Machines

- By a **machine** over an alphabet $\Sigma$, we mean
  - a collection of **states** $Q$
    together with
  - a **transition relation** $\rightarrow \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$
    - $q - \sigma \rightarrow q'$ means that $(q, \sigma, q')$ is in the relation
  - an **initial state** $q_0 \in Q$
  - and a set of **accepting states** $F \subseteq Q$. 


Behavior of a Machine

- A machine \textit{starts} in state \( q_0 \).

- From a current state \( q \) it can \textit{change state} to \( q' \) \textit{with input} \( \sigma \) provided that \( q - \sigma \rightarrow q' \).

- From a current state \( q \) it can \textit{change state} to \( q' \) \textit{spontaneously} provided that \( q - \varepsilon \rightarrow q' \).

- The machine \textit{accepts} a string \( x \in \Sigma^* \) provided there is \textit{some} path from \( q_0 \) to \textit{some} \( q \in F \) that \textit{spells out} \( x \).
Example of a Machine

Accepted: 0 0 1 0 0 1

Not accepted: 1 0 1 0
Another Example of a Machine

Accepted: $1^p$ where $p$ is prime

Not accepted: $1^q$ where $q$ is composite
Languages for Machines

- If $M$ is a machine, then $L(M)$ is the **language accepted by $M$**, defined as the set of finite strings spelled out by all paths from the initial state to some accepting state.
The Language of a Machine *State*

- If q is a state, then the language $L_q$ is defined to be the set of strings spelled out in going from q to some accepting state.

- Hence the language for the initial state is the language for the machine.
Equivalence of States

Two states are defined to be **equivalent**

$q \equiv q'$

iff their languages are equal:

$L_q = L_{q'}$. 
Equivalence Relations Review

\[\equiv\] is an equivalence relation, meaning:

- \(\forall q \in Q ( q \equiv q )\) [We drop the \(\in Q\) for brevity below.]

- \(\forall q \forall q' ( q \equiv q' \rightarrow q' \equiv q )\)

- \(\forall q \forall q' \forall q'' ( ( q \equiv q' \land q' \equiv q'' ) \rightarrow q \equiv q'' )\)
Partitions

- Every equivalence relation determines a partition on $Q$.

- A partition is a set of subsets of $Q$ such that:
  - No two subsets intersect.
  - The union of the subsets is all of $Q$.

- The partition determined by $\equiv$ is given by
  $\{\{q' \mid q' \equiv q\} \mid q \in Q\}$.

- The elements of the partition, sets of the form $\{q' \mid q' \equiv q\}$ are called the **equivalence classes** of $\equiv$. 
Partition Determines Relation

- Every partition determines an equivalence relation:

  If $P$ is a partition, then define: $q' \equiv q$ iff $\exists S \in P$ ($q \in S$ and $q' \in S$).

- Verify that the 3 equivalence relation properties hold.
Rank of a Partition

- The rank of a partition is just the number of equivalence classes.

- If the state set is finite, the rank of the corresponding equivalence partition is also finite.

- If the state set is infinite, the rank could still be finite.
Machines for Languages

- A language $L \subseteq \Sigma^*$ can be viewed as a machine:
  - The states are elements of $\Sigma^*$.
  - The initial state is $\varepsilon$.
  - The transitions are $x - \sigma \to x\sigma$.
  - The accepting states are the elements of $L$.

- As with any machine, there are languages $L_x$ for each $x$ in $\Sigma^*$. Incidentally, $L_x = \{ z \mid xz \in L \}$
Equivalence of Strings modulo a Language

- With respect to a language \( L \), two strings are equivalent

\[ x \equiv_L y \]

iff the languages of their corresponding states are equal:

\[ L_x = \{ z \mid xz \in L \} = \{ z \mid yz \in L \} = L_y \]
Congruence

- The relation $\equiv_L$ has the additional property of being a **congruence**:

  $$x \equiv_L y \text{ implies } \forall z \in \Sigma^* (xz \equiv_L yz).$$

- By induction, a necessary and sufficient condition for $\equiv_L$ to be a congruence is:

  $$x \equiv_L y \text{ implies } \forall \sigma \in \Sigma (xz \equiv_L yz).$$
Congruence Pictured

Equivalence classes

Applying the same input sequence to congruent states yields congruent states.

Partition of the congruence
Finite-State Machines (FSMs)

- A machine is finite-state iff its state set is finite.
Determinism

- A machine is deterministic iff:
  - There are no spontaneous state changes, and
  - For each $q, q', q'' \in Q$ and $\sigma \in \Sigma$
    
    \[
    \text{if } q - \sigma \rightarrow q' \text{ and } q - \sigma \rightarrow q'', \text{ then } q' = q''.
    \]

    In other words, there is a **partial function** $\delta: Q \times \Sigma \rightarrow Q$ such that
    
    \[
    q - \sigma \rightarrow q' \text{ iff } \delta(q, \sigma) = q'.
    \]

    “Partial” means that $\delta(q, \sigma)$ could be **undefined** for some pairs $(q, \sigma)$. 
DFA

- A finite-state machine that is also deterministic is called a DFA (for deterministic finite-state acceptor).
Finite-State Languages

- Say a language L is finite-state iff it is accepted by some DFA.

- The equivalence partition for a finite-state language is guaranteed to be finite rank.

- Conversely, if the equivalence partition for a language is finite rank, then there is a DFA that accepts that language.
Language in terms of Equivalence Classes

- Consider a finite-state language.
- Its equivalence partition must be finite-rank.
- The language itself must be the union of some of the equivalence classes.

\[ \sum^* \]

\[ V \cup Y \] could be the language
Myhill-Nerode Theorem

- $L \subseteq \Sigma^*$ is a finite-state language iff
- $L$ is the union of some equivalence classes of some congruence relation on $\Sigma^*$ of finite rank.
Proof of Myhill-Nerode

- Suppose \( L \) is a finite-state language.
- Let \( M \) be a DFA accepting \( L \).
- Let \( M \) be the a DFA.
- Let \( \delta \) be the state-transition function described earlier, i.e. \( \delta(q, \sigma) = q' \) means there is a transition from \( q \) to \( q' \) via letter \( \sigma \).

- Extend \( \delta: Q \times \Sigma \rightarrow Q \) to \( \delta^*: Q \times \Sigma^* \rightarrow Q \), as follows:
  - \( \forall q \in Q \exists q' \in Q \) \( \delta(q', \epsilon) = q \)
  - \( \forall q \in Q \forall x \in \Sigma^* \forall \sigma \in \Sigma \) \( \delta^*(q, \sigma x) = \delta^*(\delta(q, \sigma), x) \)

- Claim: \( x \equiv_L y \) iff \( \delta^*(q_0, x) \equiv \delta^*(q_0, y) \).
Lemma

∀z ∈ Σ* ∀x ∈ Σ*
∀q∈Q δ*(q, xz) = δ*(δ*(q, x), z)

Proof is by induction on x.

Basis x = ε: δ*(q, ε) = q and xz = z, so
δ*(q, xz) = δ*(q, z) = δ*(δ*(q, ε), z)
Lemma

- Induction step:

Assume $\forall q \in Q \; \delta^*(q, xz) = \delta^*(\delta^*(q, x), z)$.

Show $\forall q \in Q \; \forall \sigma \in \Sigma \; \delta^*(q, (\sigma x) z) = \delta^*(\delta^*(q, \sigma x), z)$.

But $\delta^*(q, (\sigma x) z)$

$= \delta^*(q, \sigma (xz))$ \hspace{1cm} \text{associativity of concat.}

$= \delta^*(\delta(q, \sigma), xz)$ \hspace{1cm} \text{definition of } \delta^*$

$= \delta^*(\delta^*(\delta(q, \sigma), x), z)$ \hspace{1cm} \text{induction hypothesis}

$= \delta^*(\delta^*(q, \sigma x), z)$ \hspace{1cm} \text{definition of } \delta^*$
Proof of Claim

- $x \equiv_L y$ iff (by definition of $\equiv_L$)
- $L_x = L_y$ iff
- $\forall z \in \Sigma^* \ (xz \in L \iff yz \in L)$ iff
- $\forall z \in \Sigma^* \ (\delta^*(q_0, xz) \in F \equiv \delta^*(q_0, yz) \in F)$, where $F$ is the set of accepting states of $M$ iff
- $\forall z \in \Sigma^*$
  - $(\delta^*(\delta^*(q_0, x), z) \in F \iff \delta^*(\delta^*(q_0, y), z) \in F)$ iff
- $\delta^*(q_0, x) \equiv \delta^*(q_0, y)$
Impact of Claim

- The claim shows that two strings are equivalent iff the states to which the machine is taken when reading those state are also equivalent.

- But the set of states is finite, so the set of equivalence classes, i.e. the partition, must be finite as well.

- Hence the partition on $\Sigma^*$ is also finite, it being in one-one correspondence with the partition on states.
Claim: If $\equiv_L$ has finite-rank, then there is a DFA accepting $L$.

- The DFA $M$ is simply constructed as follows:
  - The states of $M$ are the equivalence classes of $\equiv_L$.
  - Let $[x]$ denote the equivalence class of $x \in \Sigma^*$.
  - The initial state of $M$ is $[\varepsilon]$.
  - The accepting states are $[x]$ for $x \in L$.
  - The transitions are defined by the function:
    $$\forall x \in \Sigma^* \; \forall \sigma \in \Sigma \; \delta([x], \sigma) = [x\sigma]$$
  - $\delta$ is well defined because $\equiv_L$ is a congruence relation.
Automata

- Colloquially, an automaton (plural “automata”) is an autonomous device (such as a robot or wind-up toy).

- In CS, the term has a more specific meaning: that of an abstract mathematical machine that can perform a specific function.
Uses of Automata

- There are many uses, one of which is to specify algorithms for accepting languages.

- An automaton **accepts a language** if it can tell, for any given input string, whether or not the string is in the language.
Example: Compilers, etc.

- Every compiler contains an automaton, that tells whether or not the input string is well-formed, i.e. is in the language that it compiles.

- Every pattern search program is effectively an automaton for recognizing patterns.
Finite-State Automata
(FSA or DFA, they are the same)

- An automaton is finite-state if its behavior is representable by transitions between a states in a finite set, some of which are designated accepting and others not.

- Each automaton has a designated start state.
Examples of FSA

- An FSA capable of accepting exactly the strings ending with 1.

![Diagram of FSA accepting strings ending with 1]
Examples of FSA

- An FSA capable of accepting exactly the strings containing no two consecutive 1’s.
Thing to Check

- For each combination of a state and a symbol, there should be exactly one arrow leaving the state with that symbol.

- This is the “deterministic” (“D”) in DFA.

- If this property does not hold, better fix it; your automaton might be wrong.
Application

- One way to implement a search is to construct, perhaps on the fly, an automaton that accepts the corresponding language, then simulate the automaton on the given input.
Two Ways to Define Specific Languages

- Give an FSA that accepts the language.
- Give a regular expression for the language.
Remarkable Fact

- The preceding two ways are equivalent.

- Equivalent here means that the two methods define the same family of languages.
Application of this Theory

- Sometimes it’s easier to give an automaton for a language.
- Sometimes it’s easier to give a regular expression.
- It would be nice to be able to go from one to the other more-or-less freely.
Regular Expression from DFA

- Label the States
- Identify each state with the set of paths from the start state to it. This set is a language.
- The language accepted by the DFA is the union of the paths to each of the accepting states, in this case $L \cup M$. 
Deriving Closed Forms

- View the acceptor as a set of regular-expression equations:
  - $L = L0 \cup M0 \cup \varepsilon$
  - $M = L1$
  - $N = M1 \cup N(0 \cup 1)$
  - The $\varepsilon$ is on the RHS of the starting state only.
  - We want to solve (e.g. using Arden’s Rule) for $L$ and $M$, and take the union of the solutions.
Solving RE Equations

- **Solve** for L and M:
  - $L = L_0 \cup M_0 \cup \varepsilon$
  - $M = L_1$
  - $N = M_1 \cup N(0 \cup 1)$

- **Substitution** Operation:
  - A LHS variable can be replaced with its RHS, so replacing $M$ in the $L$ equation:
  - $L = L_0 \cup L_{10} \cup \varepsilon$, or more simply
  - $L = L(0 \cup 10) \cup \varepsilon$

- **Elimination** Operation:
  - An equation of the form $L = L_0 \cup B$ has the solution $L = B\varepsilon$, so:
  - $L = \varepsilon(0 \cup 10)^*$, or more simply $L = (0 \cup 10)^*$

- Substitution again:
  - $M = L_1$
  - $M = (0 \cup 10)^*$
The language accepted by the DFA below is

- $L \cup M$
- which is $(0 \cup 10)^* \cup (0 \cup 10)^*1$
- or more simply
- $(0 \cup 10)^*(\varepsilon \cup 1)$
DFA ⇒ RE Algorithm

- Express the FSA as a set of RE equations
  - Each state is a variable.
  - Each variable is equated to a union of expressions showing how to get to that state in one step from other states.
  - The start state has $\epsilon$ on the RHS as well.

- Solve the RE equations for the variables:
  - The variables, along with their equations, are solved for one at a time.
  - Choose a variable for elimination.
  - Express that variable in terms of the remaining variables only, using the $*$ operator ($L = LA \cup B$ has the solution $L = BA*$).
  - Substitute the solution for all occurrences of the variable in the remaining equations.
  - Repeat the above steps until no variables remain.

- Work backward, substituting the solutions found for other variables, until each variable is expressed in closed form.
Another Example

- **Solve:**
  - \( L = L_1 \cup M_0 \cup N_0 \cup \epsilon \)
  - \( M = L_0 \cup M_1 \cup N_1 \)
  - \( N = L_1 \cup M_1 \cup N_0 \)

- Note that these equations don’t really correspond to a deterministic machine, but it doesn’t matter.

- Eliminate \( N \), using \( N = (L_1 \cup M_1)^0^* \)
  - \( L = L_1 \cup M_0 \cup (L_1 \cup M_1)^0^*^0 \cup \epsilon \)
  - \( M = L_0 \cup M_1 \cup (L_1 \cup M_1)^0^*^1 \)

- Regroup:
  - \( L = L(1 \cup 10^*0) \cup M(0 \cup 10^*0) \cup \epsilon \)
  - \( M = L(0 \cup 10^*1) \cup M(1 \cup 10^*1) \)
Solution, continued

- **Solving:**
  - \( L = L(1 \cup 10*0) \cup M(0 \cup 10*0) \cup \varepsilon \)
  - \( M = L(0 \cup 10*1) \cup M(1 \cup 10*1) \)
- **Eliminate** \( M \) **using** \( M = L(0 \cup 10*1) \cup M(1 \cup 10*1) \), giving:
  - \( L = L(1 \cup 10*0) \cup L(0 \cup 10*1) (1 \cup 10*1)(0 \cup 10*0) \cup \varepsilon \)
- **Regrouping:**
  - \( L = L((1 \cup 10*0) \cup (0 \cup 10*1) (1 \cup 10*1)(0 \cup 10*0)) \cup \varepsilon \)
- **Solving:**
  - \( L = ((1 \cup 10*0) \cup (0 \cup 10*1) (1 \cup 10*1) (0 \cup 10*0))* \)
- **Working backward:**
  - \( M = ((1 \cup 10*0) \cup (0 \cup 10*1) (1 \cup 10*1) (0 \cup 10*0))* (0 \cup 10*1) (1 \cup 10*1) \)
  - \( N = (L1 \cup M1)0* = ... \)
Graphical Alternative Viewpoint

- The DFA is interpreted graphically as a set of regular-expression equations.

- After an initial setup, nodes are eliminated one at a time, replacing paths through them with regular expressions.

- At completion, there is one arc between a pair of nodes, labeled with the regular expression for the language of the DFA.
DFA → RE Example
Step 1: Add Isolated Start and End States

View each arc as having a regular expression label, not just a single symbol.
Ultimate goal

The figure shows a transition from state $s$ to state $e$ labeled as "RE for DFA."
Elimination Step

- Pick a node for elimination (other than start and end).

- Union to the regular expression of each pair of nodes having a path through the chosen node an additional expression component representing those paths.
Elimination Step Illustrated

Before:

\[ P = \text{paths from } f \text{ to } t \]

\[
\begin{align*}
  &f \quad R \quad x \quad S \quad T \\
  \text{from-state} & \quad \text{eliminate} & \quad \text{to-state} \\
  &t
\end{align*}
\]

added paths from \( f \) to \( t \):
Elimination Step Illustrated

After:

\[ P \cup RS^*T \]

from-state  \( f \)  eliminate  \( x \)  to-state  \( t \)

This has to be done for all pairs \( f, t \) including the case where \( f = t \).
Eliminate c
c Eliminated
Eliminate a
a Eliminated
Eliminate d
d Eliminated

0 ∪ 1(01)*(1 ∪ 00) → b

11 ∪ (0 ∪ 10)(01)*(1 ∪ 00) → b

b → 1 → e

b → d

d
Eliminate $b$

\[ s \rightarrow b \rightarrow e \]

\[ 0 \cup 1(01)^*(1 \cup 00) \rightarrow b \rightarrow 11 \cup (0 \cup 10)(01)^*(1 \cup 00) \]
b Eliminated (= done)

\[(0 \cup 1(01)^*(1 \cup 00)) (11 \cup (0 \cup 10)(01)^*(1 \cup 00))^* 1\]
Summary so far

- The language accepted by an DFA is a regular language.
- We haven’t yet shown that the converse is true.