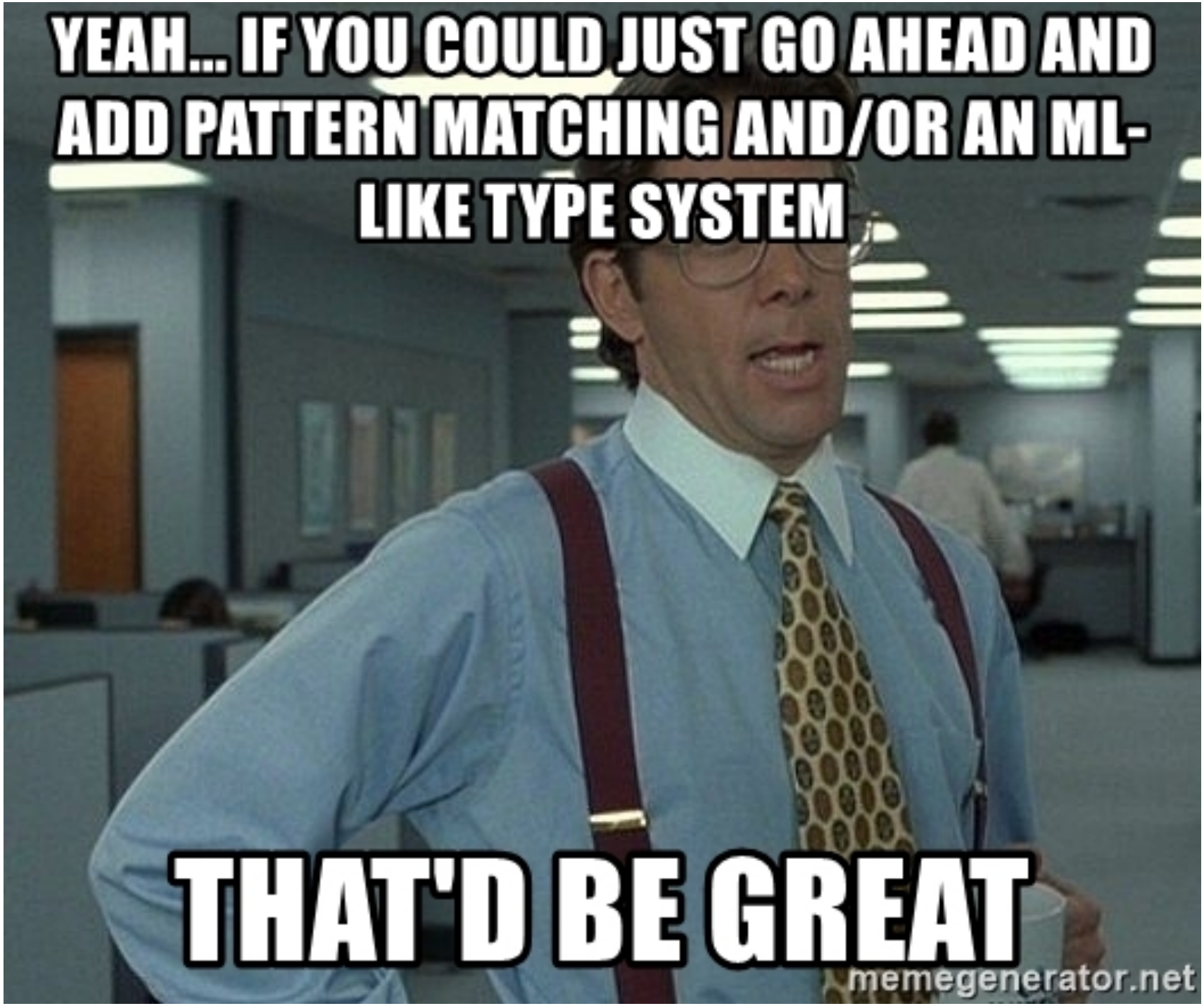
CS 10:
Problem solving via Object Oriented Programming

Pattern Matching


YEAH... IF YOU COULD JUST GO AHEAD AND ADD PATTERN MATCHING AND/OR AN ML-LIKE TYPE SYSTEM
THAT'D BE GREAT
memegenerator.net

Agenda

1. Pattern matching to validate input
   - Regular expressions
   - Deterministic/Non-Deterministic Finite Automata (DFA/NFA)
2. Finite State Machines (FSM) to model complex systems

Pattern matching goal: ensure input passes a validation check

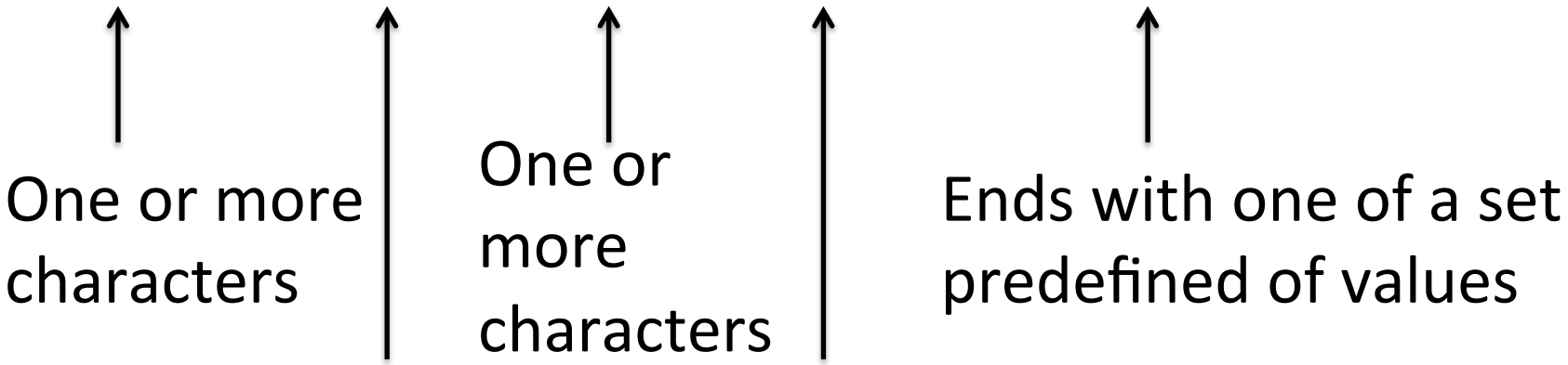
Pattern matching process:

- Given some input (e.g., a series of characters)
- Also given a pattern that describes what constitutes valid input
- Then check to see if a particular input "passes" validation check (or in other words, input matches the pattern)

Sometimes it is useful to be able to detect or require patterns

Email addresses follow a pattern:
mailbox@domain.TLD
example: tjp@cs.dartmouth.edu

We can specify a pattern or rules for email addresses:
<characters> @ <characters>.<com | edu | org | …>



Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”

Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”

Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel

Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches “cat” or “cot”

Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches “cat” or “cot”
Character classes [c_1-c_2] and [^c_1-c_2]	Alternative characters and excluded characters	[a-c] matches “a” or “b” or “c”, while [^a-c] matches any but abc

Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches “cat” or “cot”
Character classes [c_1-c_2] and [^c_1-c_2]	Alternative characters and excluded characters	[a-c] matches “a” or “b” or “c”, while [^a-c] matches any but abc
Repetition: R*	Matches 0 or more times	“ca*t” matches “ct”, “cat”, “caat”

Regular expressions (regex) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches “cat” or “cot”
Character classes [c_1-c_2] and [^c_1-c_2]	Alternative characters and excluded characters	[a-c] matches “a” or “b” or “c”, while [^a-c] matches any but abc
Repetition: R*	Matches 0 or more times	“ca*t” matches “ct”, “cat”, “caat”
Non-zero repetition: R+	Matches 1 or more times	“ca+t” matches “cat” or “caat” or “caaat”, but not “ct”

We can use regex to see if an email address is valid

Email addresses follow a pattern:
mailbox@domain.TLD
example: tjp@cs.dartmouth.edu

We can specify a pattern or rules for email addresses:
<characters> @ <characters>.<com | edu | org | …>

As a simple RegEx: [a-z]+@[a-z.]* [a-z]+. (com | edu | org …)

Check:
tjp@cs.dartmouth.edu -- valid
Blob.x -- invalid

This simple regex has some issues dealing with real email addresses

Turns out a robust email address validator is quite complicated

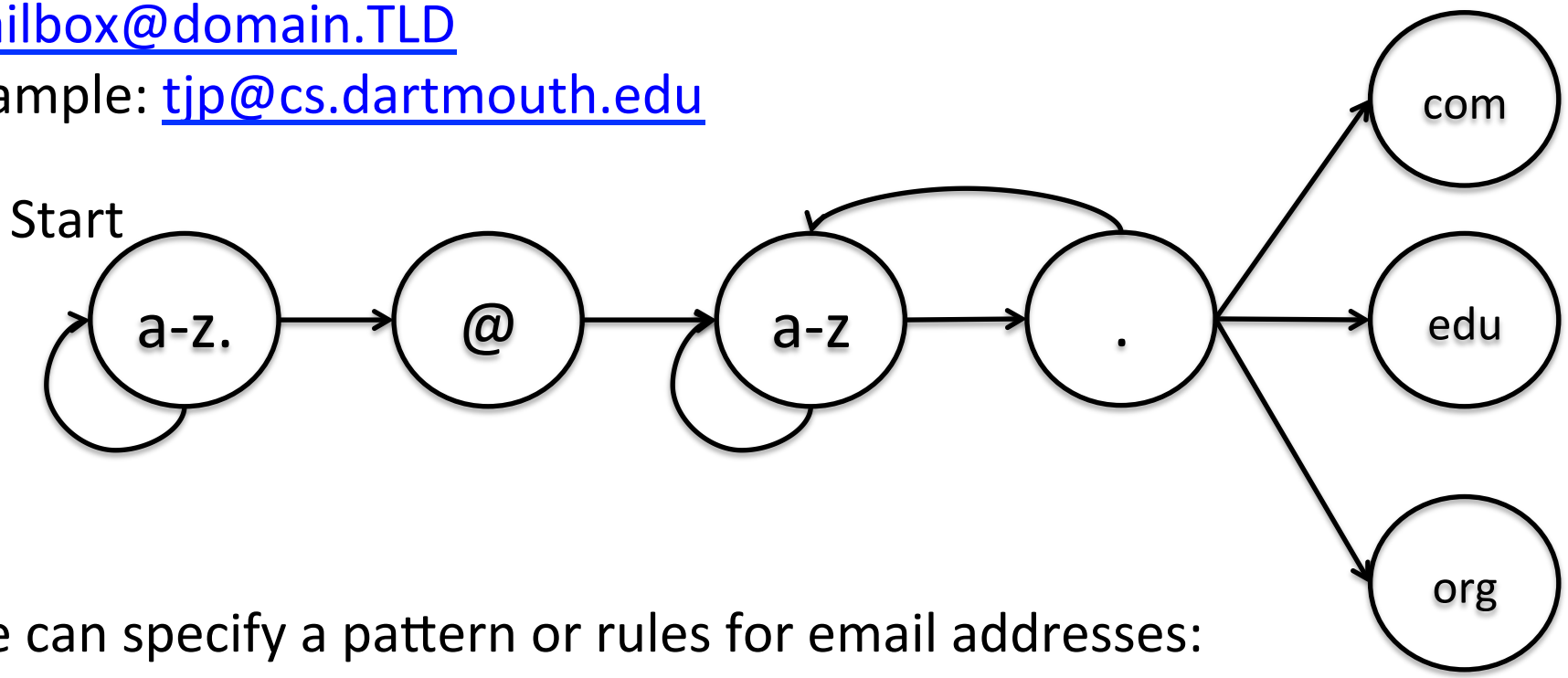
```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}
~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-
z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:
25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c
\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
\])
```

- **Hard to understand what this does**
- **We can use a graph to make things easier to understand**

Source: IETF RFC2822

A Graph can implement a regex

Email addresses follow a pattern:
mailbox@domain.TLD
example: tjp@cs.dartmouth.edu



We can specify a pattern or rules for email addresses:
<characters> @ <characters>.<com | edu | org | …>

A Graph can represent the pattern for email addresses
Sample addresses can be easily verified if in correct form

Key points

1. We can define a set of rules that must be followed

2. We may be able to represent those rules with a Graph

Agenda

1. Pattern matching to validate input
   - Regular expressions

   - Deterministic/Non-Deterministic Finite Automata (DFA/NFA)

2. Finite State Machines (FSM) to model complex systems

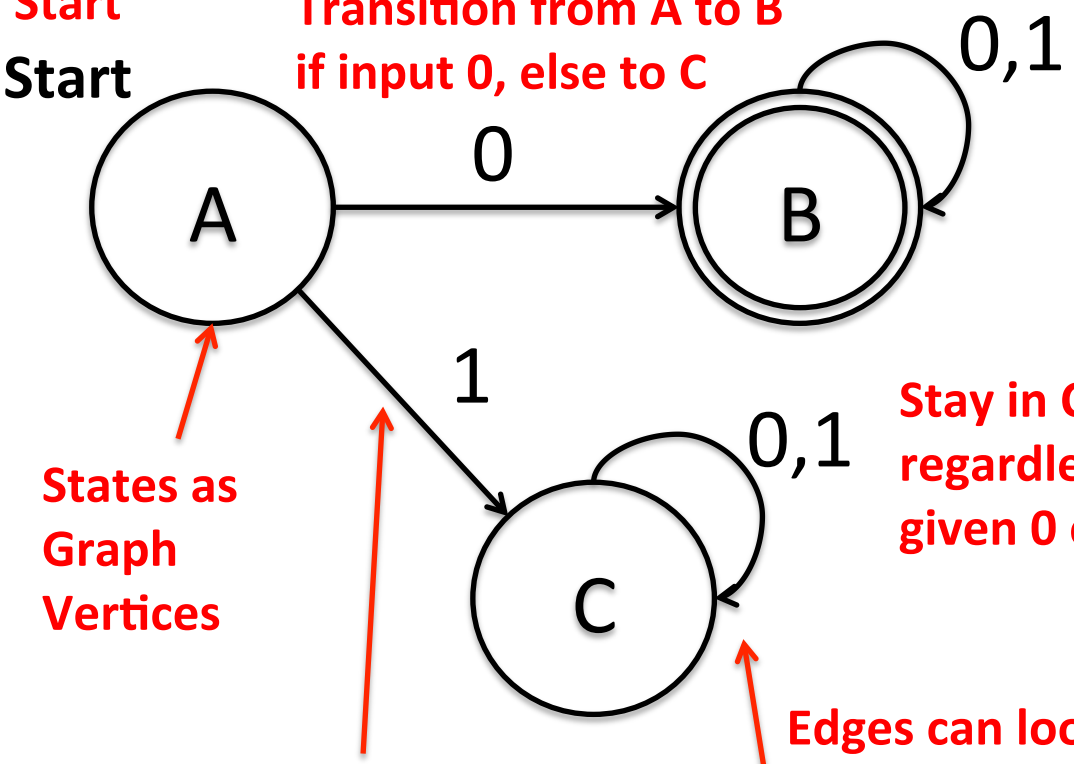
We can model States as Vertices and Transitions as Edges in a directed Graph

Finite Automata validating input

Begin at Start

Transition from A to B if input 0, else to C

Set of input symbols called alphabet



Double circle indicates valid end States, non-double circle States are invalid end States

States as Graph Vertices

Stay in C regardless if given 0 or 1

Edges as transitions between States based on input

Edges can loop back to same vertex ("self loop")

Operation:

- Begin at *Start* State
- Read character of input
- Follow graph according to input
- Continue until no more input characters
- If at valid end State, input valid, else invalid

What does this do?

- Accepts any input starting with 0

Finite Automata (FA) are formally defined as 5-tuple of States, Transitions, and inputs

Finite Automata as 5-tuple ($Q, \sum, \delta, q_0, F$)

FA = ($Q, \sum, \delta, q_0, F$)

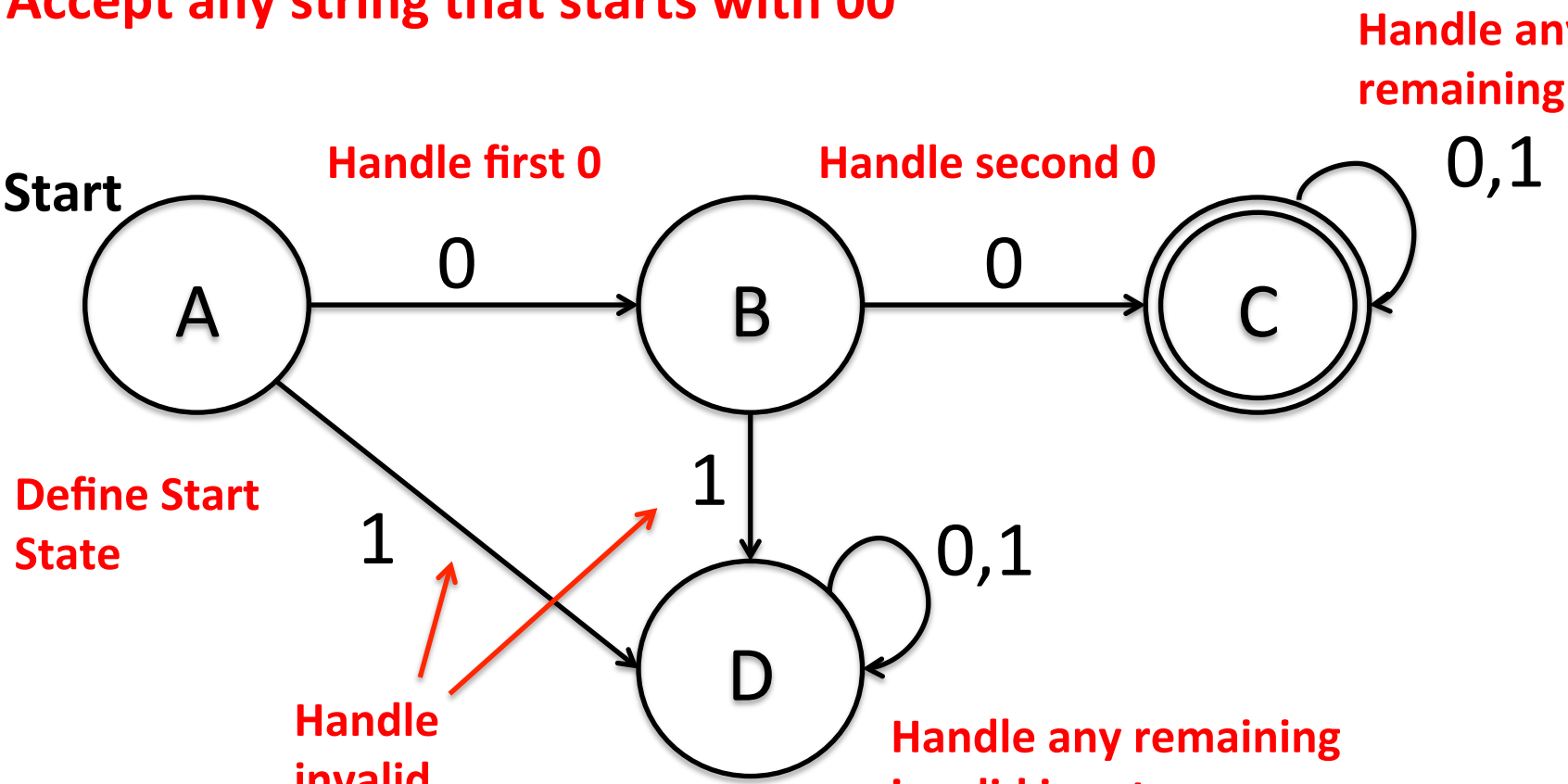
- **Q** – finite set of States (vertices in graph)
- **∑** – complete set of possible input symbols (called the *alphabet*)
- **δ** – transition function where $\delta: Q \times \sum \rightarrow Q$ (given current State Q and input symbol ∑, transition to next State Q according to δ)
- **q_0** – initial State; $q_0 \in Q$ (means q_0 is *an element of* Q)
- **F** is a set of valid end States; $F \subseteq Q$ (means F is *a subset of* Q)

We say *FA* "accepts" (validates) input $A=a_1a_2a_3...a_n$ if sequence of States $R=r_0r_1r_2...r_n$ exists in *Q* such that:

- $r_0=q_0$ //initial State is Start
- $r_{i+1} = \delta(r_i, a_{i+1})$, for i=0,1, ..., n−1 //input leads to next State
- $r_n \in F$ //last State is an element of the valid end States

We can build FAs to validate or reject input

Accept any string that starts with 00



Handle any remaining

Handle first 0

Handle second 0

Define Start State

Handle invalid input on first two characters

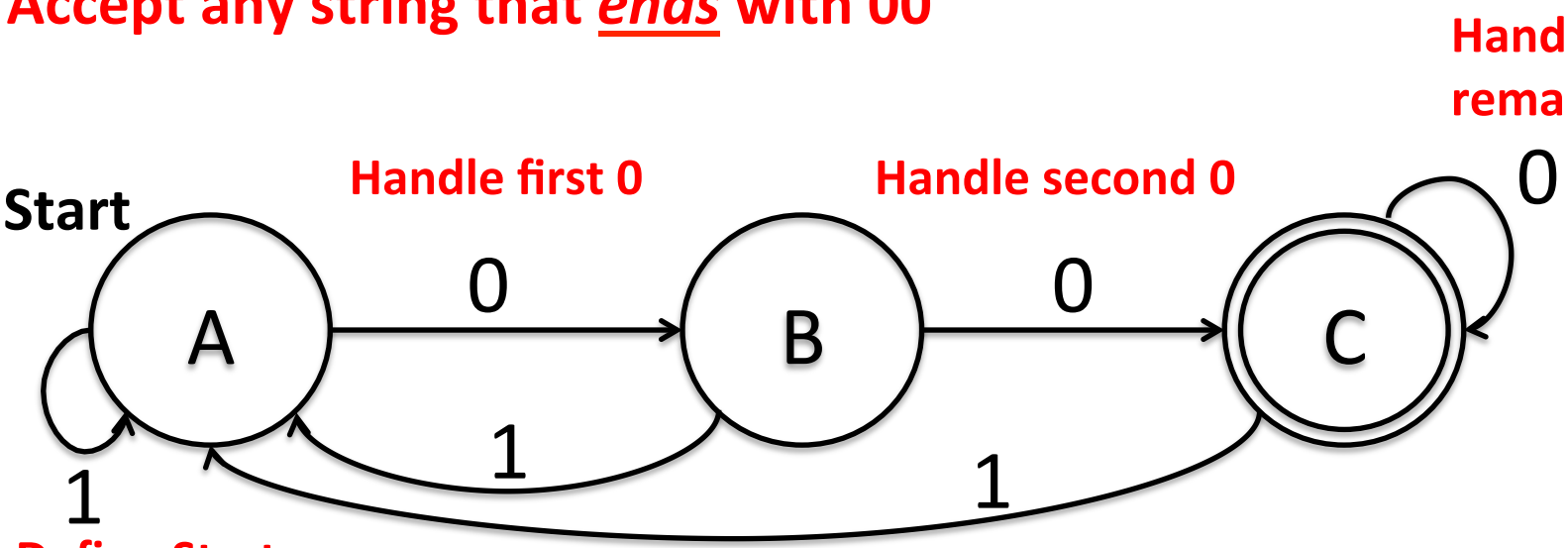
Handle any remaining invalid input

Vertices with no escape sometimes called a "trap"

Adapted from: https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/1.pdf

FAs can demonstrate "recent memory"

Accept any string that *ends* with 00



Handle any remaining 0s

Handle first 0

Handle second 0

Define Start State

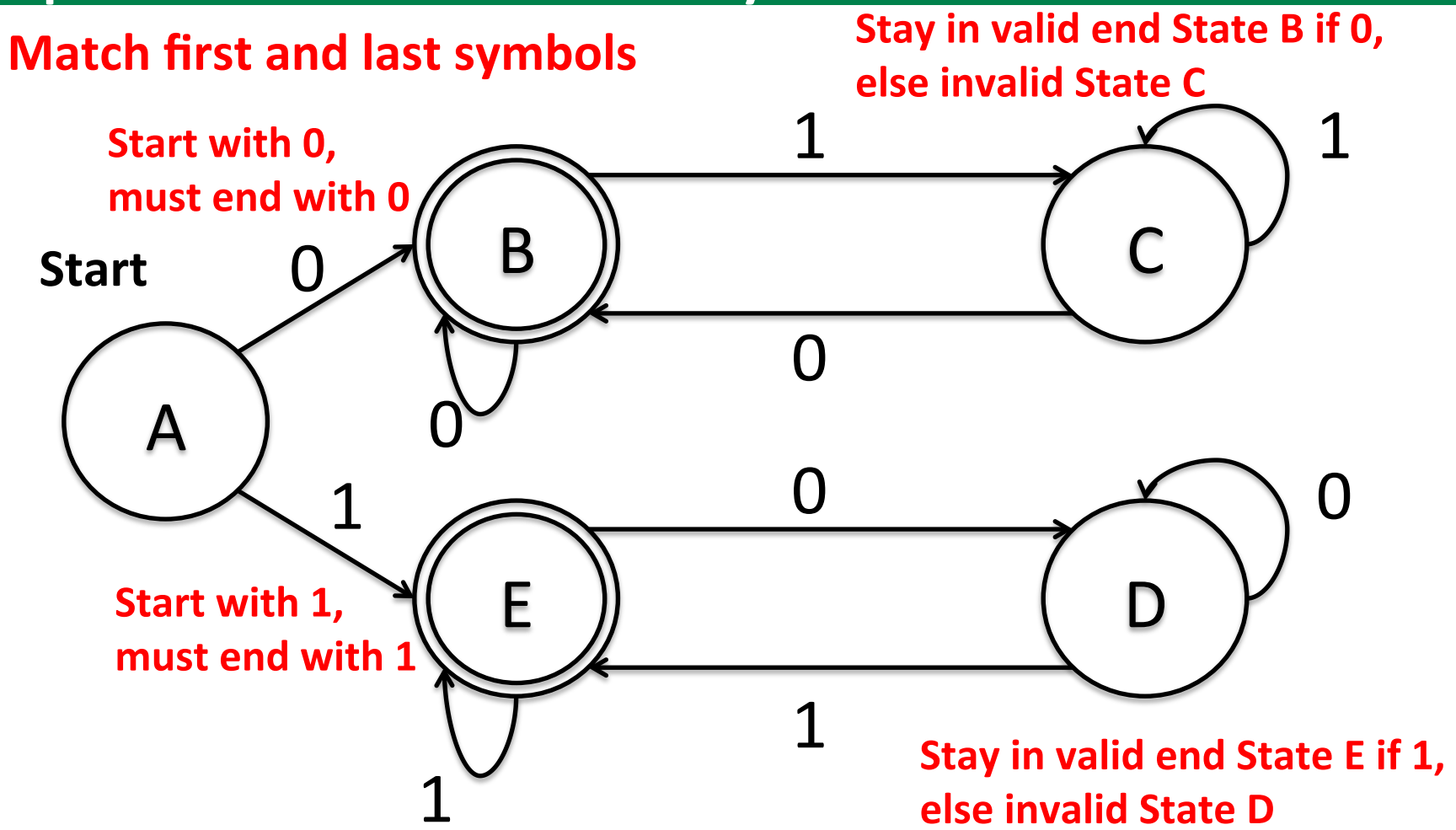
Handle any remaining invalid input

Stay here if input is 1

Not a valid end State

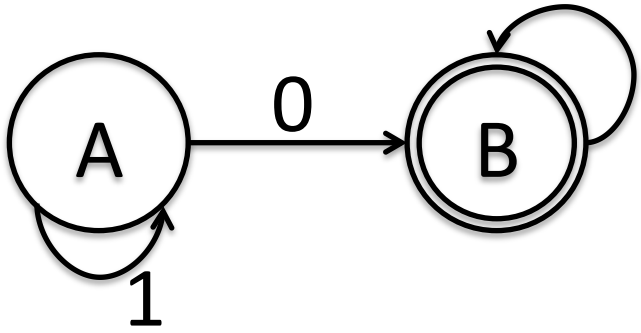
Adapted from: https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/1.pdf

Can split FA into pieces to demonstrate "permanent memory"

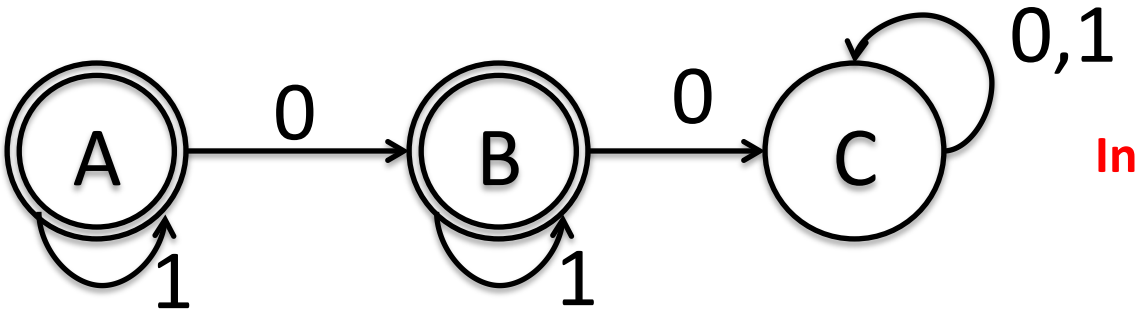


Adapted from: https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/1.pdf

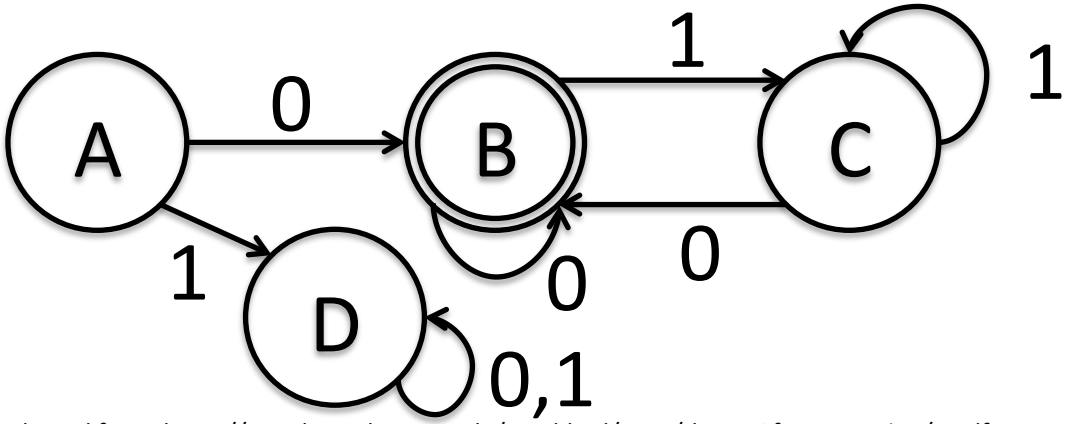
What do these FAs do?



Input has at least one 0



Input has at most one 0

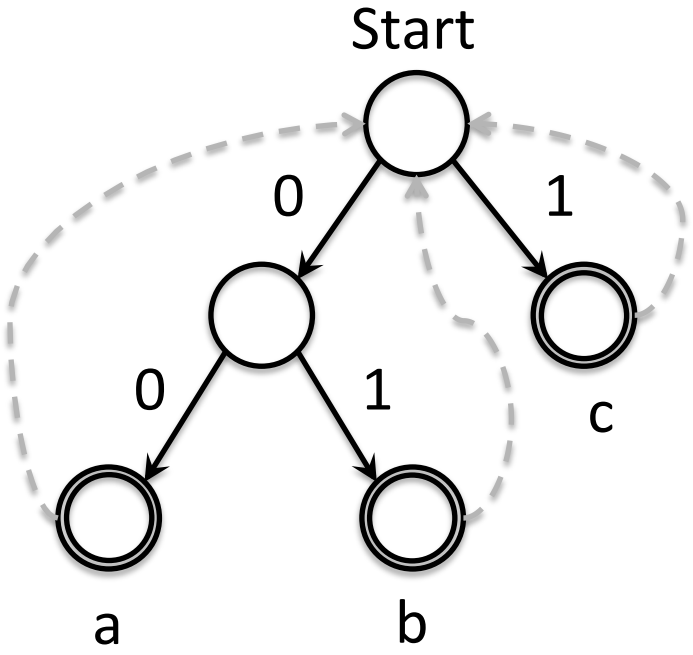


Input starts and ends with 0
(and a single 0 counts)

Adapted from: https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/1.pdf

With a slight modification, Finite Automata can validate input like Huffman

Finite Automata validating input

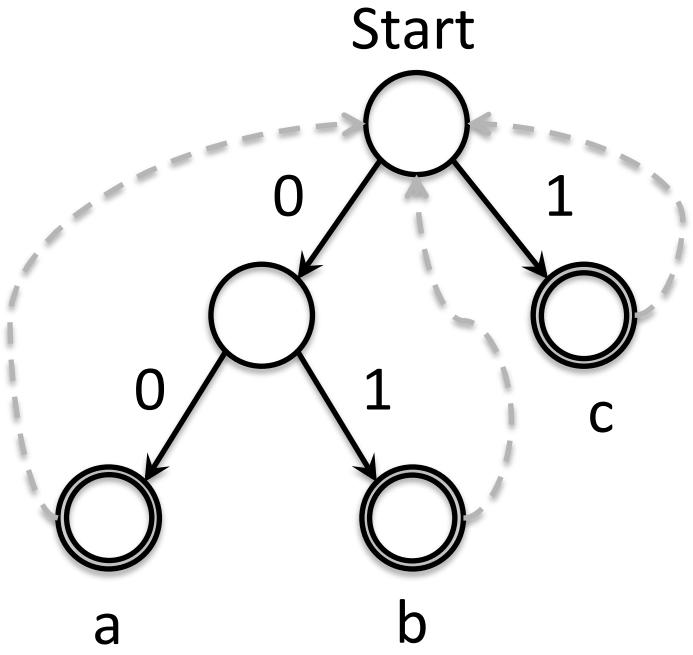


Input	Result
00	a

Leaves represent valid end states
Here can loop back to root from leaf (this is not common)
Invalid if input ends and not at valid end state (leaves here)
This is an extension of Huffman, go back to root after finding leaf

With a slight modification, Finite Automata can validate input like Huffman

Finite Automata validating input



Input	Result
00	a
01	b

Leaves represent valid end states

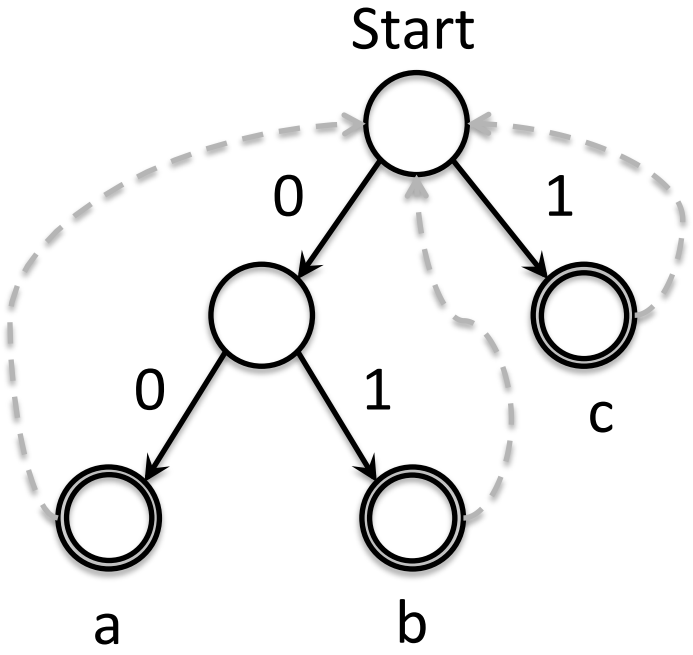
Here can loop back to root from leaf (this is not common)

Invalid if input ends and not at end state

This is an extension of Huffman, go back to root after finding leaf

With a slight modification, Finite Automata can validate input like Huffman

Finite Automata validating input



Input	Result
00	a
01	b
1	c

Leaves represent valid end states

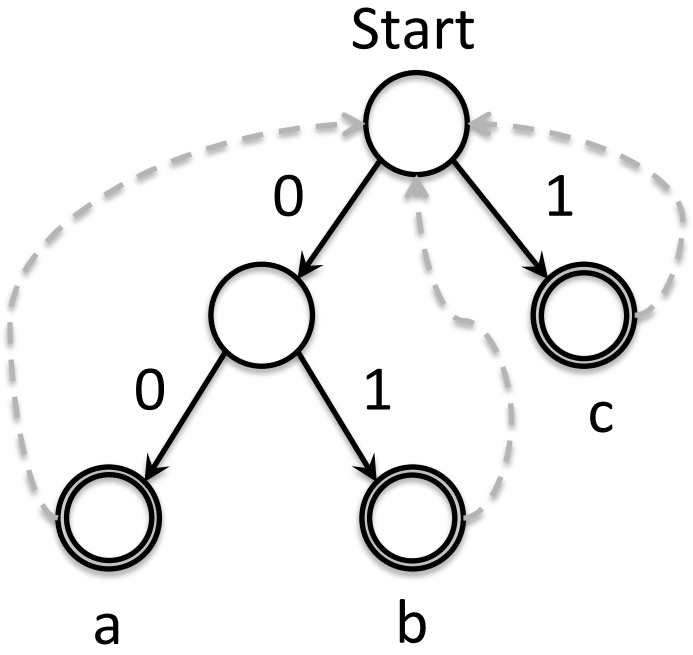
Here can loop back to root from leaf (this is not common)

Invalid if input ends and not at end state

This is an extension of Huffman, go back to root after finding leaf

With a slight modification, Finite Automata can validate input like Huffman

Finite Automata validating input



Input	Result
00	a
01	b
1	c
0	Invalid

Leaves represent valid end states

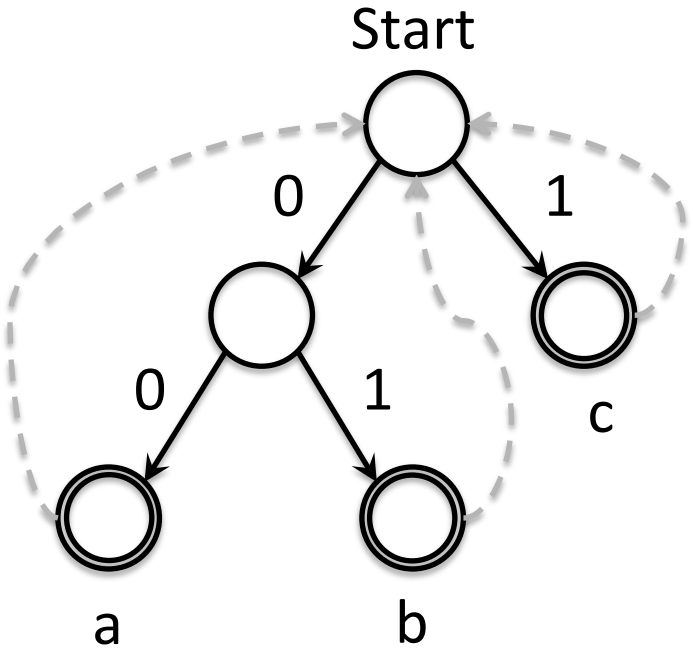
Here can loop back to root from leaf (this is not common)

Invalid if input ends and not at end state

This is an extension of Huffman, go back to root after finding leaf

With a slight modification, Finite Automata can validate input like Huffman

Finite Automata validating input



Input	Result
00	a
01	b
1	c
0	Invalid
001100	acca

Leaves represent valid end states

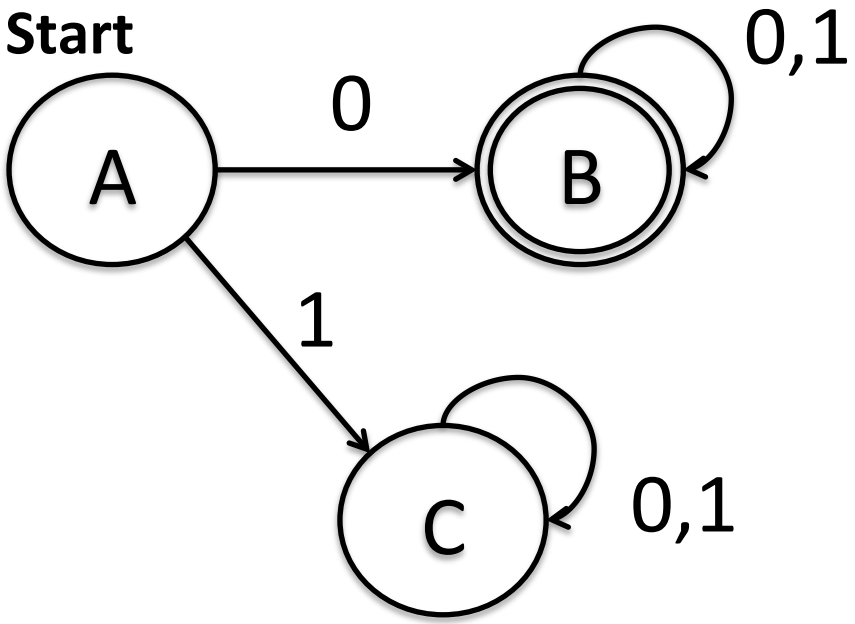
Here can loop back to root from leaf (this is not common)

Invalid if input ends and not at end state

This is an extension of Huffman, go back to root after finding leaf

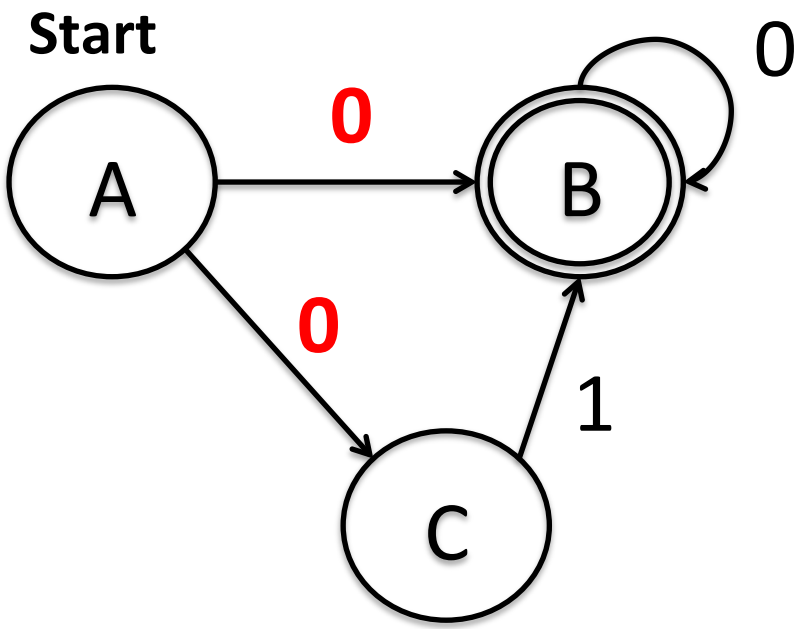
Finite Automata come in two flavors, Deterministic and Nondeterministic

Deterministic Finite Automaton (DFA)



- Exactly one transition for each possible input
- No ambiguity

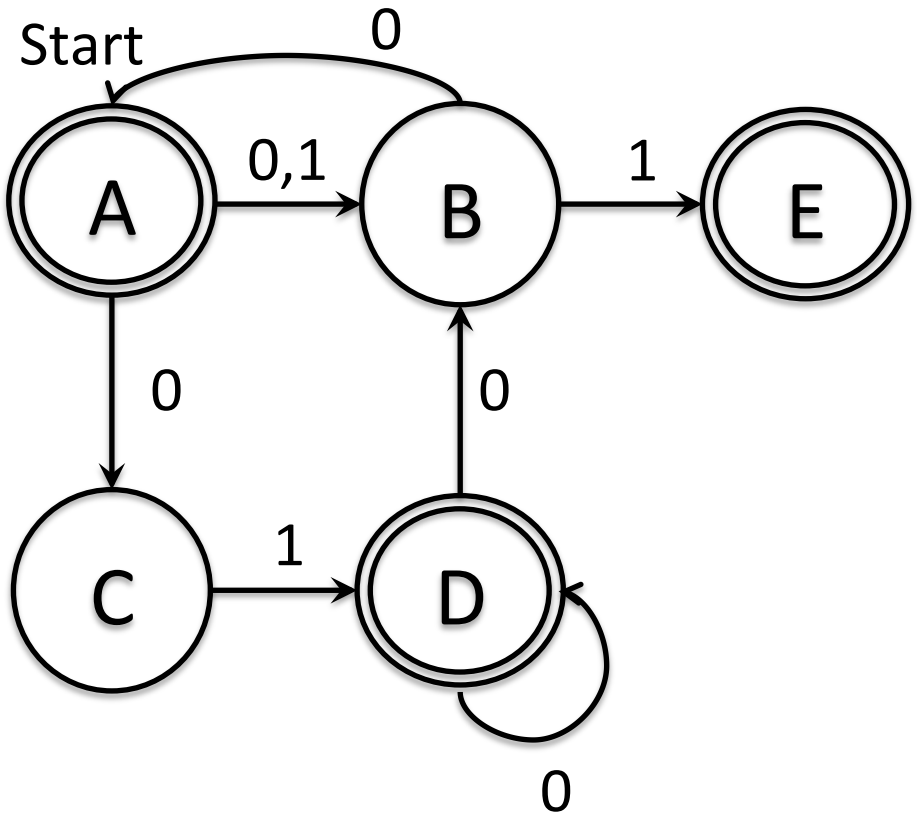
Nondeterministic Finite Automaton (NFA)



- May have <u>0, 1, or more</u> choices for each transition
- Unspecified inputs are invalid
- True if end in any valid State

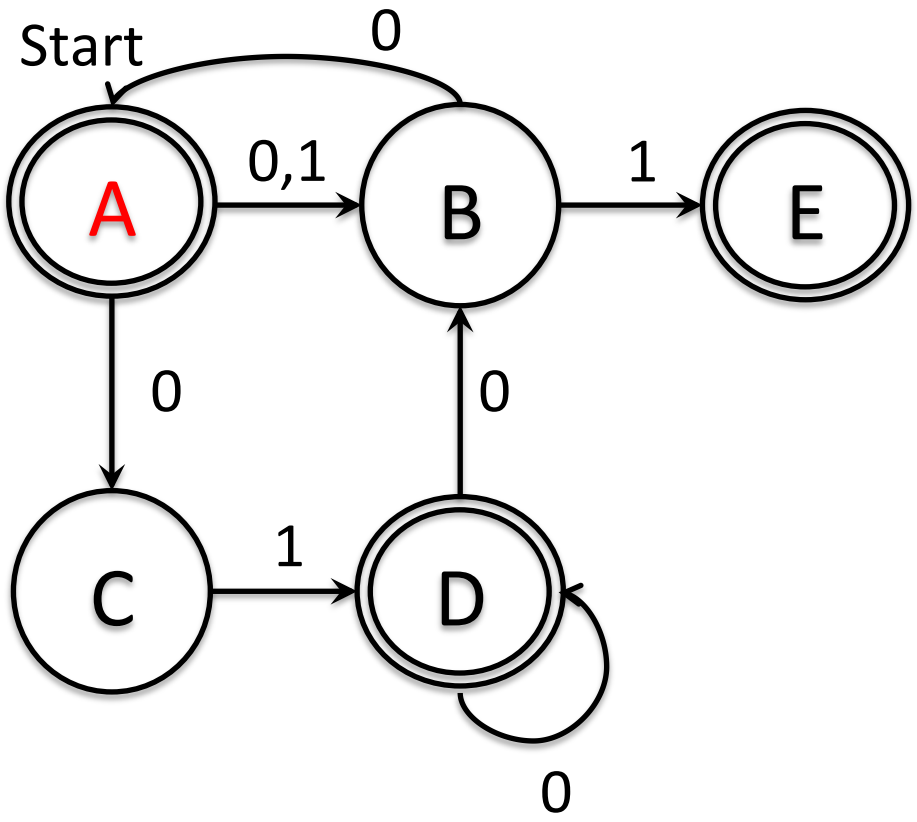
Sometimes we cannot map from a State a single next State

NFAs can have multiple next States



Sometimes we cannot map from a State a single next State

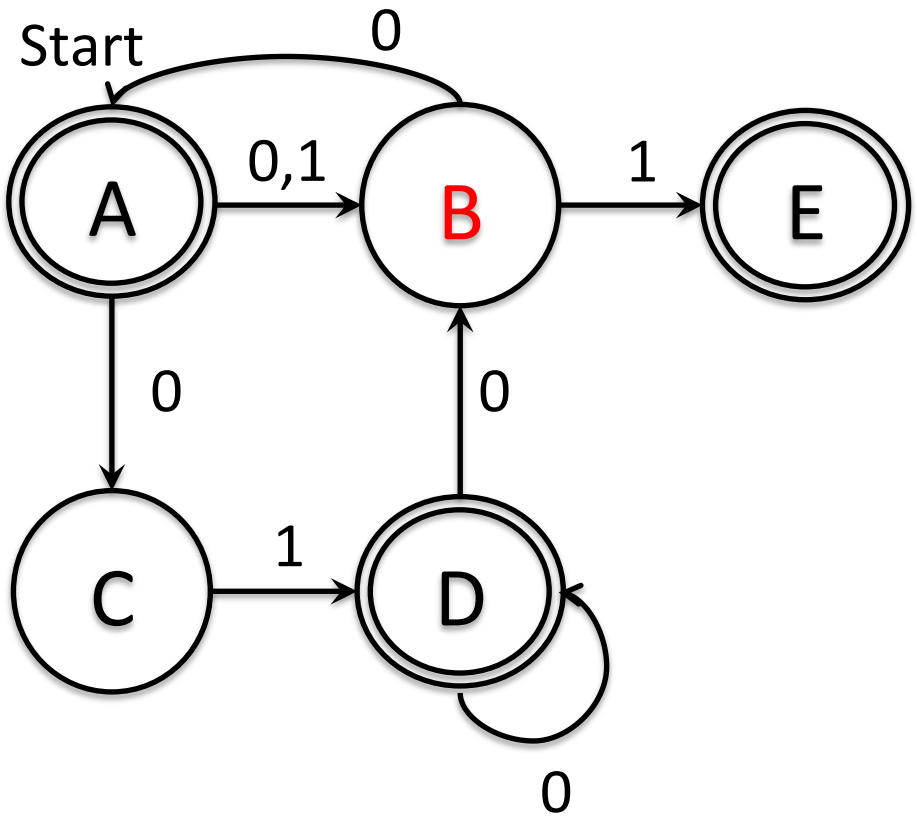
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}

Sometimes we cannot map from a State a single next State

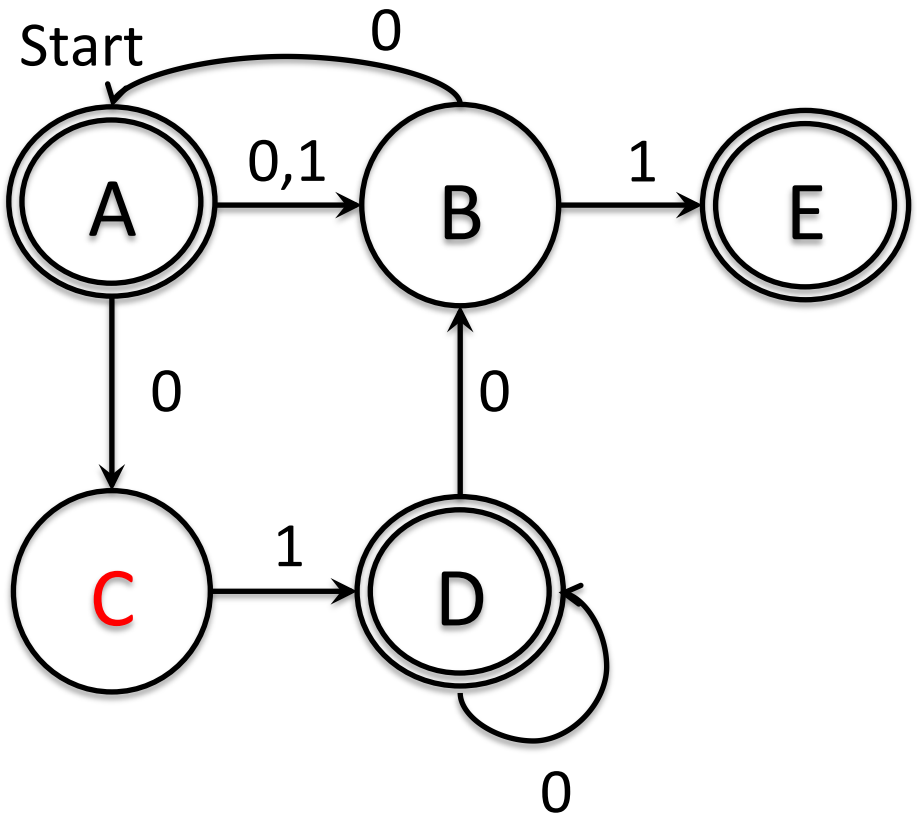
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}

Sometimes we cannot map from a State a single next State

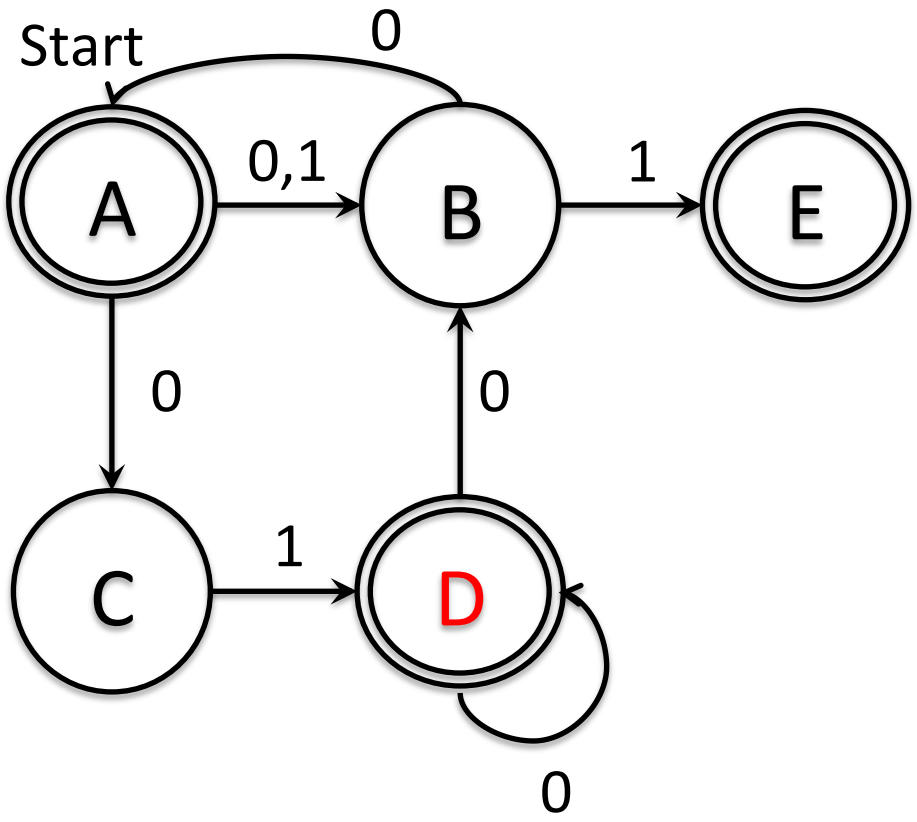
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}
C	0	{}
C	1	{D}

Sometimes we cannot map from a State a single next State

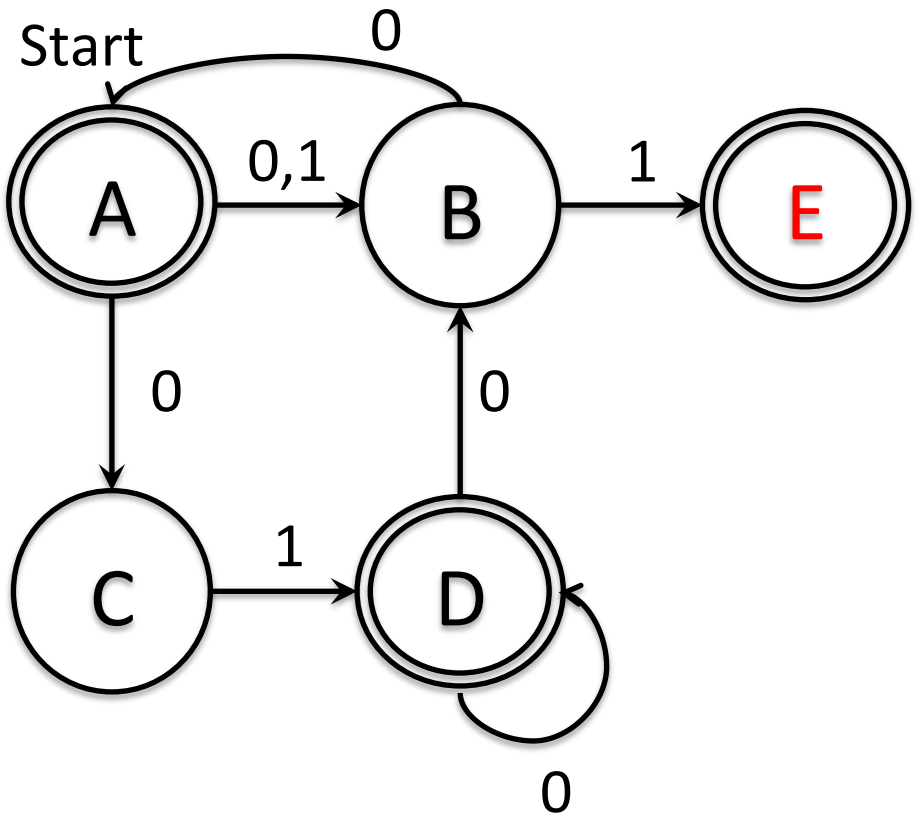
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}
C	0	{}
C	1	{D}
D	0	{B,D}
D	1	{}

Sometimes we cannot map from a State a single next State

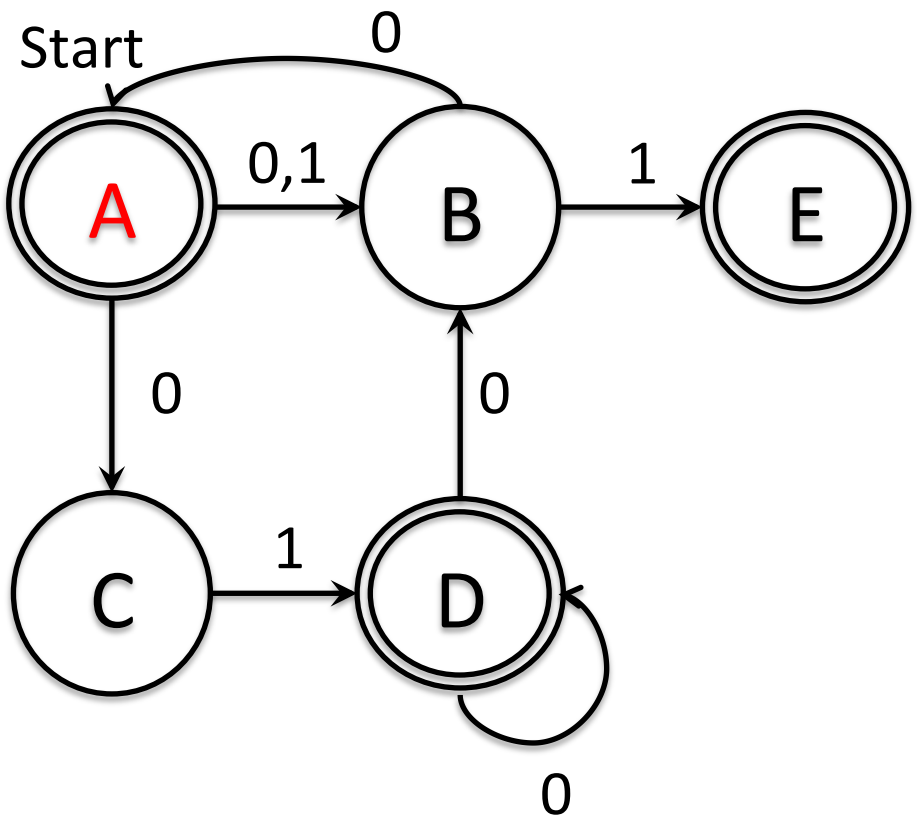
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}
C	0	{}
C	1	{D}
D	0	{B,D}
D	1	{}
E	0	{}
E	1	{}

In that case, must keep track of all possible States

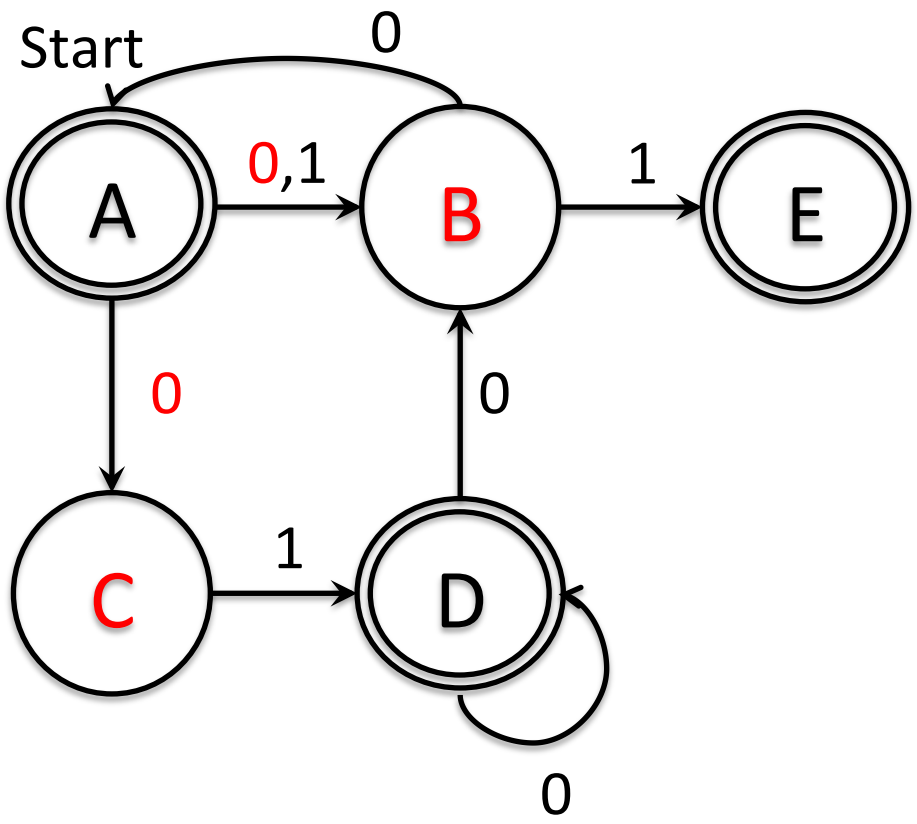
NFAs can have multiple next States



Input	Possible States
Start	{A}

In that case, must keep track of all possible States

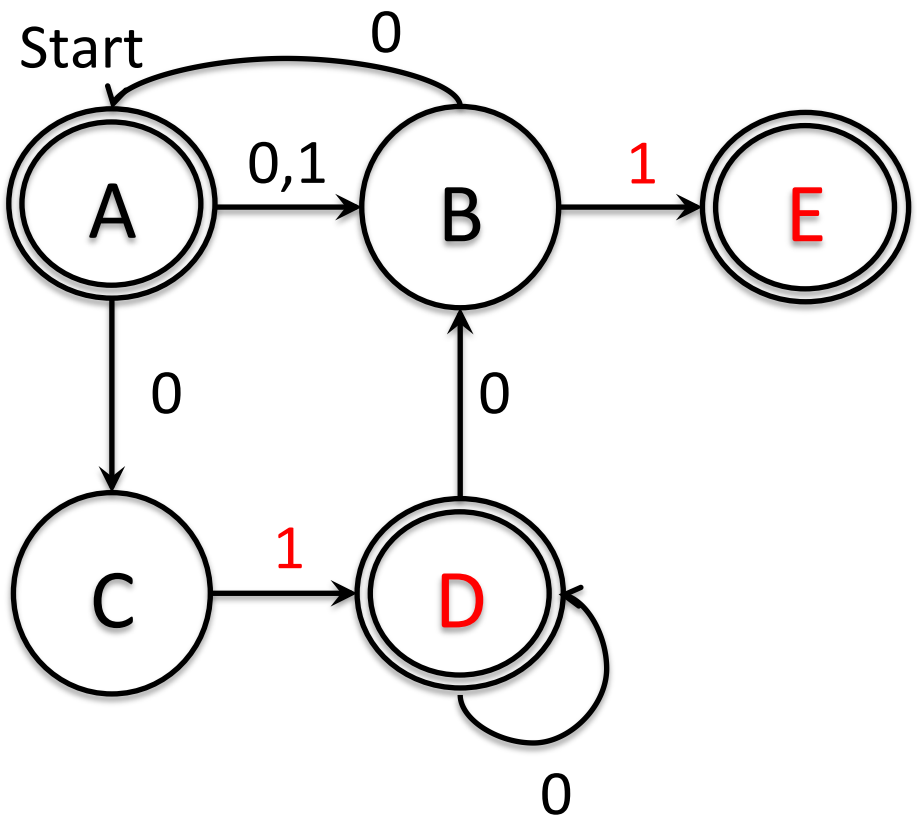
NFAs can have multiple next States



Input	Possible States
Start	{A}
0	{B,C}

In that case, must keep track of all possible States

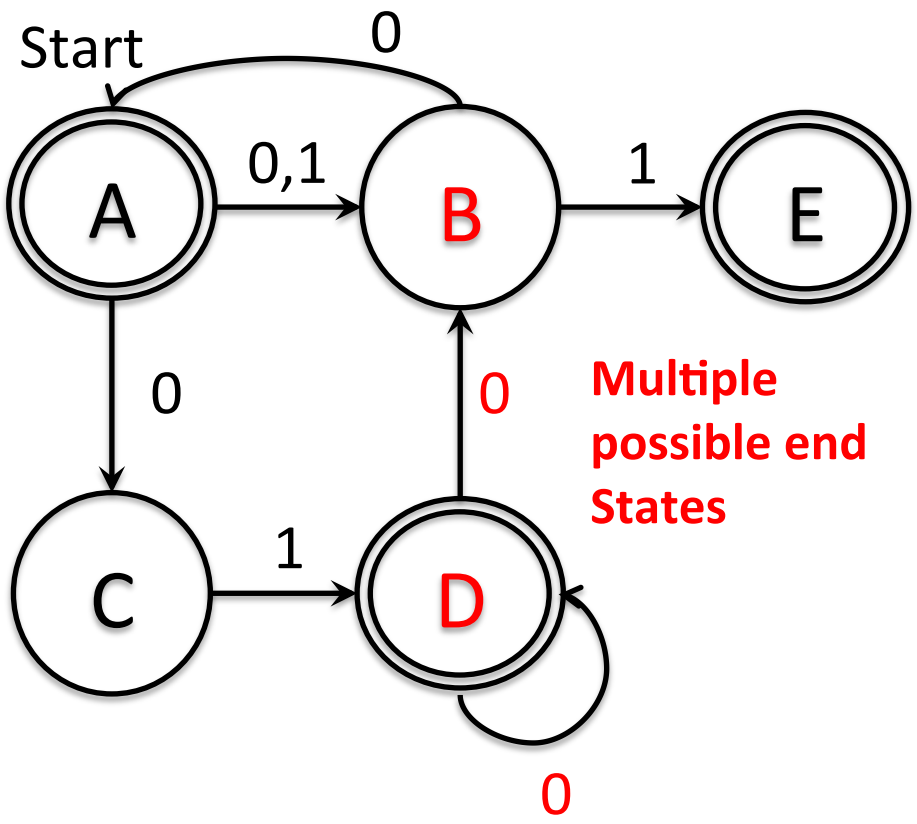
NFAs can have multiple next States



Input	Possible States
Start	{A}
0	{B,C}
1	{E,D}

In that case, must keep track of all possible States

NFAs can have multiple next States



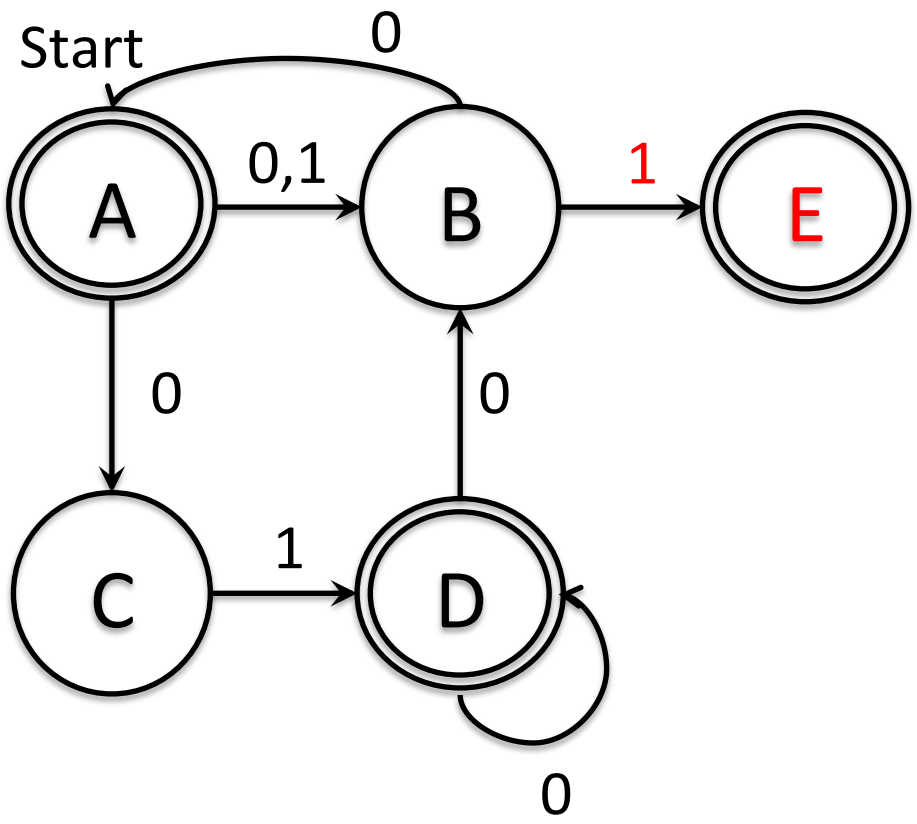
Multiple possible end States

Input	**Possible States**
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}

If end now, input is valid because D is valid end State

In that case, must keep track of all possible States

NFAs can have multiple next States

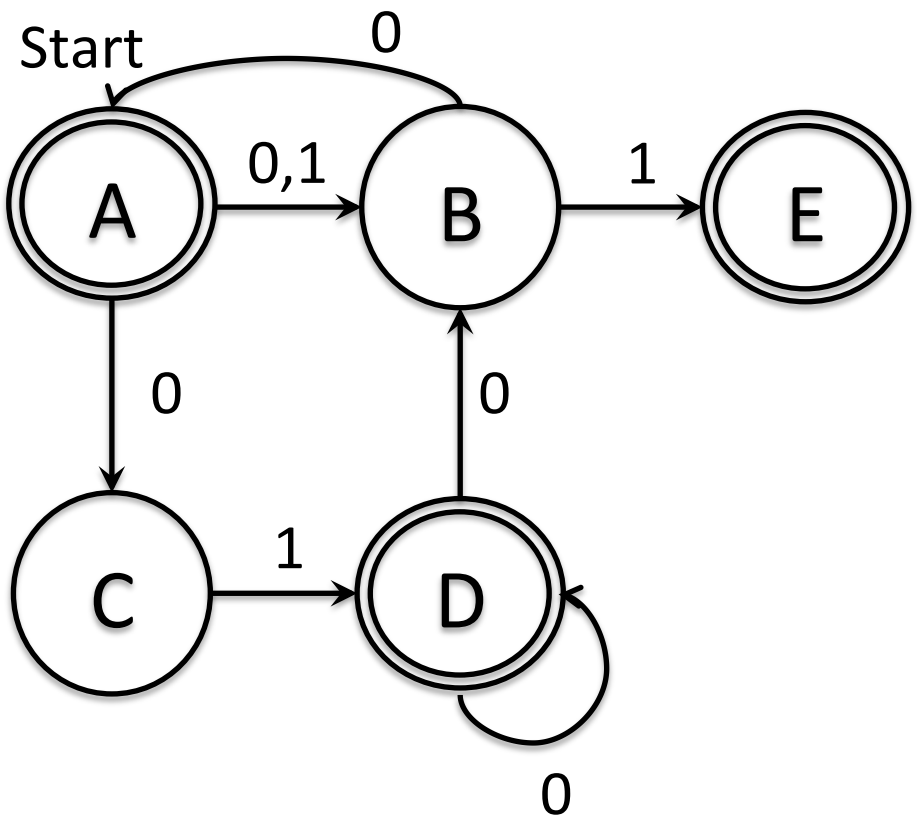


Input	Possible States
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}
1	{E}

Still have valid end State, if input ends now, return true

In that case, must keep track of all possible States

NFAs can have multiple next States

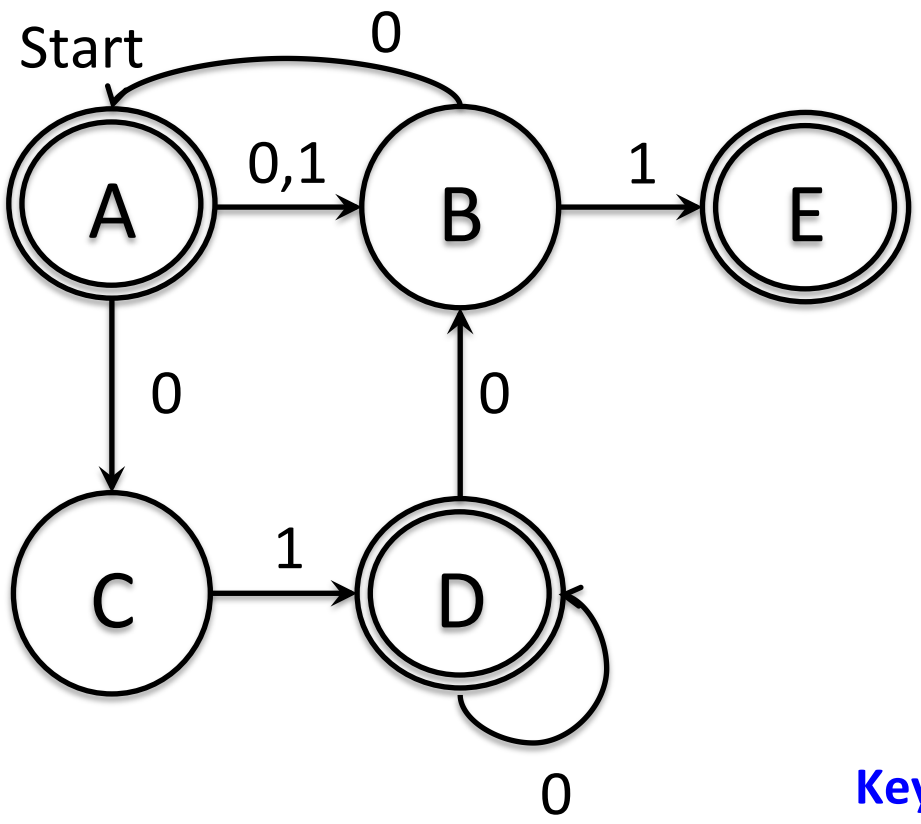


Input	Possible States
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}
1	{E}
0	{}

No valid States, return false

In that case, must keep track of all possible States

NFAs can have multiple next States

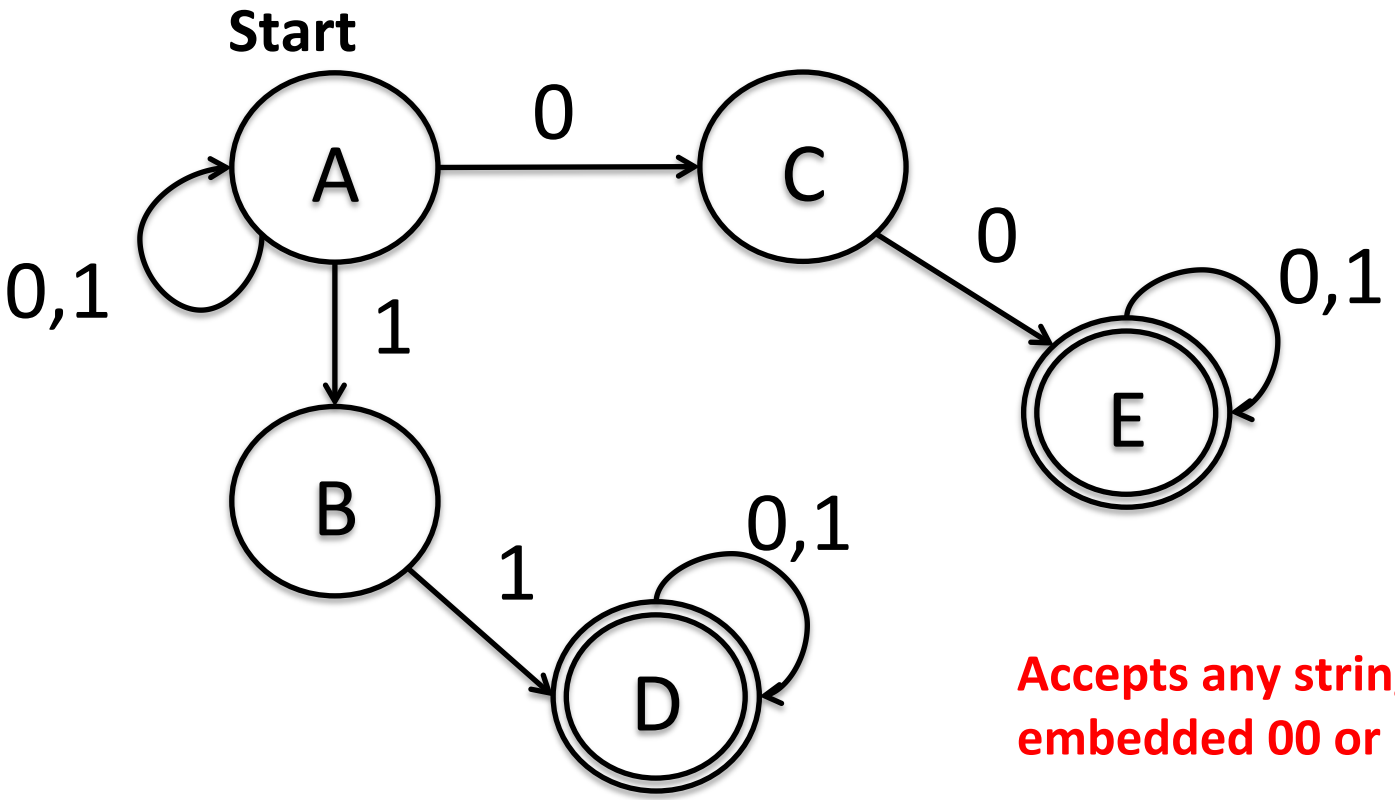


Input	Possible States
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}
1	{E}
0	{}

No valid States, return false

Key point: kept track of *all* possible States as input processed
If *any* ending state is valid, then accept input

One more practice before looking at code, what does this NFA do?



Accepts any string with embedded 00 or 11

DFA.java creates Deterministic Finite Automata

```
17 public class DFA {
18     String start; //assume only one starting position
19     Set<String> ends; //possibly multiple end states, hence the set ins
20     Map<String, Map<Character,String>> transitions; // state -> (charac
21
22     /**
23      * Constructs the DFA from the arrays, as specified in the overall
24      */
25     DFA(String[] ss, String[] ts) {
26         ends = new TreeSet<String>();
27         transitions = new TreeMap<String, Map<Character,String>>();
28
29         // Parse states
30         for (String v : ss) {
31             String[] pieces = v.split(","); //pieces[0] = state name,
32             //look for start and end markers
33             if (pieces.length>1) {
34                 if (pieces[1].equals("S")) {
35                     start = pieces[0];
36                 }
37                 else if (pieces[1].equals("E")) {
38                     ends.add(pieces[0]);
39                 }
40             }
41         }
42
43         // Parse transitions
44         for (String e : ts) {
45             String[] pieces = e.split(","); //pieces[0] = starting fror
46             String from = pieces[0];
47             String to = pieces[1];
48             if (!transitions.containsKey(from)) {
49                 transitions.put(from, new TreeMap<Character,String>())
50             }
51             for (int i=2; i<pieces.length; i++) { //could be multiple i
52                 transitions.get(from).put(pieces[i].charAt(0), to);
53             }
54         }
55
56         System.out.println("start:"+start);
57         System.out.println("end:"+ends);
58         System.out.println("transitions:"+transitions);
59     }
```

- Store *start* node (there will be only one)
- Store valid end states in Set (could be multiple valid end States)
- Track Transitions with Map of Maps
  - Key for outer Map is State
  - Value for outer Map another Map
  - Inner Map has Character as Key, next State as Value
  - So, given a State and a Character, can look up next State

- Parse States in String[] *ss = {"A,S","B,E","C"}*
- States will be in form:
  - <Char>, S indicates starting State (e.g., "A,S" means A is the Start)
  - <Char>, E indicates ending State (e.g., "B,E" means B is *an* end State)
  - <Char> indicates non-starting or ending state (e.g., "C")

DFA.java creates Deterministic Finite Automata

```
17 public class DFA {
18     String start; //assume only one starting position
19     Set<String> ends; //possibly multiple end states, hence the set ins
20     Map<String, Map<Character,String>> transitions; // state -> (charac
21
22     /**
23      * Constructs the DFA from the arrays, as specified in the overall
24      */
25     DFA(String[] ss, String[] ts) {
26         ends = new TreeSet<String>();
27         transitions = new TreeMap<String, Map<Character,String>>();
28
29         // Parse states
30         for (String v : ss) {
31             String[] pieces = v.split(","); //pieces[0] = state name,
32             //look for start and end markers
33             if (pieces.length>1) {
34                 if (pieces[1].equals("S")) {
35                     start = pieces[0];
36                 }
37                 else if (pieces[1].equals("E")) {
38                     ends.add(pieces[0]);
39                 }
40             }
41         }
42
43         // Parse transitions
44         for (String e : ts) {
45             String[] pieces = e.split(","); //pieces[0] = starting fror
46             String from = pieces[0];
47             String to = pieces[1];
48             if (!transitions.containsKey(from)) {
49                 transitions.put(from, new TreeMap<Character,String>())
50             }
51             for (int i=2; i<pieces.length; i++) { //could be multiple i
52                 transitions.get(from).put(pieces[i].charAt(0), to);
53             }
54         }
55
56         System.out.println("start:"+start);
57         System.out.println("end:"+ends);
58         System.out.println("transitions:"+transitions);
59     }
```

- **Parse Transitions in String[] *ts = {"A,B,0"…***
- **Transition in form:**
  - **<State1>,<State2>,<Char>,<Char>**
  - **Means transition from State1 to State2 if see character <Char>**
  - **"A,B,0" means transition from State A to State B if given Character 0**

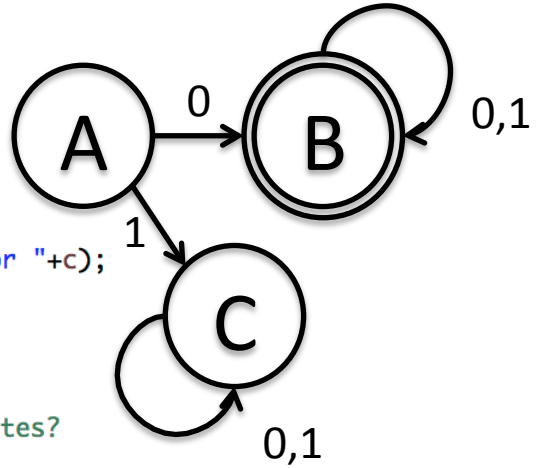
DFA.java creates Deterministic Finite Automata

```
17 public class DFA {
18     String start; //assume only one starting position
19     Set<String> ends; //possibly multiple end states, hence the set ins
20     Map<String, Map<Character,String>> transitions; // state -> (charac
21
22     /**
23      * Constructs the DFA from the arrays, as specified in the overall
24      */
25     DFA(String[] ss, String[] ts) {
26         ends = new TreeSet<String>();
27         transitions = new TreeMap<String, Map<Character,String>>();
28
29         // Parse states
30         for (String v : ss) {
31             String[] pieces = v.split(","); //pieces[0] = state name, 
32             //look for start and end markers
33             if (pieces.length>1) {
34                 if (pieces[1].equals("S")) {
35                     start = pieces[0];
36                 }
37                 else if (pieces[1].equals("E")) {
38                     ends.add(pieces[0]);
39                 }
40             }
41         }
42
43         // Parse transitions
44         for (String e : ts) {
45             String[] pieces = e.split(","); //pieces[0] = starting fro
46             String from = pieces[0];
47             String to = pieces[1];
48             if (!transitions.containsKey(from)) {
49                 transitions.put(from, new TreeMap<Character,String>())
50             }
51             for (int i=2; i<pieces.length; i++) { //could be multiple 
52                 transitions.get(from).put(pieces[i].charAt(0), to);
53             }
54         }
55
56         System.out.println("start:"+start);
57         System.out.println("end:"+ends);
58         System.out.println("transitions:"+transitions);
59     }
```

- **Parse Transitions in String[] *ts = {"A,B,0"…***
- **Transition in form:**
  - **<State1>,<State2>,<Char>,<Char>**
  - **Means transition from State1 to State2 if see character <Char>**
  - **"A,B,0" means transition from State A to State B if given Character 0**
- **Add Transitions to Map called transitions**

DFA.java creates Deterministic Finite Automata

```
65    public boolean match(String s) {
66        String curr = start; // where we are now
67        for (int i=0; i<s.length(); i++) {
68            char c = s.charAt(i);
69            if (!transitions.get(curr).containsKey(c)) {
70                System.out.println("This isn't a DFA! No transition from "+curr+" for "+c);
71                return false;
72            }
73            curr = transitions.get(curr).get(c); // take a step according to c
74        }
75        return ends.contains(curr); // did we end up in one of the desired final states?
76    }
77
78    /**
79     * Helper method to test matching against a bunch of strings, printing the results
80     */
81    public void test(String[] inputs) {
82        for (String s : inputs)
83            System.out.println(s + ":" + match(s));
84    }
85
86    public static void main(String[] args) {
87        String[] ss1 = { "A,S", "B,E", "C" };
88        String[] ts1 = { "A,B,0", "A,C,1", "B,B,0,1", "C,C,0,1" };
89        DFA dfa1 = new DFA(ss1, ts1);
90
91        String[] testsT1 = { "0", "00", "00000", "0010101" };
92        dfa1.test(testsT1);
93        String[] testsF1 = { "", "1", "1100110" };
94        dfa1.test(testsF1);
95    }
96 }
```



- **Create 3 States:**
  - **A (start), B (end), C**
- **Create transitions between States based on input**

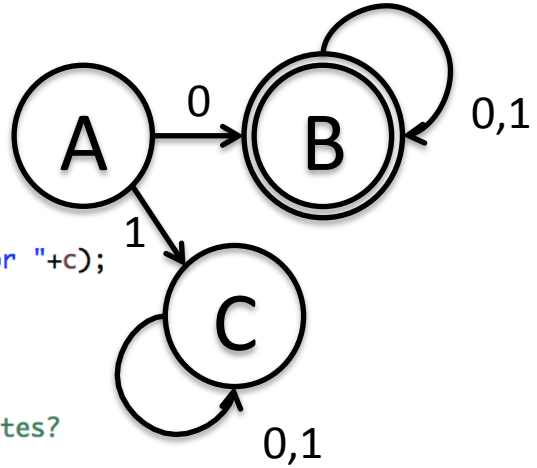
Transitions Map

A	0	B
	1	C
B	0	B
	1	B
C	0	C
	1	C

DFA.java creates Deterministic Finite Automata

```
65    public boolean match(String s) {
66        String curr = start; // where we are now
67        for (int i=0; i<s.length(); i++) {
68            char c = s.charAt(i);
69            if (!transitions.get(curr).containsKey(c)) {
70                System.out.println("This isn't a DFA! No transition from "+curr+" for "+c);
71                return false;
72            }
73            curr = transitions.get(curr).get(c); // take a step according to c
74        }
75        return ends.contains(curr); // did we end up in one of the desired final states?
76    }
77
78    /**
79     * Helper method to test matching against a bunch of strings, printing the results
80     */
81    public void test(String[] inputs) {
82        for (String s : inputs)
83            System.out.println(s + ":" + match(s));
84    }
85
86    public static void main(String[] args) {
87        String[] ss1 = { "A,S", "B,E", "C" };
88        String[] ts1 = { "A,B,0", "A,C,1", "B,B,0,1", "C,C,0,1" };
89        DFA dfa1 = new DFA(ss1, ts1);
90
91        String[] testsT1 = { "0", "00", "00000", "0010101" };
92        dfa1.test(testsT1);
93        String[] testsF1 = { "", "1", "1100110" };
94        dfa1.test(testsF1);
95    }
96 }
```

- Match test string s
- Start at start (A)
- Follow transitions



- Create 3 States:
  - A (start), B (end), C
- Create transitions between States based on input

Transitions Map

A	0	B
	1	C
B	0	B
	1	B
C	0	C
	1	C

All true

All false

NFA.java creates Non-Deterministic Finite Automata

```
16 public class NFA {
17     String start;
18     Set<String> ends;
19     Map<String, Map<Character,List<String>>> transitions; // state -> (character -> [next states])
20     // note the difference from DFA: can have multiple different transitions from state for character
21
22     /**
23      * Constructs the DFA from the arrays, as specified in the overall header
24      */
25     NFA(String[] ss, String[] ts) {
26         ends = new TreeSet<String>();
27         transitions = new TreeMap<String, Map<Character,List<String>>>();
28
29         // States
30         for (String v : ss) {
31             String[] pieces = v.split(",");
32             if (pieces.length>1) {
33                 if (pieces[1].equals("S")) start = pieces[0];
34                 else if (pieces[1].equals("E")) ends.add(pieces[0]);
35             }
36         }
37
38         // Transitions
39         for (String e : ts) {
40             String[] pieces = e.split(",");
41             String from = pieces[0], to = pieces[1];
42             if (!transitions.containsKey(from)) transitions.put(from, new TreeMap<Character,List<String>>
43             for (int i=2; i<pieces.length; i++) {
44                 char c = pieces[i].charAt(0);
45                 // difference from DFA: list of next states
46                 if (!transitions.get(from).containsKey(c)) transitions.get(from).put(c, new ArrayList<Str
47                 transitions.get(from).get(c).add(to);
48             }
49         }
50
51         System.out.println("start:"+start);
52         System.out.println("end:"+ends);
53         System.out.println("transitions:"+transitions);
54     }
```

- **Like DFA, but transitions are a Map of Map of Lists**
- **State -> Character -> Next possible states for this Character (could be more than one)**
- **Add List of next States in constructor**

NFA.java creates Non-Deterministic Finite Automata

```
60    public boolean match(String s) {
61        // difference from DFA: multiple current states
62        Set<String> currStates = new TreeSet<String>();
63        currStates.add(start);
64        for (int i=0; i<s.length(); i++) {
65            char c = s.charAt(i);
66            Set<String> nextStates = new TreeSet<String>();
67            // transition from each current state to each of its next s
68            for (String state : currStates)
69                if (transitions.get(state).containsKey(c))
70                    nextStates.addAll(transitions.get(state).get(c));
71            if (nextStates.isEmpty()) return false; // no way forward f
72            currStates = nextStates;
73        }
74        // end up in multiple states -- accept if any is an end state
75        for (String state : currStates) {
76            if (ends.contains(state)) return true;
77        }
78        return false;
79    }
```

Set *currStates tracks* all possible States given input so far
Initially set to start

Keep a Set of all possible States that could be reached from all *currStates* given input

addAll adds all items in List to nextStates Set

- **Given input and all possible current States, track all possible next states**
- **Return false if no valid next states**
- **Update *currStates* to *nextStates***

After processing all input, see if *<u>any</u>* State in *currState* is a valid end state
If yes, then return true, else false

PS-5 is similar to this!

Agenda

1. Pattern matching to validate input
   - Regular expressions
   - Deterministic/Non-Deterministic Finite Automata (DFA/NFA)
2. Finite State Machines (FSM) to model complex systems

Finite State Machines (FSM) work like FAs, but track the State of a complex system

Finite State Machine (FSM)

1. Enumerate all States possible for the system
2. Enumerate all possible Events that can occur
3. Map Transition from each State to another State (possibly the same State) given any Event
4. Start at known State
5. Transition to new State as Events occur
6. You now track the current state of the system

Sensors detect arrival and departure of cars in parking spaces

One sensor in each parking space (11,000 total sensors in San Fran)

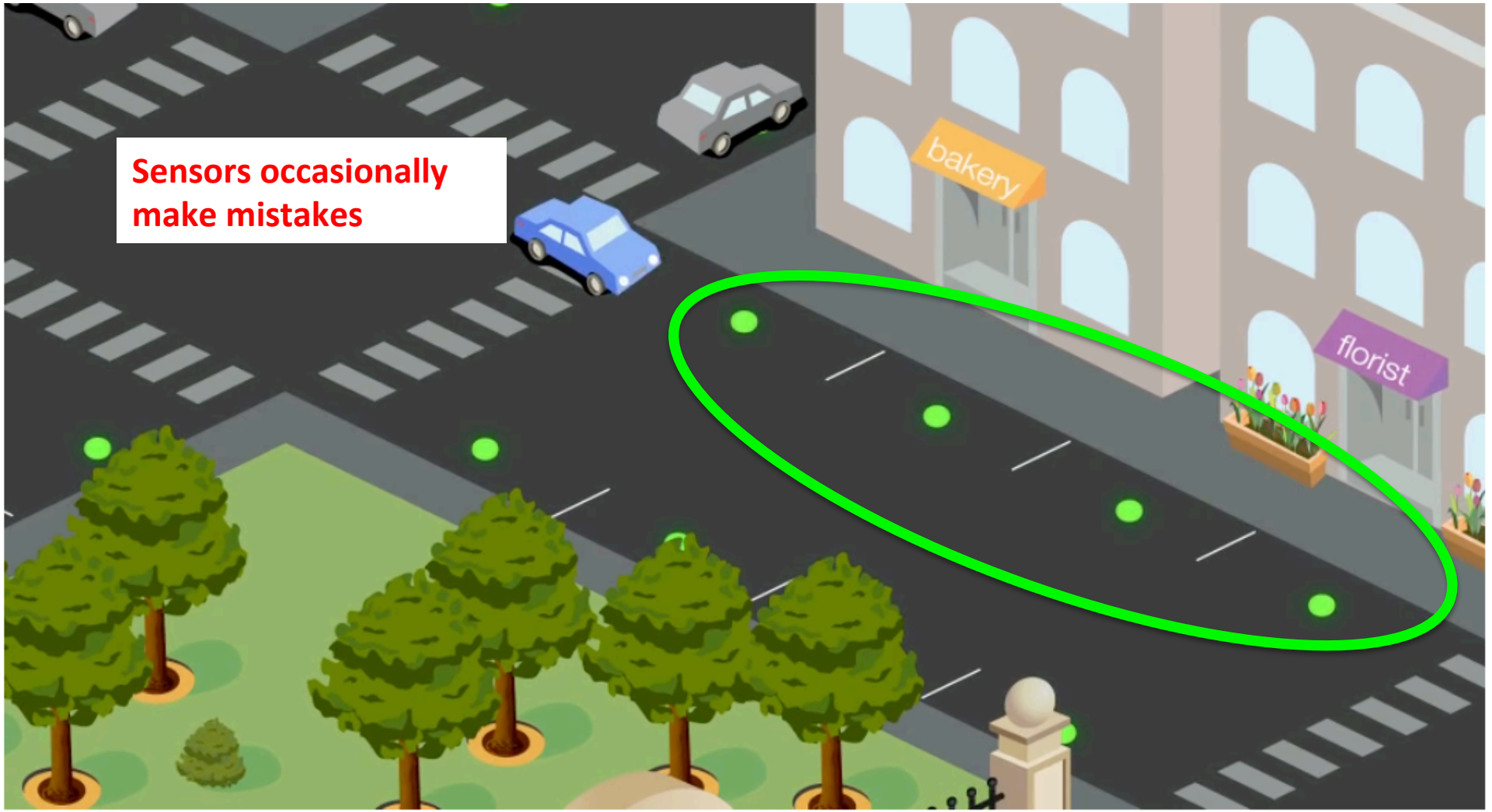


Image: Fybr.com

Parking meters detect payments and payment expirations

One parking meter per parking space

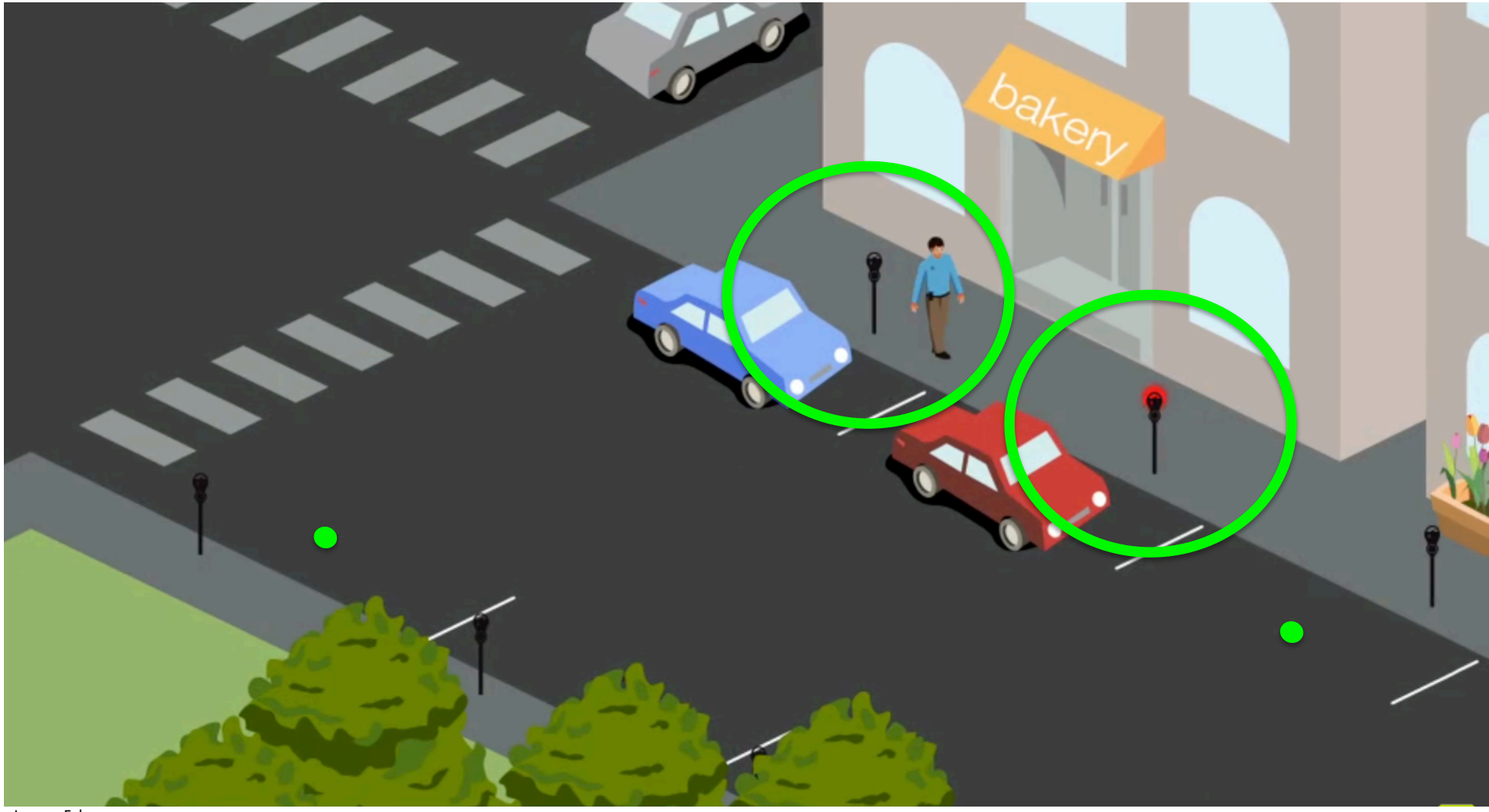


Image: Fybr.com

Aggregate sensor data to show drivers where they can find parking in real time

Fisherman's Wharf in San Francisco, CA

Green < 75% occupied, yellow = 75-90% occupied, red > 90% occupied

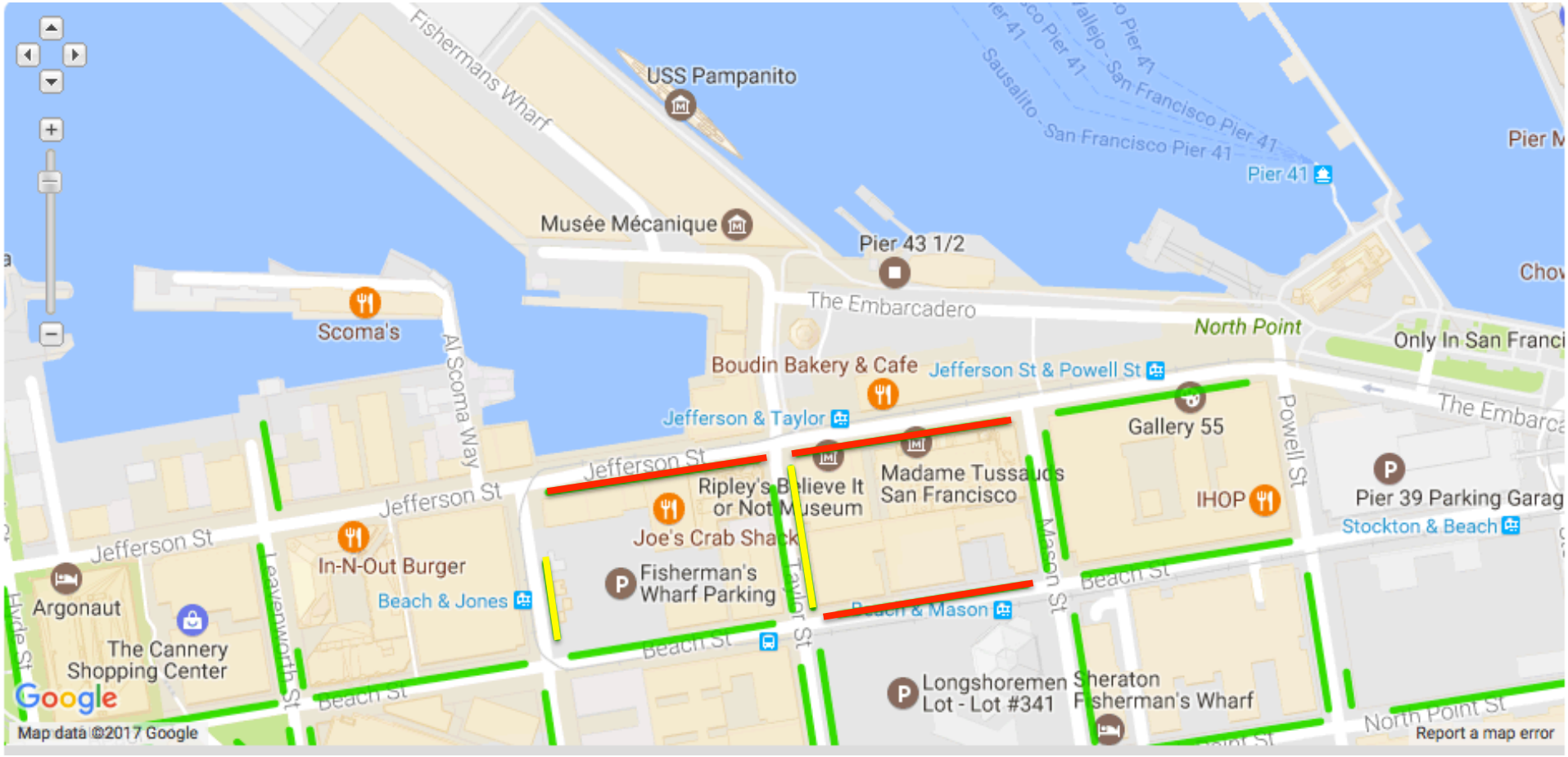


Image: sfpark.org

The parking space could be modeled with a complicated if-then structure

Simplified automobile parking

		Occupancy	
		Vacant	**Occupied**
Payment status	**Not Paid**	Vacant Not paid	Occupied Not paid
	Paid	Vacant Paid	Occupied Paid

```
void handleEvent(Event e) {
    if (event=="Payment") {
        if (occupancy=="Occupied" && payment=="Not Paid") {
            //set time on meter
        elseif (occupancy="Occupied" && payment=="Paid") {
            //increment time on meter
        ...
```

Error prone and inflexible

Handle every event, from every state

Combination of occupancy and payments leads to four States for each space

Simplified automobile parking

Four possible States

Four Events: Arrival/Departure Payment/Expiration

Start at vacant and not paid

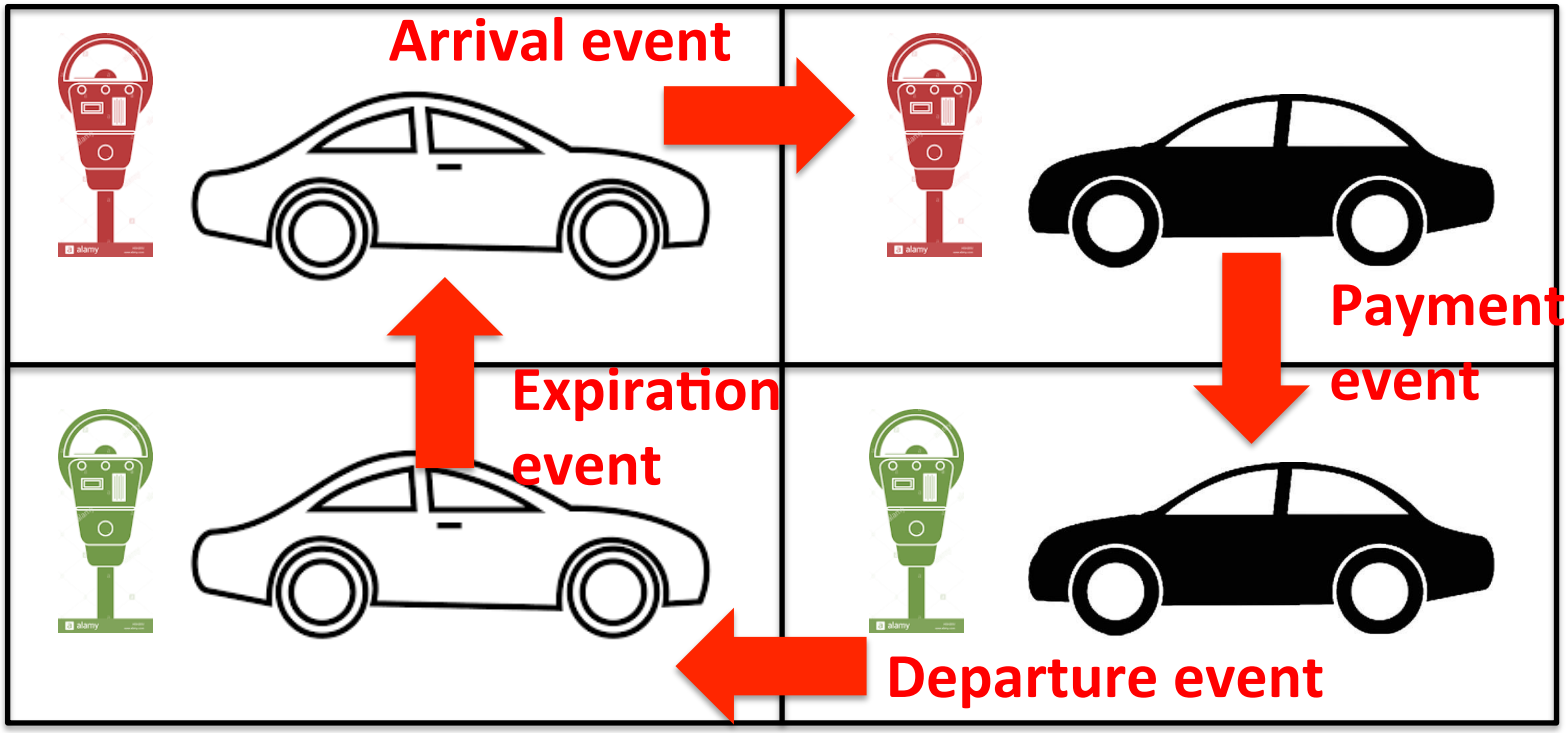
Occupied Status

Vacant | **Occupied**

Paid Status

Not Paid

Paid

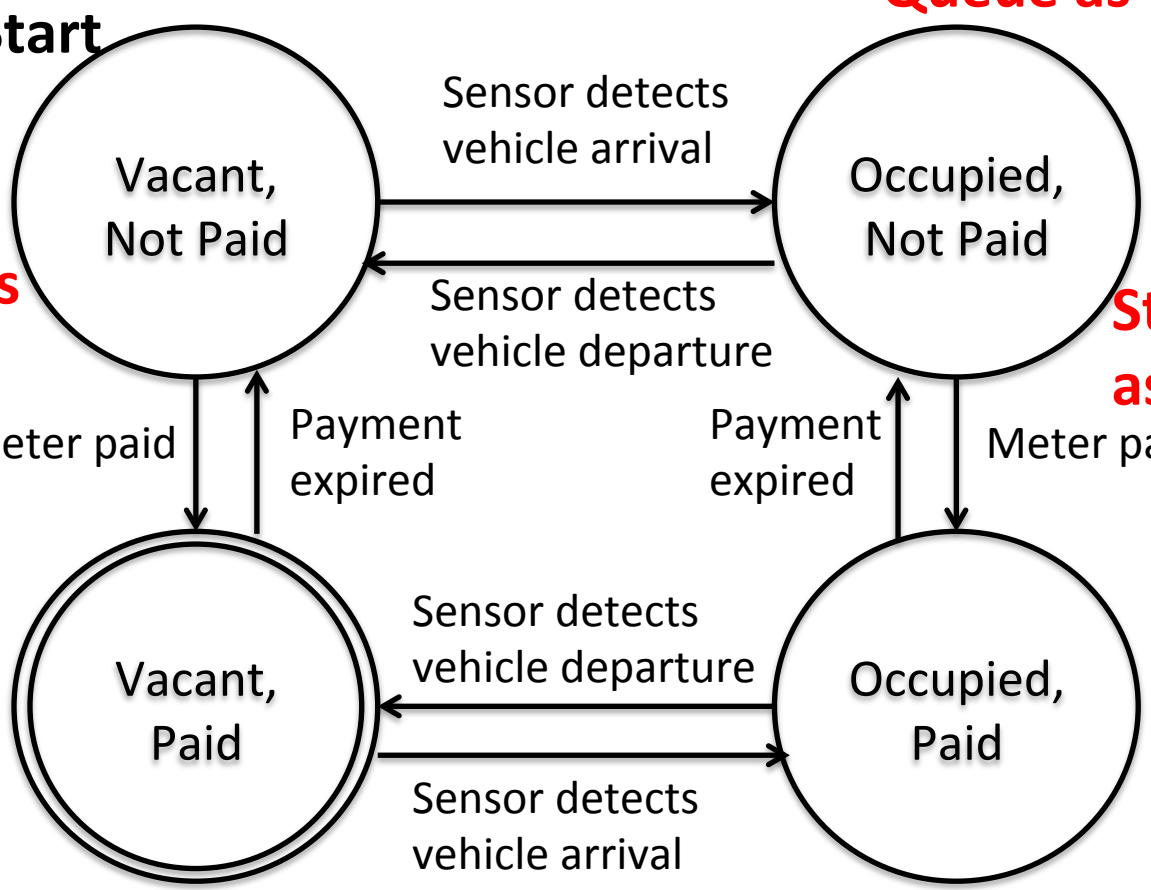


Events cause the system to transition between States

The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking

Events processed from Queue as they occur



Model four States as FSM vertices

States transition as Events happen

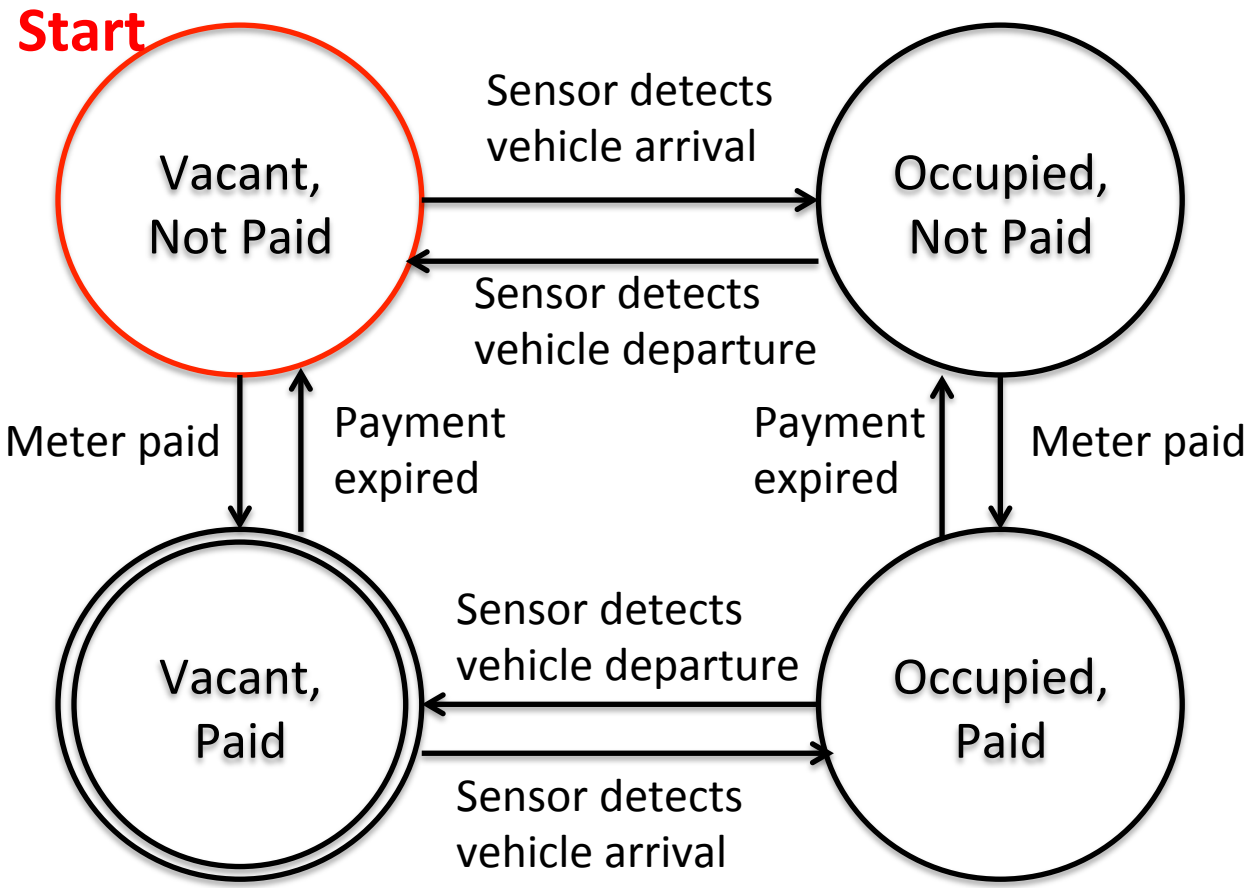
Model the Transition from each State for each Event (self loops not shown)

Current State is combination of paid status and occupancy

The parking space could be modeled more simply with a Finite Automata

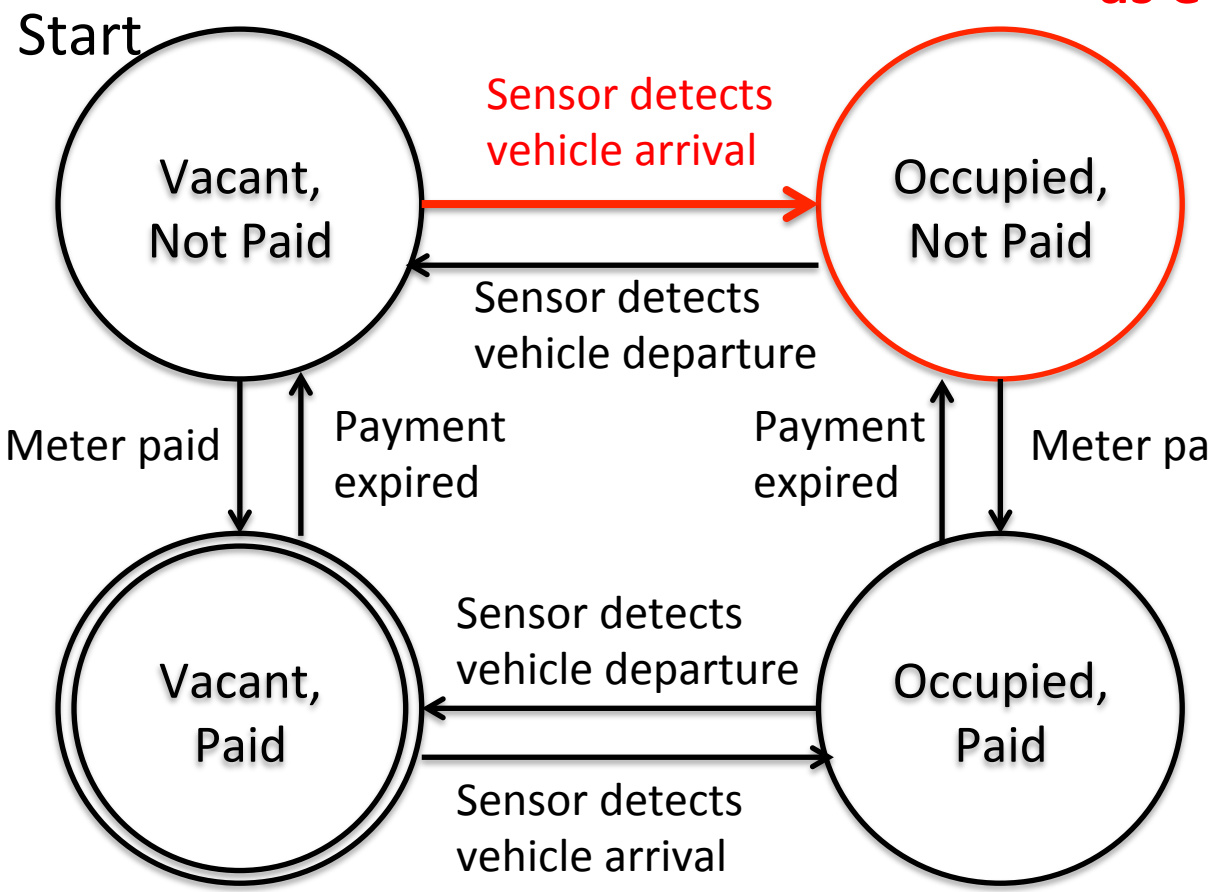
Simplified automobile parking

States transition as events happen



The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking

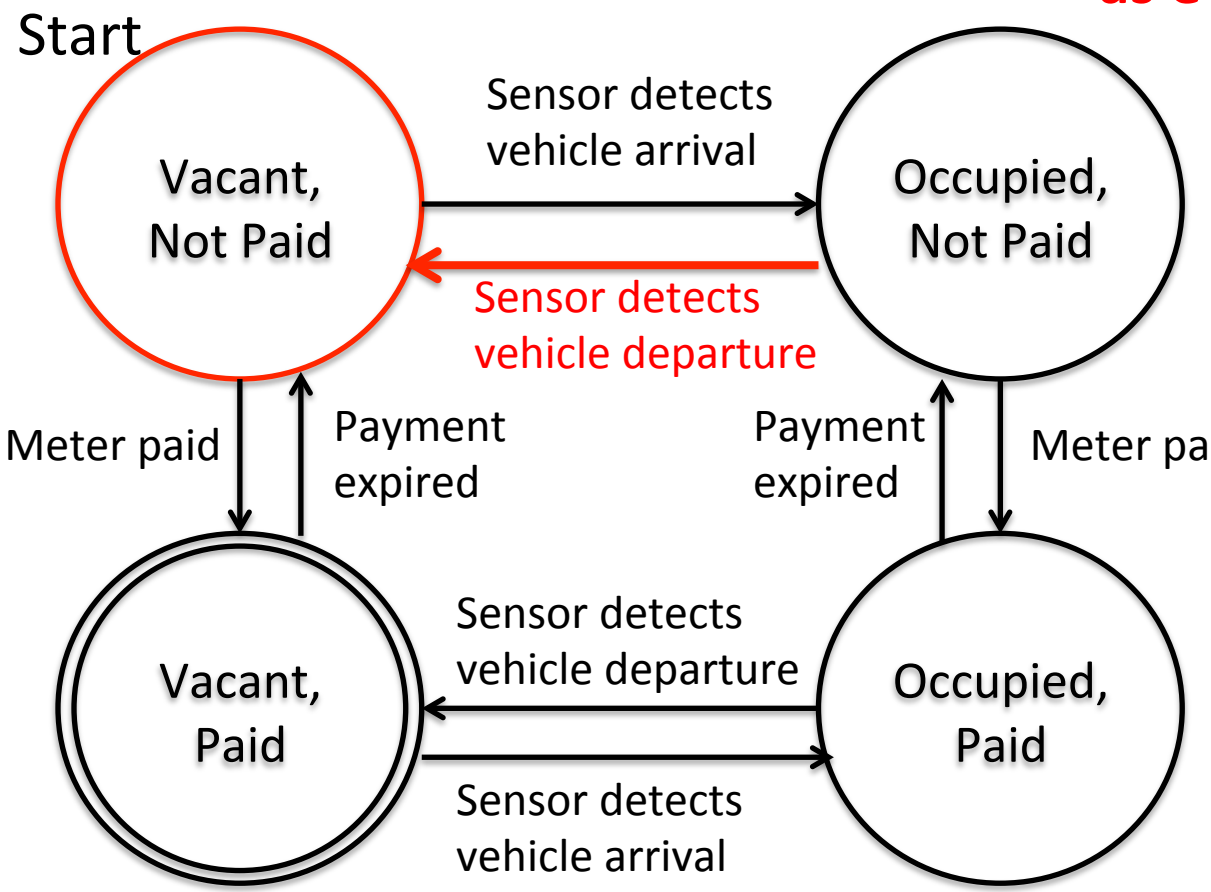


States transition as events happen

- **Sensor detects arrival**
- **Transition to Occupied Not Paid**

The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking

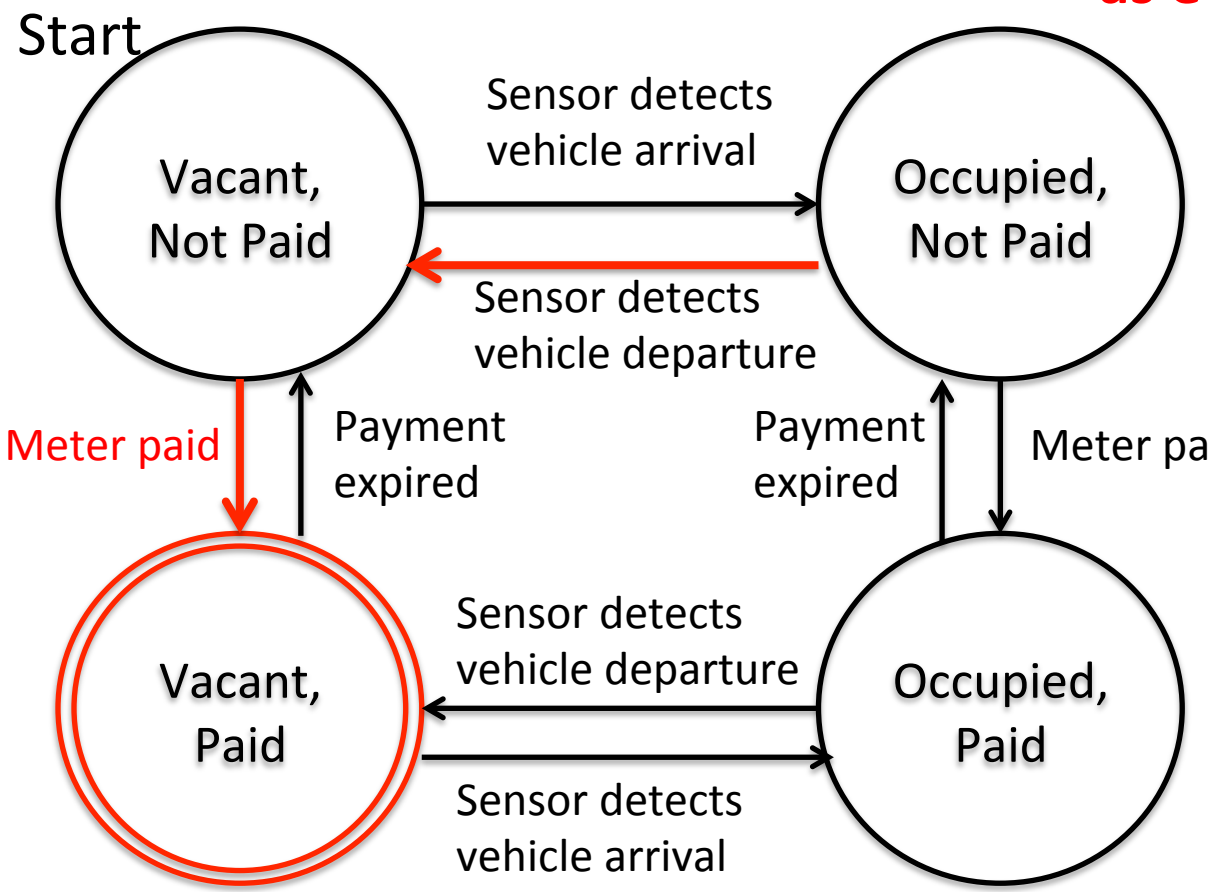


States transition as events happen

- **Sensor detects arrival**
- **Transition to Occupied Not Paid**
- **Sensor detects departure**
- **Transition to Vacant Not Paid**

The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking

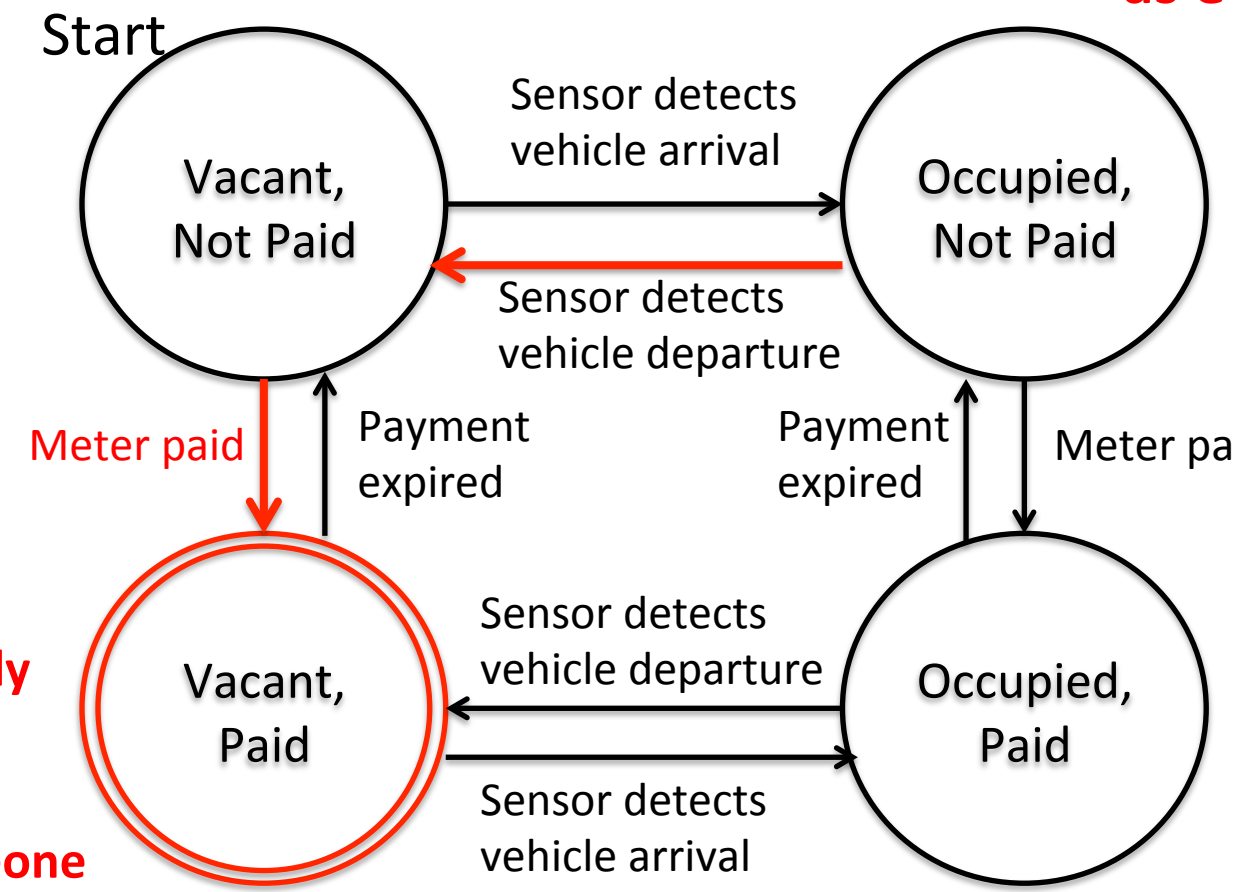


States transition as events happen

- **Sensor detects arrival**
- **Transition to Occupied Not Paid**
- **Sensor detects departure**
- **Transition to Vacant Not Paid**
- **Meter paid, but no arrival**

The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking



States transition as events happen

- **Sensor detects arrival**
- **Transition to Occupied Not Paid**
- **Sensor detects departure**
- **Transition to Vacant Not Paid**
- **Meter paid, but no arrival**

Sensor probably erroneously detected departure, send someone to figure out why!

Tracking the State of each space allows San Francisco to monitor city-wide parking

Fisherman's Wharf in San Francisco, CA

Green < 75% occupied, yellow = 75-90% occupied, red > 90% occupied

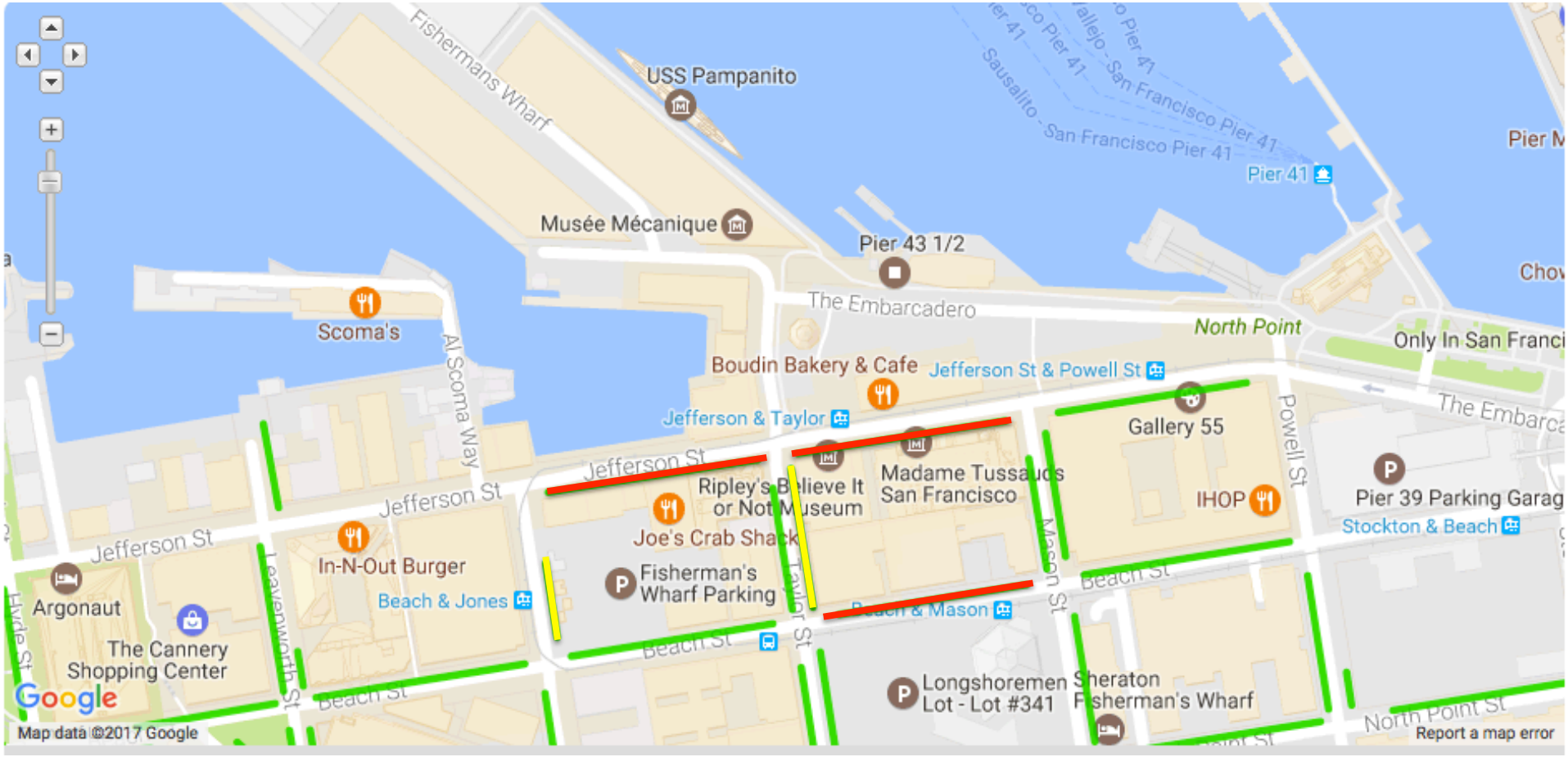


Image: sfpark.org