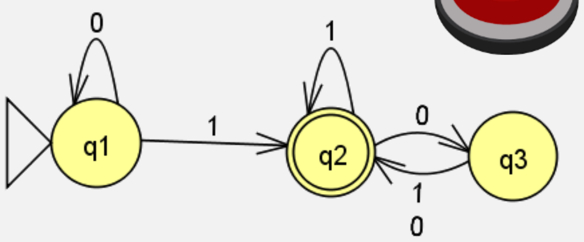
CS 420

Computing With DFAs

Monday, February 5, 2024

UMass Boston Computer Science



Announcements

- HW 1
  - <u>Due</u>: ~~Wed 2/7~~ Mon 2/12 12pm (noon)

- TAs and (new!) office hours

Office hours will be held weekly **in-person**, in McCormack, 3rd Floor, at these times:

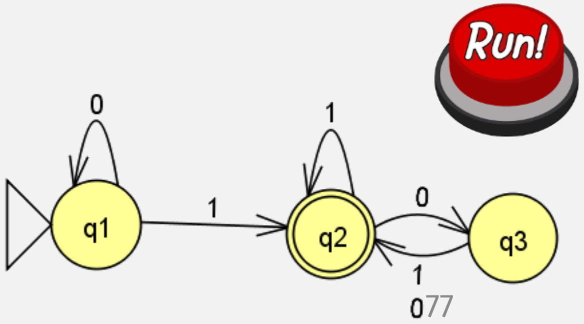
- Thu 2:00-3:30pm EST (Jean Gerard), room 0139
- Thu 3:30-5:00pm EST (Richard Chang), room 0139
- Fri 2:00-3:30pm EST (Prof Chang), room 0201-03

Office hours will be held weekly **via Zoom** during these times:

- Thu 3:30-5:00pm EST (Prof Chang) (see Blackboard for Zoom link)
- Sat 12:00-1:30pm EST (Anna Bosunova) (see Blackboard for Zoom link)

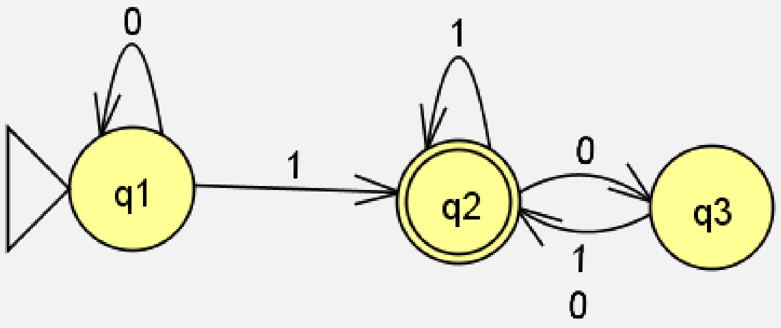
Drop-ins are fine, but emailing in advance if you can would be helpful.

These will usually be group meetings, but one-on-ones are available upon request.



Previously

Computation with DFAs (JFLAP demo)

- DFA: 

- Input: “`1101`”

HINT: always work out concrete examples to understand how a machine works

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and <u>Input</u> = **string of chars**, e.g. "`1101`"

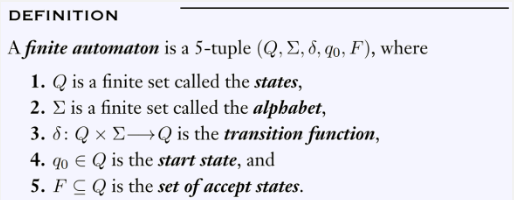
To **run** the automata / "program":

- <u>Start</u> in **"start state"**

- <u>Repeat</u>**:**
  - <u>Read</u> **1 char** from **input;**
  - <u>Change</u> **state** according to the <u>transition</u> **table**

- <u>Result</u> **of computation =**
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

DFA Computation Rules


DEFINITION

A *finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Informally	*Formally (i.e., mathematically)*

Given

- A **DFA** (~ a "Program") → • $M =$
- and <u>Input</u> = **string of chars**, e.g. "`1101`" → • $w =$

To **run** the automata / "program":

- <u>Start</u> in **"start state"**
- <u>Repeat</u>:
  - <u>Read</u> **1 char** from **input;**
  - <u>Change</u> **state** according to the <u>**transition**</u> **table**
- <u>Result</u> **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "`1101`"

To **run** the automata / "program":

- Start in "start state"
- Repeat:
  - Read **1 char** from **input**;
  - Change **state** according to the **transition table**
- Result of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally *(i.e., mathematically)*

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$, the sequence of states in the **computation**, where:

- $r_0 = q_0$
- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists . . . with $r_n \in F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

Informally

Given
- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "`1101`"

To **run** the automata / "program":
- Start in **"start state"**

- **Repeat:**
  - **Read 1 char** from **input;**
  - **Change state** according to the **transition table**

- Result of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \boxed{\delta,} q_0, F)$
- $w = \boxed{w_1} w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$,
the **sequence of states** in the **computation**, where:

- $\boxed{r_0} = q_0$

- $r_i =$

if i=1, $r_1 = \delta(r_0, w_1)$
if i=2, $r_2 = \delta(r_1, w_2)$
if i=3, $r_3 = \delta(r_2, w_3)$

- M ***accepts*** w if
  sequence of states $r_0, r_1, \ldots, r_n$ in Q exists …
  with $r_n \in F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

Informally	*Formally (i.e., mathematically)*

Given
- A **DFA** (~ a "Program")
- and <u>Input</u> = **string of chars**, e.g. "1101"

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$,
the <u>**sequence of states**</u> in the **computation**, where:

To **run** the automata / "program":
- <u>Start</u> in **"start state"**

- $r_0 = q_0$

- <u>**Repeat:**</u>
  - <u>**Read**</u> **1 char** from **input;**
  - <u>**Change**</u> **state** according to the <u>**transition**</u> **table**

- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

- <u>Result</u> of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

- M ***accepts*** w if
  sequence of states $r_0, r_1, \ldots, r_n$ in Q exists ...
  with $r_n \in F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

Informally

Given
- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "1101"

To **run** the automata / "program":
- Start in **"start state"**
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the **transition table**
- Result **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally *(i.e., mathematically)*

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

This is still a little "informal"

A **run** is represented by **variables** $r_0, \ldots, r_n$,
the **sequence of states** in the **computation,** where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if
  sequence of states $r_0, r_1, \ldots, r_n$ in Q exists ...
  with $r_n \in F$

This is still a little "informal"

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

An Extended Transition Function

set of pairs

* = "0 or more"

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \to Q$

- Domain:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range:
  - **Output state** (doesn't have to be an accept state)

Σ^* = set of all possible strings!

(Defined recursively)

- **Base case:** ...

Interlude: Recursive Definitions

```
function factorial( n )
{
  if ( n == 0 )
       return 1;
  else
       return n * factorial( n - 1 );
}
```

Base case

Recursive case

Function is called before it is fully defined!

Recursive call with “smaller” argument

- Why is this <u>allowed</u>?
  - It’s a “**feature**” (i.e., an **axiom!**) of the **programming language**
- Why does this “<u>work</u>”? (Why doesn’t it loop forever?)
  - Because the **recursive call always has** a **“smaller” argument** …
  - … and so **eventually reaches** the **base case** and **stops**

Recursive Definitions

A **Natural Number** is either:

Use of definition before it is fully defined!

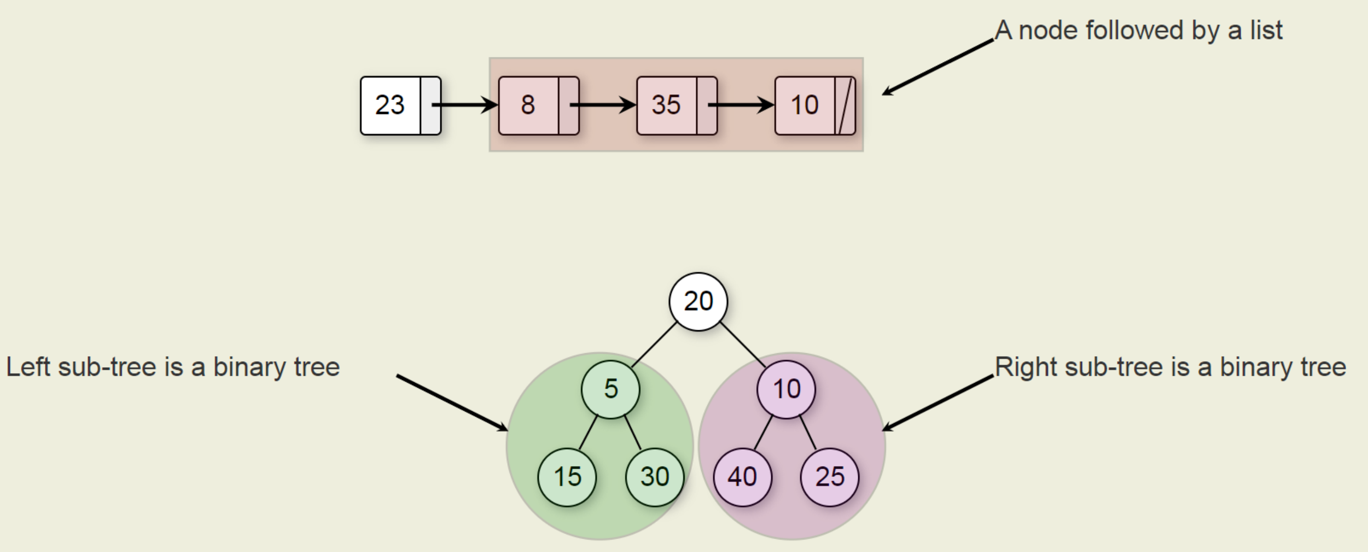
Base case → • **Zero**, or

Recursive case → • the **Successor** of a **Natural Number** ← “smaller” argument

<u>Examples</u>

- **Zero**
- **Successor** of **Zero** (= “one”)
- **Successor** of **Successor** of **Zero** (= “two”)
- **Successor** of **Successor** of **Successor** of **Zero** (= “three”) …

Recursive Data Definitions



Recursive definitions have:
- <u>base case</u> and
- <u>recursive case</u>
  (with a "smaller" object)

```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

This is a <u>recursive definition:</u> **Node is used before it is fully defined** (but must be "smaller")

Strings Are Defined Recursively

Base case

Recursive case

A **String** is either:

- the **empty string** (ε), or
- xa (non-empty string) where
  - x is a **string**
  - a is a "**char**" in Σ

"smaller" argument

Remember: **all strings** are **formed with "chars"** from some **alphabet** set Σ

Σ^* = set of all possible strings!

Recursive Data ⇒ Recursive Functions

A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case

```
function factorial( n )
{
  if ( n == 0 )
      return 1;
  else
      return n * factorial( n - 1 );
}
```

Recursive case must have "smaller" argument

Recursive functions are **recursive** because ... its **input data** is **recursively defined!**

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- <u>Domain</u>:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - **Output state** (doesn't have to be an accept state)

Recursive Input Data
needs
Recursive Function

(Defined recursively)

Base case

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

- Base case $\hat{\delta}(q, \varepsilon) =$

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- <u>Domain</u>:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - **Output state** (doesn't have to be an accept state)

Recursive Input Data
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "char" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$

Recursive call

"smaller" argument

Recursive case

string

char

- Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \to Q$

- <u>Domain</u>:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - **Output state** (doesn't have to be an accept state)

Recursive Input Data needs Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$
- Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

Previously

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "1101"

To **run** the automata / "program":

- Start in "**start state**"
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- Result of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** r_0, ..., r_n ,
the sequence of states in the **computation,** where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists . . . with $r_n \in F$

This is still a little "informal"

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "1101"

To **run** the automata / "program":

- Start in **"start state"**
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- Result **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** r_0**, …,** r_n **,**
the sequence of states in the **computation,** where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if $\hat{\delta}(q_0, w) \in F$

sequence of states $r_0, r_1, \ldots, r_n$ in Q exists …
with $r_n \in F$

Definition of Accepting Computations

An **accepting computation**, for DFA $M = (Q, \Sigma, \delta, q_0, F)$ and **string** w:

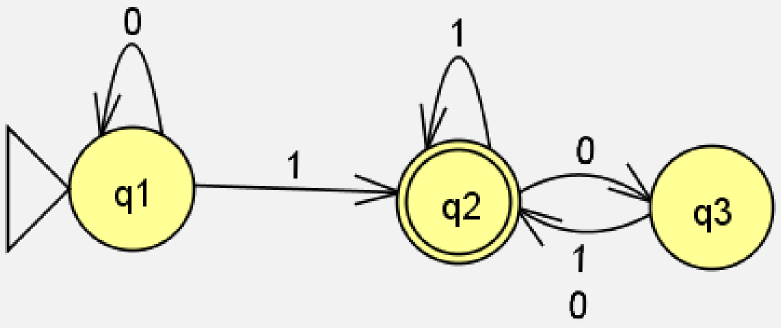
1. **starts** in the **start state** q_0
2. **goes through** a **valid sequence** of **states according to** δ
3. **ends** in an **accept state**

<u>All 3</u> **must be true** for a **computation**
to be an **accepting computation!**

M ***accepts*** w if $\hat{\delta}(q_0, w) \in F$

Accepting Computation or Not?

DFA:



- $\hat{\delta}$(q1, **`1101`**)
  - Yes
- $\hat{\delta}$(q1, **`110`**)
  - **No** (doesn't end in accept state)
- $\hat{\delta}$(q2, **`101`**)
  - **No** (doesn't start in start state)

Alphabets, Strings, Languages

Alphabet specifies "all possible strings"

- An **alphabet** is a non-empty finite set of symbols

(impossible to have strings with non-alphabet chars)

$$\Sigma_1 = \{0,1\}$$

$$\Sigma_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$$

- A **string** is a finite sequence of **symbols** from an **alphabet**

01001   abracadabra ε ← Empty string (length 0)

- A **language** is a set of **strings**

Languages can be infinite

$$A = \{\text{good}, \text{bad}\}$$

$\emptyset$ $\{\ \}$

Empty set is a language

$$A = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}$$

"the set of all …"   "such that …"

Computation and Languages

- The **language** of a **machine** is the **set of all strings** that it **accepts**

- E.g., A **DFA** M ***accepts*** w if $\hat{\delta}(q_0, w) \in F$

- **Language** of M = $L(M)$ = $\{w \mid M \text{ accepts } w\}$

"the set of all …"

"such that …"

Machine and Language Terminology

DFA M ***accepts*** w ← string

M ***recognizes language*** A ← Set of strings

if $A = \{w |\ M \text{ accepts } w\}$

Computation and Classes of Languages

- The **language** of a **machine** = <u>**set** of **all strings**</u> that it **accepts**
  - E.g., every **DFA** is **associated with** a **language**
- A **computation model** = <u>**set** of **machines**</u> it **defines**
  - E.g., **all possible DFAs** are a **computation model**
- Thus: a **computation model** = <u>**set** of **languages**</u>

Regular Languages: Definition

If a **deterministic finite automata** (**DFA**) <u>recognizes</u> a language,
then **that language** is called a **regular language.**

A ***language*** is a set of strings.

M ***recognizes language*** A
if $A = \{w |\ M \text{ accepts } w\}$

A Language, Regular or Not?

- If given: a DFA M
  - We know: $L(M)$, the language recognized by M, is a **regular language**

If a DFA <u>recognizes</u> a language,
then **that language** is called a **regular language**.

(modus ponens)

- If given: a Language A
  - Is A a regular language?
    - Not necessarily!
  - How do we determine, i.e., *prove*, that A is a regular language?

An Inference Rule: Modus Ponens

Premises

- If P then Q
- P is true

Conclusion

- Q must also be true

Example **Premises**

- If there is an DFA recognizing language A, then A is a regular language
- There is an DFA M where $L(M) = A$

Conclusion

- A is a regular language!

A Language, Regular or Not?

- If given: a DFA M
  - We know: $L(M)$, the language recognized by M, is a regular language

If a DFA <u>recognizes</u> a language,
then that language is called a **regular language**.

- If given: a Language A
  - Is A a regular language?
    - Not necessarily!
  - How do we determine, i.e., *prove*, that A is a regular language?

Prove there is a DFA recognizing A!

Language: strs with odd # **1**s

HINT: always work out concrete examples to understand a language

Example	In the language?
1	Yes
0	No
01	Yes
11	No
1101	Yes
ε	no

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

If a DFA recognizes a language,
then that language is called a **regular language**.

How to prove the language is regular?

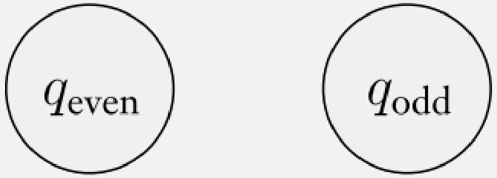
Prove there's a DFA recognizing it!

Designing Finite Automata: Tips

- Input is read only once, one char at a time
- Must decide accept/reject after that
- States = the machine's **memory**!
  - # states must be decided in advance
  - Think about what information must be remembered.
- Every state/symbol pair must have a transition (for DFAs)
- Come up with examples!

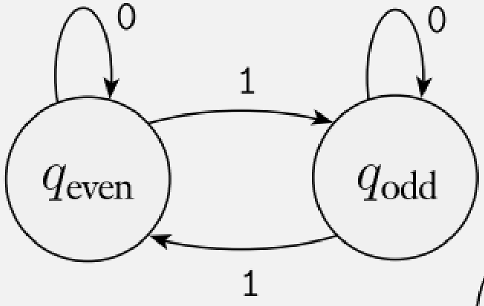
Design a DFA: accept strs with odd # **1**s

- <u>States</u>:
  - 2 states:
    - seen even **1**s so far
    - seen odds **1**s so far

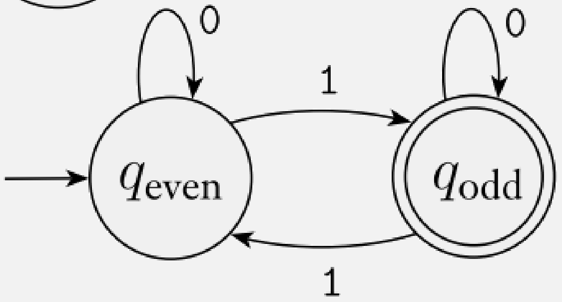


- <u>Alphabet</u>: `0` and `1`

- <u>Transitions</u>:



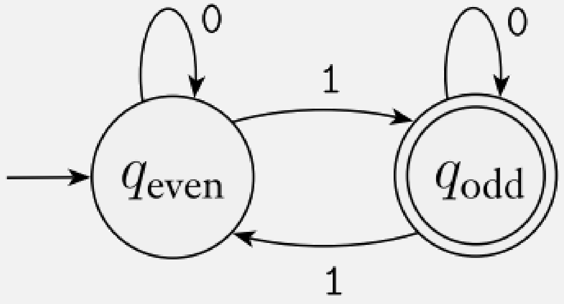
- <u>Start</u> / <u>Accept</u> states:



“Prove” that DFA recognizes a language

Example	In the language?
1	Yes
0	No
01	Yes
11	No
1101	Yes
ε	no

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



In this class, **a table like this is sufficient to “prove” that a DFA recognizes a language**

Submit 2/5 in-class work to gradescope