Announcements

- **HW 1**
  - **Due:** 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.
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 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
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

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

- $M =$
- $w =$

**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. $\Sigma$ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.
DFA Computation Rules

<table>
<thead>
<tr>
<th>Informally</th>
<th>Formally (i.e., mathematically)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given</td>
<td>$M = (Q, \Sigma, \delta, q_0, F)$</td>
</tr>
<tr>
<td>• A DFA (~ a “Program”)</td>
<td>• $w = w_1 w_2 \cdots w_n$</td>
</tr>
<tr>
<td>• and Input = string of chars, e.g. “1101”</td>
<td>A run is represented by variables $r_0, \ldots, r_n$,</td>
</tr>
<tr>
<td></td>
<td>the sequence of states in the computation, where:</td>
</tr>
<tr>
<td>To run the automata / “program”:</td>
<td>• $r_0 = q_0$</td>
</tr>
<tr>
<td>• Start in “start state”</td>
<td>$M$ accepts $w$ if</td>
</tr>
<tr>
<td>• Repeat:</td>
<td>sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists \ldots</td>
</tr>
<tr>
<td>• Read 1 char from input;</td>
<td>with $r_n \in F$ [81]</td>
</tr>
<tr>
<td>• Change state according to the transition table</td>
<td>• $M$ accepts $w$ if</td>
</tr>
<tr>
<td>• Result of computation =</td>
<td>sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists \ldots</td>
</tr>
<tr>
<td>• Accept if last state is Accept state</td>
<td>with $r_n \in F$ [81]</td>
</tr>
<tr>
<td>• Reject otherwise</td>
<td>$M$ accepts $w$ if</td>
</tr>
</tbody>
</table>


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:

• $r_0 = q_0$

• $r_i =$
  • 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 $r_n \in F$
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:
- \( r_0 = q_0 \)
- \( r_i = \delta(r_{i-1}, w_i) \), for \( i = 1, \ldots, n \)

\[ M \text{ accepts } w \text{ if } \]
- sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists . . .
  with \( r_n \in F \)
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)**

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1w_2 \cdots w_n \)

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

- \( r_0 = q_0 \)

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

- \( M \) accepts \( w \) if the sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists ... with \( r_n \in F \)
An Extended Transition Function

Define **extended transition function:**

- **Domain:**
  - Input state \( q \in Q \) (doesn’t have to be start state)
  - Input string \( w = w_1w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range:**
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- **Base case:** ...

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

\[ \hat{\delta} : Q \times \Sigma^* \longrightarrow Q \]

\( \Sigma^* = \text{set of all possible strings!} \)

\( * = \text{“0 or more”} \)

set of pairs
Interlude: Recursive Definitions

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

- **Why is this allowed?**
  - It’s a “feature” (i.e., an axiom!) of the programming language

- **Why does this “work”?** (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:
- Zero, or
- the **Successor** of a **Natural Number**

Examples
- 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:
- base case and
- recursive case
  (with a “smaller” object)

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

This is a recursive definition: Node is used before it is fully defined (but must be “smaller”)
Strings Are Defined Recursively

A String is either:
- the **empty string** ($\varepsilon$), or
- $xa$ (non-empty string) where
  - $x$ is a **string**
  - $a$ is a “char” in $\Sigma$

**Base case**

**Recursive case**

**Remember**: all strings are formed with “chars” from some **alphabet** set $\Sigma$

$\Sigma^* = \text{set of all possible strings!}$
Recursive Data ⇒ Recursive Functions

A Natural Number is either:
• Zero, or
• the Successor of a Natural Number

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

- **Base case** if \( n = 0 \)
- **Recursive case** otherwise

Recursive functions are recursive because its input data is recursively defined!
An Extended Transition Function

Define **extended transition function:**

- **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)

(Defined recursively)

- Base case: \( \hat{\delta}(q, \varepsilon) = \)
An Extended Transition Function

Define **extended transition function:**

- **Domain:**
  - Input state $q \in Q$ (doesn’t have to be start state)
  - Input string $w = w_1w_2 \cdots w_n$ where $w_i \in \Sigma$
- **Range:**
  - Output state (doesn’t have to be an accept state)

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

A String is either:
- the **empty string** ($\varepsilon$), or
- $xa$ (non-empty string) where
  - $x$ is a **string**
  - $a$ is a “char” in $\Sigma$
An Extended Transition Function

Define **extended transition function**: 

- **Domain:**
  - **Input state** \( q \in Q \) (doesn’t have to be start state)
  - **Input string** \( w = w_1w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range:**
  - **Output state** (doesn’t have to be an accept state)

(Defined recursively)

- **Base case** \( \hat{\delta}(q, \varepsilon) = q \)
- **Recursive Case** 
  \[
  \hat{\delta}(q, w_1 \cdots w_n) = \delta(\hat{\delta}(q, w_1 \cdots w_{n-1}), w_n)
  \]

\( \hat{\delta} : Q \times \Sigma^* \to Q \)

\( \delta : Q \times \Sigma \to Q \) 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)**

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

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

- \( r_0 = q_0 \)

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

- \( M \) accepts \( w \) if sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists such that \( r_n \in F \).
DFA Computation Rules

<table>
<thead>
<tr>
<th>Informally</th>
<th>Formally (i.e., mathematically)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given</td>
<td>$M = (Q, \Sigma, \delta, q_0, F)$</td>
</tr>
<tr>
<td>A DFA (~ a “Program”)</td>
<td>$w = w_1w_2 \cdots w_n$</td>
</tr>
<tr>
<td>and Input = string of chars, e.g. “1101”</td>
<td>A run is represented by variables $r_0, \ldots, r_n$, the sequence of states in the computation, where:</td>
</tr>
<tr>
<td>To run the automata / “program”:</td>
<td>$r_0 = q_0$</td>
</tr>
<tr>
<td>Start in “start state”</td>
<td>$r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$</td>
</tr>
<tr>
<td>Repeat:</td>
<td>$M$ accepts $w$ if $\hat{\delta}(q_0, w) \in F$</td>
</tr>
<tr>
<td>• Read 1 char from input;</td>
<td>sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists with $r_n \in F$</td>
</tr>
<tr>
<td>• Change state according to the transition table</td>
<td>$100$</td>
</tr>
<tr>
<td>Result of computation =</td>
<td></td>
</tr>
<tr>
<td>• Accept if last state is Accept state</td>
<td></td>
</tr>
<tr>
<td>• Reject otherwise</td>
<td></td>
</tr>
</tbody>
</table>
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 $\delta$

3. ends in an accept state

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

All 3 must be true for a computation to be an accepting computation!
Accepting Computation or Not?

DFA:

\[ \hat{\delta}(q_1, 1101) \]
- Yes

\[ \hat{\delta}(q_1, 110) \]
- No (doesn’t end in accept state)

\[ \hat{\delta}(q_2, 101) \]
- No (doesn’t start in start state)
Alphabets, Strings, Languages

• **An alphabet** is a **non-empty finite set** of symbols
  \[ \Sigma_1 = \{0,1\} \]
  \[ \Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\} \]

• A **string** is a **finite sequence** of symbols from an alphabet
  
  01001       abracadabra       \( \varepsilon \)

• A **language** is a **set** of strings
  
  \[ A = \{ \text{good}, \text{bad}\} \]
  \[ \emptyset \{ \} \]

  Empty set is a language

  Languages can be infinite

  “the set of all ...”

  “such that ...”

Alphabet specifies “all possible strings”

(impossible to have strings with non-alphabet chars)
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 $\delta(q_0, w) \in F$

• Language of $M = L(M) = \{w | M$ accepts $w\}$
Machine and Language Terminology

DFA \( M \) accepts \( w \) \[\text{string}\]

\( M \) recognizes language \( A \) \[\text{Set of strings}\]

If \( A = \{ w \mid M \text{ accepts } w \} \)
Computation and Classes of Languages

• The **language** of a machine = **set of all strings** that it accepts
  
  • E.g., every DFA is associated with a language

• A **computation model** = **set of machines** it defines
  
  • E.g., all possible DFAs are a computation model

• Thus: a **computation model** = **set of languages**
Regular Languages: Definition

If a **deterministic finite automata (DFA)** recognizes a language, then that language is called 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 recognizes 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 recognizes 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 # 1s

<table>
<thead>
<tr>
<th>Example</th>
<th>In the language?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>01</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>No</td>
</tr>
<tr>
<td>1101</td>
<td>Yes</td>
</tr>
<tr>
<td>ε</td>
<td>no</td>
</tr>
</tbody>
</table>

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

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

HINT: always work out concrete examples to understand a 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 # 1s

• **States:**
  - 2 states:
    - seen even 1s so far
    - seen odds 1s so far

• **Alphabet:** 0 and 1

• **Transitions:**

• **Start / Accept states:**
“Prove” that DFA recognizes a language

<table>
<thead>
<tr>
<th>Example</th>
<th>In the language?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>01</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>No</td>
</tr>
<tr>
<td>1101</td>
<td>Yes</td>
</tr>
<tr>
<td>ε</td>
<td>no</td>
</tr>
</tbody>
</table>

\[ \Sigma = \{0, 1\} \]
Submit 2/5 in-class work to gradescope