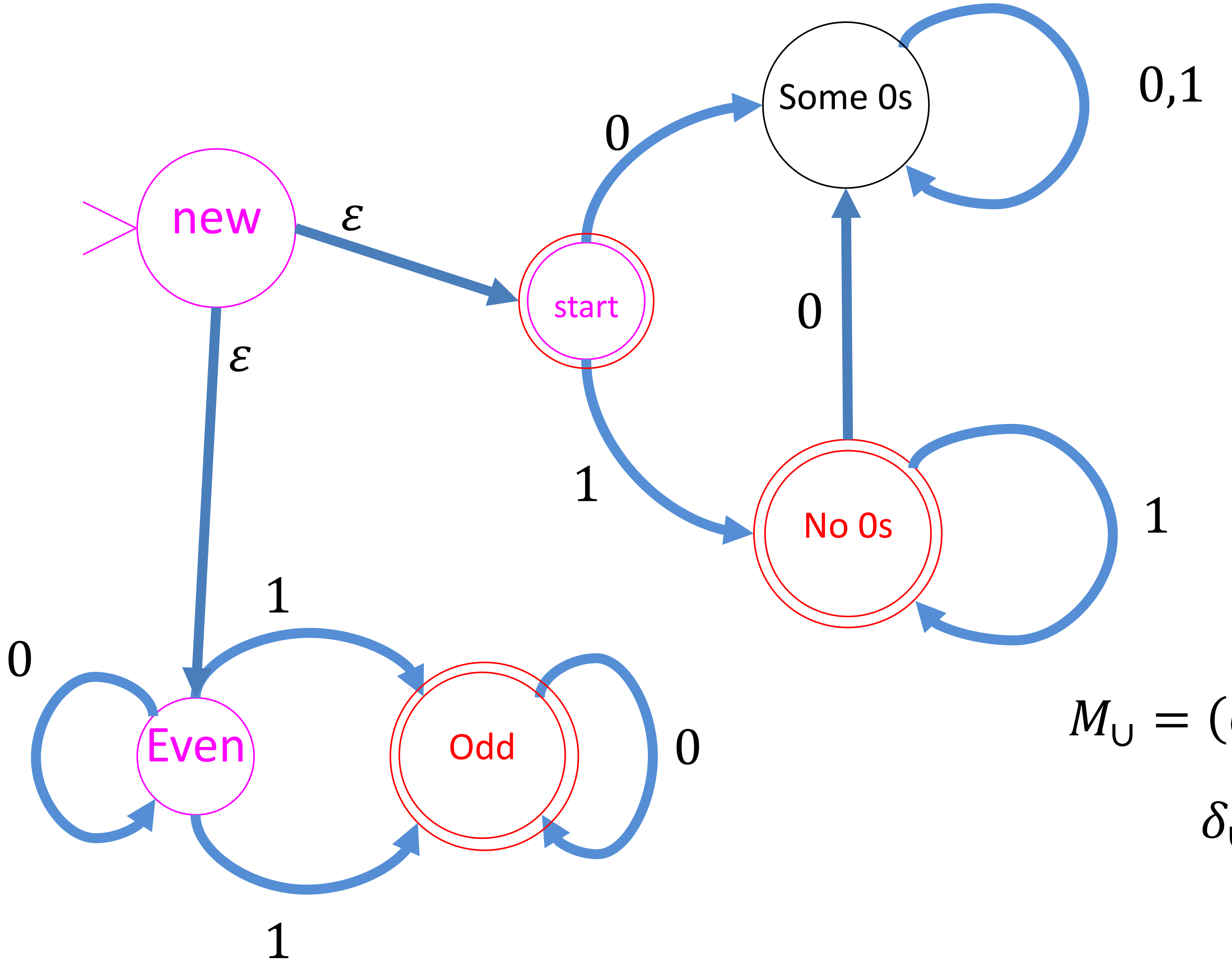
Non-Deterministic Finite State Automaton

- Implementation:
  - Finite number of states
  - One start state
  - “Final” states
  - Transitions: (partial) function mapping **state-character (or epsilon) pairs to sets of states**
- Execution:
  - Start in the initial “state”
  - **Enter every state reachable without consuming input (ε-transitions)**
  - Read each character once, in order (no looking back)
  - Transition to new **states** once per character (based on current states and character)
  - **Enter every state reachable without consuming input (ε-transitions)**
  - Return True if **any** state you end is final
    - Return False if **every** state you end in is non-final

Union Using Non-Determinism



Union Using Non-Determinism

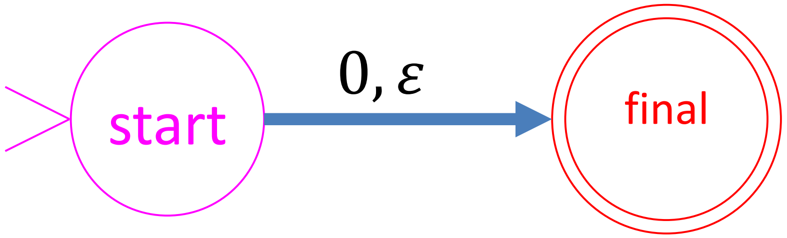


$$M_\cup = (Q_1 \cup Q_2 \cup \{new\}, \Sigma, \delta_\cup, new, F_1 \cup F_2)$$

$$\delta_\cup(q,\sigma) = \begin{cases} \{\delta_1(q,\sigma)\} \text{ if } q \in Q_1 \\ \{\delta_2(q,\sigma)\} \text{ if } q \in Q_2 \end{cases}$$

$$\delta_\cup(new,\varepsilon) = \{start, even\}$$

What’s the language?



NFA Example

$$\{w \in \{0,1\}^* | w \text{ contains } 0101\}$$