Formal Languages and Automata Theory

CSE 5210

Introduction
CSE 5210 – Formal Languages and Automata Theory

Catalog Data:

This course presents the theory of finite automata: finite-state machines, pushdown automata, bounded linear automata, and Turing machines.

The corresponding of languages: regular, context-free, context-sensitive and recursively enumerable languages is presented.

Goals:

This course introduces the students to the theory underlying the development of machines and languages.

The student is introduced to the main classes of machines and languages they recognize.

Questions of what can and cannot be computed with these machines are addressed.

Besides learning abstract theory, which is useful in its own right, students will be exposed to data and control structures that are useful in many applications of computer science.
Introduction

Formal language theory provides the foundation for the definition of programming languages and compiler design.

Formal language theory has its roots in linguistics, mathematical logic, and computer science.

Noam Chomsky developed formal systems called grammars for generating syntactically correct sentences. Grammars provide a mechanism for describing natural languages and have become the primary tool for the formal specification of programming languages.

In the 1930's, A. Turing studied an abstract machine that had all the capabilities of today's computers.

Turing's goal was to describe precisely the boundary between what a computing machine could do and what it could not do.

The study of computability was motivated by two fundamental questions:
- What is an algorithm?
- What are the capabilities and limitations of algorithmic computation?

The question of whether a problem is algorithmically solvable should be independent of the model of computation used. Either there is an algorithmic solution to a problem or there is no such solution.

Many different computational models have been proposed, including recursive functions, lambda calculus of Alonzo Church, Markov systems, and the abstract machines developed by A. Turing.

The Turing machine is used as a framework for the study of computation due to its simplicity and similarities to the modern computer.
1.1 Why Study Automata Theory?

1.1.1 Introduction to Finite Automata.

Finite automata are a useful model for many important kinds of hardware and software. Below is a list of some of the most important kinds:

- Software for designing and checking the behavior of digital circuits.
- The “lexical analyzer” of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuations.
- Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases, or other patterns.
- Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols for secure exchange of information.

A Finite Automaton modeling an on/ off switch
1.1.2 Structural Representation.

There are two important notations that are not automaton-like, but play an important role in the study of automaton and their applications.

Grammars are useful models when designing software that processes data with a recursive structure.

\[ E \implies E + E \]

Regular Expressions also denote the structure of data, especially test strings. The styles of these expressions differs significantly from that of grammars.

\[ [A-Z][a-z]*[ ][A-Z][A-Z]\]

The UNIX-style regular expression \( E = \) represents capitalized words followed by a space and two capital letters.

1.1.3 Automata and Complexity.

Automata are essential for the study of the limits of computation. There are two important issues:

1. What can a computer do? This study is called decidability.

2. What can a computer do efficiently? This study is called intractability.
Chapter 1 - Automata: The Methods and the Madness

Short Review of Set Theory:

The assumption is that everyone is familiar with fundamentals of set theory. This section serves as a brief review of the concepts and notation.

The symbol $\in$ signifies membership.
- $x \in X$  
  x is a member or element of the set X.
- $x \notin X$  
  x is not a member or element of the set X.

Sets are denoted by capital letters.
A, B, C

Brackets { } are used to indicate a set definition.
- $A = \{ 1, 2, 3, 4 \}$

A set Y is a subset of X.
- $Y \subseteq X$

If $Y \subseteq X$, and $Y \neq X$ then Y is called a proper subset of X.

Example:
#1 Let $X = \{ 1, 2, 3 \}$

The subsets of X are:
- $\emptyset$  
- $\{1\}$  
- $\{2\}$  
- $\{3\}$  
- $\{1, 2\}$  
- $\{2, 3\}$  
- $\{3, 1\}$  
- $\{1, 2, 3\}$
Short Review of Set Theory (cont.):

The union of two sets is defined by

\[ X \cup Y = \{ z \mid z \in X \text{ or } z \in Y \} \]

Example:

#1 Let \( X = \{ 1, 2, 3 \} \) and \( Y = \{ 2, 4, 5, 6 \} \)

\[ X \cup Y = \{ 1, 2, 3, 4, 5, 6 \} \]

#2 Let \( X = \{ 1, 3, 4, 5, 8 \} \) and \( Y = \{ 2, 4, 5, 6 \} \)

\[ X \cup Y = \{ 1, 2, 3, 4, 5, 6, 8 \} \]

The intersection of two sets is defined by

\[ X \cap Y = \{ z \mid z \in X \text{ and } z \in Y \} \]

The two sets whose intersection is empty are said to be disjoint.

Example:

#1 Let \( X = \{ 1, 2, 3 \} \) and \( Y = \{ 2, 4, 5, 6 \} \)

\[ X \cap Y = \{ 2 \} \]

#2 Let \( X = \{ 1, 2, 4, 5, 8 \} \) and \( Y = \{ 2, 4, 5, 6 \} \)

\[ X \cap Y = \{ 2, 4, 5 \} \]
Chapter 1 - Automata: The Methods and the Madness

Short Review of Set Theory (cont.):

The union and intersection of n sets, \( X_1, X_2, X_3, \ldots, X_n \) are defined by

\[
\begin{align*}
\bigcup_{i=1}^{n} X_i &= X_1 \cup X_2 \cup X_3 \cup \ldots \cup X_n \\
\bigcap_{i=1}^{n} X_i &= X_1 \cap X_2 \cap X_3 \cap \ldots \cap X_n
\end{align*}
\]

Let \( X \) be a subset of \( U \). The complement of \( X \) with respect to \( U \) is the set of elements in \( U \) but not in \( X \).

Example #1
Let \( X = \{0, 1, 2, 3\} \) and \( Y = \{2, 3, 4, 5, 6, 8\} \).

\( X \) and \( Y \) denote the complement of \( X \) and \( Y \) with respect to \( \mathbb{N} \).

\[
X \cup Y = \{0, 1, 2, 3, 4, 5, 6, 8\} \quad \text{then} \quad X = \{n \mid n > 3\}
\]
Chapter 1 - Automata: The Methods and the Madness

Countable and Uncountable Sets:

Cardinality is the measure that compares the size of sets. Cardinality can be obtained by counting the elements of the set.

A set that has the same cardinality as the set of natural numbers is **countably infinite** or **denumerable**

Sets \{ a, b, c \} and \{ 1, 2, 3 \} comprises of the same number of elements, where a -> 1, b -> 2, c -> 3.

**Example #1**

There are as many even numbers as there are natural numbers. Use mapping function \( F(n) = n \times 2 \)

1 * 2 = 2  
2 * 2 = 4  
3 * 2 = 6  
4 * 2 = 8

**Note:** There are as many odd numbers as there are natural numbers. Use mapping function \( F(n) = (n \times 2) + 1 \)

**Observation:** \( n \) (even numbers) + \( n \) (odd numbers) ≠ 2 \( n \) (natural numbers)

Our intuition can fail in regard to cardinality and infinite sets.
1.2 Introduction to Formal Proof

1.2.1 Deductive Proofs

- A deductive proof consists of a sequence of statements whose truth leads us from some initial statement, called the hypothesis or the given statement, to a conclusion statement.

- The hypothesis may be true or false, typically depending on values of its parameter.

- The theorem that is proved when we go from a hypothesis $H$ to a conclusion $C$ is the statement “if $H$ then $C$”. We say that $C$ is deduced from $H$.

Example:
Theorem: If $x > 4$, then $2^x > x^2$.

The hypothesis $H$ is “$x \geq 4$”, and the conclusion $C$ is $2^x \geq x^2$.

Informal Proof:

As $x$ grows larger than 4, the left side, $2^x$ doubles each time $x$ increases by 1. The right side, $x^2$, grows by the ratio $((x + 1) / x)^2$. If $x \geq 4$, then $((x + 1) / x)$ cannot be greater than 1.25, and therefore $((x + 1) / x)^2$ cannot be bigger than 1.5625. Since $1.5625 < 2$, each time $x$ increases above 4 the left side $2^x$ grows more than the right side $x^2$.

Thus, as long as we start from a value like $x = 4$ where the inequality $2^x \geq x^2$ is already satisfied, we can increase $x$ as much as we like, and the inequality will still be satisfied.
Chapter 1 - Automata: The Methods and the Madness

1.2.2 Reduction to Definitions

If you are not sure how to start a proof, convert all terms in the hypothesis to their definitions.

Example:

1. A set $S$ is finite if there exists an integer $n$ such that $S$ is exactly $n$ elements. We write $|S| = n$, where $|S|$ is used to denote the number of elements in a set $S$. If the set $S$ is not finite, we say $S$ is infinite.

2. If $S$ and $T$ are both subsets of some set $U$, then $T$ is the complement of $S$ if $S \cup T = U$ and $S \cap T = \emptyset$; that is, each element of $U$ is in exactly one of $S$ and $T$. $T$ consists of exactly those elements of $U$ that are not in $S$.

Theorem:

Let $S$ be a finite subset of some infinite set $U$. Let $T$ be the complement of $S$ with respect to $U$.

Then $T$ is infinite.

Proof:

Intuitively, this theorem says that if you have an infinite supply of something ($U$), and you take a finite amount away ($S$), then you still have an infinite amount left.

We know that $S \cup T$ and $S$ and $T$ are disjoint, so $|S| + |T| = |U|$. Since $S$ is finite, $|S| = n$ for some integer $n$, and since $U$ is infinite, there is no integer $p$ such that $|U| = p$. So assume that $T$ is finite; that is, $|T| = m$ for some integer $m$. Then $|U| = |S| + |T| = n + m$, which contradicts the given statement that there is no integer $p$ equal to $|U|$.
1.3 Additional Forms of Proof

1.3.1 Proving Equivalences About Sets.

The commutative law of union says that we can take the union of two sets $R$ and $S$ in either order. That is $R \cup S = S \cup R$.

In this case, $E$ is the expression $R \cup S$ and $F$ is the expression $S \cup R$.

The commutative law of union says that $E = F$.

Example:

Let $R = \{a, b, c, x\}$, $S = \{a, c, d\}$, and $T = \{c, e, f\}$

Show that: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

$\{a, b, c, x\} \cup (\{a, c, d\} \cap \{c, e, f\}) = (\{a, b, c, x\} \cup \{a, c, d\}) \cap (\{a, b, c, x\} \cup \{c, e, f\})$

$\{a, b, c, x\} \cup \{c\} = \{a, b, c, d, x\} \cap \{a, b, c, e, f, x\}$

$\{a, b, c\} = \{a, b, c, x\}$
Theorem 1.10: \( R \cup (S \cap T) = (R \cup S) \cap (R \cup T) \)

Proof:
The two set-expressions involved are \( E = R \cup (S \cap T) \) and \( F = (R \cup S) \cap (R \cup T) \)

In the “if” part we assume that \( x \) is in \( E \), and show that it is in \( F \).
1. \( X \) is in \( R \cup (S \cap T) \) given
2. \( X \) is in \( R \) or \( x \) is in \( (S \cap T) \) (1) and definition of union
3. \( X \) is in \( R \) or \( x \) is in both \( S \) and \( T \) (2) and definition of intersection
4. \( X \) is in \( R \cup S \) (3) and definition of union
5. \( X \) is in \( R \cup T \) (3) and definition of union
6. \( X \) is in \( (R \cup S) \cap (R \cup T) \) (4), (5), and definition of intersection

We must prove the “only-if” part of the theorem. Assume \( x \) is in \( F \) and show that it is in \( E \).
1. \( X \) is in \( (R \cup S) \cap (R \cup T) \) given
2. \( X \) is in \( R \cup S \) (1) and definition of intersection
3. \( X \) is in \( R \cup T \) (1) and definition of intersection
4. \( X \) is in \( R \) or \( x \) is in both \( S \) and \( T \) (2), (3) and reasoning about unions
5. \( X \) is in \( R \) or \( x \) is in \( S \cap T \) (4) and definition of intersection
6. \( X \) is in \( R \cup (S \cap T) \) (5), and definition of union

Since we proved both parts of the if-and-only-if statement, the distributive law of union over intersection is proved.
1.4 Inductive Proofs

Example 1:

Prove that \( 1 + 2 + 3 + \ldots + n = \frac{n(n + 1)}{2} \)

Basis:
- \( P(n) = P(1) = 1 \)
- \( P(n) = P(2) = 1 + 2 = 3 \)

Inductive Step:
- \( P(k) = 1 + 2 + 3 + \ldots + k = \frac{k(k + 1)}{2} \)
- \( P(k + 1) = 1 + 2 + 3 + \ldots + k + (k + 1) = \frac{(k + 1)(k + 2)}{2} \)

substitute

\[ P(k + 1) = \frac{k(k + 1)}{2} + (k + 1) = \frac{(k + 1)(k + 2)}{2} \]
Example 1 (cont.)

\[
\begin{align*}
P(k + 1) &= \frac{k^2 + k + 2k + 2}{2} = \frac{(k + 1)(k + 2)}{2} \\
P(k + 1) &= \frac{k^2 + 3k + 2}{2} = \frac{(k + 1)(k + 2)}{2} \\
P(k + 1) &= \frac{(k + 1)(k + 2)}{2} = \frac{(k + 1)(k + 2)}{2}
\end{align*}
\]
Example #2

Hypothesis:

\[
\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n + 1)(2n + 1)}{6}
\]

Proof:

\[
(k + 1)^3 = (k + 1)(k + 1)(k + 1)
\]
\[
(k + 1)^3 = k^3 + 3k^2 + 3k + 1
\]

Subtract \(k^3\)

\[
(k + 1)^3 - k^3 = 3k^2 + 3k + 1
\]

Sum on \(k\)

\[
\sum_{k=1}^{n} (k + 1)^3 - k^3 = \sum_{k=1}^{n} (3k^2 + 3k + 1)
\]

Expanding out the left side yields

\[
[ (1 + 1)^3 - 1^3 ] + [ (2 + 1)^3 - 2^3 ] + [ (3 + 1)^3 - 3^3 ] + \ldots
\]
\[
[2^3 - 1^3 ] + [3^3 - 2^3 ] + [4^3 - 3^3 ] + \ldots + [ (n + 1)^3 - n^3 ]
\]
Example 2 (cont.)

Each term cancels part of the next term, it is a telescoping sum.

\[-1 + (n + 1)^3 = \sum_{k=1}^{n} (3k^2 + 3k + 1)\]

\[-1 + (n + 1)^3 = 3 \sum_{k=1}^{n} k^2 + 3 \sum_{k=1}^{n} k + \sum_{k=1}^{n} 1\]

\[-1 + (n + 1)^3 = \frac{n(n + 1)^3}{3} + \frac{n(n + 1)}{2} + n\]

\[3 \sum_{k=1}^{n} k^2 = 3 \frac{n(n + 1)(2n + 1)}{6}\]
1.5 The Central Concepts of Automata Theory

1.5.1 Alphabets
An alphabet is a finite, nonempty set of symbols. We use the symbol $\Sigma$ for an alphabet.

$\Sigma = \{0, 1\}$, the binary alphabet

$\Sigma = \{a, b, \ldots, z\}$, the set of all lower-case letters.

1.5.2 Strings
A string is a finite sequence of symbols chosen from some alphabet.

“01101” is a string from $\Sigma = \{0, 1\}$

The empty string is the string with zero occurrences of symbols. The string is denoted $\epsilon$.

The standard notation for the length of a string $w$ is $|w|$.

The set of all strings over an alphabet $\Sigma$ is denoted $\Sigma^*$.

1.5.3 Languages
A set of strings all of which are chosen from some $\Sigma^*$, where $\Sigma$ is some particular alphabet, is called a language.

If $\Sigma$ is an alphabet, and $L \subseteq \Sigma^*$, then $L$ is a language over $\Sigma$. 
Chapter 1 - Automata: The Methods and the Madness

Concatenation of Strings

Let $x$ and $y$ be strings. Then $xy$ denotes the concatenation of $x$ and $y$.

If $x$ is the string composed of $i$ symbols $x = a_1a_2 \ldots a_i$ and $y$ is the string composed of $j$ symbols $y = b_1b_2 \ldots b_j$, then $xy$ is the string of length $i + j$: $xy = a_1a_2 \ldots a_i b_1b_2 \ldots b_j$.

1.5.4 Problems

In automata theory, a problem is the question of deciding whether a given string is a member of some particular language.

More precisely, if $\Sigma$ is an alphabet, and $L$ is a language over $\Sigma$, then the problem $L$ is:

*Given a string $w$ in $\Sigma^*$, decide whether or not $w$ is in $L$.***
Formal Languages and Automata Theory

CSE 5210
Session 2
Finite Machines and Regular Languages

A machine can be represented as shown below:

The input to the machine is a string, and the notation of the string is usually $u$, $v$, or $w$.

A string over a set $X$ is a finite sequence of elements from $X$.

The set of elements from which the strings are built is called the alphabet $\sum$ of the language. An alphabet consist of a finite set of indivisible objects.

The string that contains no elements is called the null string and is denoted $\lambda$. 
The representation of the machine can be expanded to show internal functions.

**Initializer and Terminators:**

An initializer for a machine is a one to one function that determines the machines initial configuration.

All variables must be initialized to some known value before starting any computation.

The initialization is performed by the compiler or converter. The compiler takes the input string u and concatenates it to the string C that represents the operation of the machine.

All machines must have some type of terminator to halt the computation. This may be an empty stack, end of file, or reaching a final state of a computation.
A program $P$ for machine $M$ consists of:

- initializer $\alpha$
- terminator $w$
- A finite set of instructions $\Gamma$ for $M$, which basically is algorithm $D$

**Three types of Programs**

**Acceptor:**
An acceptor is a non-deterministic program that takes a string as input and either accepts it or does not. If a machine $M$ accepts a language $L$, than we call $M$ an acceptor for $L$.

A common application of a program is to test whether a string belongs to a language $L$.

**Types of Acceptors:**
- Non-Deterministic Finite Acceptor
- Non-Deterministic Counter Acceptor
- Non-Deterministic Stack Acceptor
- Deterministic Turing Acceptor
- Deterministic Finite Acceptor
- Deterministic Counter Acceptor
- Deterministic Stack Acceptor
- Deterministic Turing Acceptor
Acceptor:

The machine computes for ever.

Recognizer:

If a machine $m$ decides a language $L$, then we call $M$ a recognizer for $L$. 
Transducers:

A Transducer is a program that computes a relation from $\Sigma^*$ to $\zeta^*$.

Let $M$ be a machine $[\text{control, input, output, } d_4, \ldots, d_k]$ with input alphabet $\Sigma$ and output alphabet $\zeta$.

A program for $M$ is a transducer if its initializer and terminator satisfy the following conditions:

The initial state is equal to the argument; i.e., $\alpha_{\text{input}} = I_{\Sigma^*}$

For $d \neq \text{input}$, the initial state of $d$ does not depend on the argument, i.e., $\alpha_d = \Sigma^* \times \{s_d\}$
for some states $s_d$ in the realm of $d$.

The result is equal to the outputs final state; i.e., $w_{\text{output}} = I_{\Sigma^*}$

For $d \neq \text{output}$, the result does not depend on the state that $d$ ends up in; i.e. $w_d = s_{d} \times \zeta$
for some subset $s_d$ of the realm of $d$.

An example for a transducer is a Turing Machine that copies an input string to an output Tape.
The machine does not have to halt in any particular state.
When the machine halts the computation, the output string is on the output tape ready for the user to read.
The machine may have halted in a state other than the output state.
Definition Strings

Let $\Sigma$ be an alphabet. $\Sigma^*$ is the set of strings over $\Sigma$, is defined recursively as follows:

Basis: $\lambda \in \Sigma^*$

Recursive Step: If $w \in \Sigma^*$ and $a \in \Sigma$, then $wa \in \Sigma^*$

Closure: $w \in \Sigma^*$ only if it can be obtained from $\lambda$ by a finite number of applications of the recursive step.

Example #1

Let $\Sigma = \{a, b, c\}$.

The elements of $\Sigma^*$ are
Length 0: $\lambda$
Length 1: a b c
Length 2: aa ab ac ba bb bc ca cb cc
Length 3: aaa aab aac aba abb abc aca acb acc
              baa bab bac bba bbb bbc bca bcb bcc
              caa cab cac cba cbb cbc cca ccb ccc
Definition:

A language over an alphabet $\Sigma$ is a subset $\Sigma^*$. 

Since strings are the elements of a language, we must examine the properties of strings and the operations on them.

Concatenation is the binary operation of taking two strings and gluing them together to construct a new string.

Concatenation is the fundamental operation in the generation of strings.

**Definition concatenation:**

Let $u, v \in \Sigma^*$. The concatenation of $u$ and $v$, written $uv$, is a binary operation on $\Sigma^*$ defined as follows:

**Basis:** If $\text{length}(v) = 0$, then $v = \lambda$ and $uv = u$

**Recursive step:** Let $v$ be a string with length $(v) = n > 0$. Then $v = wa$, for some string $w$ with length $n - 1$ and $a \in \Sigma$, and $uv = (uw)a$.

**Example #1**

Let $u = ab$, $v = ca$, and $w = bb$. Then

$uv = abca$  
$vw = cabb$  
$(uv)w = abcabb$  
$u(vw) = abcabb$
Finite Specification of Languages:

A language has been defined as a set of strings over an alphabet. Languages of interest do not consist of arbitrary sets of strings but rather of strings having specified form.

The specification of a language requires an unambiguous description of the strings of the language.

A finite language can be explicitly defined by enumerating its elements.

Example:
The language \( L \) of strings over \( \{a, b\} \) in which each string begins with an a and has even length is defined by:

**Basis:** \( \text{aa, ab} \in L \).

**Recursive Step:** If \( u \in L \), then \( u\text{aa, } u\text{ab, } u\text{ba, } u\text{bb} \in L \).

**Closure:** A string \( u \in L \) only if it can be obtained from the basis elements by finite number of applications of the recursive step.

Example:
The language \( L \) consists of strings over \( \{a, b\} \) in which each occurrence of b is immediately preceded by an a.
\( \lambda, a, abaab \) are in \( L \), but \( bb, bab, abb \) are not in \( L \).

**Basis:** \( \lambda \in L \).

**Recursive Step:** If \( u \in L \), then \( u\text{a, } u\text{ab} \in L \).

**Closure:** A string \( u \in L \) only if it can be obtained from the basis element by finite number of applications of the recursive step.
Definition Kleene Star:

Let \( X \) be a set. Then

\[
\infty x^* = \bigcup_{i} X^i \quad \text{and} \quad \infty X^+ = \bigcup_{i} X^i
\]

\( i = 0 \quad \text{and} \quad i = 1 \)

\( X^* \) contains all strings that can be built from the elements of \( x \).
If \( X \) is an alphabet, \( X^+ \) is the set of all non-null strings over \( X \).
\( X^+ = XX^* \).

Example:
The language \( L = \{a, b\}^* \{bb\} \{a, b\}^* \) consists of the strings over \( \{a, b\} \) that contain the substring \( bb \).
The concatenation of the set \( \{bb\} \) ensures the presence of \( bb \) in every string in \( L \).
The sets \( \{a, b\}^* \) permit any number of \( a \)'s and \( b \)'s in any order, to precede and follow the occurrence of \( bb \).

Example:
Let \( L_1 = \{bb\} \) and \( L_2 = \{\lambda, bb, bbbb\} \) be languages over \( \{b\} \).
The languages \( L_1^* \) and \( L_2^* \) both contain precisely the strings consisting of an even number of \( b \)'s.
Note that \( \lambda \), with length zero, is an element of \( L_1^* \) and \( L_2^* \).
Regular Sets and Expressions:

A set is regular if it can be generated from the empty set, the set containing the null string, and the elements of the alphabet using union, concatenation, and the Kleene Star operation.

Definition of a Regular Set:

Let \( \Sigma \) be an alphabet. The regular sets over \( \Sigma \) are defined recursively as follows:

**Basis:**  \( \emptyset, \{ \lambda \} \) and \( \{ a \} \), for every \( a \in \Sigma \), are regular sets over \( \Sigma \).

**Recursive step:** Let \( X \) and \( Y \) be regular sets over \( \Sigma \). The following sets are regular sets over \( \Sigma \):

\[
X \cup Y \\
XY \\
X^* 
\]

Closure: \( X \) is a regular set over \( \Sigma \) only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

Example:

The Language \( L = \{ a, b \}^* \{ bb \} \{ a, b \}^* \) comprises of the set of strings containing the substrings \( bb \), is a regular set over \( \{ a, b \} \). From the basis of the definition, \( \{ a \} \) and \( \{ b \} \) are regular sets.

Applying union and the Kleene star operation produces \( \{ a, b \}^* \), the set of all strings over \( \{ a, b \} \).

By concatenation, \( \{ b \} \{ b \} = \{ bb \} \) is regular.

Applying concatenation twice yields \( \{ a, b \}^* \{ bb \} \{ a, b \}^* \).
Definition of Regular Expressions:

Let \( \Sigma \) be an alphabet. The regular expressions over \( \Sigma \) are defined recursively as follows:

**Basis:** \( \emptyset, \{ \lambda \} \text{ and } \{a\}, \) for every \( a \in \Sigma, \) are regular expressions over \( \Sigma. \)

**Recursive step:** Let \( U \) and \( V \) be regular expressions over \( \Sigma. \) The following are regular expressions over \( \Sigma. \)

- \( (u \cup v) \)
- \( (uv) \)
- \( (X^*) \)

Closure: \( u \) is a regular expression over \( \Sigma \) only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

Since union and concatenation are associative, parenthesis can be omitted from expressions.

**Regular Expressions:**

- \( \emptyset \) is a regular expression
- \( \{ \lambda \} \) is a regular expression
- If \( c \) is a character, then \( (c) \) is a regular expression
- If \( L_1 \) and \( L_2 \) are regular expression, then \( L_1 \cup L_2 \) is a regular expression
- If \( L_1 \) and \( L_2 \) are regular expression, then \( L_1 \otimes L_2 \) is a regular expression
- If \( L_1 \) is a regular expression, then \( L_1 \ast \) is a regular expression.
Example:

The set \( \{ \text{bawab} \mid w \in \{a, b\}^* \} \) is regular over \( \{a, b\} \).

The following table demonstrates the recursive generation of a regular set and the corresponding regular expression.

<table>
<thead>
<tr>
<th>Set</th>
<th>Expression</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. {a}</td>
<td>a</td>
<td>Basis</td>
</tr>
<tr>
<td>2. {b}</td>
<td>b</td>
<td>Basis</td>
</tr>
<tr>
<td>3. {a} {b}</td>
<td>ab</td>
<td>1, 2, concatenation</td>
</tr>
<tr>
<td>4. {a} \cup {b} = {a, b}</td>
<td>a \cup b</td>
<td>1, 2, union</td>
</tr>
<tr>
<td>5. {b} {a} = {ba}</td>
<td>ba</td>
<td>2, 1, concatenation</td>
</tr>
<tr>
<td>6. {a, b}*</td>
<td>(a \cup b)*</td>
<td>4, Kleene star</td>
</tr>
<tr>
<td>7. {ba} {a, b}*</td>
<td>ba (a \cup b)*</td>
<td>5, 6, concatenation</td>
</tr>
<tr>
<td>8. {ba} {a, b}* {ab}</td>
<td>ba (a \cup b)* \ab</td>
<td>7, 3, concatenation</td>
</tr>
</tbody>
</table>
Deterministic Finite Automata

A finite automaton has a set of states, and its control moves from state to state in response to external inputs.

The term “deterministic” refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state.

Definition of Deterministic Finite Automata (DFA):

A deterministic finite automaton (DFA) is a quintuple $M = (Q, \Sigma, \sigma, q_0, F)$ where

- $Q$ is a finite set of states
- $\Sigma$ is a finite set called the alphabet
- $\sigma$ is a set of transition functions
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is set of states called the final or acceptance state.

Definition of $L(M)$:

Let $M = (Q, \Sigma, \sigma, q_0, F)$ be a DFA.

The language of $M$, denoted $L(M)$, is the set of strings in $\Sigma^*$ accepted by $M$.

A DFA can be considered to be a language acceptor, the language recognized by the machine is simply the set of strings accepted by its computations.

A DFA reads the input in a left to right manner; once an input symbol has been processed, it has no further effect on the computation.
Finite Machines and Regular Languages

Example 1:
Let $M$ be a DFA where

$Q = \{A, B, C\}$
$\Sigma = \{a, b\}$
Start State = $A$
Final State = $C$
$\sigma = \{\sigma(A, a) = A, \sigma(A, b) = B, \sigma(B, a) = A, \sigma(B, b) = C, \sigma(C, a) = C, \sigma(C, b) = C\}$

Suppose $u = \text{"ababb"}$ is in the Language $L(M)$ then $M$ performs the following computations.

$\sigma(A, ababb) = A, \sigma(A, babb) = B, \sigma(B, abb) = A, \sigma(A, bb) = B, \sigma(B, b) = B, \sigma(C, e) = \text{Accept}$.

The string "ababb" is accepted since the halting state of the computation, which is also the terminal state of the path that spells ababb, is the accepting state $C$.

The strings that are accepted must have two consecutive bb's.
Given the DFA $M$ in example 1, one can derive the language $L(M)$ directly from $M$.

A is the start state of $M$ and it can process either symbol a or b. Since $M$ always returns to state $A$ given input a, $\sigma(A, a) = A$, it represents the expression $(a)^*$. The transition functions $\sigma(A, b) = B$ and $\sigma(B, a) = A$ allow the machine to process $(ba)^*$. The combination of the first two expressions is $(a^* + (ba)^*)^*$.

C is the final state of $M$. The final state can be reached only after consuming two consecutive bb’s. In the final state $C$ the machine processes zero or more occurrences of symbols a and b. $(a + b)^*$, which provides the expression $(bb)(a + b)^*$. Therefore, we derive the final expression

$$L(M) = (a^* + (ba)^*)^* (bb)(a + b)^*$$

$L(M)$ can be simplified to $L(M) = (a \cup b)^* bb (a \cup b)^*$.
Finite Machines and Regular Languages

Example 2:
Let M be a DFA where

- \( Q = \{A, B, C\} \)
- \( \Sigma = \{a, b\} \)
- Start State = A
- Final State = \{A, C\}
- \( \sigma = \{\sigma (A, a) = A, \sigma (A, b) = B, \sigma (B, b) = C, \sigma (C, a) = C, \sigma (C, b) = C\} \)

Suppose \( u = \text{"aabba"} \) is in the Language \( L(M) \) then M performs the following computations.

- \( \sigma (A, aabba) = A, \sigma (A, abba) = A, \sigma (A, bba) = B, \sigma (B, ba) = C, \sigma (C, a) = C, \sigma (C, \epsilon) = \text{Accept.} \)

The string "aabba" is accepted since the halting state of the computation, which is also the terminal state of the path that spells aabba, is the accepting state C.

The strings that are accepted must contain two consecutive bb's.
Given the DFA $M$ in example 2, one can derive the language $L(M)$ directly from $M$.

$A$ is the start state of $M$ and it can process either symbol $a$ or $b$. Since $M$ always returns to state $A$ given input $a$, $\sigma(A, a) = A$, it represents the expression $(a)^*$. 

Since $A$ is also an acceptance state of $M$, $M$ can process strings that either comprise of the $\lambda$ string or strings that only contain $a$'s. This generates the expression $(a^* + \lambda)$. 

After $M$ consumed all leading $a$'s in the string, it can transition to state $C$ on input $bb$. This generates the expression $(a^* + \lambda) + (a^*bb)$. 

Once machine $M$ is in the final state $C$, $M$ can process zero or more occurrences of symbols $a$ and $b$. $(a + b)^*$. 

Therefore, we derive the final expression $(a^* + \lambda) + (a^*bb) (a + b)^*$ 

$L(M) = a^* \cup \lambda \cup (a^*bb) (a \cup b)^*$
Finite Machines and Regular Languages

Example 2:
Let $M$ be a NFA where

$Q = \{A, B, C\}$
$\Sigma = \{a, b\}$
Start State = $A$
Final State = $C$
$\sigma = \{\sigma(A, a) = A, \sigma(A, b) = \{A, B\}, \sigma(B, b) = C\}$

Suppose $u = \text{"abb"}$ is in the Language $L(M)$ then $M$ performs the following computations.

1. $M$ is in state $A$ and has input $a$. Transition function $\sigma(A, a) = A$ applies.
2. $M$ is in state $A$ and has input $b$. Transition function $(A, b) = A$ and $(A, b) = B$ applies. A second thread is started. $M$ is in two states.
3. M is in state A and B and has input b. Transition function $\sigma(B, b) = C$ takes the second thread to the acceptance state C. The first thread follows the transition function $\sigma(A, b) = A$ to state A. A third thread is started with the transition function $\sigma(A, b) = B$.

4. M is in state A, B and the acceptance state C. On input $\varepsilon$ the first and third thread dies. The second thread is in state C, therefore M accepts.
The remainder of session #3 comes directly from the text.

**Chapter 2 Finite Automata**

Read section 2.1 through 2.2.3 -- With the exception of Transition Tables we already covered this material in class.

We will cover the following material in class.

- 2.2.4 Extending the Transition Function to Strings.
- 2.3 Nondeterministic Finite Automata.
  - skip 2.3.3
    - read 2.3.5 and 2.36, but I will do it differently.

- 2.5 Finite Automata with Epsilon-Transitions
  - skip 2.5.4 and 2.5.5

**Chapter 3 Regular Expressions and Languages**

Read section 3.1 -- We covered this material already in class

We will cover the following material in class.

- 3.2 Finite Automata and Regular Expressions
  - 3.2.1 From DFA's to Regular Expressions -- see notes in session 3
  - 3.2.2 Eliminating States -- skip until next week. I will do this differently.
3.2.3 Converting Regular Expressions to Automata – read material, I will do it differently, much simpler.

3.4 Algebraic Laws for Regular Expressions.
Finite Machines and Regular Languages

Example 1:
Let M1 be a DFA where

\[ Q = \{ A \} \]
\[ \Sigma = \{ a \} \]
Start State = A
Final State = A
\[ \sigma = \{ \sigma (A, a) = A \} \]

L (M1) = a* u λ

M1 accepts the empty string and those strings that comprise of the symbol a.
The string |v| = 0 is in L (M1).
The strings “a”, “aa”, “aaa”, “aa/a” are in L (M1).
Finite Machines and Regular Languages

Example 2:
Let M2 be a DFA where

\[ Q = \{A, B, C\} \]
\[ \Sigma = \{a, b, c\} \]
Start State = A
Final State = C
\[ \sigma = \{\sigma (A, a) = A, \sigma (A, b) = B, \sigma (B, b) = B, \sigma (B, c) = C\} \]

\[ L(M2) = a^* bc \]

The strings “abc”, “aabc”, “aaabc”, “aabc”, “bc” are in L(M2).

M2 is an incompletely specified DFA. The string w = “abca” is not in L. M2 will reject string w because M2 is unable to process the final a from state C.

We can avoid this problem by designing the machine slightly differently. Add one additional state which is connected to the final acceptance state by an arc \{a, b, c\}. 
Finite Machines and Regular Languages

Example 3: Let M3 be a DFA where

- $Q = \{A, B, C, D\}$
- $\Sigma = \{a, b, c\}$
- Start State = A
- Final State = C
- $\sigma = \{\sigma(A, a) = A, \sigma(A, b) = B, \sigma(B, b) = B, \sigma(B, c) = C, \sigma(C, a) = D, \sigma(C, b) = D, \sigma(C, c) = D\}$

$L(M3) = a^* bc$

The strings “abc”, “aabc”, “aaabc”, “ababc”, “bc” are in $L(M3)$.

The string $w = “abcaa”$ is not in $L(M3)$. 

\[ \langle u \rangle \quad a \quad b \quad c \quad a, b, c \]

\[ \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \]

\[ A \quad B \quad C \quad D \]
Finite Machines and Regular Languages

Lambda Transitions

The transition from state to state, in both deterministic and nondeterministic automata, were initiated by the processing of an input symbol. The definition of NFA is relaxed to allow state transitions without requiring input to be processed. A transition of this form is called a lambda transition. The class of nondeterministic machines that utilize lambda transitions is denoted NFA-$\lambda$.

The definition of halting must be extended to include the possibility that a computation may continue using lambda transitions after the input string has been completely processed. Lambda moves can be used to construct complex machines from simpler machines.

Let $M_1$ and $M_3$ be machines defined in example 1 and example 3. We can combine both machines to build $M_4 = M_1 \cup M_3$.

$L(M_4) = (a^* \cup \lambda) \cup (a^* \cdot bc)$
Example 4

\[ L(M1) = (a \cup b)^* bc (a \cup b)^* \]

\[ L(M2) = (b \cup ab)^* (a \cup \lambda) \]
Algebraic Laws for Regular Expressions

Commutativity is the property of an operator that says we can switch the order of its operands and get the same result.

\[ L + M = M + L \]
The commutative law for union says that we may take the union of two languages in either order.

\[(L + M) + N = L + (M + N)\]
The associative law for union says that we may take the union of three languages either by taking the union of the first two initially, or taking the union of the last two initially.

\[(LM)N = L(MN)\]
The associative law for concatenation says that we can concatenate three languages by concatenating either the first two or the last two initially.

A distributive law involves two operators, and asserts that one operator can be pushed down to be applied to each argument of the other operator individually.

\[ L(M + N) = LM + LN \]
The left distributive law of concatenation over union.

\[(M + N)L = ML + NL \]
The right distributive law of concatenation over union.
Laws Involving Closures

\((L^*)^* = L^*\)
This law says that closing an expression that is already closed does not change the language.
The language of \((L^*)^*\) is all strings created by concatenating strings in the language \(L^*\).
But those strings are themselves composed of strings from \(L\).
Thus the string \((L^*)^*\) is also a concatenation of strings from \(L\) and therefore in the language of \(L^*\).

\(\emptyset^* = \varepsilon\)
The closure of \(\emptyset\) contains only the string \(\varepsilon\).

\(\varepsilon^* = \varepsilon\)
The only string that can be formed by concatenating any number of copies of the empty string is the empty string itself.

\(L^+ = LL^* = L^*L\)
\(L^+\) is defined to be \(L + LL + LLL + \ldots\)
Also \(L^* = \varepsilon + L + LL + LLL + \ldots\)
\(LL^* = L \varepsilon + LL + LLL + LLLL + \ldots\)

\((L + M)^* = (L^* M^*)^*\)
This law says that if we have any two languages \(L\) and \(M\), and we close their union, we get the same language as if we take the language \(L^* M^*\), that is, all strings composed of zero or more choices from \(L\) followed by zero or more choices from \(M\), and close that language.
Example 5
We combined machines from Example 1 and Example 2 using \( \lambda \) transitions, where
\[ L(M1) = a^* \cup \lambda \] and \[ L(M3) = a^* bc \]

The new NFA- \( \lambda \) processed the language \( L(M4) = (a^* \cup \lambda) \cup (a^* bc) \)

The Algebraic laws for Regular Expressions allow us to simplify \( L(M4) \).
\[ L(M4) = L(M1) \cup L(M3) \]
\[ L(M4) = L(M1 \cup M3) \]
\[ L(M4) = ((a^* \cup \lambda) \cup (a^* bc)) \]
\[ L(M4) = (a^* \cup a^* bc \cup \lambda) \]
\[ L(M4) = a^* \cup a^* (bc \cup \lambda) \]
\[ L(M4) = a^* (bc \cup \lambda) \]
Example 6

Given two languages $L(M_1) = (a \cup b)^* \, bb \, (a \cup b)^*$ and $L(M_2) = (b \cup ab)^* \, (a \cup \lambda)^*$
determine $L(M) = L(M_1) \cap L(M_2)$.

An input string is accepted only if it consists of a string from $L(M_1)$ concatenated with one from $L(M_2)$.
The lambda transition allows the computation to enter $M_2$ whenever a prefix of the input string is accepted by $M_1$. 
DFA Minimization

Definition
Let $M = (Q, \Sigma, \sigma, q_0, F)$ be a DFA.
States $q_i$ and $q_j$ are equivalent if $\sigma(q_i, u) \in F$ if, and only if, $\sigma(q_j, u) \in F$ for all $u \in \Sigma^*$.

Two states are equivalent are called indistinguishable.
Two states that are not equivalent are said to be distinguishable.

States $q_i$ and $q_j$ are distinguishable if there is a string $u$ such that $\sigma(q_i, u) \in F$ and $\sigma(q_j, u) \notin F$, or vice versa.
The dotted lines entering $q_i$ and $q_j$ indicate that the method of reaching a state is irrelevant; equivalence depends only upon computations from the state.
Theorem (Kleene)

A language $L$ is accepted by a DFA with alphabet $\Sigma$ if, and only if, $L$ is a regular set over $\Sigma$.

$L$ is a regular language if and only if $L$ is an NFA language.

$L$ is a regular language if and only if $L$ is a DFA language.

Corollary

If $M$ is an NFA, then $M$ accepts a regular language.

Lemma

Every regular language is an NFA language.

Definition Equivalence

Let $L$ be a language. Two strings $x_1$ and $x_2$ are prefix equivalent with respect to $L$ if

$$(\forall u \in \Sigma^*) [ x_1 u \in L \iff x_2 u \in L ]$$

Note:

If $x_1$ and $x_2$ are two substrings and they are not equivalent (the contents of the substrings are different), then the DFA that accepts these two substrings must have at least two different sets of control states to process these strings.
Equivalence of NFA and DFA

- Every deterministic program for machine M is also a non-deterministic program for M.
- Non-deterministic programs are at least as powerful as deterministic programs.
- Any NFA with \( n \) control states can be simulated by a DFA with \( 2^n \) control states.
- The exponential growth is unavoidable.
- The ability of DFA to simulate NFA is important. It allows us to define regular languages more precisely.
- Regular Languages are exactly the set of DFA languages and, because DFA (acceptors) and DFR (recognizers) are equivalent, they are exactly the set of DFR languages.

Minimizing DFRs

- Computer Scientists put a lot of effort into minimizing the use of physical resources and time.
- We are always trying to optimize programs that run as fast as possible and use as little memory as possible. Reducing physical requirements such as memory, and the number of processors required reduces the overall weight of a machine. This reduces cost, and for flight vehicles it also reduces fuel requirements.
- We say that a program for a DFR is minimal if no DFR for L has fewer control states than P.
- Standardize - Eliminate null instructions and EOF tests. The only instruction performed should be a Scan on the input device/ string. Eliminate unreachable states. All states are final states.
- Merge equivalent states - Every program that accepts \( L(P) \) has at least as many control states as \( P \) has inequivalent states. Merging equivalent states into a single state produces the minimal program that is equivalent to \( P \).
The Myhill-Nerode Theorem

Kleene’s theorem established the relationship between regular languages and finite automata. In this section regularity is characterized by the existence of an equivalence relation on the strings of the language.

**Definition**

Let $L$ be a language over $\Sigma$. Strings $u, v \in \Sigma^*$ are indistinguishable in $L$ if, for every $w \in \Sigma^*$, either both $uw$ and $vw$ are in $L$ or neither $uw$ nor $vw$ is in $L$.

Using membership in $L$ as the criterion for differentiating strings, $u$ and $v$ are distinguishable if there is some string $w$ whose concatenation with $u$ and $v$ is sufficient to produce strings with different membership values in the language $L$.

**Example**

Let $L$ be the regular language $a (a \cup b) (bb)^*$. Strings $aa$ and $ab$ are indistinguishable since $aaw$ and $abw$ are in $L$ if, and only if, $w$ consists of an even number of $b$'s.

The pair of strings $b$ and $ba$ are also indistinguishable in $L$ since $bw$ and $baw$ are not in $L$ for any string $w$.

Strings $a$ and $ab$ are distinguishable in $L$ since concatenating $bb$ to $a$ produces $abb \notin L$ and to $abbb \in L$.

The equivalence classes of $L$ are:

<table>
<thead>
<tr>
<th>Representative Element</th>
<th>Equivalence Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\lambda]$</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>$[a]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$[aa]$</td>
<td>$a (a \cup b) (bb)^*$</td>
</tr>
<tr>
<td>$[aab]$</td>
<td>$a (a \cup b) b (bb)^*$</td>
</tr>
</tbody>
</table>
The Myhill-Nerode Theorem

Example
Let M be the DFA that accepts the language \(a^* ba^* (ba^* ba^*)^*\), the set of strings with an odd number of b's.

The equivalence classes of \(L\) are:

<table>
<thead>
<tr>
<th>State</th>
<th>Equivalence Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(a^*)</td>
</tr>
<tr>
<td>B</td>
<td>(a^* ba^* (ba^* ba^<em>)^</em>)</td>
</tr>
<tr>
<td>C</td>
<td>(a^* ba^* ba^* ((ba^* ba^<em>)^</em>)</td>
</tr>
</tbody>
</table>
Deterministic Finite Automata

- We have introduced the class of machines known as deterministic finite automata and have discussed their association with lexical analyzers. Example: Deciding a Floating Point Number.

- We determined the extend to which a string recognition algorithm can be constructed. Does a transition table provide the flexibility for general string processing?

- A deterministic finite automaton divides the class of all input strings into two groups: those strings that are acceptable and those strings that are not.

- If $M$ is the deterministic finite automaton $(S, \Sigma, \delta, q_1, F)$ the collection of strings it accepts constitutes a language over $\Sigma$. We denote this language by $L(M)$, which is read as “the language accepted by $M$”.

- $L(M)$ is not merely any collection of strings accepted by $M$, but the collection of all strings accepted by $M$, no more no less.

- A language of the form $L(M)$ for some finite automaton $M$ is called a regular language.

- Special sets of regular languages is the collection of all strings of finite length over an alphabet $\Sigma$, which we presented as $\Sigma^*$, and the collection containing no strings, known as the empty language, which we represent by $\emptyset$.

- An additional example of a regular language is the language consisting of the empty string, denoted by $\{\lambda\}$. Please note the difference between $\emptyset$ and $\{\lambda\}$. $L(M) = \{\lambda\}$ consist of one string and $L(M) = \emptyset$ consist of no string.
Theorem

For any alphabet $\Sigma$, there is a language that is not equal to $L(M)$ for any deterministic finite automaton $M$.

Proof

Since any arc in a deterministic finite automaton that is labeled by a symbol outside of $\Sigma$ would have no effect on the processing of a string in $\Sigma^*$, we can consider only machines with alphabet $\Sigma$.

But the collection of deterministic finite automata with alphabet $\Sigma$ is countable since we can systematically list all possible machines with one state, followed by all machines with two states, followed by those with three states, etc.

On the other hand, the number of languages over alphabet $\Sigma$ is uncountable since the infinite set $\Sigma^*$ has an uncountable number of subsets.

Thus, there are more languages than there are deterministic finite automata. Consequently each deterministic finite automaton accepts only one language, there must be languages that are not accepted by any such machine.
Theorem (Pumping Theorem)
If a language contains strings of the form $x^n y^n$ for arbitrarily large integers $n$, then it must contain strings of the form $x^m y^n$ where $m$ and $n$ are not equal.

Proof
Suppose $M$ is a deterministic finite automaton such that $L(M)$ contains $x^n y^n$ for arbitrarily large $n$. Then, there must be a positive integer $k$ that is larger than the number of states in $M$ and such that $x^k y^k$ is in $L(M)$. Since there are more symbols in $x^k$ than there are states in $M$, the process of accepting $x^k y^k$ will result in some state $M$ being traversed more than once before any $y$'s in the string are reached (note: $x$ is processed first). That is, in reading some of the $x$'s, a circular path will be traversed in the machine's transition diagram. If $j$ is the number of $x$'s read while traversing this path, then the machine can accept the string $x^{k+j} y^k$ by traversing this path an extra time. Hence, there is a positive integer $m$, namely $k + j$, that is not equal to $k$, such that $x^m y^k$ is in $L(M)$.

Discussion:
The immediate consequence is that $L(M) = \{ x^n y^n : n \in \mathbb{N} \}$ is not a regular language!

The consequence of $L(M) = \{ x^n y^n : n \in \mathbb{N} \}$ not being a regular language is that expressions $((a+b)/c)$ and $(a+(b/c))$ cannot be processed (lexical analysis) by a DFA. The DFA is required to count the number of left brackets and right brackets in the expression. We do not want to accept this string if the count of left and right brackets is different.

For the expression $(a+b)/c$, let the left bracket “(“ be represented by the symbol $x$, and the right bracket “)” be represented by symbol $y$. $n$ represents the number of occurrences of symbols $x$ and $y$, and $u$ represents the remainder of the string “$a+b/c$”.

Then the expression can be shown as
$$S = x^n \ u \ y^n$$

By the definition of the pumping theorem $L(M) = x^n \ u \ y^n$ is not a regular language.
Turing Machines (TM)

- Generalize the class of CFLs:
Another Part of the Hierarchy:

- Regular Languages
- Context-Free Languages - $\lambda$
- Context-Sensitive Languages
- Recursive Languages
- Recursively Enumerable Languages
- Non-Recursively Enumerable Languages
• TMs model the computing capability of a general purpose computer, which informally can be described as:
  • Effective procedure
    • Finitely describable
    • Well defined, discrete, “mechanical” steps
    • Always terminates
  • Computable function
    • A function computable by an effective procedure

• TMs formalize the above notion.

• **Church-Turing Thesis:** There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.
  • There are many other computing models, but all are equivalent to or subsumed by TMs. *There is no more powerful machine* (Technically cannot be proved).

• DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.
Deterministic Turing Machine (DTM)

- One-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- A string is placed on the tape, right padded infinitely with blanks, read/write head is positioned at the left end.
- In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its’ tape head one cell left or right.
- Many modifications possible.
Formal Definition of a DTM

- A DTM is a seven-tuple:

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \]

- **Q** : A finite set of states
- **\Gamma** : A finite tape alphabet
- **B** : A distinguished blank symbol, which is in \( \Gamma \)
- **\Sigma** : A finite input alphabet, which is a subset of \( \Gamma \) – \{B\}
- **q_0** : The initial/starting state, \( q_0 \) is in \( Q \)
- **F** : A set of final/accepting states, which is a subset of \( Q \)
- **\delta** : A next-move function, which is a mapping (i.e., may be undefined) from \( Q \times \Gamma \) to \( Q \times \Gamma \times \{L,R\} \)

Intuitively, \( \delta(q,s) \) specifies the next state, symbol to be written, and the direction of tape head movement by \( M \) after reading symbol \( s \) while in state \( q \).
**Example #1:** \( \{ 0^n1^n \mid n \geq 1 \} \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Y</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( (q_1, X, R) )</td>
<td>( - )</td>
<td>( - )</td>
<td>( (q_3, Y, R) )</td>
<td>( - )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( (q_1, 0, R) )</td>
<td>( (q_2, Y, L) )</td>
<td>( - )</td>
<td>( (q_1, Y, R) )</td>
<td>( - )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( (q_2, 0, L) )</td>
<td>( - )</td>
<td>( (q_0, X, R) )</td>
<td>( (q_2, Y, L) )</td>
<td>( - )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( - )</td>
<td>( - )</td>
<td>( - )</td>
<td>( (q_3, Y, R) )</td>
<td>( (q_4, B, R) )</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>( - )</td>
<td>( - )</td>
<td>( - )</td>
<td>( - )</td>
<td>( - )</td>
</tr>
</tbody>
</table>

**Sample Computation:** (on 0011)

\[
q_0 \quad 0011 \quad \mid \quad Xq_1 \quad 011 \\
\mid \quad X0q_1 \quad 11 \\
\mid \quad Xq_2 \quad 0Y1 \\
\mid \quad q_2X0Y1 \\
\mid \quad Xq_0 \quad 0Y1 \\
\mid \quad XXq_1 \quad Y1 \\
\mid \quad XXYq_1 \quad 1 \\
\mid \quad XXq_2 \quad YY \\
\mid \quad Xq_2XYY \\
\mid \quad XXq_0 \quad YY \\
\mid \quad XXYq_3 \quad Y \\
\mid \quad XXYYq_3 \\
\mid \quad XXYYBq_4
\]
Example #1: \( \{0^n1^n \mid n \geq 1 \} \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Y</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( (q_1, X, R) )</td>
<td>-</td>
<td>-</td>
<td>(q_3, Y, R)</td>
<td>-</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( (q_1, 0, R) )</td>
<td>(q_2, Y, L)</td>
<td>-</td>
<td>(q_1, Y, R)</td>
<td>-</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( (q_2, 0, L) )</td>
<td>-</td>
<td>(q_0, X, R)</td>
<td>(q_2, Y, L)</td>
<td>-</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(q_3, Y, R)</td>
<td>(q_4, B, R)</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- The TM basically matches up 0’s and 1’s
- \( q_1 \) is the “scan right” state
- \( q_2 \) is the “scan left” state
- \( q_4 \) is the final state

Other Examples:

- 000111
- 00
- 11
- 001
- 011
Example #2: \{w \mid w \text{ is in } \{0, 1\}^* \text{ and } w \text{ ends with a } 0\}

0
00
10
10110
Not \lambda

Q = \{q_0, q_1, q_2\}
\Gamma = \{0, 1, B\}
\Sigma = \{0, 1\}
F = \{q_2\}
\delta:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_0</td>
<td>(q_0, 0, R)</td>
<td>(q_0, 1, R)</td>
<td>(q_1, B, L)</td>
</tr>
<tr>
<td>q_1</td>
<td>(q_2, 0, R)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>q_2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- q_0 is the “scan right” state
- q_1 is the verify 0 state
Formal Definitions for DTMs

- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM.

- **Definition:** An *instantaneous description* (ID) is a triple $\alpha_1q\alpha_2$, where:
  - $q$, the current state, is in $Q$
  - $\alpha_1\alpha_2$, is in $\Gamma^*$, and is the current tape contents up to the rightmost non-blank symbol, or the symbol to the left of the tape head, whichever is rightmost
  - The tape head is currently scanning the first symbol of $\alpha_2$
  - At the start of a computation $\alpha_1 = \lambda$
  - If $\alpha_2 = \lambda$ then a blank is being scanned
• Suppose the following is the current ID of a DTM

\[ x_1x_2 \ldots x_{i-1}qx_ix_{i+1} \ldots x_n \]

Case 1) \( \delta(q, x_i) = (p, y, L) \)

(a) if \( i = 1 \) then the computation terminates

(b) else \( x_1x_2 \ldots x_{i-1}qx_ix_{i+1} \ldots x_n \mid \quad x_1x_2 \ldots x_{i-2}px_ix_{i+1} \ldots x_n \)

• If any suffix of \( x_{i-1}yx_{i+1} \ldots x_n \) is blank then it is deleted.

Case 2) \( \delta(q, x_i) = (p, y, R) \)

\[ x_1x_2 \ldots x_{i-1}qx_ix_{i+1} \ldots x_n \mid \quad x_1x_2 \ldots x_{i-1}yx_{i+1} \ldots x_n \]

• If \( i > n \) then the ID increases in length by 1 symbol

\[ x_1x_2 \ldots x_{n-1}q \mid \quad x_1x_2 \ldots x_nyp \]
- **Definition:** Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \) be a TM, and let \( w \) be a string in \( \Sigma^* \). Then \( w \) is accepted by \( M \) iff

\[
q_0w \xrightarrow{\ast} \alpha_1p\alpha_2
\]

where \( p \) is in \( F \) and \( \alpha_1 \) and \( \alpha_2 \) are in \( \Gamma^* \)

- **Definition:** Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \) be a TM. The *language accepted by \( M \)*, denoted \( L(M) \), is the set

\[
\{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}
\]

- **Notes:**
  - In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.
  - Given the above definition, no final state of an TM need have any exiting transitions. *Henceforth, this is our assumption.*
  - If \( x \) is not in \( L(M) \) then \( M \) may enter an infinite loop, or halt (i.e., terminate in a non-final state, or go off the left end of the tape).
  - Some TMs halt on all inputs, while others may not. In either case the language defined by by TM is still well defined.
**Definition:** Let $L$ be a language. Then $L$ is *recursively enumerable* if there exists a TM $M$ such that $L = L(M)$.

- If $L$ is r.e. then $L = L(M)$ for some TM $M$, and
  - If $x$ is in $L$ then $M$ halts and accepts.
  - If $x$ is not in $L$ then $M$ may halt in a non-accepting state, loop forever, or move off the left end of the tape.

**Definition:** Let $L$ be a language. Then $L$ is *recursive* if there exists a TM $M$ such that $L = L(M)$ and $M$ halts on all inputs.

- If $L$ is recursive then $L = L(M)$ for some TM $M$, and
  - If $x$ is in $L$ then $M$ halts and accepts.
  - If $x$ is not in $L$ then $M$ halts and rejects (i.e., halts in a non-accepting state, or goes off the left end of the tape).

**Notes:**

- The set of all recursive languages is a subset of the set of all recursively enumerable languages.
- Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.
Recall the Hierarchy:

- Regular Languages
- Context-Free Languages - $\lambda$
- Context-Sensitive Languages
- Recursive Languages
- Recursively Enumerable Languages
- Non-Recursively Enumerable Languages
• **Observation:** Let L be an r.e. language. Then there is an infinite list $M_0, M_1, \ldots$ of TMs such that $L = L(M_i)$.

• **Question:** Let L be a recursive language, and $M_0, M_1, \ldots$ a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does $M_i$ always halt?

• **Answer:** Maybe, maybe not, but at least one in the list does.

• **Question:** Let L be a recursive enumerable language, and $M_0, M_1, \ldots$ a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does $M_i$ always halt?

• **Answer:** Maybe, maybe not. Depending on L, none might halt or some may halt.
  - If L is also recursive then L is recursively enumerable.

• **Question:** Let L be a recursive enumerable language that is not recursive (L is in r.e. – r), and $M_0, M_1, \ldots$ a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does $M_i$ always halt?

• **Answer:** No! If it did, then L would not be in r.e. – r, it would be recursive.
• Let $M$ be a TM.

  • Question: Is $L(M)$ r.e.?
  • Answer: Yes! By definition it is!

  • Question: Is $L(M)$ recursive?
  • Answer: Don’t know, we don’t have enough information.

  • Question: Is $L(M)$ in r.e – r?
  • Answer: Don’t know, we don’t have enough information.
• Let $M$ be a TM that halts on all inputs:

  • Question: Is $L(M)$ recursively enumerable?
  • Answer: Yes! By definition it is!

  • Question: Is $L(M)$ recursive?
  • Answer: Yes! By definition it is!

  • Question: Is $L(M)$ in r.e – r?
  • Answer: No! It can’t be. Since $M$ always halts, $L(M)$ is recursive.
• Let M be a TM.

  • As noted previously, L(M) is recursively enumerable, but may or may not be recursive.

  • Question: Suppose that L(M) is recursive. Does that mean that M always halts?
  • Answer: Not necessarily. However, some TM M' must exist such that L(M') = L(M) and M' always halts.

  • Question: Suppose that L(M) is in r.e. – r. Does M always halt?
  • Answer: No! If it did then L(M) would be recursive and therefore not in r.e. – r.
Let $M$ be a TM, and suppose that $M$ loops forever on some string $x$.

- Question: Is $L(M)$ recursively enumerable?
  - Answer: Yes! By definition it is.

- Question: Is $L(M)$ recursive?
  - Answer: Don’t know. Although $M$ doesn’t always halt, some other TM $M'$ may exist such that $L(M') = L(M)$ and $M'$ always halts.

- Question: Is $L(M)$ in r.e. – r?
  - Answer: Don’t know.
Modifications of the Basic TM Model

- **Other (Extended) TM Models:**
  - Two-way infinite tapes
  - Multiple tapes and tape heads
  - Non-Deterministic TMs
  - Multi-Dimensional TMs (n-dimensional tape)
  - Multi-Heads
  - Multiple tracks

_All of these extensions are equivalent to the basic TM model_
Closure Properties for Recursive and Recursively Enumerable Languages

- **TMs Model General Purpose Computers:**
  - If a TM can do it, so can a GP computer
  - If a GP computer can do it, then so can a TM

*If you want to know if a TM can do X, then an equivalent question is, can a general purpose computer do X.*
- **TM Block Diagrams:**
  - If $L$ is a recursive language, then a TM $M$ that accepts $L$ and always halts can be pictorially represented by a “chip” that has one input and two outputs.

```
  w   →   M   →   yes
      |       |      |
      v       v      
o
```

- If $L$ is a recursively enumerable language, then a TM $M$ that accepts $L$ can be pictorially represented by a “chip” that has one output.

```
  w   →   M   →   yes
```

- Conceivably, $M$ could be provided with an output for “no,” but this output cannot be counted on. Consequently, we simply ignore it.
Theorem: The recursive languages are closed with respect to complementation, i.e., if \( L \) is a recursive language, then so is \( \overline{L} = \Sigma^* \overline{\epsilon} \).

Proof: Let \( M \) be a TM such that \( L = L(M) \) and \( M \) always halts. Construct TM \( M' \) as follows:

Note That:
- \( M' \) accepts iff \( M \) does not
- \( M' \) always halts since \( M \) always halts

From this it follows that the complement of \( L \) is recursive. \( \square \)

Question: How is the construction achieved? Do we simply complement the final states in the TM? No! A string in \( L \) could end up in the complement of \( L \).
- Suppose \( q_5 \) is an accepting state in \( M \), but \( q_0 \) is not.
- If we simply complemented the final and non-final states, then \( q_0 \) would be an accepting state in \( M' \) but \( q_5 \) would not.
- Since \( q_0 \) is an accepting state, by definition all strings are accepted by \( M' \)
• **Theorem:** The recursive languages are closed with respect to union, i.e., if \( L_1 \) and \( L_2 \) are recursive languages, then so is 
\[
L_3 = L_1 \cup L_2
\]

• **Proof:** Let \( M_1 \) and \( M_2 \) be TMs such that \( L_1 = L(M_1) \) and \( L_2 = L(M_2) \) and \( M_1 \) and \( M_2 \) always halt. Construct TM \( M' \) as follows:

![Diagram of TMs](image)

• **Note That:**
  - \( L(M') = L(M_1) \cup L(M_2) \)
  - \( L(M') \) is a subset of \( L(M_1) \cup L(M_2) \)
  - \( L(M_1) \cup L(M_2) \) is a subset of \( L(M') \)
  - \( M' \) always halts since \( M_1 \) and \( M_2 \) always halt

It follows from this that \( L_3 = L_1 \cup L_2 \) is recursive. \( \square \)
**Theorem:** The recursive enumerable languages are closed with respect to union, i.e., if \( L_1 \) and \( L_2 \) are recursively enumerable languages, then so is \( L_3 = L_1 \cup L_2 \).

**Proof:** Let \( M_1 \) and \( M_2 \) be TMs such that \( L_1 = L(M_1) \) and \( L_2 = L(M_2) \). Construct \( M' \) as follows:

![Diagram of two TMs running in parallel](image)

- **Note That:**
  - \( L(M') = L(M_1) \cup L(M_2) \)
  - \( L(M') \) is a subset of \( L(M_1) \cup L(M_2) \)
  - \( L(M_1) \cup L(M_2) \) is a subset of \( L(M') \)
  - \( M' \) halts and accepts iff \( M_1 \) or \( M_2 \) halts and accepts

It follows from this that \( L_3 = L_1 \cup L_2 \) is recursively enumerable. \( \square \)

**Question:** How do you run two TMs in parallel?
Suppose $M_1$ and $M_2$ had outputs for “no” in the previous construction, and these were transferred to the “no” output for $M'$

**Question:** What would happen if $w$ was in $L(M_1)$ but not in $L(M_2)$?

**Answer:** You could get two outputs – one “yes” and one “no.”

- This is not an argument that “no” outputs should not be provided for a TM accepting an r.e. language, but rather just an indication that more complex output logic is necessary.
- As before, for the sake of convenience the “no” output will be ignored.
• **Theorem:** If \( L \) and \( \overline{L} \) are both recursively enumerable then \( L \) (and therefore \( \overline{L} \)) is recursive.

• **Proof:** Let \( M_1 \) and \( M_2 \) be TMs such that \( L = L(M_1) \) and \( \overline{L} = L(M_2) \). Construct \( M' \) as follows:

![Diagram](image)

• **Note That:**
  - \( L(M') = L \)
    - \( L(M') \) is a subset of \( L \)
    - \( L \) is a subset of \( L(M') \)
  - \( M' \) always halts since \( M_1 \) or \( M_2 \) halts for any given string

It follows from this that \( L \) (and therefore its’ complement) is recursive. \( \square \)
• **Corollary:** Let $L$ be a subset of $\Sigma^*$. Then one of the following must be true:

  - Both $L$ and $\overline{L}$ are recursive.
  - One of $L$ and $\overline{L}$ is recursively enumerable but not recursive, and the other is not recursively enumerable, or
  - Neither $L$ nor $\overline{L}$ is recursively enumerable,
In terms of the hierarchy: (possibility #1)
• **In terms of the hierarchy:** (possibility #2)
In terms of the hierarchy: (possibility #3)
• **In terms of the hierarchy:** (Impossibility #1)

Non-Recursively Enumerable Languages

\[ L \quad \bar{L} \]

Recursively Enumerable Languages

Recursive Languages
• **In terms of the hierarchy:** (Impossibility #2)

Non-Recursively Enumerable Languages

\[
\overline{L}
\]

Recursively Enumerable Languages

\[
L
\]

Recursive Languages
- **In terms of the hierarchy**: (Impossibility #3)
• **Note:** This gives/identifies three approaches to show that a language is not recursive.
  • Show that the language’s complement is not recursive
  • Show that the language’s complement is recursively enumerable but not recursive
  • Show that the language’s complement is not recursively enumerable
The Halting Problem - Background

- **Definition:** A decision problem is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.
  - Given a list of numbers, is that list sorted?
  - Given a number x, is x even?
  - Given a C program, does that C program contain any syntax errors?
  - Given a TM (or C program), does that TM contain an infinite loop?

From a practical perspective, many decision problems do not seem all that interesting. However, from a theoretical perspective they are for the following two reasons:
  - Decision problems are more convenient/easier to work with when proving complexity results.
  - Non-decision counter-parts are typically at least as difficult to solve.

- **Notes:**
  - The following terms and phrases are analogous:

    | Algorithm          | A halting TM program |
    | Decision Problem   | A language           |
    | (un)Decidable      | (non)Recursive       |
Statement of the Halting Problem

- **Practical Form:** (P1)
  Input: Program P and input I.
  Question: Does P terminate on input I?

- **Theoretical Form:** (P2)
  Input: Turing machine M with input alphabet $\Sigma$ and string w in $\Sigma^*$.
  Question: Does M halt on w?

- **A Related Problem We Will Consider First:** (P3)
  Input: Turing machine M with input alphabet $\Sigma$ and one final state, and string w in $\Sigma^*$.
  Question: Is w in L(M)?

- **Analogy:**
  Input: DFA M with input alphabet $\Sigma$ and string w in $\Sigma^*$.
  Question: Is w in L(M)?

  Is this problem decidable? Yes!
• **Over-All Approach:**

  • We will show that a language $L_d$ is not recursively enumerable
  • From this it will follow that $\overline{L}_d$ is not recursive
  • Using this we will show that a language $L_u$ is not recursive
  • From this it will follow that the halting problem is undecidable.

• **As We Will See:**

  • P3 will correspond to the language $L_u$
  • Proving P3 (un)decidable is equivalent to proving $L_u$ (non)recursive
Converting the Problem to a Language

- Let $M = (Q, \Sigma, \Gamma, \delta, q_1, B, \{q_n\})$ be a TM, where

  \[ Q = \{q_1, q_2, \ldots, q_n\} \]
  \[ \Sigma = \{x_1, x_2\} = \{0, 1\} \]
  \[ \Gamma = \{x_1, x_2, x_3\} = \{0, 1, B\} \]

- Encode:

  \[ \delta(q_i, x_j) = (q_k, x_l, d_m) \]
  where $q_i$ and $q_k$ are in $Q$
  $x_j$ and $x_l$ are in $\Sigma$, 
  and $d_m$ is in $\{L, R\} = \{d_1, d_2\}$

  as:

  \[ 0^i10^j10^k10^l10^m \]

- The TM $M$ can then be encoded as:

  \[ 111\text{code}_111\text{code}_211\text{code}_311 \ldots 11\text{code}_r111 \]

  where each code $i$ is one transitions’ encoding. Let this encoding of $M$ be denoted by $<M>$. 
Less Formally:

Every state, tape symbol, and movement symbol is encoded as a sequence of 0’s:

\[ q_1, \quad 0 \]
\[ q_2, \quad 00 \]
\[ q_3, \quad 000 \]

\[ \vdots \]

\[ 0, \quad 0 \]
\[ 1, \quad 00 \]
\[ B, \quad 000 \]

\[ L, \quad 0 \]
\[ R, \quad 00 \]

Note that 1’s are not used to represent the above, since 1 is used as a special separator symbol.

Example:

\[ \delta(q_2, 1) = (q_3, 0, R) \]

Is encoded as:

00100100010100
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$(q_1, 0, R)$</td>
<td>$(q_1, 1, R)$</td>
<td>$(q_2, B, L)$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$(q_3, 0, R)$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$q_3$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

$11101010100110100101000100100010100010110010100110010100010011001010001010011001010001001100101000100110111$

$011000001110001$

$111111$

$111111$
• **Definition:**

\[ L_t = \{ x \mid x \text{ is in } \{0, 1\}^* \text{ and } x \text{ encodes a TM} \} \]

• Question: Is \( L_t \) recursive?
• Answer: Yes.

• Question: Is \( L_t \) decidable:
• Answer: Yes (same question).

• **Definition:** (similarly)

\[ L_{df} = \{ x \mid x \text{ is in } \{0, 1\}^* \text{ and } x \text{ encodes a DFA} \} \]

• Question: Is \( L_{df} \) recursive?
• Answer: Yes.

• Question: Is \( L_{df} \) decidable:
• Answer: Yes (same question).
The Universal Language

- Define the language $L_u$ as follows:
  \[ L_u = \{ x \mid x \text{ is in } \{0,1\}^* \text{ and } x = <M,w> \text{ where } M \text{ is a TM encoding and } w \text{ is in } L(M) \} \]

- Let $x$ be in $\{0,1\}^*$. Then either:
  1. $x$ doesn’t have a TM prefix, in which case $x$ is not in $L_u$
  1. $x$ has a TM prefix, i.e., $x = <M,w>$ and either:
     a) $w$ is not in $L(M)$, in which case $x$ is not in $L_u$
     a) $w$ is in $L(M)$, in which case $x$ is in $L_u$
Recall:

\[
\begin{array}{c|c|c|c}
& 0 & 1 & B \\
q_1 & (q_1, 0, R) & (q_1, 1, R) & (q_2, B, L) \\
q_2 & (q_3, 0, R) & - & - \\
q_3 & - & - & - \\
\end{array}
\]

Which of the following are in \( L_u \)?

1110101010011010010101100100110010010100100110110010100111001

111111

01100001110001

111111
• **Compare P3 and L_u:**

(P3):
Input: Turing machine M with input alphabet $\Sigma$ and one final state, and string $w$ in $\Sigma^*$. 
Question: Is $w$ in $L(M)$?

$L_u = \{x \mid x \text{ is in } \{0, 1\}^* \text{ and } x = \langle M, w \rangle \text{ where } M \text{ is a TM encoding and } w \text{ is in } L(M)\}$

• **Notes:**
  - $L_u$ is P3 expressed as a language
  - Asking if $L_u$ is recursive is the same as asking if P3 is decidable.
  - We will show that $L_u$ is not recursive, and from this it will follow that P3 is un-decidable.
  - From this we can further show that the halting problem is un-decidable.
  - Note that $L_u$ is recursive if $M$ is a DFA.
• Define another language $L_d$ as follows:

\[ L_d = \{ x \mid x \text{ is in } \{0, 1\}^* \text{ and either } x \text{ is not a TM or } x \text{ is a TM, call it } M, \]
\[ \text{and } x \text{ is not in } L(M) \} \tag{1} \]

• Let $x$ be in $\{0, 1\}^*$. Then either:

1. $x$ is not a TM, in which case $x$ is in $L_d$

1. $x$ is a TM, call it $M$, and either:

   a) $x$ is not in $L(M)$, in which case $x$ is in $L_d$

   a) $x$ is in $L(M)$, in which case $x$ is not in $L_d$
- **Recall:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁</td>
<td>(q₁, 0, R)</td>
<td>(q₁, 1, R)</td>
<td>(q₂, B, L)</td>
</tr>
<tr>
<td>q₂</td>
<td>(q₃, 0, R)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>q₃</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **Which of the following are in L_d?**

1110101010100110100101001001101000100010001011001010001010011101100011110001

01100001110001

111111
Lemma: $L_d$ is not recursively enumerable:

Proof: (by contradiction)
Suppose that $L_d$ were recursively enumerable. In other words, that there existed a TM $M$ such that:

$$L_d = L(M) \quad (2)$$

Now suppose that $w_j$ is a string encoding of $M$.

Case 1) $w_j$ is in $L_d$ \hspace{1cm} (3)

By definition of $L_d$ given in (1), either $w_j$ does not encode a TM, or $w_j$ does encode a TM, call it $M$, and $w_j$ is not in $L(M)$. But we know that $w_j$ encodes a TM (that’s were it came from). Therefore:

$$w_j \text{ is not in } L(M) \quad (4)$$

But then (2) and (4) imply that $w_j$ is not in $L_d$ contradicting (3).

Case 2) $w_j$ is not in $L_d$ \hspace{1cm} (5)

By definition of $L_d$ given in (1), $w_j$ encodes a TM, call it $M$, and:

$$w_j \text{ is in } L(M) \quad (6)$$

But then (2) and (6) imply that $w_j$ is in $L_d$ contradicting (5).

Since both case 1) and case 2) lead to a contradiction, no TM $M$ such that $L_d = L(M)$ can exist. Therefore $L_d$ is not recursively enumerable. □
• **Note:**

\[ \overline{L_d} = \{x \mid x \text{ is in } \{0, 1\}^*, x \text{ encodes a TM, call it } M, \text{ and } x \text{ is in } L(M)\} \]

\[ \overline{L_d} \]

• **Corollary:** \( \overline{L_d} \) is not recursive.

• **Proof:** If \( \overline{L_d} \) were recursive, then \( L_d \) would be recursive, and therefore recursively enumerable, a contradiction. \( \square \)
Theorem: $L_u$ is not recursive.

Proof: (by contradiction)

Suppose that $L_u$ is recursive. Recall that:

$$L_u = \{x \mid x \text{ is in } \{0, 1\}^* \text{ and } x = <M, w> \text{ where } M \text{ is a TM encoding and } w \text{ is in } L(M)\}$$

Suppose that $L_u = L(M')$ where $M'$ is a TM that always halts. Construct an algorithm (i.e., a TM that always halts) for $L_d$ as follows:

Suppose that $M'$ always halts and $L_u = L(M')$. It follows that:

- $M''$ always halts
- $L(M'') = \overline{L_d}$

$\overline{L_d}$ would therefore be recursive, a contradiction. $\Box$
The over-all logic of the proof is as follows:

1. If \( L_u \) is recursive, then so is \( \overline{L_d} \)

2. \( \overline{L_d} \) is not recursive

3. It follows that \( L_u \) is not recursive.

The second point was established by the corollary.
The first point was established by the theorem on the preceding slide.