CS334: Theory of Computation Notes

Steven DeFalco

Fall 2023

Contents

1 Introduction

1.1 Automata, Computability, and Complexity

The central question of ***complexity theory*** is *what makes some problems computationally hard and others easy?*; the answer is unknown. Cryptography is unique in that is specifically requires computational problems that are hard, rather than easy. In ***complexity theory***, the objective is to classify problems as easy ones and hard ones; whereas in ***computability*** theory, the classification of probelms is by those that are solvable and those that are not.
Automata theory deals with the definitions and properties of mathematical models of computations.

1.2 Mathematical Notions and Terminology

1.2.1 Sets

A ***set*** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even othber sets. The objects in a set are called its *elements* or *members*. One way to describe a set is to list the set's elements inside braces. Thus the set

$$S = \{7, 21, 57\}$$

contains the elements 7, 21, and 59. The symbols $\in$ and $\notin$ denote set membership and nonmembership. We say that A is a ***subset*** of B, written $A \subseteq B$, if every member of A is also a member of B. We say that A is a ***proper subset*** of B, written $A \subset B$, if A is a subset of B and not equal to B.

The order of describing a set doesn't matter, nor does repetition of its members. If we do want to take the number of occurrences of members into account, we call the group a ***multiset*** instead of a set. An ***infinite set*** contains infinitely many elements.

We write the set of ***natural numbers*** N as

$$\{1, 2, 3, \dots\}$$

. The set of ***integers*** Z is written as

$$\{\dots, -2, -1, 0, 1, 2, \dots\}$$

The set with zero members is called the ***empty set*** and is written $\emptyset$. A set with one member is sometimes called a ***singleton set*** and a set with two members is called an ***unordered pair***.

When we want to describe a set containing elements according to some rule, we write $\{n \,|\, \text{rule about} n\}$.

If we have two sets A and B, the ***union*** of A and B, written $A \cup B$, is the set we get by combining all the leements in A and B into a single set. The ***intersection*** of A and B, written $A \cap B$, is the set of elements that are both A and B. The ***complement*** of A, written $\overline{A}$, is the set of elelements under consideration that are *not* in A.

1.2.2 Sequences and Tuples

A ***sequence*** of objects is a list of these objects in some order. We usually designate a sequence by writing the list within parentheses. For example, the sequence 7,21,57 would be written

$$(7, 21, 57)$$

. The order does matter in a set.

Finite sequences often are called ***tuples***. A sequence with k elements is a ***k-tuple***. A 2-tuple is also called an ***ordered pair***.

Sets and sequences may appear as elements of other sets and sequences. For example, the ***power set*** of A is the set of all subsets of A. If A is the set 0,1, the power set of A is the set $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.

If A and B are two sets, the ***Cartesian product*** or defcross product of A and B, written $A \times B$, is the set of all ordered pairs wherein the first element is a member of A and the second elemetn is a member of B.

1.2.3 Functions and Relations

A ***function*** is an object that sets up an input-output relationship. A function takes an input and produces and output. In every function, the same input always produces the same output.

A function is also called a ***mapping***, and, if $f(a) = b$, we say that f maps a to b.

The set of possible inputs to the function is called its ***domain***. The outputs of a function come from a set called the ***range***. The notation for saying that f is a function with domain D and range R is

$$f : D \rightarrow R$$

When the domain of a function f is $A_1 \times \cdots \times A_k$ for some sets $A_1, \ldots, A_k$, the input to f is a k-tuple and we call the a_i the ***arguments*** to f. A function with k arguments is called a ***k-ary function***, and k is called the ***arity*** of the function. If k is 1, f has a single argument and f called a ***unary function***. If k is 2, f is a ***binaryh function***. Certain familiar binary functions are written in a special ***infix notation***, with the symbol for the function placed between its two arguments, rather than in ***prefix notation***, with the symbol preceding.

A ***predicate*** or ***property*** is a function whose range is $\{\text{TRUE}, \text{FALSE}\}$. For example, let *even* be a property that is TRUE if its input is an even number and FALSE if its input is an odd number.

A property whose domain is a set of k-tuples $A \times \cdots \times A$ is called a ***relation***, a ***k-ary relation***, or a ***k-ary relation on A***.

A special type of binaryh relation, called 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 three conditions:

1. R is ***reflexive*** if for every x, xRx;
2. R is ***symmetric*** if for every x and y, xRy implies yRx; and
3. R is ***transitive*** if for every x,y, and z, xRy and yRz implies xRz.

1.2.4 Graphs

An ***undirected graph***, or simply a ***graph***, is a set of points with lines connecting some of the points. The points are called ***nodes*** or ***vertices***, and the lines are called ***edges***.

The number of edges at a particular node is the ***degree*** of that node. No more thatn one edge is allowed between any two nodes. We may allow an edge from a node to itself, called a ***self-loop***.

We say that graph G is a ***subgraph*** of graph H is the nodes of G are a subset of the nodes of H, and the edges of G are the edges of H on the corresponding nodes.

A ***path*** in a graph is a seqeunce of nodes connected by edges. A ***simple path*** is a path that doesn't repeat any nodes. A graph is ***connected*** if every two nodes have a path between them. A path is a ***cycle*** if it starts and ends in the same node. A ***simple cycle*** is one that contains at least three nodes and repeates only the first and last nodes. A graph is a ***tree*** if it is connected and has no simple cycles. A tree may contain a specially designated node called the ***root***. The nodes of degree 1 in a tree, other than the root, are called the ***leaves*** of the tree.

A ***directed graph*** has arrows instead of lines. The number of arrows pointing from a particular node is the ***outdegree*** of that node, and the number of arrows pointing to a particular node is the ***indegree***.

A path in which all the arrows point in the same direction as its steps is called a ***directed graph***. A directed graph is ***strongly connected*** if a directed path connects every two nodes.

1.2.5 Strings and Languages

Strings of characters are fundamental building blocks in computer science. The alphabet over which the strings are defined may vary with the application. For our purposes, we define an ***alphabet*** to be any nonemtpy finite set. The members of the alphabet are ***symbols*** of the alphabet. We generally use capital Greek letter Σ and Γ to dersignate alphabets.

A **string over an alphabet** is a finite sequence of syumbols from that alphabet, usually written next to one another and not separated by commas. If w is a string over Σ, the ***length*** of w, written $|\,w\,|$ is the number of symbols that it contains. The string of length zero is called the ***empty string*** and is written ϵ. The ***reverse*** of w, written w^R, is the string obtained by writing w in the opposite order. String z is a ***substring*** of w if z appears consecutively within w.

If we have a string x of length m and string y of length n, the ***concatenation*** of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1 \cdots x_m y_1 \cdots y_n$.

The ***lexicographic order*** of strings is the same as the familiar dictionary order. We'll occasionally use a modified lexicographic order, called ***shortlex order*** or simply ***string order***, that is identical to lexicographic order, except that shorter strings precede longer strings. Thus the string ordering of all strings over the alphabet $\{0, 1\}$ is

$$(\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots)$$

Say that string x is a ***prefix*** of string y if a string z exists where $xz = y$, and that x is a ***proper prefix*** of y if in addition $x \neq y$. A ***language*** is a set of strings. A language is a ***prefix-free*** if no member is a proper prefix of another member.

1.2.6 Boolean Logic

Boolean logic is a mathematical system built around the two values TRUE and FALSE. The values TRUE and FALSE are called the ***Boolean values*** and are often represented by the values 1 and 0.

We can manipulate Boolean values with the ***boolean operations***. The simplest boolean operation is the ***negation*** or ***NOT*** operation, designated with the symbol $\neg$. The negation of a Boolean value is the opposite value. We designate the ***conjunction*** or ***AND*** operation with the symbol $\wedge$. The conjunction of two Boolean values is 1 if both of those values are 1. The ***disjunction*** or ***OR*** operation is designated with the symbol $\vee$. The disjunction of two Boolean values is 1 if either of those values is 1.

The ***exclusive or*** or ***XOR*** operation is designated by the $\oplus$ symbol and is 1 if either but not both of its two operands is 1. The ***equality*** operation, written $\leftrightarrow$, is 1 if both if its operandws have the same value. Finally, the ***implication*** operation is designated by the symbol $\rightarrow$ and is 0 if its first operand is 1 and its second operand is 0; otherwise, $\rightarrow$ is 1.

The ***distributive law*** for AND and OR comes in handy when we manipulate Boolean expression; it comes in two forms:

- $P \wedge (Q \vee R)$ equals $(P \wedge Q) \vee (P \wedge R)$, and its dual
- $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$

1.3 Definitions, Theorems, and Proofs

Definitions describe the objects and notions that we use. When defining some object, we must make clear what constitutes that object and what does not.

After we have defined various objects and notions, we usually make ***mathematical statements*** about them. Typically, a statement expresses that some object has a certain property.

A ***proof*** is a convincing logical argument that a statement is true.

A ***theorem*** is a mathematical statement proved true. Occasionally, we prove statements that are interesting only because they assist in the proof of another, more significant statement. Such statements are called ***lemmas***. Occasionally a theorem or its proof may allow us to conclude easily that other, related statements are true. These statements are called ***corollaries*** of the theorem.

1.3.1 Finding Proofs

The only way to determine the truth or falsity of a mathematical statement is with a mathematical proof. Experimenting with examples is especially helpful. Thus if the statement says that all objects of a certain type have a particular property, pick a few objects of that type and observe that they actually do have that property. After doing so, try to find an object that fails to have the property, called a ***counterexample***. If the statement actually is true, you will not be able to find a counterexample.

1.4 Types of Proofs

Several types of arguments arise frequently in mathematical proofs. Here, we describe a few that often occur in the theory of computation.

1.4.1 Proof by Construction

Many theorems state that a particular type of object exists. One way to prove such a theorem is by demonstrating how to construct the object. This technique is a ***proof by construction***.

1.4.2 Proof by Contradiction

In one common form of argument for proving a theorem, we assume that the theorem is false and then show that this assumption leads to an obviously false consequence, called a contradiction.

1.4.3 Proof by Induction

Proof by induction is an advanced method used to show that all elements of an infinite set have a specified property. For example we may use a proof by induction to show that an arithmetic expression computes a desired quantity for every assignment to its variables, or that a program works correctly at all steps or for all inputs.

Every proof by induction consists of two parts, the ***basis*** and the ***induction step***. Each part is an individual proof on its own. In the induction step, the assumption that $P(i)$ is true is called the ***induction hypothesis***.

Basis: Prove that $P(i)$ is true. **Induction step:** For each $i \geq 1$, assume that $P(i)$ is true and use this assumption to show that $P(i + 1)$ is true.

2 Regular Languages

Real computers are quite complicated: too much so to allow us to set up a manageable mathematical theory of them directly. Instead, we use an idealized computer called a ***computational model***. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. Thus we will use several different computation models, depending on the features we want to focus on. We begin with the simplest model, called the ***finite state machine*** or ***finite automaton***.

2.1 Finite Automatata

Finite automata are good models for computers with an extremely limited amount of memory. Finite automatat and their probabilistic counterpart ***Markov***

chains are useful tools when we are attempting to recognize patterns in data.

Finite automata can be represented by a ***state diagram***. The ***start state*** is indicated by the arrow point at it from nowhere. The ***accept state*** is the one with a double circle. The arrows going from one state to another are called ***transitions***. When this such automation receives an input, it processes that and produces an output; the output is either ***accept*** or ***reject***.

2.1.1 Formal Definition of Finite Automaton

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a tart state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language, a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a ***transition function***, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, that means that if the automaton is in state x when it reads a 1, it then moves to state y. We can indicate the same thing with the transition function by saying that $\delta(x, 1) = y$. This notation is a kind of mathematical shorthand.

Definition 2.1 (Finite automaton) A 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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

If A is the set of all strings that machine M accepts, we say that A is the ***language of machine M*** and write $L(M) = A$. We say that ***M recognizes A*** or that ***M accepts A***. Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language-namely, the empty language $\emptyset$.

2.1.2 Formal Definition of Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \dots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n-1$, and
3. $r_n \in F$

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w \mid M \text{ accepts } w\}$.

Definition 2.2 In state q, if the next input symbol is a, the next state is $\delta(q, a)$. ***Define:*** $\Delta : Q \times \Sigma* \to Q$ recursively as follows:

$\Delta(q, \epsilon) = q$
$\Delta(q, ax) = \Delta(\delta(q, a)), a \in \Sigma, x \in \Sigma*$

$M = (Q, \Sigma, \delta, q_0, F)$ ***accepts*** a string $w = w_1 w_2 \cdots w_n$ iff $\Delta(q_0, w) \in F$

Definition 2.3 (Regular Language) A language that is recognized by some finite automaton

2.1.3 Designing Finite Automata

Try putting yourself in the place of the machine you are trying to design and then see how you would go about performing the machine's task. Suppose that you are given some language and want to design a finite automaton that recognizes it. Pretending to be the automaton, you receive an inpuit string and must determine whether it is a member of the language the automaton is supposed to recognize. First, in order to make these decisions, you have to figure out what you need to remember about the string as you are reading it. For many languages, you don't need to remember the entire input. You need to remember only certain crucial information. Exactly which information is crucial depends on the particular language considered.

Once you have determined the necessary information to remember about the string as it is being read, you represent this information as a finite lit of possibilities. Then you assign a state to each of the possibilities. Next, you assign the transitions by seeing how to go from one possibility to another upon reading a symbol. Next, you set the start state to be the stae corresponding to the possibility associated with having seen 0 symbols so far. Last, set the accept states to be those corresponding to the possibilities where you want to accept the input string.

2.1.4 The Regular Operations

In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operatioons on languages, called the ***regular operations***, and use them to study properties of the regular languages.
[Regular Operations] Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- ***Union***: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- ***Concatenation***: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- ***Star***: $A* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

The union operation takes all the strings in both A and B and lumps them together into one language. The concatenation operation attaches a string from A in front of a string from B in all possible ways to get the strings in the new language. The star operation is ***unary operation*** instead of a ***binary operation***. It works by attaching any number of strings (including 0) in A together to get a string in the new language. Even if A doesn't contain the empty string, $A*$ *does* contain the empty string.

Example 2.1 Let $A = \{oreo, ginger\}$ and $B = \{cookie, icecream\}$...
$A \cup B = \{oreo, ginger, cookie, icecream\}$
$A \circ B = \{oreocookie, oreoicecream, gingercookie, gingericecream\}$
$A* = \{\epsilon, oreo, ginger, \ldots\}$

Proof. **Proof:** Let A, B be regular languages recognized by DFAs M_1 and M_2.
Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$
Define a new DFA $M = (Q, \Sigma, \delta, q, F)$:
$Q = Q_1 \times Q_2$ each state of M is a pair of states, one from M_1, and the other from M_2
$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$ a transition in M tracks transitions in both M_1, M_2
$q_0 = (q_1, q_2)$ start M in the start states of M_1, M_2
$F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$ accept if one of the machines end in an accept state
$\cdots$ A formal proof of correctness proceeds by induction on the length of the input string$\cdots$ □

2.2 Nondeterminism

When a machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this ***deterministic*** computation. In a ***nondeterministic*** machine, several choices may exist for the next state at any point. Nondeterminism is a generalization of determinism, so every determinsitic finite automaton is atuomatically a nondeterministic finite automaton.

Machine is initially in start state. From the start state, clones appear in all states reachable using only ϵ transitions. Each clone in its current state.

1. Reads the next input symbol a
2. If there is no outgoing transition labeeld a: the clone dies
3. For each outoging transition labeled a: a clone appears at the next state
4. ...

Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. NFAs may violate that rule. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol. Additionally, in a DFA, labels on the transition arrows are symbols from the alphabet. An NFA may have arrows labeled with members of the alphabet or ϵ. Zero, one, or many arrows may exit from each state with the label ϵ.

So how does an NFA compute… Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. After reading said symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ϵ symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ϵ-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

Definition 2.4 (Nondeterministic Finite State Automata (NFA)) a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states
2. Σ is a finite alphabet
3. $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states

For any set Q we write $P(Q)$ to be the collection of all subsets of Q. Here $P(Q)$ is called the ***power set*** of Q. For any alphabet Σ, we write Σ_ϵ to be $\Sigma \cup \{\epsilon\}$. Thus, we can write the formal description of the type of the transitioon function in an NFA as $\delta : Q \times \Sigma_\epsilon \to P(Q)$.

Definition 2.5 (Compuitation for an NFA) Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then we say that N *accepts* w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ϵ and a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q with three conditions.

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

Condition 1 says that the machine starts out in the start state. Condition 2 says that state r_{i+1} is one of the allowable next states when N is in the state r_i and reading y_{i+1}. Observe that $\delta(r_i, y_{i+1})$ is the *set* of allowable next states and so we say that r_{i+1} is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

2.3 Equivalence of NFAs and DFAs

Deterministic and nondeterministic finite automato recognize the same class of languages. Say that two machines are ***equivalent*** if they recognize the same language.

Theorem 2.1 Every nondeterministic finite automaton has an equivalent deterministic finite automaton

Lemma 1 A language is regular if and only if some nondeterminstic finite automaton recognizes it.
One part of the conditioon states that a language is regular if some NFA recognizes it. Consequently, if an NFA recognizes some language, so does some DFA, and hence the language is regular. The condition also states that a language is regular only if some NFA recognizes it. That is, if a language is regular, some NFA must be recognizing it.

For a NFA with n states, the equivalent NFA will need to have 2^n states.

2.4 Closure under the Regular Operations

Theorem 2.2 The class of regular languages is closed under the union operation. In other words, if A, B are both regular langauges then $A \cup B$ is regular as well.

Theorem 2.3 The class of regular languages is closed under the concatenation operation.

Theorem 2.4 The class of regular languages is closed under the star operation.

2.5 Regular Expressions

We can use the regular operations to build up expressions describing languages, which are called ***regular expressions***. The value of a regular expression is a language.

Definition 2.6 (Formal Definition of a Regular Expression) Say that R is a ***regular expression*** if R is...

1. a for some a in the alphabet Σ
2. ϵ
3. $\emptyset$
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*), where R_1 and R_2 are regular expressions

In items 1 and 2, the regular expressions a and ϵ represent the languages $\{a\}$ and $\{\epsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Remark The expression ϵ represents the language containing a single string—namely, the empty string—wheras $\emptyset$ represents the language that doesn't contain any strings.

Remark If we let R be any regular expression, we have the following identities.

- $R \cup \emptyset = R$
  Adding the empty language to any other language will not change it
- $R \circ \epsilon = R$
  Joining the empty string to any string will not change it

However, exchanging $\emptyset$ and ϵ in the preceding identities may cause the equalities to fail.

- $R \cup \epsilon$ may not equal R.
  For example if $R = 0$, then $L(R) = \{0\} but L(R \cup \epsilon) = \{0, \epsilon\}$.
- $R \circ \emptyset$ may not equal R.
  For example if $R = 0$, then $L(R) = \{0\}$ but $L(R \circ \emptyset) = \emptyset$ (i.e. $R \cup \epsilon = R iff \epsilon \in R$).
- $R \circ \emptyset = \emptyset$
- $R \circ \epsilon = R$

- $\emptyset^* = \{\epsilon\}$

Example 2.2

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

- $L_1 = \{$3rd last bit is 1$\}$
  $(0 \cup 1)^*1(0 \cup 1)(0 \cup 1)$ or $\Sigma^*1\Sigma\Sigma$
- $L_2 = \{$strings in which every 0 is followed by at least one 1$\}$
  $(01 \cup 1)^*$ or $1^*(011^*)^*$
- $L_3 = \{$strings starting with 0 and followed by any number of 1s OR strings with no 0$\}$
  $01^* \cup 1^*$ or $(0 \cup \epsilon)1^*$
- $L_4 = \{$strings with lengths that are multiples of 3$\}$
  $((0 \cup 1)(0 \cup 1)(0 \cup 1))^*$ or $(\Sigma\Sigma\Sigma)^*$
- $L_5 = \{$strings that end with 01$\}$
  Σ^*01

Elemental objects in a programming language, called ***tokens***, such as the variable names and constants, may be described with regular expressions. Once the syntax of a programming language has been described with a regular expression in terms of its toekns, automatic systems can generate the ***lexical analyzer***, the part of a compiler that initially process the input program.

2.6 Equivalence with Finite Automata

Regular expressions and finite automata are equivalent in their descriptive power. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa.

Theorem 2.5 A language is regular if and only if some regular expression describes it.

Lemma 2 If a language is described by a regular expression, then it is regular.

Lemma 3 If a language is regular, then it is described by a regular expression

Definition 2.7 (Generalized nondeterministic finite automaton) are nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only memebers of the alphabet or ϵ. The GFNA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GFNA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow.

Like an NFA, but the *start state* has no incoming transition and has outoging transitions to every state. The *accept state* has no outgoing transition, has incoming transitions from every state, and is distinct from the start state. All the *other states* have outgoing transitions to every state. Transitions are labeled with regular expressions.

Definition 2.8 (DFA to GFNA Conversion) 1. Convert the DFA into a generalized NFA (DFA $M \to$ GNFA G)

- Create a new start state with outgoing ϵ-transition to the start state of M
- Create a new accept state with incoming ϵ-transitions from every accept state of M
- If there are multiple labels on any transition, replace it with the union of the labels

2. Derive the regular expression from the GNFA.
The idea is to manipulate labels so that the label on (s, a) describes all strings accepted by the GNFA. Elimintae states one-at-a-time updating labels at each step. /

Definition 2.9 (Formal Definition of GFNA) A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the fintie set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to R$ is transition function
4. q_{start} is the start state
5. q_{accept} is the accept state

A GFNA accepts a string w in Σ^* if $w = w_1 w_2 \dots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \dots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_k = q_{\text{accept}}$ is the accept state
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_i - 1, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Theorem 2.6 For every regular language L, there exists a *unique* minimal DFA M such that $L = L(M)$. However, this is *not true* for NFAs; there is not necessarily a unique minimal DFA.

Definition 2.10 (Indistinguishable States) States p and q are indistinguishable: $p \sim q$ if for every string w, the computation starting in state p reaches an accept state if and only if the computation from q reaches an accept state.

Lemma 4 The indistinguishability relation, $\sim$, is an equivalence relation. **Proof**...

- **Reflexive**: $p \sim p$
- **Symmetric**: $p \sim q \leftrightarrow q \sim p$

- **Transitive**: $(p \sim q \cap q \sim r) \to p \sim r$

Lemma 5 (The Distinguishability Lemma) If for some symbol $\delta(p,a) \not\sim \delta(q,a)$ then $p \not\sim q$

Definition 2.11 (Finding Distinguishable Pairs of States) .
Initialize: Create an empty set D which will store pairs of distnguishable states
Step 0:
Add all pairs (p,a) where exactly one is an accept state to D. While True do: Scan over all pairs (p,q) of states not yet distinguished. For each alphabet symbol a. If the pair $(\delta(p,a), \delta(q,a))$ is a distinguished, add (p,q) to D. If no new pair of states is added to $D\dots$ break. Return D

2.7 Nonregular Languages

The ***pumping lemma*** states taht all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value called the ***pumping length***. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

Theorem 2.7 (Pumping Lemma) If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of lenght at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

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

Recall the notation where $|s|$ represents the length of the string s, y^i means that i copies of y are concatenated together, and y^0 equals ϵ.

When s is divided into xyz, either x or z may be ϵ, but condition 2 says that $y \neq \epsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p. It is an extract technical condition that we occasionally find useful when proving certain languagtes to be nonregular.

3 Context-Free Languages

In this chapter, we present ***context-free grammers***, a more powerful method of describing languages. Suych grammars can describe features that have a recursive structure, which makes them useful in a variety of applications.

The collection of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages of context-free grammars and study the properties of context-free lagnuages. We also introduce ***pushdown automata***, a class of machines recognizing the context-gree languages.

3.1 Context-Free Grammars

A grammar consists of a collection of ***substitution rules***, also called ***productions***. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analagous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the ***start variabele***. It usually occurs on the left-hand side of the topmost rule.

You use a grammar to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

Definition 3.1 (LL(1) Grammar) a language where you process the input left-to-right with the leftmost derivation first and lookahead 1 symbol at a time.

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a ***derivation***. A derivation of string 000#111 in grammar G_1 is

$$A \to 0A1 \to 00A11 \to 000A111 \to 000B111 \to 000\#111$$

You may also represent the same information pictorially with a ***parse tree***.

Remark To get the derived string when given a parse tree, take the pre-order traversal of leaves in the parse tree.

Definition 3.2 (Formal Definition of a Context-Free Grammar) A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***
2. Σ is a finite set, disjoint from V, called the ***terminals***

3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals

4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \to uwv$. Say that u ***derives*** v, written $u \to v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \to u_1 \to u_2 \to \cdots \to u_k \to v$$

The ***language of the grammar*** is $\{w \in \Sigma^* \mid S \to w\}$.

3.1.1 Designing Context-Free Grammars

First, many CFLs are the union of simple CFLs. If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct indivdual grammars for each piece. These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars.

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalence CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \to \epsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same langauge that the DFA recognizes.

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n 1^n \mid n \geq 0\}$ because a machine would need to rememeber the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \to uRv$, which generates strings wherein the portion containing the u's corresponds to the poertion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. Any time the symbol a appears, an entire parenthesized expressin might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

3.1.2 Ambiguity

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings. This result may be undesirable for certain applications, such as programming languages, where a program should have a unique interpretation. If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously, we say that the grammar is *ambiguous*.

Definition 3.3 A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar we can find an unambigous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called ***inherently ambiguous***.

Example 3.1 $\{0^i 1^j 2^k : i = j \text{ or } j = k\}$ is inherently ambiguous

3.1.3 Chomsky Normal Form

When working with context-free grammars, it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the Chomsky normal form. Chomsky normal form is useful in giving algorithms for working with context-free grammars. This form of a grammar is preferred because the parse tree will be a **binary tree**.

Definition 3.4 (Chomsky Normal Form) A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$

$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \epsilon$, where S is the start variable.

Theorem 3.1 Any context-free language is generated by a context-free grammar in Chomsky normal form.

Definition 3.5 (Pumping Lemma for Context-Free Languages) .
$\forall$ context-free language A
$\exists p$ (the pumping length)
$\forall s \in A, |s| \geq p \to \exists u, v, x, y, z : s = uvxyz$ where

1. $\forall i \geq 0, uv^i xy^i z \in A$

2. $|vy| > 0$
3. $|vxy| \leq p$

Proof: Since A is a CFL, there is a grammar $G = (V, \Sigma, R, S)$ that generates A. Let $b \geq 2$ be the maximum length of the RHS of a rule in G. Then, each node in the parse tree has at most b children. Thus, the maximum length of any string generted by a parse tree of height h is b^h.

If we choose $h = |V| + 1$ one of the $|V|$ non-terminals must repeat. Select $p = b^h = b^{|V|+1}$ so that every string $s, |s| \geq p$, has a repeated non-terminal along a root to-leaf path. We've chosen $p = b^{|V|+1}$ so that condition 1 will be satisfied. Now we show that the string $s(|s| \geq p)$ can be divided into 5 pieces to satisfy 2 and 3.

Consider the smallest parse tree of s (fewest number of nodes) with the repeated non-terminal. To satisfy condition 3, choose R which occurs twice within the bottom $|V| + 1$ non-terminals along any path. Then $|vxy \leq b^{|v|+1} = p$.

3.2 Pushdown Automata

We will introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammar. This means that we can prove a language is context free by giving either a context-free grammar generating it or a pushdown automata recognizing it.

A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol *pushes down* all the other symnbols on the stack. At any time the symbol on the top of the stack can be read and removed. THe remaining symbols then move back up. Writing a symbol on the stack is referred to as ***pushing*** the symbol, and removing a symbol is referred to as ***popping*** it. Note that all access to the stack, for both reading and writing, may be done only at the top.

3.2.1 Formal Definition of a Pushdown Automaton

Definition 3.6 (Formal Definition of a Pushdown Automaton) A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where $Q, \Sigma, \Gamma, andF$ are all finite sets, and

1. Q is the set of states

2. Σ is the input alphabet

3. Γ is the stack alphabet

4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$ is the transition function

5. $q_0 \in Q$ is the start state

6. $F \subseteq Q$ is the set of accept states

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\epsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \epsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.

2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.

3. $r_m \in F$. This condition states that an accept state occurs at the input end.

Remark The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack.

Remark (At each step of reading a PDA) ...

1. Choose whether or not to read an input symbol.
    - Input head advances if input symbol is read

2. Choose whether or not to pop the stack

3. Choose whether or not to push symbol on stack

4. Choose next state, pushed symbol depending on
    - Current state
    - Input symbol (if one was read)
    - Stack symbol (if one was popped)

The PDA **accepts the input** when:

- One or more computation paths end in an accept state
- All the input is consumed
- (no requirement for the stack)

The PDA **rejects** when every computation path:

- Ends in a non-accepting state, or
- is incomplete (the transition function is undefined at some point during computation)

Remark (Language for transition function) Some PDA transition function $a, b \to c$ says that in the current state... if we read symbol a as input, and can pop symbol b from the top of the stack, then we take this transition and push symbol c to the top of the stack. If we do not read symbol a or symbol b is not at the top of the stack, then we cannot take this transition. We may also use ϵ in place of any symbol to describe the lack of a requirement to accept/do anything with that symbol.

Example 3.2 (Transition function) For example some transition function $\epsilon, \epsilon \to S$ allows us to start in the current state and push a S symbol to the stack without transitioning state or requiring any pops from the stack

Definition 3.7 (Deterministic Pushdown Automata) A deterministic-PDA has the same properties as a normal (non-deterministic) PDA except for the transition function. Each state must have exactly one transition defined for every possible input symbol. That is the transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \cup \{\epsilon\}$ for every $q \in Q, a \in \Sigma, x \in \Gamma$ exactly one of $\delta(q, a, x), \delta(q, \epsilon, x), \delta(q, a, \epsilon), \delta(q, \epsilon, \epsilon)$

3.2.2 Equivalence with Context-Free Grammars

We will show that contexdt-free grammars and pushdown automata are equivalent in power. Both are capable of describing the class of context-free languages.

Theorem 3.2 A language is context free if and only if some pushdown automaton recognizes it.

Lemma 6 If a language is context free, then some pushdown automaton recognizes it.

Lemma 7 If a pushdown automaton recognizes some language, then it is context free.

3.3 Non-Context-Free Languages

Here we present a similar pumping lemma for context-free languages. It states that every context-free language has a special value called the ***pumping length*** such that all longer strings in the language can be *pumped*. *Pumped* means that the string can be divided into five parts so that the second and the fourth parts may be repreated together any number of times and the resulting string still remains in the language.

Theorem 3.3 (Pumping lemma for context-free languages) If A is a context-free language, then there is 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 conditions

1. for each $i \geq 0, uv^ixy^iz \in A$,
2. $|vy| > 0$,
3. $|vxy| \leq p$

When s is being divided into $uvxyz$, condition 2 says that either v or y is not the empty string. Otherwise the theorem would be trivially true. Condition 3 states that the pieces v,x, and y together have length at most p. This technical condition sometimes is useful in proving that certain languages are not context free.

4 The Church-Turing Thesis

4.1 Turing Machines

Turing machines are fimilar to finite automaton but with an unlimited and unrestricted memory; they are much more accurate models of a general purpose computer. The ***turing machine*** uses an infinite tapes as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape. Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the macnine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs *accept* and *reject* are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

The following list summarizes the differences between finite automata and turing machines.

1. A turing machine can both write on the tape and read from it
2. The read-write head can move both to the left and to the right
3. The tape is infinite
4. The special states for rejecting and accepting take effect immediately

The heart of the definition of a Turing machine is the transition function δ because it tells us how the machine gets from one step to the next. For a Turing machine, δ takes the form $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. That is, when the machine is in a certain state q and the head is over a tape square containing a symbol a, and if $\delta(q,a) = (r,b,L)$, the machine writes the symbol b replacing the a, and goes to state r. The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case, the L indicates a move to the left.

Definition 4.1 (Formal Definition of a Turing Machine) A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
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. $q_{\text{accept}} \in Q$ is the accept state
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ computes as follows. Initially, M receives its input $w = w_1 w_2 \dots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank. The head starts on the leftmost square of the tape. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it enters either the accept or reject states, at which points it halts. If neither occurs, M goes on forever.

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a ***configuration*** of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet Γ, we write uqv for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v.

Say that configuration C_1 ***yields*** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. We define this notion formally as follows.
Suppose that we have a, b, and c in Γ, as well as u and v in Γ^* and states q_i and q_j. In that case, uaq_ibv and uq_jacv are two configurations. Say that

$$uaq_ibv \text{ yields } uq_jacv$$

if in the transition function $\delta(q_i, b) = (q_j, c, L)$. That handles the case where teh Turing machine moves leftward. For a rigthward move, say that

$$uaq_ibv \text{ yields } uacq_jv$$

if $\delta(q_i, b) = (q_j, c, R)$.

Special cases occur when the head is at one of the ends of the configuratino. For the left-hand end, the configuration q_ibv yeilds q_jcv if the transition is left-moving (because we prevent the machine from going off the left-hand end of the tape), and it yields cq_jv for the right-moving transition. FOr the right-hand end, the configuration uaq_i is equivalent to $uaq_i\textvisiblespace$ because we assume that blanks follow the part of the tape represented in the configuration. Thus we can handle this case as before, with the head no longer at the right-hand end.

The ***start configuration*** of M on input w is the configuration q_0w, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape. In an ***accepting configuration***, the state of the configuration is q_{accept}. In a ***rejecting configuration***, the state of the configuration is q_{reject}. Accepting and rejecting configurations are ***halting conditions*** and do not yield further configurations. Because the machine is defined to halt when in the states q_{accept} and q_{reject}, we equivalently could have defined the transition function to have the more complicated form $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R\}$, where Q' is Q without q_{accept} and q_{reject}. A Turing machine M ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where

1. C_1 is the start configuration of M on input w
2. each C_i yields C_{i+1}
3. C_k is an accepting configuration

The collection of strings that M accepts is ***the language of M***, or ***the language recognized by M***, denoted $L(M)$.

Definition 4.2 We call a language ***Turing-recognizable*** if some Turing machine recognizes it.

When we start a Turing machine on an input, three outcomes are possible. The machine may *accept*, *reject*, or *loop*. By ***loop*** we mean that the machine simply does not halt. Looping may entail any simply or complex behavior that never leads to a halting state.

A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called ***deciders*** because they always make a decision to accept or reject. A decider that recognizes some language also is said to ***decide*** that language.

Definition 4.3 Call a language ***Turing-decidable*** or simply ***decidable*** if some machine decides it.

5 Decidability

5.1 Decidable Languages

5.1.1 Decidable Problems Concerning Regular Languages

This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

$$A_{\text{DFA}} = \{\langle B, w\rangle | B \text{ is a DFA that accepts input string } w\}$$

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}. Showing that the language is decidable is the same as showing that the computational problem is decidable.

Theorem 5.1 A_{DFA} is a decidable language.

We simply need to present a TM M that decides A_{DFA}.
$M =$ On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept.* If it ends in a nonaccepting state, *reject.*

Theorem 5.2 A_{NFA} is a decidable language.

We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: Have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before basing it to M.
$N =$ On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the standard procedure.
2. Run TM M from the previous theorem on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject.*

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure.

Theorem 5.3 A_{REX} is a decidable language.

The following TM P decides A_{REX}.
$P =$ On input $\langle R, w\rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion.
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*.

Theorem 5.4 E_{DFA} is a decidable language.

A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible. To test this condition, we can design a TM T that uses a marking algorithm.
$T =$ On input $\langle A\rangle$, where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked
4. If no accept state is marked, *accept*; otherwise, *reject*.

Theorem 5.5 EQ_{DFA} is a decidable language.

5.1.2 Decidable Problems Concerning Context-Free Languages

Here, we begin to describe algorithms to determine wheter a CFG generates a particular string and to determine wheter the language of a CFG is empty. Let

$$A_{\text{CFG}} = \{\langle G, w\rangle | G \text{ is a CFG that generates } w\}$$

.

Theorem 5.6 A_{CFG} is a decidable language.

The TM S for a A_{CFG} follows:
$S =$ On input $\langle G, w\rangle$, where G is a CFG and w is a string.

1. Convert G to an equivalent grammar in Chomsky Normal Form.
2. List all derivations with $2n - 1$ steps, where n is the lenght of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*.

Theorem 5.7 E_{CFG} is a decidable langauge.

Theorem 5.8 Every context-free language is decidable.

Definition 5.1 (Language Enumerators) An enumerator ϵ for a language A is TM that, starting witha blank input tape, prints all and only strings in A. ϵ never halts when A is infinite. ϵ may print a string multiple times but must print every string in A eventually (in finite time). So, if A is finite, all its strings will be printed within a finite amount of time—just wait long enough!

Theorem 5.9 A language A is TM-recognizable if and only if there is an enumerator ϵ for it.

$$\text{L is enumerable} \Leftrightarrow \text{L is TM-recognizable}$$

5.2 Undecidability

One theorem that establishes the undecidability of a specific language: the problem of determining whether a Turing machine accets a given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG}. But, whereas A_{DFA} and A_{CFG} were decidable, A_{TM} is not. Let

$$A_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$$

.

Theorem 5.10 A_{TM} is undecidable. We can show, however, that A_{TM} is Turing-recognizable. The following Turing machine U recognizes A_{TM}.
U = On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*.

Note that this machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was no halting on w, it could *reject* in this case. However, an algorithm has no way to make this determination.

5.2.1 The Diagonalization Method

If we have two infinite sets, how can we tell whether one is larger than the other or whether they are the same size? Cantor observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set. This method compares the sizes without resorting to counting.

Definition 5.2 (Diagonalization) Assume that we have sets A and B and a function f from A to B. Say that f is ***one-to-one*** if it never maps two different elementts to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is ***onto*** if it hits every element of B—that is, for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the ***same size*** if there is a one-to-one; into function $f : A \to B$. A function that is both one-to-one and onto is called a ***correspondence***. In a correspondence, every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.

Alternative common terminology for these types of functions is ***injective*** for one-to-one, ***surjective*** for onto, and ***bijective*** for one-to-one and onto.

Definition 5.3 (Countability) A set A is ***countable*** if either it is finite or it has the same size as N

Theorem 5.11 The set of real numbers, R, is uncountable.

Theorem 5.12 Some languages are not Turing-recognizable

5.2.2 A Turing-Unrecognizable Language

If both a language and its complement are Turing-recognizable, the language is decidable. Hence for any undecidable language, either it or its complement is not Turing-recognizable. Recall that the complement of a language is the language consisting of all strings that are not in the language. We say that a language is ***co-Turing-recognizable*** if it is the complement of a Turing-recognizable language.

Theorem 5.13 A language is decidable if and only if it is Turing-recognizable and co-Turing-recognizable. In other words, a language is decidable exactly when both it and its complement are Turing-recognizable.

6 Reducibility

A ***reduction*** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem. Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A.

6.1 Undecidable Problems from Language Theory

Let's consider $HALT_{TM}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input. THis problem is widely known as the ***halting problem***. We use the undecidability of A_{TM} to prove the undecidability of the halting problem by reducing A_{TM} to $HALT_{TM}$. Let

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ ahlts on input } w$$

Theorem 6.1 $HALT_{TM}$ is undecidable.
Proof: Let's assume for the purpose of obtaining a contradiction that TM R decides $HALT_{TM}$. We construct TM S to decide A_{TM}, with S operating as follows.
S = On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*.

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{TM}$ also must be undecidable.

Theorem 6.2 E_{TM} is undecidable where

$$E_{TM} = \{\langle M\rangle | M \text{ is a TM and } L(M) = \emptyset$$

Proof: Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.
M_1 = On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does.

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same. Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.
S = On input $\langle M, w\rangle$, and encoding of a TM M and a string w:

1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1\rangle$.
3. If R accepts, *reject*; if R rejects, *accept*.

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it only needs to add extra states to M that perform the $x = w$ test. If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

Theorem 6.3 $REGULAR_{TM}$ is undecidable where

$$REGULAR_{TM} = \{\langle M\rangle | M \text{ is a TM and } L(M) \text{ is a regular language}$$

6.1.1 Reductions via Computation Histories

The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input. It is a complete record of the machine.

Definition 6.1 Let M be a Turing machine and w an input string. An ***accepting computation history*** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} acording to the rules of M. A ***rejecting computation history*** for M on w is defined similarly, except that C_l is a rejecting configuration.

Computation histories are finite sequences. If M doesn't halt on w, no accepting or rejecting computation history exists for M on w. Deterministic machines have at most one computation history on any given input. Nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches.

Definition 6.2 A ***linear bounded automaton*** is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to mofve its head off either end of the input, the head stays where it is—in the same way that the head will not move off the left-hand end of an ordinary Turing machine's type.

Lemma 8 Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape length of n.

Theorem 6.4 A_{LBA} is decidable.
Proof: The algorithm that decides A_{LBA} is as follows.
$L =$ On input $\langle M, w\rangle$, where M is an LBA and w is a string.

1. Simulate M on w for qng^n steps or until it halts.
2. If M has halted, *accept* if it has accepted and *reject* if it has rejected. If it has not halted, *reject*.

If M on w has not halted within qng^n steps it must be repeating a configuration and therefore looping. That is why our algorithm rejects in this instance.

6.2 Mapping Reducibility

6.2.1 Computable Functions

A Turing machine computes a fucntion by starting with the input to the function on the tape and halting with the output of the function on the tape.

Definition 6.3 A function $f : \Sigma^* \to \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

6.2.2 Formal Definition of Mapping Reducibility

Definition 6.4 Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is s a computable function $f : \Sigma^* \to \Sigma^*$, where for every w,

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called the ***reduction*** from A to B.

A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem.

Theorem 6.5 If $A \leq_m B$ and B is decidable, then A is decidable.
Proof: We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows:
$N =$ On input w:

1. Compute $f(w)$.
2. Run M on input $f(2)$ and output whatever M outputs.

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Therefore, N works as desired.

Theorem 6.6 If $A \leq_m B$ and A is undecidable, then B is undecidable.

Theorem 6.7 If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

Theorem 6.8 If $A \leq_m B$ and B is not Turing-recognizable, then A is not Turing-recognizable.

Theorem 6.9 EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

7 Advanced Topics in Computability Theory

7.1 The Recursion Theorem

The recursion theorem is a mathematical result that plays an important role in advanced work in the theory of computability. It has connnections to mathetmatical logic, the theory of self-reproducing systems, and even computer viruses.

Lemma 9 There is a computable function $q : \Sigma^* \to \Sigma^*$ where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and then halts.

Theorem 7.1 (Recursion Theorem) Let T be a Turing machine that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a Turing machine R that computes a function $r : \Sigma^* \to \Sigma^*$, where for every w,

$$r(w) = t(\langle R \rangle, w)$$

The statement of this theorem seems a bit technical, but it actually represents something quite simple. To make a Turing machine that can obtain its own description and then compute with it, we need only make a machine, called T in the statement, that receives the description of the machine as an extra input. Then the recursion theorem produces a new machine R, which operates exactly as T does but with R's description filled in automatically.

Theorem 7.2 A_{TM} is undecidable.

Definition 7.1 If M is a Turing machine, then we say that the length of the description $\langle M \rangle$ of M is the number of symbols in the string describing M. Say that M is ***minimal*** if there is no Turing machine equivalent to M that has a shorter description. Let

$$MIN_{TM} = \{\langle M \rangle \mid M \text{ is a minimal TM}\}$$

Theorem 7.3 MIN_{TM} is not Turing-recognizable.

Theorem 7.4 Let $t : \Sigma^* \to \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.
In this theorem, t plays the role of the transformation, and F is the fixed point.

Proof: Let F be the following Turing machine.
F = On input w:

1. Obtain, via the recursion theorem, own description $\langle F \rangle$.
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
3. Simulate G on w

Clearly $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G.

8 Time Complexity

8.1 The Class NP

One remarkable discovery shows that the complexities of many problems are linked. A polynomial time algorithm for one such problem can be used to solve an entire class of problems.

The $HAMPATH$ problem has a feauter called ***polynomial verifiability*** that is important for understanding its complexity. Even though we don't know of a fast (i.e. polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow, we could easily convince someone else of its existence simply by presenting it. In other words, *verifying* the existence of a Hamiltonian path may be much easier than *determining* its existence.

Definition 8.1 (Verifier) A ***verifier*** for a language A is an algorithm V, where

$$A = \{w|V \text{ accept } \langle w, c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

A verifier uses additional information, represented by the symbol c to verify that a string w is a member of A. This information is called a ***certificate*** or ***proof*** of membership in A. Observe that for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound.

Definition 8.2 ***NP*** is the class of languages that have polynomial time verifiers.

The term NP comes from ***nondeterministic polynomial time*** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines.

Theorem 8.1 A language is in NP if and only if it is decided by some nondeterministic polynomial time Turing machine.

We define the nondeterministic time complexity class $NTIM(t(n))$ as analagous to the deterministic time complexity class $TIME(t(n))$.

Definition 8.3 $NTIME(t(n)) = \{L|L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine.

The class NP is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomially equivalent. When describing and analyzing nondeterministic polynomial time algorithms. Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model. We analyze the algorithm to show that every branch uses at most polynomially many stages.

8.1.1 The P versus NP Question

NP is the class of languages that are solvable in polynomial time on a nondeterministic Turing machine; or, equivalently, it is the class of languages whereby membership in the language can be verified in polynomial time. P is the class of languages where membership can be tested in polynomial time.

P = the class of languages for which membership can be *decided* quickly
NP = the class of languages for which membership can be *verified* quickly

There are examples of languages that are members of NP but that are not known to be in P. The power of polynomial verifiability seems to be much greater than that of polynomial decidability. But, hard as it may be to image, P and NP could be equal. We are unable to *prove* the existence of a single language in NP that is not in P.