# Finite Automata
*continued*

CMPU 240 • Language Theory and Computation • Spring 2019

## Last class:

- Introduced finite automata
- Defined deterministic finite automata and their operation

## Today:

- Introduce regular languages
- Introduce nondeterministic finite automata

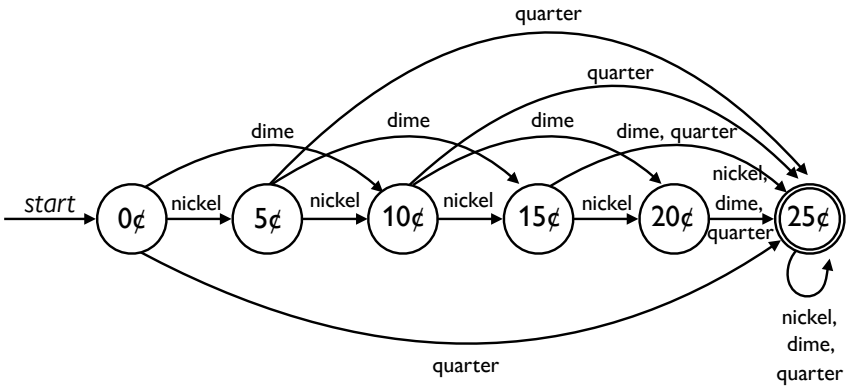
# Deterministic finite automata, *continued*

## Finite automata in action

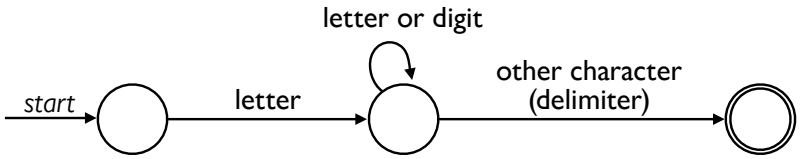
- Used in
  - Text editors and search engines for pattern matching
  - Compilers for lexical analysis
  - Web browsers for HTML parsing
- Serve as the control unit in many physical systems, including
  - Elevators, traffic signals, vending machines
  - Computer microprocessors
  - Network protocol stacks and old VCR clocks
- Play a key role in natural language processing and machine learning
  - *Markov chains* are probabilistic FAs used in part of speech tagging, speech processing, and optical character recognition


*Evan Grothjan for The New York Times*

# Finite automaton for a newspaper vending machine




# Sketch of a finite automaton that recognizes simple identifiers




When we want to design an automaton for a certain task, think as follows:

The states of the automaton represent its memory.
Use different states for different possibilities.

For example:

An automaton that accepts iff the string has an even number of 1s will have to count number of 1s mod 2

You want to have *one state for each possibility*.

An automaton that accepts iff the first symbol equals the last symbol will have to keep track of what the first symbol is

It should have *different states for different possibilities of the first symbol*.

## Conventions

It helps if we can avoid mentioning the type of every name by following some rules:

Input *symbols* are *a*, *b*, etc., or digits

*Strings* of input symbols are *u*, *v*, …, *z*

*States* are *q*, *p*, etc.

## Problems

Build an automaton to recognize the set of strings that end with "ing".

Build an automaton to recognize the set of strings with an even number of 1s.

Build an automaton to recognize the set of strings that start and end with the same symbol.

## Formal definition of a deterministic finite automaton (DFA)

A DFA is represented as a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where

$Q$: Finite set of *states*.

$\Sigma$: Finite set of *input symbols* (the *alphabet*).

$\delta$: $Q \times \Sigma \rightarrow Q$: A *transition function*.

$q_0 \in Q$: One state is the *start state* (or *initial state*).

$F \subseteq Q$: Set of zero or more *final states* (or *accepting states*).

## Transition function $\delta$

Takes two arguments: a state and an input symbol.

$\delta(q, a)$ = the state the DFA goes to when it is in state $q$ and reads input symbol $a$.

**Note**: $\delta$ is a total function; there is always a next state. You must add a "dead state" if there's no transition you want.

# Formal definition of how finite automata compute

Let $M = (Q, \Sigma, \delta, q_o, F)$ be a finite automaton.

Let $w_1w_2...w_n$ be a string where each $w_i \in \Sigma$.

$M$ accepts $w$ if a sequence of states $r_0, r_1, ..., r_n$ exists where each $r_i \in Q$ and:

**1.** $r_0 = q_0$ (it begins at the start state)

**2.** $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, ..., n-1$ (each transition in the sequence is allowed by the transition function for the corresponding input symbol.)

**3.** $r_n \in F$ (it ends in an accept state)

# Creating a finite automaton

If you think of the states as "memory", then the memory of a finite automaton is limited to the number of states.

The consequence of a finite number of states is, if $|w|$ > number of states, some state must be repeated in the execution of the FA over $w$.

If the machine can't remember all the symbols it has seen so far in an input string, it has to change state based on other information, e.g.,

$L_1$ = the set of all strings with an odd number of 1s over the alphabet {0, 1}

# Creating a finite automaton

Start by putting yourself in the place of the FA that has to make every transition choice based on a single character because it can't look ahead or rewind.

1. Define the meaning of the states:

   For $L_1$, we can designate a two-state automaton; one state applies to the situation in which the number of ones seen is even (so far), and the other to the situation where the number of ones seen is odd (so far).

2. Determine the transition function:

   In either state, if you read a 0, stay put. Whenever you read a 1, follow a link to the other node.

3. Label the start and final states

4. Test your FA.

   You can trace sample inputs or you can enter your FA in a tool like JFLAP to run examples.

# Example: Clamping logic

We can think of an accepting state as representing a "1" output and non-accepting states as representing "0" output.

A "clamping" circuit waits for a 1 input, and forever after makes a 1 output. To avoid clamping on spurious noise, we'll design an FA that waits for two 1s in a row, and only clamps then.

Shows how a state can represent the history of what has been seen on the input so far.

The states we need are:

State $q_0$, the start state, says that the most recent input (if there was one) was not a 1, and we have never seen two 1s in a row.

State $q_1$ says we have never seen 11, but the previous input was 1.

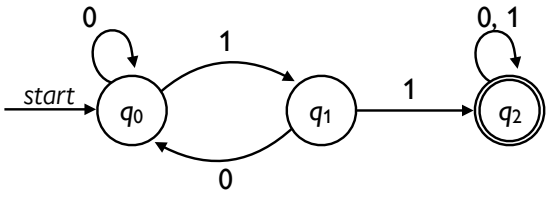
State $q_2$ is the only accepting state. It says that we have at some time seen 11.

Thus, $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$, where $\delta$ is given by:

| | 0 | 1 |
|---|---|---|
| $\rightarrow q_0$ | $q_0$ | $q_1$ |
| $q_1$ | $q_0$ | $q_2$ |
| $*q_2$ | $q_2$ | $q_2$ |

*By marking the start state with → and accepting states with *, the transition table that defines δ also specifies the entire FA.*

# Clamping logic: transition graph




# Extension of $\delta$ to paths

Intuitively, a finite automaton *accepts* a string $w = a_1a_2\ldots a_n$ if there is a path in the transition diagram that:

1. Begins at the start state
2. Ends at an accepting state
3. Has sequence of labels $a_1, a_2, \ldots, a_n$.

Formally, we extend transition function $\delta$ to $\hat{\delta}(q, w)$ where $w$ can be any string of input symbols:

*Basis*: $\hat{\delta}(q, \varepsilon) = q$

I.e., on no input, the FA doesn't go anywhere

*Induction*: $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$, where $w$ is a string and $a$ is a single symbol.

I.e., see where the FA goes on $w$, then look for the transition on the last symbol from that state.

$\hat{\delta}$ really represents paths.

That is, if $w = a_1a_2\ldots a_n$ and $\delta(p_i, a_i) = p_{i+1}$ for all $i = 0, 1, \ldots, n-1$, then $\hat{\delta}(p_0, w) = p_n$.

### *Acceptance of strings*

A finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ accepts string $w$ if

$\hat{\delta}(q_0, w) \in F$

### *Language of a finite automaton*

A finite automaton $A$ recognizes the language

$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$

## Aside: type errors

A major source of confusion when dealing with automata – or mathematics in general – is making "type errors".

**Example**: Don't confuse $A$, a FA, i.e., a program, with $L(A)$, which is of type "set of strings".

**Example**: The start state $q_0$ is of type "state", but the accepting states $F$ is of type "set of states".

## Is *a* a symbol or a string of length 1?

**Answer**: It depends on the context.

If it's being used in $\delta(q, a)$, it's a symbol.

If it's being used in $\hat{\delta}(q, a)$, it's a string.


## Acknowledgments

This lecture incorporates material from:

Nancy Ide

Michael Sipser

Jeffrey Ullman