Today: a very simple model of computation for deciding language membership (aka Y/N problem)
- given input, read it char by char
- keep track of a finite amount of memory
  \[ \text{does not depend on input} \]
- when no more input, outputs Y/N (accept/reject)

Variants used in: simple robots, network controllers, language processing

Pieces:
- each possible configuration of memory is called a state
  \[ Q = \text{set of possible states} \]
- start machine in some “starting” state \( s \in Q \)
- given the current state, the next input char tells us how to update mem, or, which state to transition to
  \[ \delta: Q \times \Sigma \rightarrow Q \]
- at the end, we are in an accepting state or rejecting state.
  \[ A = \text{set of accepting states} \] 
  \[ (Q \setminus A = \text{set of rejecting states}) \]

deterministic finite automaton (DFA)
\[ M = (Q, \Sigma, s, \delta, A) \]
  set of accepting states
For determining if, given a binary string $w$, if $\#(1, w)$ is odd.

Idea: count the # of 1s.

Problem: the amount of memory depends on the input!

Actually keep track of parity of this count.

Parity 1s:

- Parity $\leq 0$
- While input
  - Read character $a$
    - If $a = 1$:
      - Parity $\leq 1$ - Parity
      - Return (Parity = 1)

$Q = \{0, 1\}$
$\Sigma = \{0, 1\}$

$\delta(q, a) = \left\{ \begin{array}{ll}
q & \text{if } a = 0 \\
1 - q & \text{if } a = 1
\end{array} \right.$

$S = 0$ (in $Q$)
$A = \{1\}$

Graphical representation of DFAs:
- Vertices are states
- Edges represent transitions
- For each state, there is exactly one outgoing edge per $a \in \Sigma$
  (sometimes colored for visualization)
- Pointer to starting state
- pointer to starting state
- doubled circles for accepting

\[ \text{even} \quad \rightarrow 0 \quad \rightarrow 1 \quad \rightarrow \text{odd} \]

\[ \text{e.g. on input 01101} \]
\[ \text{state say:} \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow 1 \]
\[ 1 \text{ is accepting so return yes} \]
\[ \text{on input 1010?} \]
\[ \text{state req:} \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 0 \]
\[ \text{reject} \]

Program for determining if binary string contains 00.
- keep track of if we've seen 00
- keep track of prev char.
  - if prev = 0 and curr = 0, we are looking at 00.

keep track of \((\text{found}, \text{prev}) \in \{Y,N\} \times \{0, \text{not0}\}\)

```
\text{have we seen 00?}
\text{combine into Y?}
```
Given a DFA $M$, the language accepted by $M$ is $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

What does it mean for $M$ to accept $w$? After reading $w$, we ended on an accepting state.

Q: What is the state we end on?

We'd like to give it a label like $\delta^*(w) \in Q$.

Define extended transition function $\delta^*$

$\delta^* : Q \times \Sigma^* \rightarrow Q$

$\delta^*(q, \omega) = \text{state we end on if, starting from } q$, we read $\omega$.

$\delta^*(q, \omega) = \begin{cases} q & \text{if } \omega = \varepsilon \\ \delta^*(\delta(q, a), x) & \text{if } \omega = ax \\ \delta^*(\delta(q, a), x) & \text{if } \omega = \varepsilon \\ \delta^*(\delta(q, a), x) & \text{if } \omega = ax \\ \end{cases} \\
\delta^*(q, \omega) = \begin{cases} q & \text{if } \omega = \varepsilon \\ \delta^*(\delta(q, a), x) & \text{if } \omega = ax \\ \end{cases}$

$L(M) = \{ \omega \in \Sigma^* \mid \delta^*(s, \omega) \in A \}$

Def. A language is **automatic** if there is a DFA.
Def. A language is **automaton** if there is a DFA that accepts it.

Recall regular languages are closed under:
- (finite) union
- (finite) concatenation
- Kleene star

What can we do to automaton languages to get an automaton?

→ gives a way of building complex DFAs for harder languages from simpler DFAs for easier languages.

- Complement

\[
\overline{L(M)} = \Sigma^* \setminus L(M)
= \Sigma^* \setminus \left\{ w \in \Sigma^* \mid \delta^*(s, w) \in A \right\}
= \left\{ w \in \Sigma^* \mid \delta^*(s, w) \notin A \right\}
= \left\{ w \in \Sigma^* \mid \delta^*(s, w) \in Q \setminus A \right\}
\]

Given \( M = (Q, \Sigma, \delta, s, A) \),
define \( \overline{M} = (Q, \Sigma, \delta, s, Q \setminus A) \).

The preceding derivation proves that \( L(\overline{M}) = \overline{L(M)} \).

E.g.

\[
\begin{align*}
\text{specific to 3 strings} & \quad \text{spoons: automatic if regular} \\
\text{Closure properties} & \\
\text{recall regular languages are closed under:} & \\
- \text{(finite) union} & \\
- \text{(finite) concatenation} & \\
- \text{Kleene star} & \\
\text{What can we do to automaton languages to get an automaton?} & \\
\rightarrow \text{gives a way of building complex DFAs for harder languages from simpler DFAs for easier languages.} & \\
\text{- Complement} & \\
\overline{L(M)} & = \Sigma^* \setminus L(M) \\
& = \Sigma^* \setminus \left\{ w \in \Sigma^* \mid \delta^*(s, w) \in A \right\} \\
& = \left\{ w \in \Sigma^* \mid \delta^*(s, w) \notin A \right\} \\
& = \left\{ w \in \Sigma^* \mid \delta^*(s, w) \in Q \setminus A \right\} \\
\text{Given } M = (Q, \Sigma, \delta, s, A) & \\
\text{define } \overline{M} = (Q, \Sigma, \delta, s, Q \setminus A) & \\
\text{the preceding derivation proves that } L(\overline{M}) = \overline{L(M)} & \\
e.g. & \\
\begin{align*}
\text{specific to 3 strings} & \quad \text{spoons: automatic if regular} \\
\text{Closure properties} & \\
\text{recall regular languages are closed under:} & \\
- \text{(finite) union} & \\
- \text{(finite) concatenation} & \\
- \text{Kleene star} & \\
\text{What can we do to automaton languages to get an automaton?} & \\
\rightarrow \text{gives a way of building complex DFAs for harder languages from simpler DFAs for easier languages.} & \\
\text{- Complement} & \\
\overline{L(M)} & = \Sigma^* \setminus L(M) \\
& = \Sigma^* \setminus \left\{ w \in \Sigma^* \mid \delta^*(s, w) \in A \right\} \\
& = \left\{ w \in \Sigma^* \mid \delta^*(s, w) \notin A \right\} \\
& = \left\{ w \in \Sigma^* \mid \delta^*(s, w) \in Q \setminus A \right\} \\
\text{Given } M = (Q, \Sigma, \delta, s, A) & \\
\text{define } \overline{M} = (Q, \Sigma, \delta, s, Q \setminus A) & \\
\text{the preceding derivation proves that } L(\overline{M}) = \overline{L(M)} & \\
e.g. &
Intersection

Ex. \#(1,w) is odd AND contains 00

Idea: simulate keeping track of both machines, in particular, remember the state in each machine.

10011 → 0 → 1 → 1 → 1 → 0 → 1
state seq's: → a → a → b → c → c → c

Procedure: Product Construction

\[ \text{Given} \quad M_1 = (Q_1, \Sigma, \delta_1, Q_1, A_1) \]
\[ M_2 = (Q_2, \Sigma, \delta_2, Q_2, A_2) \]
\[ \text{Same!} \]

\[ M = (Q, \Sigma, \delta, Q, A) \]
\[ Q = Q_1 \times Q_2 \quad \text{states are tuples of states} \]
\[ Q = Q_1 \times Q_2, \] states are tuples of states
\[ \delta \left( (q_1, q_2), a \right) = (\delta_1(q_1, a), \delta_2(q_2, a)) \]

Thm: \( L(M) = L(M_1) \cap L(M_2) \)

Pf: (see textbook)

The DFA given by product construction might not be the smallest DFA for \( L(M_1) \cap L(M_2) \).

We don't care (for now)

- Union \( L(M_1) \cup L(M_2) \)

also by product construction except
\[ A = \{ (q_1, q_2) | q_1 \in A_1 \land q_2 \in A_2 \} \]

everything else is the same.

- Set Difference \( L(M_2) \setminus L(M_2) \)

for sets \( S, T \), \( S \setminus T = S \cap \overline{T} \).

So, given \( M_1, M_2 \),
\[ = \{ x | x \in S \text{ and } x \notin T \} \]
do product construction on $M_1, M_2$

ex. #$(1,0)$ is odd, except the set containing 00