COSC 312 Course Workbook Spring 2022

Instructor: Dr. M.W. Berry

Created by Dr. Michael W. Berry

Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville

January 25th, 2022 – May 10th, 2022

Contents

1 Theory of Computation, Introduction

Problem solving has three components: **unknowns**, **data**, and **conditions**. Solving a problem means equates to finding a way of determining the unknowns from given data such that conditions of the problem are satisfied.

Example: Find the remainder of $n/5$. unknown: int r, data: int n, condition: $n \bmod 5 = r$.

The traditional areas of the Theory of Computation (TOC):

- **Automata** – provide problem solving devices.
- **Computability** – provide framework that can characterize devices by their computing power.
- **Complexity** – provide framework to classify problems according to time/space complexity of the tools used to solve them.

Let's briefly breakdown each of these areas.

Automata:

- Abstraction of computing devices.
- How much memory can be used?
- What operations can be performed?

Computability:

- Study different computing models and identify the most powerful ones.
- Range of problems: very simple (solved by the simplest to most most powerful models) to very complex (solution cannot be provided by even the most powerful models).
- Problems can be undecidable or uncomputable. **Example**: determine if a procedure will terminate on a given input (Can any algorithm do this?).

Complexity:

- Computing problems range from easy to hard; sorting is easier than scheduling.
- **Q**: What makes some problems computationally hard and others easy?

Problem Abstraction:

- **Data** - abstracted as a **word** in a given **alphabet**.
- **Conditions** - abstracted as a set of words called a **language**.
- **Unknowns** - A Boolean variable: **true** if a word is in the language or **false** otherwise.

Example: All positive numbers divisible by 5: $S = \{5, 10, 15, \ldots\}$.

Abstraction of Data:

- Σ: alphabet – a finite, nonempty set of symbols.
- Σ^*: all words of finite length built up using Σ.
- Rules:(1) the empty word (ϵ) is in Σ^*; (2) if $w \in \Sigma^*$and $a \in \Sigma$, then $aw \in \Sigma^*$, and (3) nothing else is in Σ^*.

Example: If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots\}$.

Sets:

- A **set** is a collection of objects that is represented as a unit; objects in a set are called **elements** or **members**.
- Sets can be described formally by enumerating their elements or by providing a **property** satisfied by all the elements.

Example: $S = \{x | x \text{ is an integer divisible by } 5\}$.

Subsets:

- Given sets A and B, $A = B$ if every member of A is a member of B and vice-versa.
- For sets A and B, $A \subseteq B$ if every member of A is a member of B and A could be equal to B.
- A is a proper subset of B (i.e., $A \subset B$) if A is a subset of B and is **not** equal to B.

Infinite and Empty Sets:

- An infinite set contains infinitely many elements. You cannot write a list of all elements of an infinite set; typically define by $S = \{e | p(e)\}$.
- A set with no (zero) members is called the empty set and is denoted by $\emptyset$.

Operations with Sets:

- Given sets A and B, $A \cup B = \{x | x \in A \vee x \in B\}$.
- Given sets A and B, $A \cap B = \{x | x \in A \wedge x \in B\}$, and $A – B = \{x | x \in A \wedge x \notin B\}$. Let's draw a Venn diagram for $A \cap B$ and $A – B$.

- The **Power set** of a set A is defined as $\mathcal{P}(A) = \{B | B \subseteq A\}$. Suppose $A = \{0, 1\}$. What is $\mathcal{P}(A)$?

Sequences:

- A **sequence** of objects is a list of objects in some order; the empty sequence denoted by ϵ.
- The **length** of a sequence $x = a_1, a_2, \ldots, a_n$ is defined by $|x| = n$.
- The **concatenation** of x and $y = b_1, b_2, \ldots, b_m$ is $x \cdot y = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$.

Tuples:

- Finite sequences are often called **tuples**.
- A sequence with k elements is called a k-tuple. **Example**: $(7, 21, 57)$ is a 3-tuple.
- A 2-tuple is normally called a **pair**.

Cartesian Product (CP):

- $A \times B$ = set of all pairs whose first element is a member of A and second element is a member of B.
- $A_1 \times A_2 \times \cdots \times A_k = \{A_1 \times A_2 \times \cdots A_{k-1} \times A_k\}$.
- $A^k = A \times A \times \cdots \times A$ (CP of k A's)

Functions:

- A **function** is a mathematical object that defines some input-output relationship.
- A function f takes an input and produces output: $f(a) = b$; the same input always produces the same output.
- A function is also called a **mapping**; if $f(a) = b$, we say f maps a to b.
- The **domain** is the set of possible inputs to function f, and the **range** is the set of possible outputs of function f.
- To denote that f is a function with domain D and range R, we write $f : D \to R$.

- When the domain of a function f is $A_1 \times A_2 \times \cdots \times A_k$, the input to f is a k-tuple $(a_1, a_2, \ldots, a_k)$ and a_i, $i = 1, 2, \ldots, k$, are called the **arguments** of f.
- A k-ary function f has k arguments (k called the **arity** of the function f); unary ($k = 1$), binary ($k = 2$).

Predicates (or Properties):

- A **predicate** (or property) is a function whose ranges is the set {**True, False**}.
  Example: even: $N \rightarrow$ {True, False},
  even(n) = True, if $n = 2k$ for some k;
  even(n) = False, if $n = 2k + 1$, for some k.
- A property whose domain is a set of k-tuples $A \times A \times \cdots A$ is called a **relation** (k-ary relation).

Boolean Operations:

- Boolean values are manipulated by Boolean operations.
- Negation (NOT, $\neg$): $\neg 0 = 1$, $\neg 1 = 0$.
- Conjunction (AND, $\wedge$): $0 \wedge 0 = 0$, $0 \wedge 1 = 0$, and $1 \wedge 1 = 1$.
- Disjunction (OR, $\vee$): $0 \vee 0 = 0$, $0 \vee 1 = 1$, $1 \vee 0 = 1$, and $1 \vee 1 = 1$.

Ordered Pairs:

- For x, y elements of a set, (x, y) is an **ordered pair** if (1) $(x, y) = (u, v) \Rightarrow x = u,\ y = v$, and (2) $x = u, y = v \Rightarrow (x, y) = (u, v)$.
- A definition of the ordered pair that satisfies (1) and (2) above is $(x, y) = \{\{x\}, \{x, y\}\}$.

Set Relations:

- A **relation** on a set A is a set of ordered pairs of elements of A.
  Example: $<$ on $\mathcal{N}$ where $<$ is the set relation $\{(x, y) | x, y \in \mathcal{N} \wedge x < y\}$.

- Notation: $x \, R \, y$ for a relation R.
- Properties of Relations:
  - **Transitive**: $x \, R \, y \wedge y \, R \, z \Rightarrow x \, R \, z$
  - **Reflexive**: $x \, R \, x$
  - **Symmetric**: $x \, R \, y \Rightarrow y \, R \, x$
  - $\text{dom}(R) = \{x \in A | \, \exists \, y \in A \wedge (x, y) \in R\}$ is the **domain** of the relation R.
  - $\text{ran}(R) = \{y \in A | \, \exists \, x \in A \wedge (x, y) \in R\}$ is the **range** of the relation R.

Special Relations:

- Total order: R is a **total order** on the set A if (1) R satisfies the trichotomy ($\forall x, y \in A$, $x = y$ or $x \, R \, y$ or $y \, R \, x$), and (2) R is transitive.
  Example: $<$ is total on $\mathcal{N}$.
- Given a function $f : X \rightarrow Y$, it defines the relation $<x,y>$ where $f(x) = y$; a relation R is a function if for any x, there exists a unique y so that $x \, R \, y$.
- An **equivalence relation** captures the notion of two objects being equal in some feature.
- A **binary relation** R is an equivalence relation if R satisfies:
  1. R is reflexive ($\forall x$, $x \, R \, x$),
  2. R is symmetric ($\forall x, y$, $x \, R \, y$ iff $y \, R \, x$), and
  3. R is transitive ($\forall x, y, z$, $x \, R \, y$ and $y \, R \, z \Rightarrow x \, R \, z$).

Strings:

- An **alphabet** is a finite, nonempty set of symbols.
- A **string** over an alphabet is a finite sequence of symbols from that alphabet (symbols concatenated).
- If w is a string over Σ, the length of w (denoted $|w|$) is the number of symbols contained in w.
- The string of length zero is called the **empty string** and is denoted by ϵ.

- If $|w| = n$, we can write $w = a_1 a_2 \dots a_n$, $a_i \in \Sigma$, $i = 1, 2, \dots, n$.

String Operations:

- The **reverse** of $w = w_1 w_2 \dots w_n$, written as w^R, is defined by $w^R = w_n \dots w_2 w_1$.
- A string z is a **substring** of w if $w = xzy$ for x, y not necessarily the empty strings; x is called the **prefix** of w and y is called the **suffix** of w.
- The **concatenation** of strings x and y, where $x = x_1 x_2 \dots x_m$ and $y = y_1 y_2 \dots y_n$, is defined as the string $xy = x_1 x_2 \dots x_m y_1 y_2 \dots y_n$.
- We define $x^k = xx \dots x$, where x is repeated k times.

---

2 Finite Automata

Formal Language:

- A **formal language** is a set of strings over a given alphabet.
- How do you specify a language?
- How do you recognize strings in a language?
- How do you translate the language?

Abstraction of Problems:

1. **Data** – word in a given alphabet.
   Σ: alphabet, a **finite** non-empty set of symbols.
   Σ^*: all words of **finite length** built-up using Σ.
2. **Conditions** – set of words called a **language**.
   Any subset $L \subseteq \Sigma^*$is a **formal language**.
3. **Unknown** – a **boolean variable** that is **true**, if word is in language; **false**, otherwise.
   Given $w \in \Sigma^*$and $L \subseteq \Sigma^*$, is $w \in L$?

Formal Definition:

- Simplest computational model also referred to as a **finite-state machine** or **finite automaton** (FA).
- Representations: graphical, tabular, and mathematical.
- A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols (alphabet), the transition function δ maps $Q \times \Sigma$ to Q, $q_0 \in Q$ is the start (initial) state, and $F \subseteq Q$ is the set of accept (final) states.

FA Applications:

- Parsers for compilers
- Pattern recognition
- Speech processing and OCR
- Financial planning and market prediction

FA Computation:

- Automaton M_1 receives input symbols one-by-one (left to right).
- After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label.
- When M_1 reads the last symbol of the input, it produces the output: **accept** if M_1 is in an accept state, or **reject** if M_1 is not in an accept state.

Language Recognition:

- If L is the set of all strings that a FA M accepts, we say that L is the language of the machine M and write $L(M) = L$.
- An automaton may accept several strings, but it always recognizes **only one** language.
- If a machine accepts no strings, it still recognizes one language, namely the empty language $\emptyset$.

Formal Definition of Acceptance:

- Let Let $M = (Q, \Sigma, \delta, q_0, F)$ be a FA and $w = a_1a_2 \ldots a_n$ be a string over Σ. We say M accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ exist in Q such that
    1. $r_0 = q_0$ (where machine starts),
    2. $\delta(r_i, a_{i+1}) = r_{i+1},\ i = 0, 1, \ldots, n-1$, (transitions based on δ) and
    3. $r_n \in F$ (input accepted).

Regular Languages:

- We say that a FA M recognizes the language L if $L = \{w \mid M \text{ accepts } w\}$.
- A language is called a **regular** language, if there exists an FA that recognizes it.
- **Q**: How do you design/build an FA?

FA Design Approach:

1. Identify finite pieces of information you need, i.e., the states (possibilities).
2. Identify the condition (or alphabet) to change from one state to another.
3. Identify the starting and final/accept states.
4. Add missing transitions.

Example: Let $M_1 = (Q, \Sigma, \delta, q_1, F)$, $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, and $F = \{q_2\}$. Let's define a transition function δ for M_1 and then draw the resulting (graph-based) **state transition diagram** for M_1.

Q: What is $L(M_1)$?

Example: Let $M_2 = (Q, \Sigma, \delta, q_1, F)$, $Q = \{q_1, q_2\}$, $\Sigma = \{0, 1\}$, and $F = \{q_2\}$. Let's define a transition function δ for M_1 and then draw the resulting (graph-based) state transition diagram for M_2.

Q: What is $L(M_2)$?

Example: Now consider$M_3 = (Q, \Sigma, \delta, q_1, F)$, $Q = \{q_1, q_2\}$, $\Sigma = \{0, 1\}$, and $F = \{q_1\}$ that has the same transition function δ of M_2. Let's draw the resulting (graph-based) state transition diagram for M_3.

Q: What is $L(M_3)$?

Example: For our last DFA example, consider $M_4 = (Q, \Sigma, \delta, s, F)$, $Q = \{s, q_1, q_2, r_1, r_2\}$, $\Sigma = \{a, b\}$, and $F = \{q_1, r_1\}$. We will define a transition function δ for M_4 and then draw the resulting (graph-based) state transition diagram for M_4.

Q: What is $L(M_4)$?

3 Regular Languages

Let A and B be languages:

- **Union**: $A \cup B = \{x \mid x \in A \vee x \in B\}$
- **Concatenation**: $A \circ B = \{xy \mid x \in A \wedge y \in B\}$
- **Star**: $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \wedge x_i \in A,\ 0 \leq i \leq k\}$

Q: Is ϵ always a member of A^* regardless of the language A? ________

Q: What is another name for the language $A \circ A^*$? ________

Closures of Regular Languages:

- **Theorem**: Class of regular languages is closed under intersection. (Proof: Use cross-product construction of states.)
- **Theorem**: Class of regular languages is closed under complementation. (Proof: Swap accept/non-accept states and show FA recognizes the complement.)

3.1 *Nondeterminism*

Nondeterministic Finite Automata (NFA):

- Every step of a FA computation follows in a unique way from the proceeding step; a deterministic computation.
- Nondeterministic computation – choices exist for the next state; a nondeterministic FA (NFA).
- Ways to introduce nondeterminism:
  1. more choices for next state (zero, one, many),
  2. state may change to another state without reading any symbol.

NFA Formal Definition:

- An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols (alphabet), the transition function δ maps $Q \times \{\Sigma \cup \epsilon\}$ to $\mathcal{P}(Q)$, $q_0 \in Q$ is the start (initial) state, and $F \subseteq Q$ is the set of accept (final) states.
- Notice that the range of the transition function δ for an NFA is the power set of Q ($\mathcal{P}(Q)$).

Formal Definition of Acceptance (NFA):

- Let Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $w = y_1 y_2 \ldots y_n$ be a string over $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. We say N accepts w if a sequence of states $r_0, r_1, \ldots, r_m$ exist in Q such that

  1. $r_0 = q_0$,
  2. $\delta(r_i, y_{i+1}) = r_{i+1}$ for $i = 0, 1, \ldots, m-1$, and
  3. $r_m \in F$.

NFA Motivation:

- For some problems, they are much easier to construct than a DFA.
- NFA may actually be smaller than a DFA that performs the same task; but NFA computation is usually more expensive.
- Every NFA can be converted into an equivalent DFA (in theory, every NFA has an equivalent DFA to recognize the same language).
- NFAs can be used to show that regular languages are closed under union, concatenation, and star operations.

3.2 *Closure Properties*

Union:
Using a *schematic* proof, we will show that the class of regular languages is closed under the **union** operation (using NFAs).

Concatenation:
Using a *schematic* proof, we will show that the class of regular languages is closed under the **concatenation** operation (using NFAs). More specifically, if A_1 and A_2 are regular languages, we can show that $A_1 \circ A_2$ is also a regular language.

Star Operation:
Using a *schematic* proof, we will show that the class of regular languages is closed under the **star** operation (using NFAs). More specifically, if A is a regular language, we can show that A^* is also a regular language.

3.3 *DFA/NFA Equivalence*

NFA to DFA Conversion:

- Let Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA that recognizes the language A and construct the DFA M that also recogizes A. Define $M = (Q', \Sigma, \delta', q_0', F')$.
- For any $R \subseteq Q$, define $E(R)$ to be the collection of states that can be reached from R by going along ϵ transitions, including members of R themselves. Then,

$$E(R) = R \cup \{q \in Q \mid \exists r_1 \in R, r_2, \ldots, r_k \in Q,\ r_{i+1} \in \delta(r_i, \epsilon),\ r_k = q\}.$$

- Now, modify δ' of M to place additional links (edges) on all states that can be reached by going along ϵ edges after every step:

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)), \text{for some } r \in R\}.$$

- Finally, set $q_0' = E(\{q_0\})$ and $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.

Example: $N_1 = (Q, \Sigma, \delta, q_1, F)$, $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, and $F = \{q_1\}$.
What is N_1's start state? _______

Example: Let $N_2 = (Q, \Sigma, \delta, q_1, F)$, $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, and $F = \{q_4\}$. We will define a transition function δ for N_2 that has an ϵ transition and then draw the resulting (graph-based) state transition diagram for N_2.

Q: What is $L(N_2)$?

Example: Suppose $\Sigma = \{a, b\}$. Let F_1 be a DFA that recognizes the language $A_1 = \{w \mid w \text{ has exactly two a's}\}$ and F_2 be a DFA that recognizes the language
$A_2 = \{w \mid w \text{ has at least two b's}\}$.

Let's build the DFA $F = F_1 \times F_2$ that recognizes the language
$A = \{w \mid w \text{ has exactly two a's and has at least two b's}\}$.

Example: Suppose N_3 is an NFA defined by $N_3 = (Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, and $F = \{q_1\}$. For simplicity of notation, we will use simple integers to represent states in the corresponding state diagram for N_3. That is, $q_1 \equiv 1$, etc. Let's create an equivalent DFA $D_3 = (Q', \Sigma, \delta', q_1', F')$ that recognizes the same language that N_3 recognizes.

We will draw the state diagram for N_3 below and then derive the start state q_1' and final states F' for D_3 based on the power set of states from N_3.

Now, let's represent all the possible states of D_3 and add transitions based on δ from N_3.

Do you see any unreachable states? If so, cross them out in the state diagram for D_3 and then redraw/simplify the final state diagram for D_3 below.

4 Nonregular Languages

We say that R is a **regular expression** if R is ...

1. a for some $a \in \Sigma$,
2. ϵ (language contains only the empty string),
3. $\emptyset$ (language has no strings),
4. $(R_1 \cup R_2)$, where R_1, R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1, R_2 are regular expressions, or
6. R_1^*, where R_1 is a regular expression.

Theorem: A language is regular if and only if some regular expression (RE) describes it.

Proof. If a language R is described by a RE, then A is recognized by an NFA so A must be regular. If the language R is regular, then it is recognized by a DFA from which a RE can be deduced. □

Q: Is $B = \{0^n 1^n | n \geq 0\}$ a regular language?

A DFA that recognizes B needs to remember how many 0s have been seen so far as it reads the input; the DFA would have to keep track of an unlimited number of possibilities – can we do this with a finite number of states?

Warning: Just because a language might seem to require unbounded (infinite) memory to recognize it – it still could be regular.

Suppose you have the following two languages: $C = \{w \mid w$ has an equal number of 0s and 1s$\}$ and $D = \{w \mid w$ has an equal number of 01 and 10 substrings$\}$.

Which one is regular? _______

Q: How do we prove the nonregularity of a language?

4.1 *Pumping Lemma*

Pumping Property:

- All strings in a language can be pumped if they are at least as long as a certain value called the **pumping length**.
- Another interpretation: every string contains a section that can be repeated any number of times with the resulting string remaining in the language.
- **Example**: sqrt(sqrt(sqrt(. . . sqrt(x)...)))

Pumping Lemma:
If A is a regular language, then there exists a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$ subject to . . .

1. $\forall i \geq 0,\ xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Pumping Lemma (PL) Proof Construction:

- Let $M = (Q,\ \Sigma,\ \delta,\ q_1,\ F)$ be be a DFA that recognizes the language A. Assign a pumping length p to the number of states of M.
- Show that any string $s \in A$, $|s| \geq p$ may be broken into xyz satisfying the three PL conditions.
- If there are no strings in A of length at least p, then the PL is true because all three conditions hold for all strings of length as least p (if there are NO such strings).

Pumping Lemma:

Proof. Let $M = (Q,\ \Sigma,\ \delta,\ q_1,\ F)$ be a DFA that recognizes A and let p be the size of Q (i.e., the number of states). Let $s = s_1 s_2 \ldots s_n$ be a string over Σ with $n \geq p$ and $r = r_1 r_2 \ldots r_{n+1}$ be the sequence of states encountered while processing s, i.e., $r_{i+1} = \delta(r_i, s_i)$, $1 \leq i \leq n$.

We know that $n + 1 \geq p + 1$, why?

Then, among the first $p + 1$ elements in the sequence $r_1 r_2 \ldots r_{n+1}$, there must be a repeating state, say r_j, r_k. What principle is this based on? ____________

Let r_k be the recurring state among the first $p + 1$ states in the sequence starting with r_1, so $k \leq p + 1$. Let $x = s_1 s_2 \ldots s_{j-1}$, $y = s_j s_{j+1} \ldots s_{k-1}$, and $z = s_k s_{k+1} \ldots s_n$.

So, x takes M from r_1 to r_j, y takes M from r_j to r_j and z takes M from r_j to r_{n+1}; recall that $r_j = r_k$ and that r_{n+1} is an accept state. Therefore,

- M must accept $xy^i z$, for $i \geq 0$ (**PL Condition 1**).
- Since $j \neq k$ then $|y| > 0$. (**PL Condition 2**).
- Since $k \leq p + 1$ then $|xy| \leq p$ (**PL Condition 3**).

$\square$

4.2 *Pumping Lemma (for Regular Languages) Proofs*

Technique for using PL to prove a language A is **not** regular:

1. Assume A is regular and obtain a contradiction.
2. PL guarantees existence of a pumping length p such that all strings of length p or greater (in A) can be pumped.
3. Find $s \in A$, $|s| \geq p$ that cannot be pumped; consider all the ways of dividing s into x, y, z and show that for each division, at least one of the PL conditions fail to hold.

Example: Use the PL to prove that the language $B = \{0^n 1^n \mid n \geq 0\}$ is not regular.

Proof. Assume B is regular and let p be the pumping length for B. Choose $s = 0^p 1^p \in B$ so that clearly $s > p$. By the PL, we can partition $s = xyz$ such that for all $i \geq 0$, $xy^i z \in B$. Let's consider three possible cases for the contents of substring y:

1.

2.

3.

By obtaining a contradiction in all three of the cases above, we can conclude that **Condition 1** of the PL will be violatd regardless of the choice for y. Therefore, B cannot be regular as assumed. □

Q: How might the proof above be simplified using **Condition 3** of the PL to constrain the contents of the substring y?

Example: Use the PL to prove that the language $E = \{0^i 1^j \mid i > j\}$ is not regular.

Proof. Assume E is regular and let p be the pumping length for E. Choose $s = 0^{p+1}1^p \in E$ so that clearly $s > p$. By the PL, we can partition $s = xyz$ such that for all $i \geq 0$, $xy^iz \in E$. Start with **Condition 3** of the PL to determine the contents of substring y and see if you can get a violation of **Condition 1** of the PL for any choice of y.

□

5 Context-Free Languages

We have shown that $L = \{0^n1^n \mid n \geq 0\}$ cannot be specified by by a FA or regular expression; **Context-Free Grammars** (or CFGs) provide a more powerful way to specify languages. A CFG is a 4-tuple (V, Σ, R, S), where

- V is a finite set of symbols (variables or nonterminals),
- Σ is a finite set of symbols disjoint from V (terminals),
- R is a finite set of specification rules of the form $lhs \rightarrow rhs$, $lhs \in V$, $rhs \in (V \cup \Sigma)$, and $S \in V$ is the start variable.

Example: CFG G_1 has the following specification rules:

$$\begin{array}{lcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

The start variable for G_1 is A.

What are the nonterminals? ____________ What are the terminals? ____________

Language Specification:

A grammar generates each string of the language in three basic steps:

1. Write down start variable; *lhs* of first spec rule,
2. Find variable that is written down and a rule whose *lhs* is that variable; replace the written down variable with the *rhs* of that rule,
3. Repeat **Step 2** until no variables remain in string.

Example: Use the CFG G_1 (above) to derive the string 000#111. Show derivation and corresponding parse tree.

The language specified by the G_1 grammar is given by $L(G_1) = \{0^n\#1^n \mid n \geq 0\}$. A language generated by a CFG is called a **context-free language** (or CFL).

Direct Derivation:

If $u, v, w \in (V \cup \Sigma)^*$, i.e., are strings of variables and terminals, and $A \to w \in R$ is a grammar rule, then we say that uAv yields uwv or $uAv \Rightarrow uwv$. Alternatively, uwv is **directly derived** from uAv using the rule $A \to w$.

Derivation:

We write $u \Rightarrow v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k \in (V \cup \Sigma)^*$ exists for $k \geq 0$, and $u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$. We say that $u_1, u_2, \ldots, u_k, v$ is a **derivation** of v from u_1.

If $G = (V, \Sigma, R, S)$ is a CFG, then the language specified by G (or the language of G) is a CFL, i.e., $L(G) = \{w \in \Sigma^* \mid S \to w\}$.

Example: Suppose G_3 is a CFG defined by

$$G_3 = (\{S\}, \{a, b\}, \{S \to aSb|SS|\epsilon\}, S).$$

What is $L(G_3)$?

CFG Applications:

- compiler design and implementation,
- programming language specification,
- scanners, parsers, and code generators.

Example: Let $G_4 = (\{E, T, F\}, \{a, +, *, (,)\}, R, E)$, where R is given by

$$E \to E + T | T, T \to T * F | F, F \to (E) | a \,.$$

What is $L(G_4)$?

Now, let's create parse trees (using G_4) for the strings $a + a * a$ and $(a + a)$.

5.1 *Design Techniques*

CFG Design Technique No. 1:

- Many CFGs are unions of simpler CFGs.
- Combination involves putting all the rules together and adding the new rules

  $$s \to s_1 | s_2 | \cdots | s_k,$$

  where the variables s_i, $1 \leq i \leq k$, are the start variables of the individual grammars and the s is a new variable.

Example: Design a grammar for the language

$$\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\} .$$

CFG Design Technique No. 2:
Construct a DFA for the language first.

Conversion procedure:

1. Make a variable R_i for each state q_i of the DFA.
2. Add the rule $R_i \to R_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
3. Add the rule $R_i \to \epsilon$, if q_i is an accept state of the DFA.
4. If q_0 is the start state of the DFA, make R_0 the start variable of the CFG.

Theorem: Every regular language is context-free.

CFG Design Technique No. 3:

- Certain CFLs contain strings with two related substrings, e.g., 0^n and 1^n in $\{0^n1^n \mid n \geq 0\}$.
- Machine would need to remember an unbounded amount of information about one of the substrings.

- CFG that handles such cases uses a rule of the form $R \rightarrow uRv$.

Example: Consider the CFG $G = (\{S, B\}, \{a, b\}, \{S \rightarrow aSB|B|\epsilon, B \rightarrow bB|b\}, S)$. Let's determine $L(G)$.

Consider a few derivations . . .

S → aSB → aaS**B**B → aaS**bB**B
S → aSB → aaSB**B** → aaSB**bB**
S → aSB → aaSB**B** → aaSbB**b**
S → aSB → aa**S**BB → aaBB

B → bB → bbB → $\mathrm{b}^{\mathrm{k}-1}$B → b^{k}, k ≥ 1
S → aSB → aSb^{k}, k ≥ 1

So, $L(G) =$ _____________

Ambiguous Grammar:
If a CFG generates the same string in different ways, that string is derived ambiguously in the grammar. Such a CFG is called **ambiguous**.

Example: Consider the CFG G_5 that has the rules: $E \rightarrow E + E \mid E * E \mid (E)|a$. Let's produce two different derivation trees for $a + a * a$:

Derivation Order:
It is possible for 2 different derivations to produce the same derivation tree because they differ in the order in which they replace nonterminals (not the rules).

- **Leftmost** derivation: replace the leftmost nonterminal
- **Rightmost** derivation: replace the rightmost nonterminal at each step.

These two derivations of a string w are unique (and are equivalent to the derivation tree).

Example: We will show that a particular CFG $G = (V, \Sigma, R, S)$ with $\Sigma = \{a, b, c\}$ for the CFL $A = \{a^i b^j c^k \mid i = j \vee j = k \wedge (i, j, k \geq 0)\}$ is inherently ambiguous. That is, from the start variable (S) we need to produce the same set of terminals. Keep in mind that such a set of terminals could just be the empty string (ϵ).

5.2 *Chomsky Normal Form*

The Chomsky Normal Form (CNF) is a simplification procedure for CFGs. The format for rules in CNF has one of two forms:

- $A \rightarrow BC$, or
- $A \rightarrow a$, where a is a terminal and A, B, C are nonterminals and B, C may not be the start variable. The rule $S \rightarrow \epsilon$ for the start variable S is not excluded.

Theorem: Any CFL is generated by a CFG in CNF.

Proof. **Stage 1** – Add a new start symbol S_0 and rule $S_0 \to S$, where S was the original start variable (do not want S in the *rhs* of any rule).

Stage 2 – Eliminate all ϵ-rules.
Repeat ... (until all ϵ-rules are removed):

1. Eliminate the ϵ-rule $A \to \epsilon$, where A is not the start variable.
2. For each occurrence of A on the *rhs* of a rule, add a new rule with that occurrence of A deleted.
3. Replace the rule $B \to A$ (if present) by $B \to A \mid \epsilon$ unless the rule $B \to \epsilon$ has not been previously eliminated.

Example: To delete $A \to \epsilon$, replace $B \to uAv$ by $B \to uAv \mid uv$; replace $B \to uAvAw$ by $B \to uAvAw \mid uvAw \mid uAvw \mid uvw$.

Stage 3 – Remove all unit rules.
Repeat ... (until all unit rules are removed):

1. Remove a unit rule $A \to B$.
2. For each rule $B \to u$ that appears, add the rule $A \to u$, unless it was a previously-removed unit rule (u can be a string of variables and terminals).

Stage 4 – Convert all remaining rules.
Repeat ... (until no rules of the form $A \to u_1u_2 \dots u_k$ with $k \geq 3$ remain):

1. Replace a rule $A \to u_1u_2 \dots u_k$, $k \geq 3$, where each u_i, $1 \leq i \leq k$, is a variable or a terminal, by $A \to u_1A_1$, $A_1 \to u_2A_2$,..., $A_{k-2} \to u_{k-1}u_k$, where $A_1, A_2, \dots, A_{k-2}$ are **new variables**.
2. If $k \geq 2$, replace any terminal u_k with a new variable U_i and add the rule $U_i \to u_k$.

□

Example: Convert the CFG $G = (V, \Sigma, R, S)$, where R is given below, to CNF.

$$\begin{aligned} R:\quad & S \to ASA \mid aB \\ & A \to B \mid S \\ & B \to b \mid \epsilon \end{aligned}$$

Step 1 – new start variable.

Step 2 – remove ϵ rules.

Step 3 – remove unit rules.

Step 4 – convert remaining rules.

6 Pushdown Automata

A **pushdown automata** (or PDA) is similar to an NFA but it has a **stack**. The stack provides additional memory beyond finite memory available in control; it allows the PDA to recognize some nonregular languages.

Two options to prove that a language is context-free:

- Construct a CFG that generates it, or
- construct a PDA that recognizes it.

Some CFLs are more easily described in terms of their **generators**, whereas others are more easily described in terms of their **recognizers**. Let's draw a schematic representation of the difference between an NFA and a PDA:

Terminology:

- Writing a symbol on the stack is called **pushing** the symbol.
- Removing a symbol from the stack is called **popping** the symbol.
- All access to the stack my be done only at the top (LIFO storage device).

The primary benefit of that stack is that it can hold an unlimited amount of data; a PDA can recognize $\{0^n1^n \mid n \geq 0\}$ because it can use the stack to remember the number of 0s it has seen (read).

Informal Algorithm for $\{0^n 1^n \mid n \geq 0\}$:

1. Read symbols from the input. As each 0 is read push it onto the stack.
2. As soon as a 1 is read, pop a 0 off the stack (for each 1 read). 3. If input finishes when the stack become empty, accept; if stack becomes empty while there is still input or input finishes while the stack is not empty, reject.

A PDA may be nondeterministic. Languages such as $\{0^n 1^n \mid n \geq 0\}$ do not require nondeterminisim. However, the language $\{ww^R \mid w \in \{0,1\}^*\}$ would require nondeterminism. Why?

Formalization:

- A PDA may use different alphabets for input (Σ) and stack (Γ).
- Nondeterminism allows the PDA to make transitions on empty input. Define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$.
- The domain of the PDA transition function is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$, where Q is the set of states.
- The range of the PDA transition function is $\mathcal{P}(Q \times \Gamma_\epsilon)$.

$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$$

PDA Formal Definition:

A PDA is a 6-tuple (Q, Σ, Γ,δ, q_0, F), where Q, Σ, Γ are finite set of states, and ...

1. Q is a set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of all accept states.

PDA Computation:

A PDA M=(Q, Σ, Γ,δ, q_0, F) computes as follows ...

M inputs $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\epsilon$.
There are a sequence of states $r_0, r_1 r_2, \dots, r_m \in Q$ and a sequence of strings $s_0, s_1, \dots, s_n \in \Gamma_*$ that satisfy the following conditions ...

1. $r_0 = q_0$, $s_0 = \epsilon$; begin with start state and empty stack;
2. $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, $i = 0, 1, \dots, m-1$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma$ and $t \in \Gamma^*$;
3. $r_m \in F$; accept state encountered at end of input.

Stack notation: $a, b \to c$ or simply abc
a is read from the input, b is popped from the stack, and c is pushed onto the stack. ϵ-cases:

- If $a = \epsilon$, machine can transition without reading any input.
- If $b = \epsilon$, machine can transition without popping any symbol from the stack.
- If $c = \epsilon$, machine can transition without writing any symbol onto the stack.

Empty stack:

The PDA (by definition) does not consider the testing of an empty stack. We can achieve this by initially placing a special char (say \$) on the stack. When the PDA encounters that char (\$) again (on the stack), it knows the stack is effectively empty.

Both CFGs and PDAs specify context-free languages; we can always convert a CFG into a PDA that recognizes the language of the CFG.

CFG – specifies a program language
PDA – specifies/implements the compiler

Theorem: : A language is context-free if and only if some PDA recognizes it.

1. If a language L is CF, then there is a PDA M_L that recognizes it.
2. If a language L is recognized by a PDA M_L, then there is a CFG G_L that generates L.

Lemma: If a language is context-free then some PDA recognizes it.

Proof.

1. Let A be a CFL so A has a CFG G that generates it.
2. Need to convert G into a PDA P that accepts a string w if G generates w.
3. P operates by determining a derivation of w.

$\square$

What are the difficulties with the approach above?

Difficulties:

How do we decide which substitutions to make for a derivation? (PDA P nondeterminism can help)

- At each step of the derivation, one of the rules for a particular variable is selected non-deterministically.
- P has to start by writing the start variable on the stack and then continue working the string w.
- If while consuming the string w, P arrives at a string of terminals that equals w, then accept; otherwise, reject.

Informal Description:

Place marker symbol \$ and start variable on the stack.
Repeat:

1. If TOS (top of stack) is a **variable** symbol A, non- deterministically select a rule r such that $lhs(r) = A$ and substitute A by the string $rhs(r)$.

2. If TOS is a **terminal** symbol, a, read the next input symbol and compare it with a; if they match, pop the stack; if they do not match, reject this branch of nondeterminism.

3. If the TOS is a \$ and all the text has been read, **accept**; otherwise **reject**.

Generic State Diagram for PDA P with the following stack operations:

1. TOS = **variable**: set $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, w) \mid A \to w \in R\}$, where R is the set of CFG rules.

2. TOS = **terminal**: set $\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$.

3. TOS = **\$**: $\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$.

Recall notation: (read input, pop stack, push to stack)

Example: Let's draw the state diagram for the PDA that would recognize the language generated by the CFG G_1 having rules $S \to aTb \mid b$ and $T \to Ta \mid \epsilon$, with $\Sigma_\epsilon = \{a, b\} \cup \{\epsilon\}$.

Example: Let's draw the state diagram for the PDA that would recognize the language generated by the CFG G_2 having rules $E \to E + T \mid T$, $T \to T * F \mid F$, and $F \to (E) \mid a$, with $\Sigma_\epsilon = \{a, +, *, (,)\} \cup \{\epsilon\}$.

7 Pumping Lemma (CFLs)

Recall that the Pumpling Lemma (or PL) was used to prove that a given language is or is not regular. Another version of the PL for Context-Free Languages (or CFLs) states that every CFL has a specific value called the **pumping length** such that all longer strings in the language can be pumped. Pumping in this context means that a string can be divided into **five parts** and that the 2nd and 4th parts may be repeated any number of times and the resulting string is in the language.

Pumping Lemma for CFLs
If A is a CFL, then there exists a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces, $s = uvxyz$, satisfying the following conditions:

1. $\forall i \geq 0,\ uv^ixy^iz \in A$,
2. $|vy| > 0$ (i.e., either v or y is not ϵ),
3. $|vxy| \leq p$ (i.e., the interior cannot be larger than p).

Pumping Lemma (PL) Schematic Proof

Proof. Let A be a CFL and G be the CFG that generates A. We have to show that any sufficiently long $s \in A$ can be pumped and remain in A.

- Because $s \in A$, it is derivable from G and say has a derivation tree D_s.
- The tree D_s must be very tall (for a long s).
- So D_s contains some relatively long path from the start variable (at root) to a terminal at a leaf.
- On such a long path, some variable X must be repeated due to the *pigeonhole principle*.
- The repetition of X allows for the replacement of a subtree under the <u>second occurrence</u> of X to be replaced by the subtree under the <u>first occurrence</u> of X.

So, we may cut s into five pieces and repeat the 2nd and 4th partitions to obtain $uv^ixy^iz \in A$, for any $i \geq 0$.

□

Let's illustrate the replacement described above for *pumping up* once.

Now, let's create a similar illustration for *pumping down v* and y from the original string s.

Example: Use the PL for CFLs to prove that the language $B = \{a^n b^n c^n \mid n \geq 0\}$ is not a CFL.

Proof. Assume B is a CFL and let p be the pumping length for B. Choose $s = a^p b^p c^p \in B$ so that clearly $s > p$. By Condition 1 of the PL for CFLs, we can partition $s = uvxyz$ such that for all $i \geq 0$, $uv^i xy^i z \in B$. In order to show that s cannot be pumped, let's consider the ramifications of Condition 2 of the PL for CFLs for the contents of v and y.

□

Example: Use the PL for CFLs to prove that the language $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$ is not a CFL.

Proof. Assume C is a CFL and let p be the pumping length for C. Choose $s = a^p b^p c^p \in C$ so that clearly $s > p$. By Condition 1 of the PL for CFLs, we can partition $s = uvxyz$ such that for all $i \geq 0$, $uv^i xy^i z \in C$. Similar to the previous example, let's consider the possible contents for v and y.

1. Case when v and y contain only one type of alphabet symbol. In this case, it can be shown that one of the symbols a, b, or c cannot appear in v or y. We can then break this situation into three sub-cases:

   (a) The a's do not appear.

   (b) The b's do not appear.

(c) The c's do not appear.

2. Case when either v or y contain more than one type of alphabet symbol.

□

Example: Use the PL for CFLs to prove that the language $D = \{ww \mid w \in \{0,1\}^*\}$ is not a CFL.

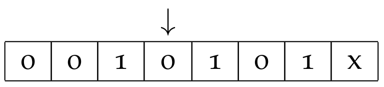
Proof. Assume D is a CFL and let p be the pumping length for D. Choose $s = 0^p 1^p 0^p 1^p \in D$ so that clearly $s > p$. By Condition 1 of the PL for CFLs, we can partition $s = uvxyz$ such that for all $i \geq 0$, $uv^i xy^i z \in D$. Using Condition 3 of the PL for CFLs, we can consider the ways that the substring vxy (the interior) can straddle the midpoint of s. We can show that under all situations, s cannot be pumped and thereby conclude that D could not be a CFL.

$\square$

8 Turing Machines

Similar to a FA with a supply of unlimited memory. A Turing Machine (TM) can do everything that a modern computing device can do; but there are problems that even a TM cannot solve.

Tape:



- Memory is modeled by a tape of symbols.
- Initially, tape contains only the input string and blanks everywhere else.
- A TM can store information by writing symbols on the tape.
- The tape can move its head left and right to read symbols.
- TM continues to move until it enters a state in which the next move is *not defined*.

TM –vs- FA, PDA:

write A TM tape allows both write and read ops; DFA and NFA have only an input tape (read-only) and the tape head moves from left to right. PDA has both an input tape and stack tape; we can read/write on the stack tape (when moves right it writes and when head moves left it erases the current symbol).

size The TM tape is infinite; the input of FA/PDA is finite; the stack of a PDA is infinite.

accept FA/PDA accept a string when it has scanned all the input symbols and enters a final state; a TM accepts a string as long as it enters a final state (one suffices).

Example: Construct a TM M_1 that tests the membership of the language $L_1 = \{w\#w \mid w \in \{0,1\}^*\}$. In other words, design a M_1 such that $M_1(w) =$ accept if and only if $w \in L_1$. Position of tape head is underlined:

$S_0, a = 0$	$\underline{0}$10#010	$S_0, a = 0$	xx$\underline{x}$#xx0
S_1	x10$\underline{\#}$010	S_1	xxx$\underline{\#}$xx0
S_2	x10#$\underline{x}$10	S_2	xxx#xx$\underline{x}$
S_3	x10$\underline{\#}$x10	S_3	xxx$\underline{\#}$xxx
S_4	$\underline{x}$10#x10	S_4	xx$\underline{x}$#xxx
$S_0, a = 1$	x$\underline{x}$0#x10	S_0	xx$\underline{x}$#xxx
S_1	xx0$\underline{\#}$x10	S_5	xxx#xxx_ (accept)
S_2	xx0#x$\underline{x}$0		
S_3	xx0$\underline{\#}$xx0		
S_4	xx$\underline{0}$#xx0		

Formal Definition of a Turing Machine (7-tuple):

A TM is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are finite sets, and

1. Q is a set of states,
2. Σ is the input alphabet and a *blank* $\notin \Sigma$.
3. Γ is the tape alphabet, *blank* $\in \Gamma$, $\Sigma \subset \Gamma$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. q_{accept} is the accept state (q_a), and q_{reject} is the reject state (q_r).

Q: How does the TM M compute?

- M receives as input $w = a_1 a_2 \ldots a_n \in \Sigma^*$, $a_i \in \Sigma$ written on the leftmost squares of the tape (rest of the tape contains *blanks*).
- The head starts on the leftmost square of the tape and the first *blank* encountered signals the end of input.
- Once M starts, it proceeds according to δ. M stays on the leftmost square (of input) even if δ indicates a left move (L) from that square.
- Computation continues until M cannot move; w is accepted if M enters q_{accept}. (M may go on forever as long as δ is defined.)

Example: Let's revisit our TM M_1 that tests the membership of the language $L_1 = \{w\#w \mid w \in \{0,1\}^*\}$.

S_0	If symbol read is a 0 or 1, replace it by x and remember the symbol as a; if the symbol is a # go to S_5; else **reject**. $\delta(S_0, 0) = (S_1(0), x, R)$, $\delta(S_0, 1) = (S_1(1), x, R)$, $\delta(S_0, \#) = (S_5, \#, R)$
$S_1(a)$	Move right until a # is found; if no # is found before *blank*, **reject**. $\delta(S_1(a), 0) = (S_1(a), 0, R)$, $\delta(S_1(a), 1) = (S_1(a), 1, R)$ $\delta(S_1(a), \#) = (S_2(a), \#, R)$ // change state
$S_2(a)$	Move right until a 0 or 1 is found; if current symbol is the same as a, replace if by x; else **reject**. $\delta(S_2(a), x) = (S_2(a), x, R)$, $\delta(S_2(0), 0) = (S_3, x, L)$ $\delta(S_2(1), 1) = (S_3, x, L)$ // change state
S_3	Move left until a # is found. $\delta(S_3, a) = (S_3, a, L)$, where $a \in \{0, 1, x\}$ $\delta(S_3, \#) = (S_4, \#, L)$ // change state
S_4	Move left until an x is found and go to S_0 $\delta(S_4, a) = (S_4, a, L)$, where $a \in \{0, 1\}$ $\delta(S_4, x) = (S_0, x, R)$ // change state
S_5	Move right until a 0, 1, or *blank* is found; **accept** if current symbol is a *blank*; **reject** if current symbol is 0 or 1. $\delta(S_5, a) = (S_5, a, R)$, where $a \in \{0, 1, x\}$ // stay in S_5 $\delta(S_5, blank) = (S_0, blank, L)$ // accept

Formalizing TM Computation:

A **configuration** C of the TM M is a 3-tuple $C = (u, q, v)$, where $q \in Q$, $u, v \in \Gamma^*$ is the tape content and the head is pointing to the first symbol of v.

A configuration C_1 yields a configuration C_2 if the TM can (legally) go from C_1 to C_2 in a single computation (step). Suppose $a, b, c \in \Sigma$; $u, v \in \Gamma^*$ and $q_i, q_j \in Q$.

1. uaq_ibv yields $uacq_jv$, if $\delta(q_i, b) = (q_j, c, R)$.
2. uaq_ibv yields uq_jacv, if $\delta(q_i, b) = (q_j, c, L)$.
3. Suppose head is at leftmost symbol of input – q_ibv yields q_jcv if the transition is left-moving, i.e., $\delta(q_i, b) = (q_j, c, L)$; q_ibv yields cq_jv if the transition is right-moving, i.e., $\delta(q_i, b) = (q_j, c, R)$.
4. Suppose head is at rightmost symbol of input – uaq_i is equivalent to uaq_ib because we assume *blanks* follow the part of the tape represented in the configuration.

Special Configurations:

- If the input of TM is w and the initial state is q_0, then $q_0 w$ is the **start** configuration.
- $uaq_{accept}bv$ is an **accepting configuration**.
- $uaq_{reject}bv$ is a **rejecting configuration**.
- $uaqcv$, where $\delta(q, c)$ is undefined, is also a rejecting configuration.
- Accepting and rejecting configurations are also called **halting configurations**.

A TM accepts the input w if a sequence of configurations $C_1, C_2, \ldots, C_n$, exists such that

1. C_1 is the start configuration, i.e., $C_1 = (\epsilon, q_0, w)$.
2. Each C_i yields C_{i+1}, $i = 1, 2, \ldots, n-1$.
3. C_n is an accepting configuration.

Let $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$; the language L is Turing-recognizable if there is a TM M that recognizes L.

Three possible cases for TM execution on input w: TM accepts w, TM rejects w, or TM does not halt.

A TM that halts on all inputs is called a **decider**. A language L is called **Turing-decidable** (or **decidable**) if some TM decides it.

Important Facts:

- Any regular language is Turing-decidable.
- Any CFL is Turing-decidable.
- Every decidable language is Turing-recognizable.
- Certain Turing-recognizable languages are not decidable (i.e., TM does not halt on all inputs).

A few high-level operations for TMs:

1. Determine if two strings are the same.
2. Compute the addition, subtraction, multiplication, division, power, log, etc. of numbers in unary form.
3. Shift a string to the right (or left).
4. Maintain a base-b counter.

Multi-tape TMs:

1. Each tape has its own head for reading/writing.
2. Initially the input is on tape 1 and the other tape(s) are blank.
3. Transition function allows for reading, writing, and moving the head on all tapes simultaneously.
4. $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$, where k is the number of tapes.
5. $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes b_1 through b_k on tapes 1 through k, respectively, and moves each head to the left (or right) as specified.

Theorem: Every multi-tape TM has an equivalent single tape TM.

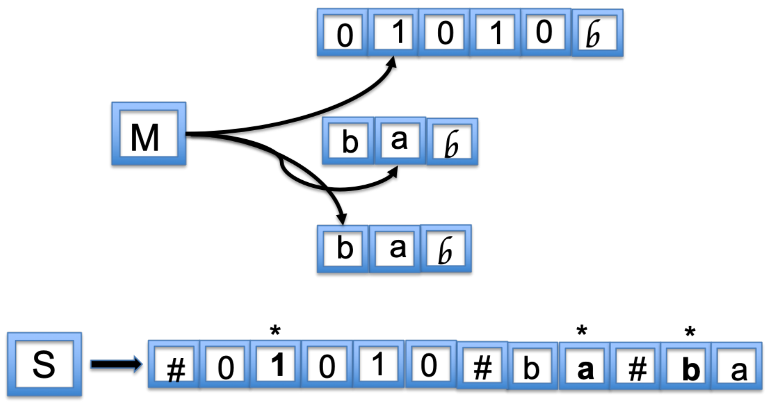
Example: 3-tape TM to recognize $L = \{0^a 1^b 2^c \mid c = \lfloor log_a(b) \rfloor, a > 1,\ b > 0\}$. Note: $b = ac$ and $a^c \leq b < a^{c+1}$.

- Tape 1: input tape.
- Tape 2: contains x.
- Tape 3: contains k, where $x = a^{k+1}$; loop while $x \leq b$.

M = “On w …”

1. Check if $w \in 00^+1^+2^*$; if not, **reject**.
2. Copy all 0’s on input tape to tape 2.
3. If number of 0’s on tape 2 exceeds number of 1s on input tape, go to Step 6.
4. Multiply number of 0’s on tape 2 by number of zeros on input tape, and keep result on tape 2.
5. Add a symbol 2 to tape 3; go to Step 3.
6. If number of 2’s on tape 3 is same as the number of 2’s on input tape, **accept**; otherwise, **reject**.

Example: Multi-tape to Single Tape (# is tape delimiter)



Demonstrations and Shorthand:

Turing showed how to compute e, π, and all real algebraic numbers, i.e., solutions to

$$a_n x^n + a_{n-1}^{n-1} \cdots + a_2 x^2 + a_1 x + a_0 = 0,$$

and roots of Bessel functions.

The number of machine configurations and symbols could be rather large for the TMs required so Turing developed a **shorthand notation** for computing (tabulating).

Example: TM that computes the infinite sequence 0101010 . . ..

m-config	Symbol	Action (Op)	Final *m*-config
A (q_1)	b (S_0)	P0,R (PS_1,R)	B (q_2)
B (q_2)	b (S_0)	R (PS_0,R)	C (q_3)
C (q3)	b (S_0)	P1,r (PS_2,R)	D (q_4)
D (q_4)	b (S_0)	R (PS_0,R)	A (q_1)

$q_1S_oS_1Rq_2$; $q_2S_0S_0Rq_3$; $q_3S_oS_2Rq_4$;$q_4S_oS_oRq_1$;

Qi → DAAA,…,A (i = no. of A's)
Sj → DCCC,…,C (j = no. of C's)

Standard Description : DADDCRDAA; DAADDRDAAA; DAAADDCCRDAAAA; DAAAADDRDA;

A → 1, C → 2, D → 3, L → 4, R → 5, N → 6, ; → 7

Description Number: 31332531173113353111731113322531111731111335317

Example: Let's construct a TM to decide $A = \{0^{2^n} \mid n \geq o\}$. So A is the language of all strings of 0's that have a length of a power of 2.

$M =$ "on some input string w do . . . "

1. Sweep left to right across the tape and cross off every other 0; if the number of 0's is odd, **reject**.
2. If in Step 1, the tape contained a single 0, **accept**.
3. Return head to the left-end of the tape.
4. Go to Step 1.

TM Execution:

1. Mark the first 0 by A: $\delta(q_1, 0) = (q_2, A, R)$.
   Cross-off next 0 after A: $\delta(q_2, 0) = (q_3, x, R)$.
   Pass 0's at the odd positions and cross-off 0's at the even positions: $\delta(q_3, 0) = (q_4, 0, R)$, $\delta(q_4, 0) = (q_3, x, R)$.
   Make sure x is invisible to $\{q_2, q_3, q_4\}$: $\delta(q, x) = (q, x, R)$ for $q \in \{q_2, q_3, q_4\}$.
   If the number of 0's is odd, **reject**: $\delta(q_4, blank) = (q_r, blank, R)$.
2. Accept for a single 0: $\delta(q_2, blank) = (q_a, blank, R)$.
3. Return head to the left-end of the tape: $\delta(q_3, blank) = (q_5, blank, L)$,
   $\delta(q_5, a) = (q_5, a, L)$ for $a \in \{0, x\}$.
4. Go to Stage 1: $\delta(q_5, A) = (q_2, A, R)$.

Let's draw a state diagram for this TM and designate q_a and q_r as the accept and reject states, respectively.

Finally, let's generate all the machine configurations that would follow the starting configuration $q_1\,0\,0\,0\,0$. Would you expect the TM to accept the tape input? _______

9 Decidability

We have used languages to represent computational problems. A language is **decidable** if there is an algorithm (i.e., a Turing decider) to decide it.

We will consider some languages that are decidable by algorithms.

1. **Membership for DFA** – test whether a particular FA accepts a given string (denoted by A_{DFA}). A_{DFA} contains encodings of all DFA's together with the strings that the DFA's accept, i.e.,

   $$A_{DFA} = \{< B, w > \mid B \text{ is a DFA that accepts } w\}.$$

   Testing whether DFA B accepts w is the same as testing whether $<B,w> \in A_{DFA}$. To show that a computational problem is <u>decidable</u> is to show that the <u>encoding</u> of the problem is decidable.

   Theorem: A_{DFA} is a decidable language.

   Proof. Construct a TM M that decides A_{DFA}.
   M = "On input $<B,w>$, where B is a DFA and w is a string:

   (a) Simulate B on w.
   (b) If simulation ends in accept state, M **accepts**; if simulation ends in a non-accept state, M **rejects**. Note: w is finite and the simulation always ends."

   □

2. **Acceptance for NFA** – define

   $$A_{NFA} = \{< B, w > \mid B \text{ is a NFA that accepts } w\}.$$

   Theorem: A_{NFA} is a decidable language.

   Proof. Construct a TM N that decides A_{NFA}.
   Because M (in previous theorem) was designed to work with DFA's, N first converts its input NFA to a DFA.

   N = "On input $<B,w>$, where B is a NFA and w is a string:

   (a) Convert NFA B to DFA C.

(b) Run TM M from previous theorem.

(c) 3. If M accepts, then N **accepts**; otherwise, N **rejects**."

□

3. **Emptiness Problem** – Test if the language of a DFA is empty. Define

$$E_{DFA} = \{< A > \mid A \text{ is a DFA and } L(A) = 0\}.$$

Theorem: E_{DFA} is a decidable language.

Proof. DFA A accepts some string if and only if it reaches a final state from the start state and travelling along the edges of the DFA. Construct a TM say T that marks the states of the DFA A using the δ function of A. We then use T to solve the emptiness problem.

T = "On input $<A>$, where A is a DFA:

(a) Mark the start state of A.

(b) Repeat until no new states get marked:

  i. Mark any state that has a transition coming into it from any state that is already marked.

  ii. If no final state is marked, T **accepts**; otherwise T **rejects**."

□

4. **Language Equality** – For two DFA's, A and B, is $L(A) = L(B)$?

$$EQ_{DFA} = \{< A, B > \mid A,\ B \text{ are DFA's and } L(A) = L(B)\}.$$

Recall that the **symmetric difference** $L(C) = [L(A) \cap \overline{L(B)}] \cup [\overline{L(A)} \cap L(B)]$ defines what is unique to each of the languages $L(A)$ and $L(B)$. If $L(C) = \emptyset$, then $L(A) = L(B)$. Let's draw a Venn diagram for $L(C)$.

Theorem: EQ_{DFA} is a decidable language.

Proof. Construct a TM F as follows:
F = "On input $<A,B>$, where A and B are DFA's:

(a) Construct DFA C that recognizes $L(C)$, the symmetric difference of $L(A)$ and $L(B)$.
(b) Run TM T from the previous theorem for E_{DFA} on the input $<C>$.
(c) If T accepts, then F **accepts**; otherwise F **rejects**."

□

Given any DFA A on Σ, can we decide if $L(A) = \Sigma^*$?

$$ALL_{DFA} = \{< A > \mid A \text{ is a DFA that recognizes } \Sigma^*\}.$$

Theorem: ALL_{DFA} is a decidable language.

Proof. Construct a TM L that decides ALL_{DFA} using the fact that $\overline{L(A)}$ is regular.
L = "On input $<A>$, where A is a DFA:

1. Construct DFA B that recognizes $\overline{L(A)}$ by swapping accept and unaccept states in A.
2. Run our previous TM T that decides the emptiness of E_{DFA} on B.
3. If T accepts, then L **accepts**; if T rejects, then L **rejects**."

□

Q: Can we describe algorithms to test whether a CFG generates a particular string?

Start with testing whether the language generated by a CFG is empty.

$$A_{CFG} = \{< G, w > \mid G \text{ is a CFG that generates the string } w\}.$$

Theorem: A_{CFG} is a decidable language.

Proof. (**Draft**) Go through all the derivations generated by G checking whether one of them is a derivation of w. But there are infinitely-many derivations?

If G does not generate w, the algorithm does not halt; so we could only produce a recognizer not a decider. □

Q: How can we redesign the recognizer into a decider and only process a finite number of derivations?

Note: If G is a CFG in CNF then for any $w \in L(G)$, where $|w| = n$, exactly $2n - 1$ steps are required for any derivation of w.

Theorem: A_{CFG} is a decidable language.

Proof. (**Revised**) Construct the TM S that decides A_{CFG}. S="On input <G,w>, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in CNF.
2. List all derivations using $2n-1$ steps, where $n = len(w)$; if $n = 0$, list all derivations in 1 step.
3. If any derivations produce w, S **accepts**; otherwise S **rejects**."

□

Emptiness Problem for CFG's

$$E_{CFG} = \{< G > \mid G \text{ is a CFG and } L(G) = 0\}.$$

Theorem: E_{CFG} is a decidable language.

To test whether $L(G)$ is empty, we need to test whether the G can generate a string of terminals. Moreover, can each variable generate a string of terminals?

We need an algorithm (i.e., TM) to cross off terminals and the variables whose grammar rules have right-hand sides comprised of those terminals.

Proof. Construct the TM R= "On input <G>, where G is a CFG:

1. Mark all terminal symbols of G.
2. Repeat until no new variable gets marked:
   Mark any variable A where G has a rule $A \rightarrow u_1 u_2 \cdots u_k$
   and each symbol u_1, u_2, ..., u_k has already been marked.

3. If the start symbol of G is not marked, R **accepts**; otherwise R **rejects**."

□

Theorem: Every CFL is decidable.

Proof. Let G be a CFG for A, i.e. $L(G) = A$.
Design a TM M_G that decides A by building a copy of G into M_G.
M_G= "On input w:

1. Run TM S from the A_{CFG} proof on the input $<G,w>$.
2. 2. If S accepts, then M_G **accepts**; otherwise M_G **rejects**."

□

Lemma: Class of CF languages is **NOT** closed under $\cap$.

Let's construct a language using the intersection that is not CF.

Lemma: Class of CF languages is **NOT** closed under complementation.

Let's construct a language using complementation that is not CF.

CFL Equality Problem:

$$EQ_{CFG} = \{< G, H > \mid G, H \text{ are CFGs and } L(G) = L(H)\}$$

Can't use symmetric difference now since CFLs are **NOT** closed under intersection and complementation.

Theorem: EQ_{CFG} is not a decidable language.

Proof. (By Contradiction) Suppose EQ_{CFG} is decidable and construct a decider M for

$$ALL_{CFG} = \{< G > \mid G \text{ is a CFG and } L(G) = \Sigma^*\}.$$

Then, M= "On input $<G>$:

1. Construct a CFG H such that $L(H) = \Sigma^*$.
2. Run the decider for EQ_{CFG} on $<G,H>$.
3. If the decider accepts, then M **accepts**; otherwise M **rejects**."

So, M decides ALL_{CFG} assuming a decider for EQ_{CFG} exists. But ALL_{CFG} can be shown to be undecidable. This is a **reducibility** argument that we will revisit later. □

Overall Methodology (for proving that a language is decidable):

- Understand relationship between languages.
- Transform relationship into an expression using closure operators on decidable languages.
- Design a TM that constructs language expressed.
- Run TM that decides that language.

---

10 Halting Problem

Membership problem–does a TM accept a given input string?

$$A_{TM} = \{< M, w > \mid M \text{ is a TM and M accepts } w\}.$$

The language A_{TM} is **not** decidable but A_{TM} is Turing-recognizable.
Let's construct a recognizer for A_{TM}:
U = "On input <M,w>, where M is a TM and w is a string:

1. Simulate M on the input w.
2. If M ever enters its accept state, U **accepts**; if M ever enters its reject state, U **rejects** Note: U loops on the input <M,w>, if M loops on w and this is why U does **not decide** A_{TM}."

If the algorithm has some way to determine that M was not halting on w, it could reject. This is known as the **Halting Problem**.

The TM U (named for "universal TM") was proposed by Alan Turing and played an important role in the development of future stored-program computers.

Undecidability–how can we prove a language is undecidable.?

For TM membership, we can exploit George Cantor's diagonalization technique (1873); he wanted to measure the size of infinite sets (i.e., count the number of elements in the set).

However, such a counting approach would not halt.

Examples of infinite sets:

- Set of strings over $\{0, 1\}$
- $\mathbb{N}$ – set of natural numbers
- $\mathbb{E}$ – set of all even natural numbers

Cantor's solution for comparing infinite sets: two infinite sets have the same size if their elements can be paired.

Two sets A and B have the same size if there is a correspondence $f : A \to B$.
We say f is **1-to-1** if it never maps two different elements of A into the same element of B, i.e., $f(a) \neq f(b)$ whenever $a \neq b$.
We say f is **onto** if it hits every element of B, i.e., for all $b \in B$, there exists $a \in A$ such that $f(a) = b$.
We conclude that f is a **correspondence** if it is both 1-to-1 and onto.

Example: Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $\mathbb{E} = \{2, 4, 6, \ldots\}$. Cantor showed that $\mathbb{N}$ and $\mathbb{E}$ have the same size by defining the correspondence $f : \mathbb{N} \to \mathbb{E}$ by $f(n) = 2n$.

A set is **countable** if either it is finite or it has the same size as $\mathbb{N}$.

Let $\mathbb{Q}$ be the set of positive rational numbers, i.e., $\mathbb{Q} = \{m/n \mid m, n \in \mathbb{N}\}$.
Is $\mathbb{Q}$ the same size as $\mathbb{N}$? Let's see if we can define a correspondence between $\mathbb{Q}$ and $\mathbb{N}$:

These are countable (infinite) sets: $\mathbb{N} \times \mathbb{N}$, $\mathbb{N}^k$ for any k, Σ^*, and any subset of a countable set.

Example of an uncountable set: $\mathbb{R}$–the set of real numbers.

Theorem: $\mathbb{R}$ is uncountable.

Proof. Suppose a correspondence $f : \mathbb{N} \to \mathbb{R}$ exists and deduce a contradiction that f cannot be a correspondence, i.e., construct $x \in \mathbb{R}$ that cannot be the image of any $n \in \mathbb{N}$. Assuming the correspondence f exists, we can list all real numbers. Now, construct $x \in (0, 1)$ as follows . . .

1. Let $x = 0.d_1\, d_2\, d_3\, d_4\, \cdots$ with an infinite number of decimals constructed by the following rule: for all $i \in \mathbb{N}$ choose d_i to be different from ith digit of $f(i)$.
2. Then, for all $i \in \mathbb{N}$, $x \neq f(i)$. So x does not belong to our list of all real numbers and f is not a correspondence (contradiction).

□

Some languages are not decidable or even Turing-recognizable.

There are countable many TMs but an uncountable number of languages; each TM can recognize a single language and there are many more languages than TMs so what can we conclude?

That there will be languages that are **NOT** recognized by an TM (these are not T-recognizable).

Examples of uncountable sets: $\mathbb{R}_1 = (0, 1)$, infinite-length binary strings ($\mathbb{B}$), integer functions $F = \{f \mid f : \mathbb{N} \to \{0, 1\}\}$, power set $\mathcal{P}(N)$, and the set of all formal languages $\mathbb{L} = \{L \mid L \subseteq \Sigma^*\}$.

Suppose $\Sigma = \{0, 1\}$ and A is the language of all strings starting with 0 over Σ.
Then, $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 11, 000, 001, 010, 011, \ldots\}$ and $A = \{0, 00, 01, 000, 001, 010, 011, \ldots\}$.
Define the characteristic function

$$\chi_A = 0101101111\ldots$$

The function χ_A runs over each element of Σ^* and returns a 1 if that element is an element of A; otherwise, it returns a 0.

Theorem: Set of all formal languages $\mathbb{L} = \{L \mid L \subseteq \Sigma^*\}$ is uncountable.

Proof. Construct a correspondence $\mathbb{B} \to \mathbb{L}$. Since each Σ^* is countable, we can write $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in L$ has a unique infinite binary sequence $\chi_A \in \mathbb{B}$ constructed as follows: the ith bit of χ_A is such that $\chi_A(i) = 1$ if $s_i \in A$ and $\chi_A(i) = 0$ if $s_i \notin A$. So, χ_A is the **characteristic function** of A in Σ^* and the function $f : \mathbb{L} \to \mathbb{B}$, where $f(A) = \chi_A$, is 1-to-1 and onto, i.e., a correspondence. Since $\mathbb{B}$ is uncountable, $\mathbb{L}$ must be uncountable. $\square$

Q: Can we construct a Turing-unrecognizable language?

Recall that A_{TM} is a Turing undecidable language but it is Turing-recognizable (TR). Constructing a TR language relies on the fact that if both a language and its complement are TR, then the language is decidable. Hence, for any undecidable language, either the language or its complement is not TR.

Theorem: $A_{TM} = \{< M, w > \mid M \text{ is a TM and M accepts } w\}$ is undecidable.

Proof. Assume A_{TM} is decidable and suppose H is a decider of A_{TM}. On input $< M, w >$, where M is a TM and w is a string, H halts and accepts if M accepts w. Similarly, H halts and rejects if M fails to accept w.

Now, construct a new TM D that uses H as a subroutine. That is, D calls H to determine what M does when its input is $w =< M >$.

So if M accepts $< M >$, then $D(< M >)$ rejects and if M rejects $< M >$ then $D(< M >)$ accepts (opposite of what H outputs). Now, run D on $< D >$, then $D(< D >)$ returns accept if D doesn't accept $< D >$ and $D(< D >)$ returns reject if D doesn't reject $< D >$. See the problem? $\square$

Let's use *diagonalization* to arrive at the same impossibility. Start by listing all TMs running on TMs as input . . .

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$
M_1	accept		accept	
M_2	accept	accept	accept	accept
M_3				
M_4	accept	accept		

(Entry (i, j) is accept if M_i accepts $< M_j >$)

Now run H . . .

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$
M_1	accept	**reject**	accept	**reject**
M_2	accept	accept	accept	accept
M_3	**reject**	**reject**	**reject**	**reject**
M_4	accept	accept	**reject**	**reject**

(Entry (i, j) is value of H on $< M_i < M_j >>$)

Now consider the result of running H when D is present . . .

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$	$< D >$
M_1	accept	reject	accept	reject	**accept**
M_2	accept	accept	accept	accept	**accept**
M_3	reject	reject	reject	reject	**reject**
M_4	accept	accept	reject	reject	**accept**
D	**reject**	**reject**	**accept**	**accept**	**?**

(D returns the opposite of $< M_i < M_i >>$)

Summary

Let's draw a Venn diagram that summarizes all the sets of languages we have studied in this course: regular, context-free, decidable, and Turing-recognizable.

11 Reducibility

- Reduction is a terminating process.
- When Problem A is reduced to Problem B, solving problem A cannot be harder than the sum of reduction and solving Problem B; the solution to Problem B should yield the solution to Problem A.
- If Problem A is reduced to Problem B that is decidable, then Problem A is decidable; the solution to Problem B solves Problem A in a finite number of steps.
- If Problem A is undecidable and is reducible to Problem B, then B is undecidable.

Example: Problem A (measuring area of a circle) reduces to Problem B (measuring r, the circle's radius) that reduces to Problem C (performing πr^2).

Example: Problem A (proving a set is uncountable) reduces to Problem B (establishing a correspondence between the set and the set of reals, $\mathbb{R}$).

Methodology:

To prove that a Problem P is undecidable by reduction:

1. Find a Problem Q known to be undecidable.
2. Assume P is decidable by a TM M_P.
3. Use TM M_P to construct a TM M_Q that solves Q: encode every instance q of Problem Q as an instance q_p of Problem P. Use M_P to solve q_p.

Since it is known that Q is undecidable, M_Q cannot exist so M_P cannot exist and P is undecidable.

This methodology works for not Turing-recognizable also.

Undecidable Problems:

Problem P: the halting problem, $HALT_{TM}$ is the problem of determining whether a TM halts on input w:

$$HALT_{TM} = \{< M, w > \mid M \text{ is a TM and TM halts on } w\}.$$

A (known) undecidable Problem Q: we have established the un-decidability of

$$A_{TM} = \{< M, w > \mid M \text{ is a TM and M accepts } w\}.$$

We can use the the undecidability of A_{TM} to show that $HALT_{TM}$ is undecidable by reducing A_{TM} to $HALT_{TM}$.

Theorem: $HALT_{TM}$ is undecidable.

Proof. Assume that TM R decides $HALT_{TM}$ and use R to construct TM S that decides A_{TM}. S="On input $< M, w >$, i.e., an encoding of M and w: Run TM R on $< M, w >$.
If R rejects, i.e., M loops on w, S **rejects**.
If R accepts, i.e., M halts on w, simulate M on w until it halts.
If M has accepted, S **accepts**; if M has rejected, S **rejects**." □

So A_{TM} has been reduced to $HALT_{TM}$.

Q: Can a TM recognize a language recognized by a simpler computational model, such as a regular language?

$REGULAR_{TM}$ is the problem of testing whether a given TM has an equivalent finite automaton:

$$REGULAR_{TM} = \{< M > \mid M \text{ is a TM } \textit{and } L(M) \text{ is regular}\}.$$

Theorem: $REGULAR_{TM}$ is undecidable.
(Proof construction):

We seek to reduce $REGULAR_{TM}$ to A_{TM}. That is, we assume that $REGULAR_{TM}$ is decidable by a TM R and use this assumption to construct the TM S that decides A_{TM}. For S: take input $< M, w >$ and modify M so that the resulting TM M_2 recognizes a regular language if and only if M accepts w.

M_2 recognizes the non-regular language $\{0^n 1^n \mid n \geq 0\}$, if M does not accept w; M_2 recognizes the regular language Σ^* if M accepts w.

Constructing M_2: M_2 accepts (automatically) all strings in $\{0^n1^n \mid n \geq 0\}$. In addition, if M accepts w, then M_2 accepts all other strings.

Theorem: $REGULAR_{TM}$ is undecidable.

Proof. Let R be a TM that decides $REGULAR_{TM}$. Construct the TM S that decides A_{TM}:

S="On input $< M, w >$ where M is a TM and w is a string:

1. Construct the code of TM M_2 as follows: M_2 = "On input x, a) if $x = 0^n1^n$ for some $n \geq 0$, accept; b) if $x \neq 0^n1^n$, run M on w and if M accepts w, then accept."
2. Run R on $< M_2 >$.
3. If R accepts, then S **accepts**; if R rejects, then S **rejects**."

So, if R decides $REGULAR_{TM}$, then S decides A_{TM}; but A_{TM} is undecidable so S cannot exist. Therefore, R cannot exist and $REGULAR_{TM}$ must be undecidable. □

Theorem: (Rice's) Let P be any property about TMs and express P as a language, i.e., P is the language of TMs having property P.

Assume that P satisfies the following two properties:

- For any TMs, M_1 and M_2, where $L(M_1) = L(M_2)$ we have $< M_1 >\in P$ if and only if $< M_2 >\in P$, i.e., membership of a TM M in P depends only on the language of M.
- There exist TMs M_1 and M_2, where $< M_1 >\in P$ and $< M_2 >\notin P$, i.e., P is not trivial – holds for some TMs but not for all.

Then, P is undecidable.

Proof. Suppose P is a decidable language satisfying both conditions of Rice's Theorem. Let T_P be a TM that decides P. Without loss of generality, assume that T_0 with $L(T_0) = \emptyset$ (always rejects) and that $< T_0 >\notin P$. (We would proceed with $\overline{P}$ instead of P if $< T_0 >\in P$.)

Since P is not trivial, there exists a TM M_1 with $< M_1 > \in P$. Now, construct the following TM X. X = "On input $< M, w >$:

1. Construct a TM M_w that accepts input x if and only if M accepts w and M_1 accepts x.
2. Run T_P on $< M_w >$; if T_P accepts, X **accepts**; otherwise X **rejects**."

Now, if $w \in A_{TM}$ and $L(M_w) = L(M_1)$, then $< M_w >$ should be accepted by T_P since $< M_1 > \in P$.

But if $w \notin A_{TM}$ and $L(M_w) = \emptyset = L(T_0)$, then $< M_w >$ should be rejected by T_P according to our assumption that $< T_0 > \notin P$.

So, we have shown that $w \in A_{TM}$ if and only if $< M_1 > \in P$. But, A_{TM} is undecidable and therefore P is also undecidable. $\square$

Example: Prove that $L_x = \{< M > \mid M$ is a TM that writes an x on some cell of the tape when started on a blank$\}$ is undecidable.

Assume L_x is decidable and let R be a TM that decides it. Then define
$S =$ "On input $< M, w >$, where M is a TM and w is an input string do:

1. Construct a new TM M_w such that on input y, we substitute X for x everywhere in $< M >$ and w, creating $< M', w' >$.
2. Run M' on the input w' so that if M' rejects, then M_w rejects; similarly if M' accepts w', then print x on the tape and have M_w accept.
3. Now, run R on $< M, w >$ so that if R accepts, then S **accepts**; similarly if R rejects then S **rejects**."

Example: Prove that $L_{UT} = \{< M > \mid M$ is a TM such that $L(M)$ is any string containing "UT"$\}$ is undecidable.

Assume L_{UT} is decidable and let R be a TM that decides it. Then define $S =$ "On input $< M, w >$, where M is a TM and w is an input string do:

1. Construct a new TM M_w with input string x.
2. Erase the input string x and replace it with the constant string w.
3. Simulate TM M on w.

Now, run R on $< M_w >$ and if R accepts, then S **accepts**; similarly if R rejects, then S **rejects**."

Example: The famous $3x + 1$ problem is based on the following function $f(x)$ over the natural numbers x:

$$f(x) = \begin{cases} 3x + 1, & \text{for odd } x, \\ x/2, & \text{for even } x. \end{cases}$$

If you start with an integer x and iterate f, you obtain a sequence

$$x,\ f(x),\ f(f(x)),\ \dots\ ,$$

and stop if you ever hit 1. For example, if $x = 17$, you would get the sequence

$$17,\ 52,\ 26,\ 13,\ 40,\ 20,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1\ .$$

Extensive computations have shown that every starting point between 1 and a large positive integer yields a sequence that ends in 1. But, the question of whether all positive starting integers yields a sequence terminating at 1 is unsolved.

Suppose that A_{TM} were decidable by a TM H. Let's use H to describe a TM that would be guaranteed to answer the $3x+1$ problem.

Define a TM S on input $< n >$, where n is a natural number. Let $S =$ "On input $< n >$, where n is a natural number, run the $3x+1$ procedure starting at n and **accept** if and when 1 is reached. If n never reaches 1, then S will not halt."

Now, define the TM T on input $< n >$ to use TM H to determine whether or not S accepts $< n >$. If S accepts, then T accepts; T will halt and reject if S does not accept.

12 Complexity

In analyzing TM_A, the time to decide the language A depends on the number of steps that TM_A moves – normally depends on several parameters.

In general,

1. The running time of an algorithm is a function of the length of the string that represents the input.
2. **Worst-case-analysis** is the longest running time of all inputs of the same length.
3. **Average-case-analysis** is the average of all running times of inputs of the same length.

Let M be a deterministic TM that halts on all inputs. The running time or time complexity of M is a function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

If $f(n)$ is the running time of M we say that M runs in time $f(n)$ and that M is an $f(n)$ time TM, where n is the length of the input.

Usually we estimate running time of an algorithm (exact time may be too complicated to determine).

Asymptotic analysis – determine running time of algorithms on large inputs; consider only the highest order term of running time expression (dominates values of other terms on large inputs).
Example:

$$f(n) = 6n^3 + 2n^2 + 10n + 100$$

We say that f is asymptotically at most n^3, and write $f(n) = O(n^3)$.

Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}$. We say that $f(n) = O(g(n))$, if positive integers c and n_0 exist such that for all $n \geq n_0$, $f(n) \leq cg(n)$. We say that $g(n)$ is an asymptotic upper bound for $f(n)$ with the suppression of constant factors.

In practice, most functions f will have an obvious highest order term $h(n)$ so that

$f(n) = O(h(n))$.

For logarithms, $\log_b(n) = \log_2(n)/\log_2(b)$ so that base really doesn't matter.
For example, if

$$f_2(n) = 3n\log_2(n) + 5n\log_2(\log_2(n)) + 2$$

we would say $f_2(n) = O(n\log n)$.

Big-Oh Expressions:

1. If $f(n) = O(n^2) + n$, then $f(n) = O(n^2)$; but if $g(n) = O(n^2)$, we cannot conclude that $f(n) = O(g(n))$.
2. If $f(n) = 2^{O(n)}$, then $f(n) = O(2^{cn})$ for some constant c.
3. If $f(n) = 2^{O(\log n)}$ then $f(n)$ is an upper bound for n^c, constant c. Why?
4. $n^{O(1)}$ representes the value n^c for constant c.
5. Polynomial bounds have the form n^c, for constant c and exponential bounds have the form a^{cn^δ}, for $a > 1$, and $c, \delta > 0$.

Small-Oh notation:
A function is asymptotically less than another function; let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \{f(n)/g(n)\} = 0 \, .$$

This means that for any real number $c > 0$, there exists an integer (size) n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$.
Examples:

Function	Small-Oh
$\sqrt{n}$	$o(n)$
n	$o(n\log(\log(n)))$
$n\log(\log(n))$	$o(n\log(n))$
$n\log(n)$	$o(n^2)$
n^2	$o(n^3)$

Note: $f(n)$ is never $o(f(n))$. In words, Big-Oh means "grows no faster than" and Small-Oh means "grows strictly slower than".

Analyzing Algorithms:

Consider a TM algorithm M_1 that decides $A = \{0^k 1^k \mid k \geq 0\}$. M_1 = "On input, string w:

1. Scan across tape and reject if 0 is found to the right of a 1.
2. Repeat as long as both 0's and 1's remain on the tape: scan across tape, crossing off a single 0 and a single 1.
3. If 0's remain after 1's have been crossed off or if 1's still remain after all 0s have been crossed off, **reject**. Otherwise, if neither 0's nor 1's remain on the tape, **accept**."

Let's analyze the complexity of M_1 for input length n:

Stage 1: Machine scans n steps to verify 0^+1^+ and then moves another n steps to reposition the head. [$2n$ steps = $O(n)$]

Stage 2: Each scan in this stage is performed in $O(n)$ time. Because each scan crosses off a 0 and a 1, at most $n/2$ scans occur; so total number of steps is $(n/2) \times O(n) = O(n^2)$.

Stage 3: Machine makes a single scan to decide whether to accept or reject. [$O(n)$ steps]

So, the running time for M_1 is $O(n) + O(n^2) + O(n) = O(n^2)$.

Let $t : \mathbb{N} \to \mathbb{N}$ be a function. The time complexity class TIME ($t(n)$) is the collection of all languages that are decidable by an $O(t(n))$ time TM.

$A = \{0^n 1^n \mid n \geq 0\}$ is decided by M_1 in $O(n^2)$ steps so $A \in$ TIME (n^2).

Q: Is there a TM that decides A asymptotically faster? Is $A \in$ TIME ($t(n)$) for $t(n) = O(n^2)$?

Note: One can cross two 0's and two 1's in Stage 2 which cuts the number of scans by half but the overall running time does not change.

Consider $M_2 = \text{TIME}(\ n \log n)$ TM, where $M_2 =$ "On input string w:

1. Scan across tape and reject if 0 is found to the right of a 1.
2. Repeat as long as some 0's and some 1's remain on the tape:
   (a) Scan across the tape checking whether total number of 0's and 1's remaining on tape is even or odd; if odd **reject**.
   (b) Scan again across the tape, crossing off every other 0 starting with the first 0 and then crossing off every other 1 starting with the first 1.
3. If no 0's and no 1's remain on tape, **accept**; otherwise **reject**."

." Analysis:

Stage 1: $O(n)$ steps

Stage 2: $(1 + \log_2 n) \times O(n) = O(n \log_2 n)$ steps

Stage 3: $O(n)$ steps

So, asymptotically $A \in \text{TIME}\ (n \log_2 n)$.

Q: What would the time complexity be using two tapes?

Consider a two-tape M_3 that decides A in linear time, i.e. $O(n)$ time.
$M_3 =$ "On input string w on tape 1:

1. Scan across tape 1 and reject if a 0 found to right of a 1.
2. Scan across 0's on tape 1 until the first 1 is found and at the same time copy those 0's to tape 2.
3. Scan across 1's on tape 1 until the end of the tape is reached. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0's are crossed off before all the 1's are read, **reject**.
4. If all 0's have been crossed off, **accept**; if any 0's remain, **reject**."

Summary:

TM	Tapes	Time Complexity
M_1	1	$O(n^2)$
M_2	1	$O(n \log n)$
M_3	2	$O(n)$

Class P:
Languages that are decidable in polynomial time on a deterministic (singe-tape) TM.

$$\mathbf{P} = \bigcup_k \text{TIME}(n^k)$$

P is invariant for all models of computation that are polynomially equivalent to the deterministic (single-tape) TM.

P roughly corresponds to the class of problems that are realistically solvable by computers.

A **verifier** for a language A is an algorithm V where $A = \{w \mid V \text{ accepts } < w, c > \text{ for some string } c\}$.

The running time of a verifier is measured in terms of the length of w. A polynomial-time verifier runs in polynomial time in length of w.

We say that a language A is polynomial verifiable if it has a **polynomial-time verifier** (PTV).

Theorem: Every CFL is a member of **P**.

Languages that have polynomial-time verifiers (PTV) belong to Class **NP**.

NP is defined as "nondeterministic polynomial time".

Hamiltonian Path Problem $\in$ **NP** (i.e., take two specified nodes in a directed graph and trace a path from one to the other that goes through each node of the graph exactly once).

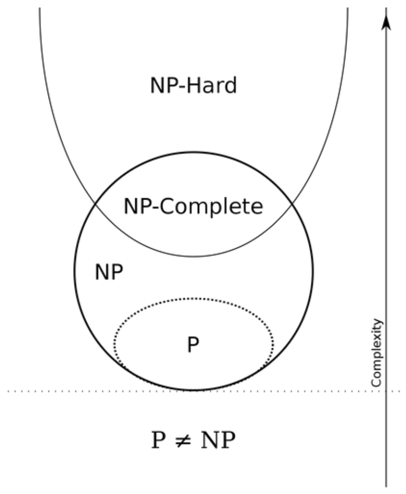
P versus **NP**:

- Problems in **P** are quickly *solvable* such as multiplication and sorting.
- Problems in **NP** are quickly *checkable* such as factoring and finding max cliques (largest complete subgraph) in a graph.
- Finding the max clique of an undirected graph having hundreds of vertices could take centuries of computer time.
- If **P** = **NP**, then the searching needed to solve **NP** problems would be eliminated.

P = **NP**?

This remains an unsolved problem in theoretical computer science. If there is a polynomial-time algorithm for certain problems in **NP**, then all problems in **NP** would be polynomial-time solvable. Such problems are called **NP-complete**. Problems referred to as **NP-hard** (see Venn diagram below) are not in NP but there is some NP-complete problem that is reducible to the NP-hard problem in polynomial time. The complexity of NP-complete problems relates to the entire class of **NP** problems. So, in one sense, problems in **NP** are *linked* to each other.

A language B is **NP-complete** if 1) $B \in$ **NP**, and 2) every $A \in$ **NP** is polynomial-time reducible to B.



Dynamic Programming:

In order to prove that every CFL is a member of **P** , we can deploy a dynamic programming algorithm to determine whether each variable in a CFG G generates each substring of a given string $w = w_1 w_2 \cdots w_n$ in the language generated by G.

The algorithm enters the solution to each subproblem into an $n \times n$ table. For $i \leq j$, the (i,j) entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$. For $i > j$, the table entries are unused.

Let G be a CFG in CNF that generates the CFL L. Assume that S is the start variable and that the algorithm (using dynamic programming) will handle the special case when w=ϵ in step (stage) 1.

D = "On input $w = w_1 w_2 \cdots w_n$:

```
For $i = 1$ to $n$:
    For each variable $A$:
        Test whether $A \rightarrow w_i$ is a rule.
        If so, place $A$ in Table($i$,$j$).
For $l = 2$ to $n$: // $l$ is length of substring
    For $i = 1$ to $n - l + 1$: // $i$ is start pos. of substring
        Let $j = i + l - 1$ // $j$ is end pos. of substring
        For $k = i$ to $j - 1$: // $k$ is the split position
            For each rule $A \rightarrow BC$
                If Table($i$,$k$) contains $B$ and
                Table($k + 1$,$j$) contains $C$, put $A$ in Table ($i$,$j$).
```

If S is in Table (1,n), **accept**; else **reject**."

Let v be the number of variables in G and assume it is independent of n (size of the input string). Assume the CFG G has r rules.

Algorithm D runs in $O(n^3)$ time so that the CFL L belongs to TIME (n^3) which is in **P** .

Example CFG : $S \to RT,\ R \to TR \mid a,\ T \to TR \mid b$

Let's generate the Table of Algorithm D for the following cases of w : *baab*, *bbab*, *abaa*.

	1	2	3	4
1	T	R, T	R, T	S
2		R	n/a	n/a
3			R	S
4				T

G: $S \to RT,\ R \to TR \mid a,\ T \to TR \mid b$

Table 1: $w = baab$

	1	2	3	4
1	T	n/a	R, T	S
2		T	R, T	S
3			R	S
4				T

G: $S \to RT,\ R \to TR \mid a,\ T \to TR \mid b$

Table 2: $w = bbab$

Satisfiability:

A Boolean formula is *satisfiable* if some assignments of 1 and 0 (true and false) to its variables makes the formula evaluate to 1.

$$\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

with $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1.

The satisfiability problem (or **SAT**) is to test whether a Boolean formula is satisfied, i.e., SAT $= \{< \phi > \mid \phi$ is a satisfiable Boolean formula$\}$; SAT is **NP-complete**.

Cook-Levin Theorem: SAT $\in$ **P** if and only if **P** $=$ **NP**.

3SAT:

A *clause* is several literals (Boolean variables) connected with $\vee$'s such as

$$x_1 \vee \bar{x_2} \vee \bar{x_3} \vee x_4 \,.$$

A Boolean formula is in **conjunctive normal form** (called a cnf-formula) if it comprises several clauses connected by $\wedge$'s. A **3cnf**-formula has clauses with three literals in each clause such as

$$(x_1 \vee \bar{x_6} \vee \bar{x_7}) \wedge (x_1 \vee \bar{x_2} \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_4} \vee \bar{x_5}) \,.$$

We define the language **3SAT** by $\{< \phi > \mid \phi$ is a satisfiable 3cnf-formula$\}$.

It can be shown that all languages in **NP** can be reduced to **3SAT** in polynomial time, i.e., **3SAT** is **NP-complete**.

To reduce (in polynomial time) **3SAT** to a given language, it is helpful to identify structures in the language that can simulate variables and clauses in Boolean formulas.

Such structures are called *gadgets*.

---

13 Lambda-Calculus

λ-calculus was developed by A. Church in the early 1930s; he used it in his proof of the undecidable E-problem that Turing also addressed.

Recall the differences between procedural/imperative programming languages (C,C++, Java, C#) and functional programming languages (Lisp, APL, Haskell, Schema, F#).

Church's first paper on λ-calculus was published in the Annals of Mathematics in April, 1932.

Notation:

- Expression: $x^2 + 5x + 7$
- Function notation: $f(x) = x^2 + 5x + 7$ or $f(y) = y^2 + 5y + 7$
- Functional value: $f(4) = 4^2 + 5(4) + 7 = 43$
- Keeping independent variable, we can write:
$$[y^2 + 5y + 7](4)$$
- For more than one variable, order matters:
$$[y^2 + 5y + 18x - 2xy^2 + 7](4, 5)$$
- Church's notation for a function of one variable: $\lambda x[M]$
  Example: $\lambda x[x^2 + 5x + 7]$
- A function $F(x)$ with a value A for the variable x can be denoted $\{F\}(A)$
- If the function has an independent variable x, the proper notation would be $\{\lambda x[M]\}(A)$;
  Example: $\{\lambda x[x^2 + 5x + 7]\}(A)$
- Function of two independent variables: $\{\lambda x \lambda y[y^2 + 5y + 18x - 2xy^2 + 7]\}(A, B)$
- $\lambda x \lambda y[M](A, B)$ can also be written as $\lambda x \lambda y.M(A, B)$

Rules of Conversion:

- You can change a bounded variable ($x \to y$), if the new variable does not collide with the formula.
- For $\{\lambda x[M]\}(N)$, if N has no expression in x you can substitute N for all occurrences of x in M.
- "conv" means *by conversion* to indicate that one formula has been converted into another equivalent formula:

$$\lambda y[y^2 + 5y + 7](A) \text{ conv } A^2 + 5A + 7$$

- "$\to$" means *is an abbreviation for*:

$$1 \to \lambda f x.f(x) \text{ or } 1 \to \{\lambda f x[f(x)]\} \text{ or } 1 \to \lambda ab.a(b)$$

- $S \to \lambda \rho f x.f(\rho(f, x))$

$$2 \to S(1) \text{ or "2 succeeds 1"}$$

- Can write $S(1) = \{\lambda \rho f x.f(\rho(f, x))\}(\lambda ab.a(b))$ so that first bound variable (ρ) is replaced by the expression for 1.

Let's derive the succession of 1, i.e., start with $S(1) = \{\lambda \rho f x.f(\rho(f, x))\}(\lambda ab.a(b))$

So, $S(2) \rightarrow S(1)$ conv ____________________ and $S(3) \rightarrow S(2)$ conv ____________________.

Addition:

- Church's student Cole Kleene developed more operators like "+" circa 1934:

$$+ \rightarrow \lambda\rho\sigma f x.\rho(f, \sigma(f, x))$$

- Let's evaluate $2 + 3$ with $2 \rightarrow \lambda ab.a(a(b))$ and $3 \rightarrow \lambda cd.c(c(c(d)))$.

  $\{+\}(2, 3)$ is . . .

  Substitute 2 for ρ and 3 for σ.

  Now, substitute f for c and x for d.

Finally, substitute f for a and $f(f(f(x)))x$ for b.

Multiplication:

- Operator definition: $\times \rightarrow \lambda\rho\sigma x.\rho(\sigma(x))$
- Let's evaluate 2×3 with $2 \rightarrow \lambda ab.a(a(b))$ and $3 \rightarrow \lambda cd.c(c(c(d)))$.

  $\{\times\}(2,3)$ is ...

  Substitute 2 for ρ and 3 for σ.

Substitute x for c.

Substitute $\lambda d.x(x(x(d)))$ for a.

Now, substitute b for d.

Finally, substitute $x(x(x(b)))$ for d.

Church proposed that all λ-definable functions are all the effectively calculable functions; Kleene studied relationships between recursive functions and λ-definable functions.

Equivalence to Turing's work on E-Problem provided in the Appendix of Turing's 1936 paper.

λ-functions in Python:

Python supports the creation of anonymous functions (not bound to a name) at runtime – called **lambda functions** although not exactly the same as the one's we just considered (and provided in Lisp).

```
>>>   (lambda x : x**2)(3)
>>>   9
```

Typically used to encapsulate specific, non-reusable code without having to write several one-line functions:

```
processFunc = collapse and
(lambda s: \ ".join(s.split())) or (lambda s: s)
```

If `collapse` is true, `processFunc(string)` will collapse whitespace or return the string unchanged.

Also used to create *jump tables*, i.e., lists or dictionaries of actions to be performed on demand:

```
>>>   L = [lambda x : x**2, lambda x : x**3,
      lambda x : x**4]
>>>   for f in L: print(f(3))
>>>   9
      27
      81
>>>   print(L[0](11))
>>>   121
```