CS 360: Introduction to the Theory of Computing

Professor Shai Ben-David

Fall 2013, University of Waterloo

Notes taken by Chris Thomson

I'd love to hear your feedback. Feel free to email me at chris@cthomson.ca.
See cthomson.ca/notes for updates. Last modified: November 17, 2014 at 7:00 PM (12e034e).

Contents

1 Introduction & Course Structure

← September 10, 2013

1.1 Course Structure

The grading scheme is 30% assignments, 30% midterm, and 40% final. If your final grade is better than your midterm grade, your final will be worth 70% and the midterm will be worth 0%.

In terms of textbooks, any textbook relating to "automata theory", "theory of computation", or "formal language theory" is acceptable.

The professor's office is in DC 1311, and his office hours are on Tuesdays from 2:00 to 3:00 pm.

This course is mathematical. Consequently, relevant background courses are mainly mathematical (mostly logic) and (almost) no programming.

1.2 Introduction

The goal of the course is to develop the theory of computation. We want to understand the mathematics that govern computers. We will explore questions like:

- What are computers?
- Which tasks can computers carry out?
- Which tasks are easy or hard for computers?

We make abstractions and use them as common properties that are relevant to many phenomenon. We aren't going to be focusing on any particular programming language or machine architecture, but instead, we'll focus on issues that underlie any automated computation.

These abstractions allow us to form conclusions that are relevant to computing in general, and not just for today's hardware and software.

This course has two key goals:

1. To familiarize you with the fundamentals of computer science theory.
2. To develop the ability to reason formally and abstractly about computing.

Throughout the course, we will follow two main threads concurrently: abstract models of computers, and computing tasks. We'll progress from simple models and simple tasks to complex models and complex tasks, and along the way connections will be made between the two.

1.3 Computing Tasks

We're going to focus on **decision problems**. Decisions problems are problems which have an input and produce a binary (yes/no) output. Some examples of decision problems are:

- Input: a number. Decision: is it a prime number?
- Input: email message. Decision: is it spam?
- Input: a graph. Decision: is it connected?

We're interested in seeing which of these questions a computer can carry out and how difficult are such tasks.

1.3.1 Modeling Decision Problems

We model decision problems using **formal languages**. A decision problem has two components:

- A domain set X.
- A language $L \subseteq X$.

Our decision problem is then "for some $x \in X$, is $x \in L$?" Re-examining our examples from earlier, we would define:

- $X = \mathbb{Z}$ (the set of all integers), and $L =$ the set of all prime numbers.
- $X =$ the set of all email messages, and $L =$ the set of all spam email messages.
- $X =$ the set of all graphs, and $L =$ the set of all connected graphs.

For further concreteness, we will fix some finite set Σ, which we will call the **alphabet**. Some examples of alphabets are $\{a, b\}$, $\{0, 1\}$, and $\{0, 1, 2, 3\}$.

$\Sigma^\star$ represents the set of all finite strings over the alphabet Σ. In many cases, we define $X = \Sigma^\star$. For example, if $\Sigma = \{0, 1\}$ then $\Sigma^\star = \{\epsilon, 0, 1, 00, 11, 01, 10, \ldots\}$. Note that $\Sigma^\star$ is an infinite set, but every member of $\Sigma^\star$ is finite.

For the time being, we will define just one operation for strings: **concatenation**. Given two strings, σ and η, we get $\sigma\eta$. That is, $\frac{\sigma \quad \eta}{\sigma\eta}$. For example, if $\sigma = 01$ and $\eta = 111$, then $\sigma\eta$ is 01111.

Languages are subsets of $\Sigma^\star$ (that is, a language is a collection of strings). Some examples of languages include:

- $L = \emptyset$. This is the **empty language**, which no strings belong to. Note that $|L| = 0$. This is analogous to an empty fridge.
- $L = \{\epsilon\}$. This is a non-empty language that contains one element ($|L| = 1$), which just happens to be ϵ. This is analogous to having a fridge that contains only an empty pop can.
- $L = \{\epsilon, 0, 01, 11111\}$.
- $L = \Sigma^\star$.

A compiler's task of checking their input (the code of a program) to see if the program is valid (if the code is in the language), can be readily seen as checking membership in such a language (namely, the language of all binary sequences that represent valid code in the given programming language).

2 Inductive Definition of Sets and Structural Induction

We'll start off by reviewing some basic notions that were covered on CS 245, to ensure that everyone's on the same page.

2.1 Formal Tools for Defining Sets

We want languages to model every computational task, some of which are complex. We need several different methods for formally defining sets, since all languages are sets.

1. **List all members of the set**. This method is precise and concrete, but it makes look-ups difficult ($O(n)$) and it fails if the set is infinitely large.
2. **Common property that characterizes the members of the set**. For example, consider the set of all even numbers. This set is not definable by a list because it's an infinite set. However, there is a common property that's shared between all members of the set: if you divide by two and examine the remainder, every member in the set will have a remainder of zero and only members of the set have this property.
3. **Inductive definitions**. Methods (1) and (2) don't offer us a way to define a set of all your blood relatives, since there's no common property that characterizes that set. Instead, we could use *inductive definitions*.

   Inductive definitions of sets requires three components:

   - A domain set X (sometimes omitted if it is obvious).
   - A core set A (as in "atoms"), such as {me}.
   - A set of operations P, such as {"son of", "father of", "mother of"}. This set is a set of functions from $X \to X$.

   Given these components, we define the **inductive set** $I(A, P)$ (say, the set of blood relatives), as all domain elements that can be reached from the core set by applying a finite sequence of operations from P.

   Definition. $I(A, P)$ – the **inductive set** defined by core A and operations P – is the smallest set satisfying:

   - Contains all members of A, and
   - Closed under the operations in P.

   Examples:

   - Let the domain be the set of all real numbers ($\mathbb{R}$), the core set be {2}, and the set of operations be {"add 2", "subtract 2"}. The defined set is the set of all even numbers, both positive and negative.
   - To define the set of all algebraic expressions, our core set would be $\{x, y, z, 1, 2, 3, \ldots, a, b, c, \ldots\}$, and our operations would be: $\dfrac{\sigma \qquad \eta}{(\sigma + \eta)}$, $\dfrac{\sigma \qquad \eta}{(\sigma \cdot \eta)}$.

   Given a set (language) defined inductively, how can we tell if some $x \in X$ is in the language or not?

Example 2.1. Let $X = \{a, b\}^\star$, $A = \{a\}$, and $P = \left\{ \dfrac{\mathrm{X}}{\mathrm{aX}}\ ,\ \dfrac{\mathrm{X}}{\mathrm{Xa}}\ ,\ \dfrac{\mathrm{X}\quad\mathrm{Y}}{\mathrm{bXY}}\ ,\ \dfrac{\mathrm{X}\quad\mathrm{Y}}{\mathrm{XbY}}\ ,\ \dfrac{\mathrm{X}\quad\mathrm{Y}}{\mathrm{XYb}} \right\}$.

A typical question would be: is $abbaa \stackrel{?}{\in} I(A, P)$? The answer: yes! However, we must show how to generate $abbaa$ from a, using the operations in P, in a finite number of steps.

1. a, belongs to the core A.
2. baa, after applying $P_3(a, a)$.
3. $abbaa$, after applying $P_4(a, baa)$.

Note that the first step must always be an element in the core set.

Is $\epsilon \in I(A, P)$? No. We could use simple reasoning to show that $\epsilon \notin I(A, P)$ (each function in P increases the length). However, in general, it is more difficult to prove non-membership of an element than it is to prove membership of another element.

Definition. A **certificate** (or **generating sequence**) for x is a sequence of elements of $X, \theta_1, \theta_2, \theta_3, \ldots, \theta_n$ such that:

- $\theta_n = x$, and
- Every θ_i in the sequence is either an element of the core set A or is the result of applying an operation from P to a θ_j, θ_l that appeared earlier in the sequence ($j < i$ and $l < i$).

← September 12, 2013

Our definition of $I(A, P)$ is still somewhat informal, however. We haven't formally defined what it means for a subset to be the *smallest* subset, and we also haven't formally defined what it means for a set to be "closed" under a set of operations.

2.2 More on Inductive Sets and Structural Induction

We say that a set B is **closed under the operations of P** if for every $f \in P$ and every $x, y \in B$, $f(x, y) \in B$ (for concreteness, we considered here an f that accepts two inputs, but it may also be a unary function, $f(x)$ or ternary $f(x, y, z)$, etc.). For example, the set of even numbers is closed under the operation $+$.

Now, let's redefine $I(A, P)$ in a more formal way:

$$I(A, P) = \bigcap \{B : A \subseteq B \ \text{ and } \ B \ \text{ is closed under } \ P\}$$

Note that $I(A, P)$ is the intersection over the *entire* collection of sets (and is therefore uniquely defined).

Example 2.2. Let $X = \mathbb{N}, A = \{10\}, P = \{+\}$.

$I(A, P)$ is defined as the intersection of all sets B such that $A \subseteq B$ and B is closed under P (that is, B is closed under $+$). So, we have:

- $B_1 = \{\text{all even numbers}\}$
- $B_2 = \{\text{all numbers divisible by 5}\}$
- $B_3 = \{\text{all numbers divisible by 10}\}$

In this case, $I(A, P) = \{10, 20, 30, \ldots\}$, since $I(A, P)$ must be the minimal set among all possible sets B (or, in other words, the multiples of 10 are exactly the numbers that belong to each and every of these sets B).

Aside.

$$\bigcap \{[0, r] : r > 0\} = \{0\}$$
$$\bigcap \{(0, r) : r > 0\} = \emptyset$$
$$\bigcup \{\{1, \ldots, n\} : n \in \mathbb{N}\} = \mathbb{N}$$
$$\bigcap \{\{1, \ldots, n\} : n \in \mathbb{N}\} = \{1\}$$

The equivalence of the two definitions of $I(A, P)$ is not immediately clear. We claim that $I(A, P)$ is closed under the operations of P, and $A \subseteq I(A, P)$.

Proof. Pick any $x, y \in I(A, P)$ and operation $f \in P$. For every B which contains A and is closed under P, $x, y \in B$ (since $I(A, P)$ is the intersection of all such B's).

Since every such B is closed under P, $f(x, y) \in B$ for every such B. Therefore, $f(x, y) \in \cap\{B : A \subseteq B \text{ and } B \text{ is closed under } P\}$, so $f(x, y) \in I(A, P)$.

Exercise: prove that $A \subseteq I(A, P)$. □

Corollary (Proof by Structural Induction Theory)**.** Any $B \subseteq X$ which contains A and is closed under P is a superset of $I(A, P)$. Namely, $I(A, P) \subseteq B$.

We could rephrase this corollary as follows: given any set B, if you want to prove that $I(A, P) \subseteq B$, it suffices to show:

- $A \subseteq B$, and
- B is closed under P.

2.2.1 Proof by Structural Induction

Let's say we want to show that for every n, $n^2 + n$ is even. A proof by usual (mathematical) induction requires us to show two things:

- Prove the claim holds for $n = 1$. This is equivalent to showing that $A \subseteq B$.
- Prove that if the claim holds for n, it also holds for $n+1$. This is equivalent to showing that B is closed under P.

How is this the same as structural induction? Usual induction shows that $\mathbb{N} = I(A, P)$, where $A = \{1\}$ and $P = \{\text{“+1”}\}$. That is, mathematical induction is just one specific case of structural induction.

2.3 Examples of Proofs by Structural Induction

Example 2.3. Imagine we have three paper cups placed on a desk, with two of them facing upwards and one upside down. You must flip exactly two cups at a time. We want to get all of the cups facing upwards. Can we prove that this goal is not achievable?

Let X be the set of all possible configurations of the cups. Let our core set A contain only the state "[up] [down] [up]". Let our set of operations P contain:

- P_1: flip the two rightmost cups.
- P_2: flip the two leftmost cups.
- P_3: flip the rightmost and leftmost cups.

We claim that the state "[up] [up] [up]" $\notin I(A, P)$.

Proof. Let B be the set of all configurations with an even number of "up" cups.

We want to show that $I(A, P) \subseteq B$. This would show that "[up] [up] [up]" is not attainable because it is not a member of B. We will use structural induction to show this.

Induction base: $A \subseteq B$.

Induction step: B is closed under P. Namely, if $c \in B$, then $P_1(c), P_2(c)$, and $P_3(c) \in B$. There are three cases:

- We flip one "up" cup and one "down" cup. This is a difference of +0 "up" cups.
- We flip two "up" cups. This causes us to remove two (which is even) from an already even number, which gives us another even number of "up" cups.
- We flip two "down" cups. This causes us to add two (which is even) to an already even number, which gives us another even number of "up" cups.

Note that these three cases show us that no matter which operation we use, it is not possible to obtain all three "up" cups, since the number of "up" cups will always be even. □

Example 2.4. Earlier, we defined $X = \{a, b\}^\star$, $A = \{a\}$, and $P = \left\{ \frac{\text{X}}{\text{aX}}\ ,\ \frac{\text{X}}{\text{Xa}}\ ,\ \frac{\text{X}\quad\text{Y}}{\text{bXY}}\ ,\ \frac{\text{X}\quad\text{Y}}{\text{XbY}}\ ,\ \frac{\text{X}\quad\text{Y}}{\text{XYb}} \right\}$.

For this example, let $\#_a(x)$ denote the number of "a"s in the string x and similarly, let $\#_b(x)$ denote the number of "b"s in the string x.

Claim: every member of $I(A, P)$ has more "a"s than "b"s. To put this more simply, let $B = \{x : \#_a(x) > \#_b(x)\}$. We wish to show that $I(A, P) \subseteq B$.

Proof. We will prove this claim by structural induction.

Induction base: $A \subseteq B$. It is trivial to see that the string "a" has more "a"s than "b"s (since it has no "b"s).

Induction step: if $x, y \in B$, then we must show that $P_1(x) \in B, P_2(x) \in B, P_3(x, y) \in B, P_4(x, y) \in B$, and $P_5(x, y) \in B$.

It is clear that for P_1 and P_2, one "a" is being added to a string that must already include more "a"s than "b"s, so this trivially holds true.

The case of P_3 isn't as immediately obvious. Assume $\#_a(x) > \#_b(x)$, and $\#_a(y) > \#_b(y)$. So, $\#_a(x) \geq \#_b(x) + 1$ and $\#_a(y) \geq \#_b(y) + 1$. This gives us $\#_a(xy) \geq \#_b(xy) + 2$.

Now, note that $\#_a(bxy) = \#_a(xy)$, and $\#_b(bxy) = \#_b(xy) + 1$. We can conclude that $\#_a(bxy) \geq \#_b(bxy) + 1 > \#_b(bxy)$.

A similar argument applies for P_4 and P_5. □

2.4 Regular Languages

For us, tasks will always be decision problems. That is, given $L \subseteq \{a, b\}^\star$ and input $x \in \{a, b\}^\star$, decide if $x \in L$.

The set of all possible tasks is $\{L : L \subseteq \{a, b\}^\star\}$. This set is an infinite set of finite strings.

Recall that not all infinite sets have the same size. The set of all finite strings $\{a, b\}^\star$ is a "small" infinite set, meaning it is **countable**. However, the set of all subsets of $\{a, b\}^\star$ is a "large" infinite set, which is **uncountable**.

The set of all possible computer programs is a subset of $\{0, 1\}^\star$. Note that since every program is a *finite* string, the set of all possible programs is countable.

In summary, we have an uncountable number of tasks, but a countable number of programs (and each program is suitable for carrying out at most a single task). This implies that there are tasks (languages) for which no program exists (namely, for such languages, L, there exists no program that will figure out correctly for every string x whether it is a member of L or not).

2.5 Defining Interesting Families of Tasks

← September 17, 2013

Definition. The **empty string** can be denoted as either ϵ or λ.

2.5.1 Operations on Languages

There are several operations that we can perform on languages.

We can use **standard set operations**, including $\bigcup, \bigcap$, complement ($\overline{A}$), and set difference ($A \backslash B$). You should be familiar with how these standard operations are defined.

Since we focus on subsets of the set of finite strings, $\Sigma^\star$, we can also use **operations**

on sets of such strings. Let's go into more detail of how these string set operations are defined.

1. **The concatenation operation**. Given $L_1, L_2 \subseteq \Sigma^\star$, we define $L_1L_2 = \{w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$.

   Example 2.5. Let $L_1 = \{0, 011\}$ and $L_2 = \{\epsilon, 110\}$. Then $L_1L_2 = \{0, 0110, 011, 011110\}$.

   Concatenation is not always commutative. Consider $L_1 = \{0\}$ and $L_2 = \{1\}$. We have $L_1L_2 = \{01\}$ but $L_2L_1 = \{10\}$, so $L_1L_2 \neq L_2L_1$.

   Can $L_1 \subseteq L_1L_2$? Yes, but only if $\epsilon \in L_2$. Otherwise, $L_1 \not\subseteq L_1L_2$.

2. **The power operation**. Given any language L, define:

   - $L^0 = \{\epsilon\}$
   - $L^{n+1} = L^nL$ for every $n > 0$

   In particular, note that $L^1 = \{\epsilon\}L = L$.

   Example 2.6. Let $L = \{0, 1\}$. Then L^n is the set of all ($\{0, 1\}$) strings.

3. **The star (Kleene) operation**. For any language L, $L^\star = \bigcup_n L^n$. That is, $L^\star$ is the set of all finite concatenations of strings from L. Note that $L^\star$ includes L^0, which means $\epsilon \in L^\star$.

Examples:

- $\{1\}^\star$ is the set of all finite strings of 1s.
- $\{0, 1\}^\star$ is the set of all finite $\{0, 1\}$ strings.
- Can $L^\star = L$? Yes. Consider $L = \{\epsilon\}$, or $(L^\star)^\star = L^\star$ for any L.
- Can $L^\star$ be finite? Yes. Consider $L = \{\epsilon\}$ or $L = \emptyset$.

2.5.2 The Set of Regular Languages

Given some finite alphabet set Σ, let $X = \Sigma^\star$. We will also define our core set to be:

$$A = \{\emptyset, \{\epsilon\}\} \cup \{\{a\} : a \in \Sigma\}$$

For instance, suppose $\Sigma = \{0, 1\}$. Then $A = \{\emptyset, \{\epsilon\}, \{0\}, \{1\}\}$. Next, we'll define our set of operations P:

$$P = \left\{ \frac{L_1 \quad L_2}{L_1 \cup L_2}, \frac{L_1 \quad L_2}{L_1 \cdot L_2}, \frac{L}{L^\star} \right\}$$

A language is said to be **regular** if it belongs to $I(A, P)$, for this A and P.

To prove a language is regular, you must be able to show how to generate it.

Example 2.7. $\{0, 1, 01\}$ is regular.

Proof.

$$\{0\} \text{ core} \tag{1}$$
$$\{1\} \text{ core} \tag{2}$$
$$\{01\} \text{ product} \tag{3}$$
$$\{0,1\} \text{ union of (1) and (2)} \tag{4}$$
$$\{0,1,01\} \text{ union of (3) and (4)} \tag{5}$$

□

Claim 1: for every (finite) string σ, $\{\sigma\}$ is a regular language. It's easy to see this is true, because you can generate the language by concatenating the languages that represent individual letters.

Claim 2: every finite language is regular. Since we know every $\{\sigma\}$ is regular, so is any finite unions of such languages.

Example 2.8. Let L_{even} be the language of all words of even length. L_{even} is regular. We can form every word in L_{even} using the expression $(\{0,1\}\{0,1\})^\star$.

Example 2.9. Let $L_{\text{equal}} = \{w : \#_0(w) = \#_1(w)\}$. L_{equal} is not regular. We can't prove this yet though.

2.6 Explicit Descriptions of Regular Languages: Regular Expressions

A **regular expression** is just a word in a language that is designed to describe languages.

2.6.1 Defining The Set of Regular Expressions

Given some alphabet Σ, let X be defined as

$$X = (\Sigma \cup \{+, \star, (,), \emptyset, \epsilon\})^\star$$

We will define our core set A as

$$A = \{\emptyset, \epsilon\} \cup \{a : a \in \Sigma\}$$

Finally, we will define our set of operations P as

$$P = \left\{ \frac{r_1 \quad r_2}{r_1 + r_2}\ ,\ \frac{r_1 \quad r_2}{r_1 r_2}\ ,\ \frac{r}{r^\star}\ ,\ \frac{r}{(r)} \right\}$$

A regular expression is any member of this $I(A, P)$.

Suppose $\Sigma = \{a, b\}$. Some regular expressions over Σ are a, ab, $a + b$, $a(a + \epsilon)^\star$, and $(a + b)(a + b)$.

Claim: in any regular expression, the number of + symbols is at least one less than the number of symbols from the set $\Sigma \cup \{\epsilon, \emptyset\}$. The proof of this (by structural induction) is left as an exercise.

2.6.2 Mapping Regular Expressions to Languages

We will define a mapping L such that L : Regular Expression $\to$ Language. We define such a mapping L recursively over the definition of regular expressions.

- $L(\epsilon) = \{\epsilon\}$
- $L(a) = \{a\}$ for every $a \in \Sigma$

Assume $L(r_1)$ and $L(r_2)$ are already defined. We then define the mappings for each operation:

- $L(r_1 r_2) = L(r_1)L(r_2)$
- $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
- $L(r_1^\star) = (L(r_1))^\star$
- $L((r_1)) = L(r_1)$

Examples:

- Let $r = (0 + 1)$. Then $L(r) = \{0, 1\}$.
- Let $r = 01$. Then $L(r) = \{01\}$.
- Let $r = (01)^\star$. Then $L(r) = \{\epsilon, 01, 010101, 01 \ldots 0101 \ldots 01, \ldots\}$.
- Let $r = (01)^\star + (10)^\star$. Then $L(r)$ is the set of all alternating strings that either start with 0 and end with 1, or start with 1 and end with 0.

Find an expression r such that $L(r)$ is the set of *all* alternating strings. One such expression is $r = (0 + \epsilon)(10)^\star(1 + \epsilon)$. Another expression to represent the same language is $r' = (01)^\star + (10)^\star + 1(01)^\star + 0(10)^\star$.

Since $L(r) = L(r')$ represent the same language, this shows that there is no *unique* regular expression to describe a language. Multiple distinct regular expressions can represent the same language.

2.6.3 Uses of Regular Expressions

Regular expressions are often useful to search for patterns in code. For example, if you were looking for all strings that start with "a" and end with "b", you could use the regular expression $a(a + b)^\star b$.

Regular expressions are also used in compilers. Compilers need to find certain patterns in the code they're given to compile, in order to perform the parsing step of the compilation.

← September 19, 2013

The lecture from September 19, 2013 is omitted. If you have notes from this lecture, feel free to email me (chris@cthomson.ca) and I'll include them here.

3 Computing Machines

← September 24, 2013

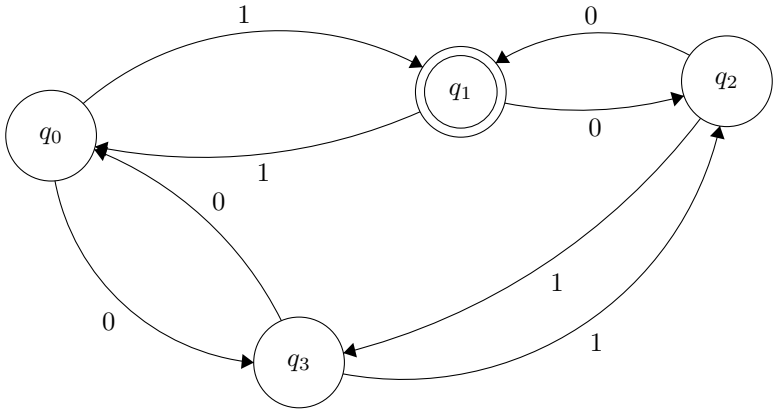
3.1 Deterministic Finite Automata

The first model of computing machines that we'll discuss are **deterministic finite automata** (also known as DFAs). A DFA is denoted by $A = (\Sigma, Q, q_0, \delta, F)$, where

- Σ is the alphabet (the set of letters) of the DFA.
- Q is the set of all states in the DFA.
- q_0 is the initial state.
- δ is the transition function.
- F is the set of accepting states. (Note that there can be more than one accepting state.)

Every DFA can be represented with a bubble diagram, to make it easier to visualize.

Example 3.1. A bubble diagram that represents the DFA that accepts all strings with an even number of 0s and an odd number of 1s is as follows:



To define $L(A)$, we extend $\delta : Q \times \Sigma \to Q$, to: $\hat{\delta} : Q \times \Sigma^\star \to Q$. Now, we define the language accepted by DFA A as:

$$L(A) = \{w : \hat{\delta}(q_0, w) \in F\}$$

Claim: For this automaton, $L(A) = \{w : \#_0(w)$ is even, and $\#_1(w)$ is odd$\}$. But how do we prove this?

Let's rephrase our claim. For every $w \in \{0,1\}^\star$, if $\#_0(w)$ is even and the $\#_1(w)$ is odd, then A accepts w, and otherwise, A does not accept w.

Since we said "for every $w \in \{0,1\}^\star$", that should be a give away that we should use structural induction for this proof.

Let's define $\{0,1\}^\star$ inductively. Note that we are *not* yet defining the set of accepted members for the DFA, but instead, we are aiming to inductively define the set of all possible

words across the alphabet $\Sigma = \{0, 1\}$.

We have our core set $A = \{\epsilon\}$ (not to be confused with the A we're using to denote the DFA). We also have two operations for any string σ,

$$P = \left\{ \frac{\sigma}{\sigma 1}\ ,\ \frac{\sigma}{\sigma 0} \right\}$$

Rather than proving our claim directly, it will be easier to prove a stronger claim, which characterizes every state of the machine. This stronger claim (four claims, in fact) will be easier to prove because we'll have more to work with in our induction hypothesis.

For every $w \in \{0, 1\}^\star$:

- If $\#_0(w)$ is even and $\#_1(w)$ is even, then $\hat{\delta}(q_0, w) = q_0$.
- If $\#_0(w)$ is even and $\#_1(w)$ is odd, then $\hat{\delta}(q_0, w) = q_1$.
- If $\#_0(w)$ is odd and $\#_1(w)$ is odd, then $\hat{\delta}(q_0, w) = q_2$.
- If $\#_0(w)$ is odd and $\#_1(w)$ is even, then $\hat{\delta}(q_0, w) = q_3$.

Proof. We will prove the four claims using structural induction on $\{0, 1\}^\star$.

Base case: $w = \epsilon$. It's easy to see that $\#_0(w)$ is even, $\#_1(w)$ is even, and indeed, $\hat{\delta}(q_0, \epsilon) = q_0$.

Induction step: assume the four claims hold for w. We need to show they hold for w_0 and w_1.

The operation w_0: if $\#_0(w_0)$ is even and $\#_1(w_0)$ is even, then $\#_0(w)$ is odd and $\#_1(w)$ is even. By the induction hypothesis, $\hat{\delta}(q_0, w) = q_3$.

By the definition of $\hat{\delta}$, $\hat{\delta}(q_0, w_0) = \delta(\hat{\delta}(q_0, w), 0) = \delta(q_3, 0) = q_0$.

In order to finish this proof, we still need to prove the other three cases for w_0 and all four cases for w_1. □

Every task carried out by a DFA A has the general form: "given some input string w, decide if $w \in L(A)$". In particular, we can say each A handles the decision problem of the language $L(A)$. DFAs perform this task by consuming one character of w at a time.

Note that not every language can be represented by a DFA. The following languages can all be handled by a DFA:

- $\{\epsilon\}$
- $\Sigma^\star$
- For every $a \in \Sigma, \{a\}$
- For every $w \in \Sigma^\star, \{w\}$

3.1.1 Closure Operations

Is the family of languages that can be computed by a DFA closed under $\cup, \cap$, set difference, concatenation, $L^\star$, etc?

Claim: if each of L_1, L_2 can be computed by some DFA, then so can $L_1 \cap L_2, L_1 \cup L_2$, and $L_1 \backslash L_2$.

Proof. Let A_1 and A_2 be DFAs such that $L(A_1) = L_1$ and $L(A_2) = L_2$. Let's construct some DFA A_3 such that $L(A_3) = L_1 \cap L_2$.

The general idea is to run the two machines, A_1 and A_2, in parallel, and if $w \in F$ in both machines, accept w for $L_1 \cap L_2$.

Given any two automata $A_1 = (\Sigma, Q_1, q_0^1, \delta_1, F_1)$ and $A_2 = (\Sigma, Q_2, q_0^2, \delta_2, F_2)$, we define the **product automaton** as

$$A_1 \times A_2 = (\Sigma, Q_1 \times Q_2, (q_0^1, q_0^2), \delta_{A_1 \times A_2}, F_{A_1 \times A_2})$$

where Σ is the same alphabet as defined previously, $Q_1 \times Q_2 = \{(p, q) : p \in Q_1, q \in Q_2\}$, and $\delta_{A_1 \times A_2}((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$.

F will be defined differently depending on which combination of L_1 and L_2 we wish to compute. For example:

- $F_{A_1 \cap A_2} = \{(p, q) : p \in F_1 \text{ and } q \in F_2\}$
- $F_{A_1 \cup A_2} = \{(p, q) : p \in F_1 \text{ or } q \in F_2\}$
- $F_{A_1 \backslash A_2} = \{(p, q) : p \in F_1 \text{ and } q \notin F_2\}$

Conclusion: if L_1 and L_2 both have DFAs, then so do $L_1 \cap L_2, L_1 \cup L_2, L_1 \backslash L_2, L_1 \Delta L_2$, etc. (any boolean operation on sets). However, note that the product automaton does not tell us if a language is closed under $L^\star$ or concatenation.

Is it true that whenever L has a DFA, then so does its complement, $L^C = \Sigma^\star \backslash L$? Yes, its DFA is $A_{\Sigma^\star \times L}$. Let $A = (\Sigma, Q, q_0, \delta, F)$ be a DFA such that $L(A) = L$, then let $A^C = (\Sigma, Q, q_0, \delta, Q \backslash F)$. Then, $L(A^C) = L^C$.

Since in our definition of a DFA we required that for every state $q \in Q$ and every letter $a \in \Sigma$, q has some outgoing edge labeled by a (we are never stuck). □

3.1.2 The Relationship Between Regular Expressions and DFAs

← September 26, 2013

Claim: for every regular expression r, there exists some DFA A such that $L(r) = L(A)$.

Proof. We will prove this claim by structural induction on the set of regular expressions.

Base case: the elements of the core set of regular expressions are $\{\epsilon, \underbrace{a, b, \ldots,}_{\Sigma}, \emptyset\}$. For each of these, we should show that there exists some DFA A such that $L(A) = L(r)$.

- $L(\epsilon) = \{\epsilon\}$. We have seen such an A.
- $L(\emptyset) = \emptyset$. We have seen such an A.
- For every $a \in \Sigma, L(a) = \{a\}$ and there exists some DFA A such that $L(A) = \{a\}$.

Inductive step: we must show that under each operation the claim continues to hold. Recall the operations we defined on regular expressions to be

$$P = \left\{ \frac{r_1 \quad r_2}{r_1 + r_2}\ ,\ \frac{r_1 \quad r_2}{r_1 r_2}\ ,\ \frac{r}{r^\star}\ ,\ \frac{r}{(r)} \right\}$$

1. Consider the + operation. Assume there are automata A_1, A_2 such that $L(r_1) = L(A_1)$ and $L(r_2) = L(A_2)$. We need to show that there exists some A_3 such that $L(A_3) = L(r_1 + r_2)$.

   Recall that $L(r_1 + r_2) = L(r_1) \cup L(r_2)$. We saw that given A_1, A_2 we can construct A_3 such that $L(A_3) = L(A_1) \cup L(A_2)$.

2. Consider the concatenation operation. Assume $L(A_1) = L(r_1)$ and $L(A_2) = L(r_2)$. We need to define A_3 such that $L(A_3) = L(r_1 r_2) = L(A_1)L(A_2)$. Note that

   $$L(A_1)L(A_2) = \{ww' : w \in L(A_1), w' \in L(A_2)\}$$

   One idea on how to approach this is to take DFAs A_1 and A_2 and merge each final state in A_1 with the initial state of A_2 (and make the states no longer final states). However, this approach will not suffice because we could then have multiple edges for the same letter, leading out of those newly-merged states.

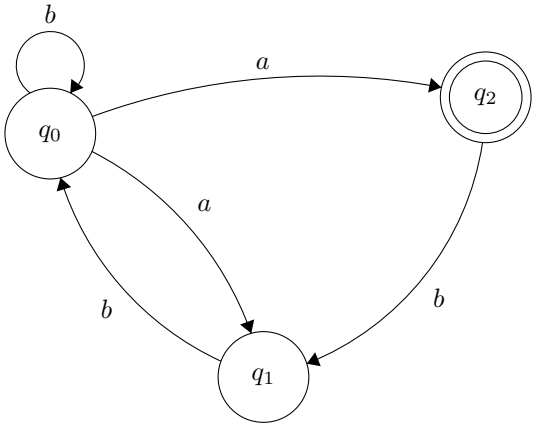
   In order to solve this problem, we need to take a detour to define a new family of machines. We'll return to this proof at a later point in time. □

3.2 Non-deterministic Finite Automata (NFAs)

NFAs are similar to deterministic finite automata, except they allow more edges on a state than there are letters in Σ. You can create multiple paths to be followed simultaneously.

We define an NFA N as $N = (\Sigma, Q, q_0, \delta, F)$ where Σ, Q, q_0, and F are exactly the same as in DFAs. However, for NFAs, we define $\Sigma : Q \times \Sigma \to \{B : B \subseteq Q\}$.

Example 3.2. Will this machine (an NFA) accept a?



Whether this machine will accept a is largely up to how we define acceptance. Generally, NFAs are defined so if there is *any* valid path that leads to a final state, the string will be accepted. So, under that definition, a would be accepted by this NFA.

3.2.1 Languages Accepted by an NFA

Let's define the language $L(N)$ that is accepted by an NFA N. Intuitively, we will accept a string w if and only if there is a path from q_0 following the letters of w and ending inside F.

We will extend our formal definition of δ to $\hat{\delta} : Q \times \Sigma^\star \to \{B \subseteq Q\}$. $\hat{\delta}(q, w)$ is the set of all states that can be reached from q after reading w.

For $w = \epsilon$, $\hat{\delta}(q, \epsilon) = \{q\}$. Assume that $\hat{\delta}(q, w)$ is already defined; say $\hat{\delta}(q, w) = \{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\} \subseteq Q$. Now, define $\hat{\delta}(q, wa)$ for any $a \in \Sigma$ as

$$\hat{\delta}(q, wa) = \bigcup_{l=1}^{k} \delta(q_{i_l}, a)$$

We can then define the language accepted by a particular NFA N as

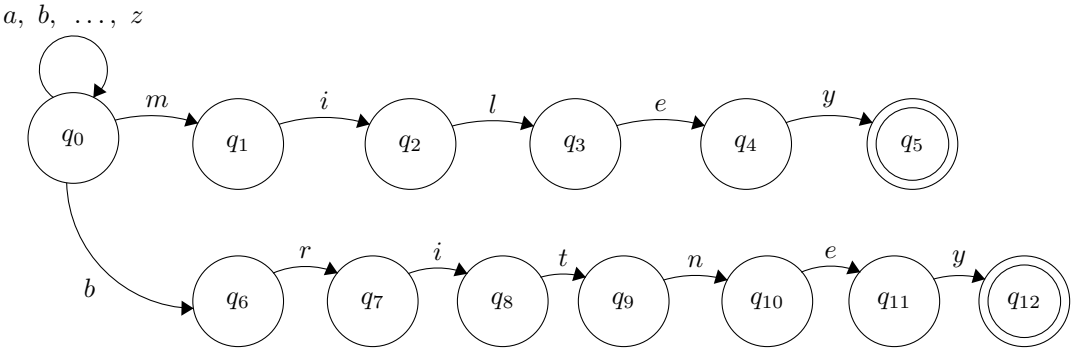
$$L(N) = \{w : \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Phrased another way, this set contains all words w which, after reading all characters in w, ended up in *at least* one accepting state.

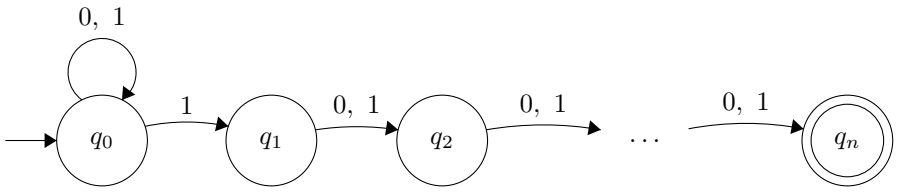
3.2.2 Uses of NFAs

NFAs sometimes give us a very compact representation of a language that would otherwise be complicated to describe as a DFA.

Example 3.3. NFAs could be used for keyword searches. For example, suppose we want to create an NFA that matches the keywords "miley" and "britney", and nothing else. We can form this NFA with the alphabet $\Sigma = a, b, \ldots, z$:



Example 3.4. Suppose we want to define an NFA for the language $L_n = \{w : \text{the } n\text{th-before-last letter in } w \text{ is } 1\}$ on the alphabet $\Sigma = \{0, 1\}$. We could form the following NFA to represent this:



When using an NFA to represent languages like L_n, we need just $n + 1$ states.

Claim: every DFA that accepts L_n must have at least 2^n states. Why? For every word w of length less than n, there should be a different state. In other words, if $w_1 \neq w_2$ and $\hat{\delta}(q_0, w_1) = \hat{\delta}(q_0, w_2)$, the DFA is bound to fail. Since $w_1 \neq w_2$, the last "1" must be in a different position in w_1 and w_2, which makes this DFA representation fail.

Claim: a language L has an NFA N such that $L(N) = L$ if and only if there is a DFA A such that $L(A) = A$. In other words, nondeterminism does not increase the power of finite automata.

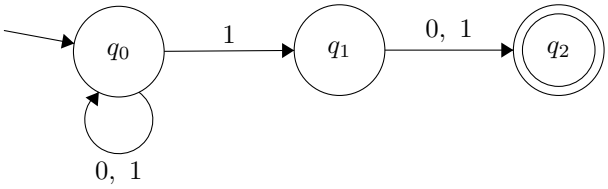
Proof. First, note that any DFA can also be viewed as an NFA. For every $q \in Q$ and $a \in \Sigma$, if $\delta_N(q, a) = p$, define $\delta_D(q, a) = \{p\}$. In other words, let $\delta_N(q, a) = \{\delta_D(q, a)\}$.

For the other direction, if we're given some NFA $N = (\Sigma^N, Q^N, q_0^N, \delta^N, F^N)$, we wish to define a DFA $A = (\Sigma^A, Q^A, q_0^A, \delta^A, F^A)$ such that $L(A) = L(N)$. In particular:

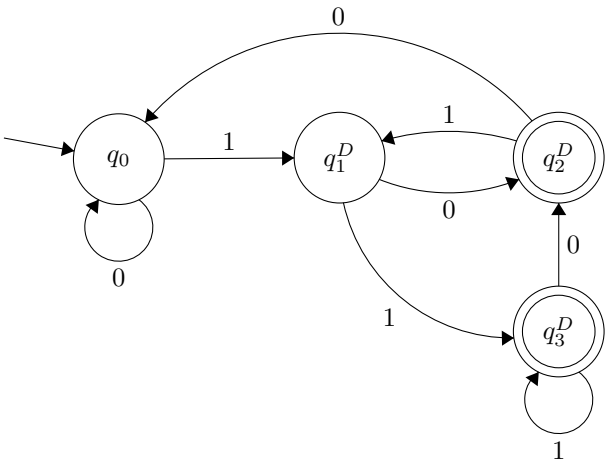
- $\Sigma^A = \Sigma^N$
- $Q^A = \{B \subseteq Q^N\}$
- $q_0^A = \{q_0^N\}$
- $\delta^A(q, a) =$ the set of all states that can be reached from q by reading a.
- $F^A = \{B \subseteq Q^N : B \cap F^N \neq \emptyset\}$

□

Example 3.5. Let's look at an example of converting an NFA to an equivalent DFA. Let N be:



N accepts all strings where the second-to-last letter is 1. Note that N is nondeterministic because q_0 has two edges for 1, and q_2 has no edges. By way of subset construction, we can construct an equivalent DFA:



where $q_0^D = \{q_0\}, q_1^D = \{q_0, q_1\}, q_2^D = \{q_0, q_2\}, q_3^D = \{q_0, q_1, q_2\}$.

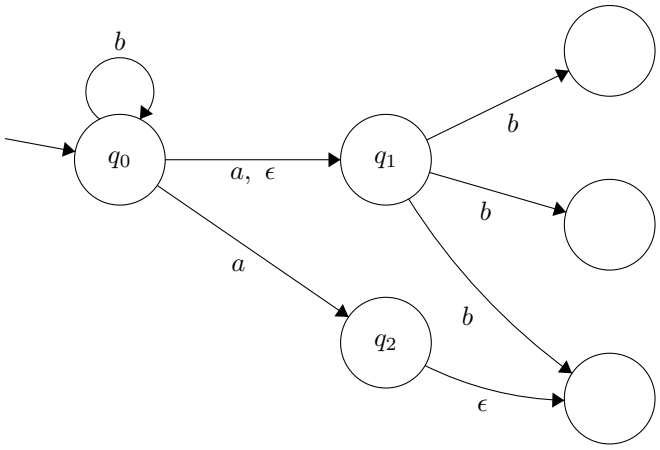
The accepting states of this DFA are all states which contain at least one accepting state from the NFA.

Note that in the worst case, an NFA with n states could be equivalent to a DFA with 2^n states. This intuitively follows from the way we construct NFAs – there are 2^n subsets of n items.

3.3 Nondeterministic Finite Automata with Epsilon Transitions

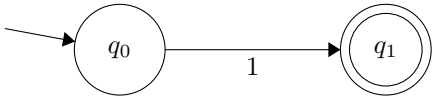
We will now introduce a third type of machine, **the ϵ-NFA**. An ϵ-transition is essentially a transition that can occur spontaneously. That is, an ϵ-transition is a transition that can be followed without reading any characters.

Example 3.6. This is what an ϵ-NFA could look like:

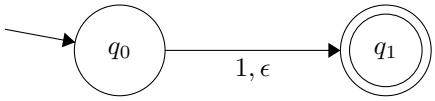


Let's consider a more simplistic example to understand ϵ-NFAs better.

Example 3.7. Consider N_1:

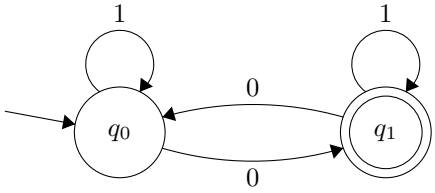


It's trivial to see that $L(N_1) = \{1\}$. Now, consider N_2, the same NFA but adding an ϵ-transition:

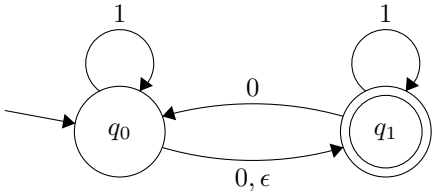


Note that now, $L(N_2) = \{\epsilon, 1\}$. That is, $L(N_2)$ contains the empty string, and the string that is just a single 1.

Let's now consider a different NFA, N_3:



$L(N_3) = \{w : \#_0(w) \text{ is odd}\}$. Finally, let's consider N_4, the same NFA but with an ϵ-transition added:



N_4 accepts everything. That is, $L(N_4) = \{0,1\}^\star$. In N_4, we can spontaneously move to q_1 initially. Then, either 1 loops in q_1 or you follow 0 to q_0, but in that case you can spontaneously move back to q_1 again.

We can define an ϵ-NFA as follows. For such a machine N, $L(N) = \{w :$ there is a way that upon reading w and using ϵ-transitions, q_0 leads to some accepting state$\}$.

Claim: a language is accepted by an ϵ-NFA if and only if it is accepted by some DFA.

The proof of this claim is the same as before, except additionally taking the ϵ-transitions into account when performing the subset construction of the DFA.

Recall the theorem we discussed a few weeks ago:

Theorem. For every regular expression r, there exists a DFA A such that $L(r) = L(A)$.

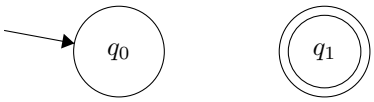
Proof. We will prove this theorem by structural induction on the set of regular expressions.

Let's prove an equivalent claim:

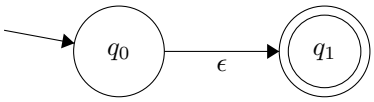
Claim: for every r, there is a DFA A such that A has only a single accepting state, no edges leading out of the accepting state, but A may have ϵ-transitions, and $L(r) = L(A)$. (Note that this isn't formally a DFA, since we allow ϵ-transitions and missing edges, but since every ϵ-NFA is equivalent to a DFA, we know that a corresponding *proper* DFA exists.)

Induction basis: $\emptyset, \epsilon$, and a for $a \in \Sigma$.

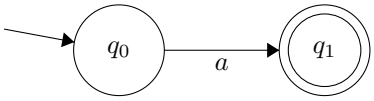
An ϵ-DFA for $L(\emptyset) = \emptyset$ is:



An ϵ-DFA for $L(\epsilon) = \{\epsilon\}$ is:



An ϵ-DFA for $L(a) = \{a\}$ is:

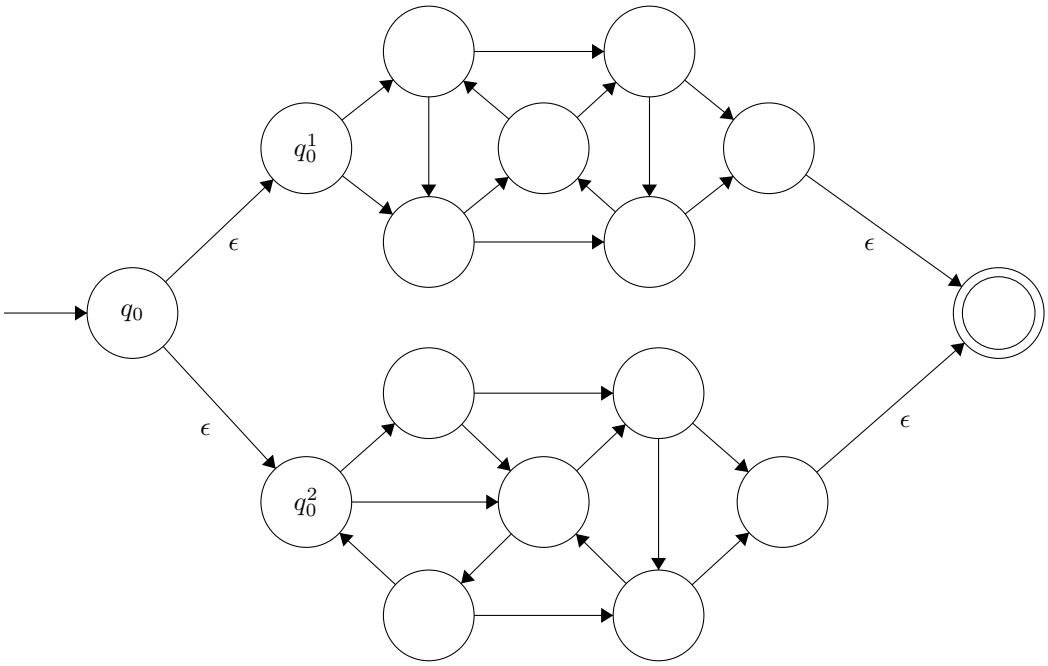


Induction step: assume r_1, r_2 are regular expressions such that there are A_1, A_2 for which $L(r_1) = L(A_1)$ and $L(r_2) = L(A_2)$. Recall the operations that we defined on regular expressions:

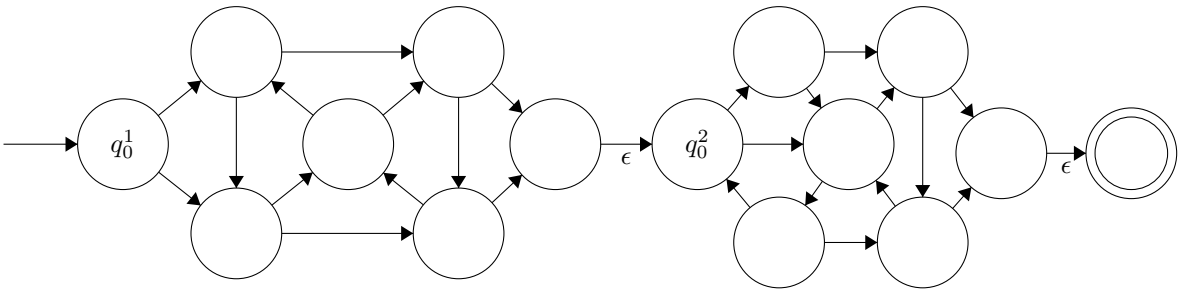
$$P = \left\{ \frac{r_1 \quad r_2}{r_1 + r_2}\ ,\ \frac{r_1 \quad r_2}{r_1 r_2}\ ,\ \frac{r_1}{(r_1)}\ ,\ \frac{r_1}{r_1^\star} \right\}$$

We must show that our claim holds for each of these operations.

Let's consider the first operation, $\frac{r_1 \quad r_2}{r_1 + r_2}$. We need to construct an automaton A_3 such that $L(A_3) = L(A_1) \cup L(A_2)$. Essentially, we want to introduce a new q_0 that is connected via ϵ-transitions to the q_0 states of A_1 and A_2. Then, convert the single final state of each of A_1 and A_2 into a non-final state, but connect that state via an ϵ-transition to a new accepting state. Here's a visualization of this construction:



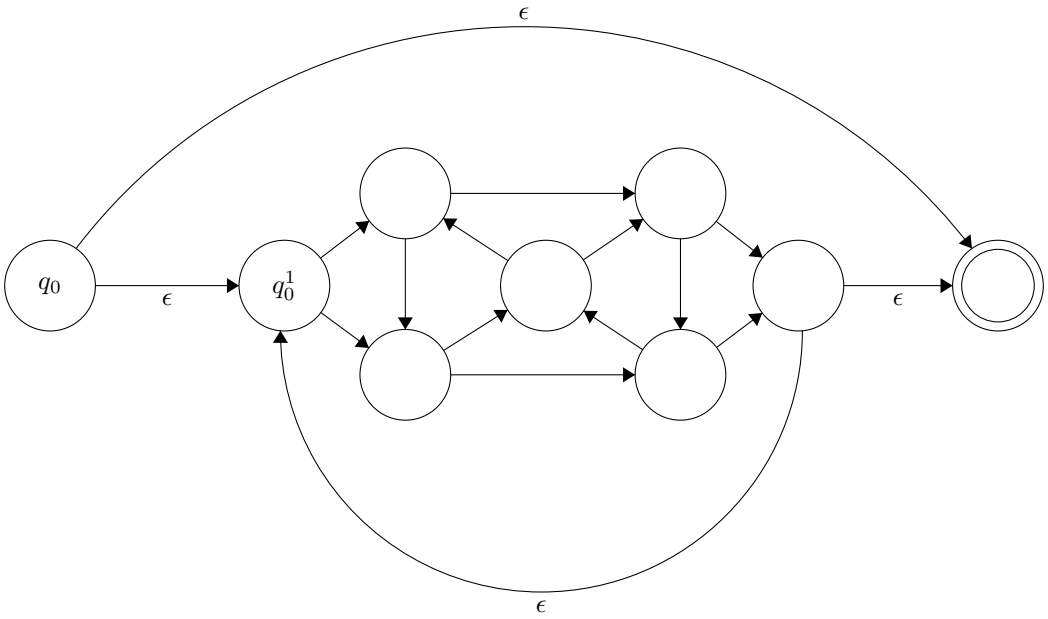
Next, let's look at the operation $\frac{r_1 \quad r_2}{r_1 r_2}$. We need to construct an automaton A_4 such that $L(A_4) = L(A_1)L(A_2)$. Essentially, we want to make the final states of A_1 and A_2 non-final, then connect the final state of A_1 to the q_0 of A_2, through an ϵ-transition. Then, the previously-final state of A_2 should connect to a new final state through an ϵ-transition. Here's a visualization of this construction:



The operation $\frac{r_1}{(r_1)}$ is trivially true. The DFA for this operation is the same as the DFA for r_1.

Finally, let's look at our last operation, $\frac{r_1}{r_1^\star}$. Assume $L(A) = L(r)$. We will define $A^\star$, an ϵ-DFA such that $L(A^\star) = L(r^\star) = (L(r))^\star$.

We will introduce a new initial state, q_0, and a new accepting state. We will connect q_0 and our new accepting state by an ϵ-transition, since $\epsilon \in L(r^\star)$. We will connect our q_0 to the initial state of A by an ϵ-transition. We will make the final state of A non-final, and connect it back to A's previously-initial state (A's q_0) by an ϵ-transition, to allow repetition. Finally, we will connect the previously-final state of A to our new accepting state by an ϵ-transition. Here's a visualization of this construction:



□

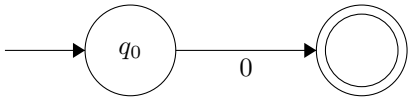
Example 3.8. Consider the regular expression $r = (0+1)^\star 1(0+1)$. This expression matches all strings with their second-to-last character being 1.

Let's construct an ϵ-DFA A such that $L(A) = L(r)$, by following the steps of the previous proof.

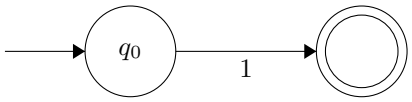
We would typically generate the regular expression r in the following order:

$$\begin{aligned} 0 \\ 1 \\ (0+1) \\ (0+1)^\star \\ (0+1)^\star 1 \\ (0+1)^\star 1(0+1) \end{aligned}$$

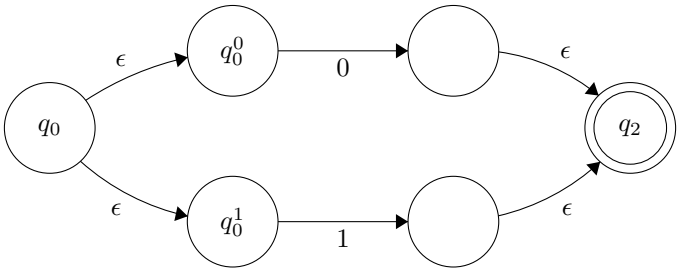
We will construct the automaton in this order as well. Let's start with an automaton for matching 0:



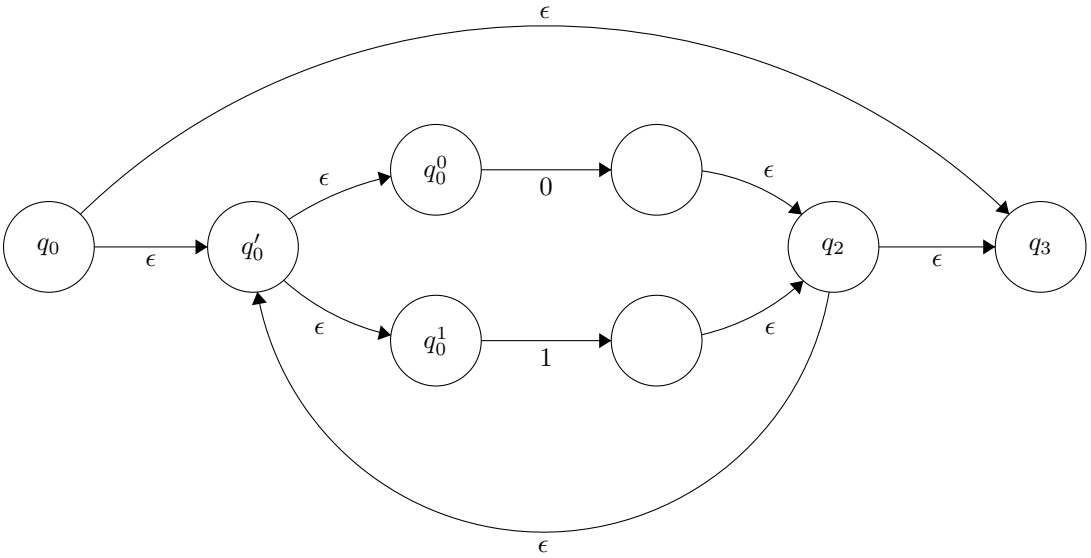
Next, we'll create an automaton for matching 1:



Next, we'll combine these two machines to create an automaton that matches $(0 + 1)$:



Next, we'll introduce the $\star$. We'll form an automaton that matches $(0 + 1)^\star$.



The rest of this construction is left as an exercise.

4 Overview of Regular Languages

← October 3, 2013

The following definitions are equivalent:

- L is regular if for some regular expression r, $L = L(r)$.
- L is regular if there exists some DFA, A, such that $L = L(A)$.
- L is regular if there exists some NFA, N, such that $L = L(N)$.
- L is regular if there exists some ϵ-NFA, N, such that $L = L(N)$.

How can we show that some language L is not regular? Let's look at an example.

Example 4.1. Let $L_0 = \{0^n 1^n : n \in \mathbb{N}\}$. We claim that L_0 is not a regular language.

Proof. Assume by way of contradiction that L_0 is regular. Therefore, it is $L(A)$ for some DFA A. Say $A = (\{0,1\}, Q, q_0, \delta, F)$. Let $k = |Q|$, the number of states in A.

Consider $w = 0^{k+2}1^{k+2}$. Clearly, $w \in L_0$. Notice that there must be some $i < j \leq k+2$ such that $\hat{\delta}(q_0, 0^i) = \hat{\delta}(q_0, 0^j)$, by the pigeonhole principle.

The Pigeonhole Principle: if you have more pigeons than holes, then you can't have every pigeon in its hole alone.

Note that

$$0^{k+2}1^{k+2} = \underbrace{0^i 0^{j-i} 0^{k+2-j}}_{0^{k+2}} 1^{k+2}$$

We can now see that $w' = 0^i 0^{j-i} 0^{j-i} 0^{k+2-j} 1^{k+2} \notin L_0$. The general idea here is that if we take a long enough word, it must visit the same state again, which means we may accept members into our language that aren't supposed to be part of the language. □

4.1 The Pumping Lemma

Lemma (The Pumping Lemma)**.** For every regular language L, there exists some number n such that for every $w \in L$, if $|w| \geq n$, then there are x, y, z so that

1. $xyz = w$,
2. $|xy| \leq n$,
3. $y \neq \epsilon$, and
4. For every k, $xy^k z \in L$

Note that the converse is *not* true. There are languages that are not regular that still satisfy these properties.

Let's begin by looking at some positive examples.

Example 4.2. Let $r = (0011)^\star$, and let $L = L(r)$. Pick $n = 5$. For every $w \in L$ such that $|w| > 5$, we have the general form

$$\underbrace{0011}_{y}\underbrace{001100110011}_{z}\ldots$$

Pick $x = \epsilon, y = 0011$, and $z =$ the rest of w. Remember: just because a language satisfies these properties, does *not* prove it is regular.

Example 4.3. Let $L_{7a} = \{w \in \{a, b\}^\star : \text{ the 7th-to-last letter in } w \text{ is a}\}$.

Let x be the first letter in w, and let y be the second letter in w. Note that pumping y does not affect the last 7 letters, so the pumped words would still be in the language, as expected.

Now, let's look at a couple of negative examples.

Example 4.4. Let $L_{\text{plus}} = \{0^s 1^t 0^r : s + t = r\}$.

Claim: L_{plus} is not regular. This is an intuitive claim since finite automata do not have memory. They only know which state they're currently in. Similarly, counting numbers > 10 on your fingers (without using memory) can only determine the count modulus 10, since we only have 10 fingers.

We need to show that the claim of the Pumping Lemma fails. Given some n, consider

$$w = 0^n 10^{n+1} \in L_{\text{plus}}$$

For every choice of x, y, z, satisfying requirements (1), (2), and (3), inevitably $x = 0^i, y = 0^j$ such that $i + j \leq n$ and $j > 0$. Then we have $z = 0^{n-(i+j)}10^{n+1}$.

Pick $k = 2$. Then we have

$$\begin{aligned} xy^k z &= 0^i 0^{2j} 0^{n-(i+j)} 10^{n+1} \\ &= 0^{n+j} 10^{n+1} \end{aligned}$$

where $j > 0$, which is not in L_{plus}.

We could rephrase the Pumping Lemma in terms of a game between two players, R and NR. R is the player who is trying to convince us that the language is regular, and NR is the player who is trying to convince us otherwise.

First, R chooses an n. NR can then fire back with a word w such that $w \in L$ and $|w| \geq n$. R can then provide an x, y, z such that $|xy| \leq n$, $xyz = w$, and $y \neq \epsilon$.

If $xy^k z \in L$ for all k, then R wins. If $xy^k z \notin L$ for some k, then NR wins.

The Pumping Lemma states that if L is regular, the player R can guarantee that she wins.

Example 4.5. Consider $L_{\text{prime}} = \{\underbrace{a \ldots a}_{p \text{ times}} : p \text{ is prime}\} = \{\underbrace{aa}_{2}, \underbrace{aaa}_{3}, \underbrace{aaaaa}_{5}, \underbrace{aaaaaaa}_{7}, \ldots\}$.

Claim: L_{prime} is not regular.

We wish to show that the player NR can defeat the player R, regardless of how well R plays the game. Let n be the number picked by the player R. Let p be a prime number greater than $n+2$.

Pick $w = a^p \in L_{\text{prime}}$. Suppose R picks some xyz satisfying requirements (1), (2), and (3). Let $x = a^r, y = a^m$, and $z = a^{p-(r+m)}$ where $m \geq 1$ and $r + m \leq n$.

Player NR now picks $k = (p - m)$. Then we have

$$\begin{aligned} xy^k z &= a^r a^{m(p-m)} a^{p-(r+m)} \\ &= a^{m(p-m)+(p-m)} \end{aligned}$$

Note that $(p-m)(m+1)$ is not prime, because it's two factors, where $p - m > 1$ (since $m < n < p$), and $m + 1 > 1$ (since $m \geq 1$). Therefore, this string cannot have a prime length, so $xy^k z \notin L_{\text{prime}}$.

← October 8, 2013

Example 4.6. Let $L_{klk} = \{0^k 1^l 0^k : k, l \in \mathbb{N}\}$. Intuitively, this language is not regular because a machine with finite memory cannot remember how many zeros it counted for the first run of 0s.

Claim: L_{klk} is not a regular language.

Proof. We will prove this claim with the Pumping Lemma. We need to show that player NR can win, regardless of what player R does.

Given some n (chosen by player R), our first try is to choose $w = 0^{2n}$. In this case, we may attempt to construct $x = \epsilon, z = \epsilon, y = 0^{2n}$, but that violates the second condition of the Pumping Lemma. We may also try $x = \epsilon, y = 00, z = 0^{2n-2} =$ the rest of the word. For any k that NR picks, $xy^k z = (00)^k 0^{2n-2} = 0^{2(k+n-1)} \in L_{klk}$. R won in this case, which is not what we want. Note that just because R won in this case, does not prove that L_{klk} is regular.

Let's try another word. Given some n (chosen by player R), we will choose $w = 0^n 1^n 0^n$. Any choice of x, y such that string $|xy| \leq n$ will have $z = 0^i 1^n 0^n$. Now pick k arbitrarily; say $k = 2$. Then since 0^j for some $j \geq 1$, $xy^2 z = 0^{n+j} 1^n 0^n \notin L_{klk}$. □

Example 4.7. Let $L_< = \{w : \#_0(w) > \#_1(w)\}$. Intuitively, this language is non-regular because we can't check this condition with unbounded memory. In some *particular* cases, such as $w = 0101010101$, we *can* verify membership with bounded memory. Such words would not be good candidates for w in our proof.

Claim: $L_<$ is not regular.

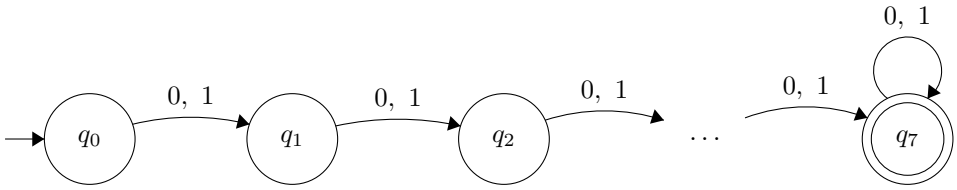
Proof. We will prove this claim with the Pumping Lemma. We need to show that player NR always wins.

Given some n, pick $w = 1^n 0^{n+1}$. Now, since $|xy| \leq n$, player R is forced to choose $x = 1^j, y = 1^i, z = 1^{n-(i+j)} 0^{n+1}$, for $i \geq 1$. Any $k > 1$ will then result in $xy^k z \notin L_<$. □

Claim: let L be a non-empty language that for some $s \in \mathbb{N}$ accepts no words of length $\leq s$. Then, if a DFA A computes L (that is, $L = L(A)$), then inevitably, A has more than s states.

For example, $L_{>6} = \{w : |w| > 6\}$ is not $L(A)$ for any DFA with at most 6 states.

Note that languages of this form *are* regular. We only need to count up to s, which would require s states. For instance, $L_{>6}$ is regular because the following DFA computes it:



Proof. Note that if $L = L(A)$ for some DFA A, then in the Pumping Lemma, one can pick $n = |Q^A|$ (the number of states in A).

Assume by way of contradiction that for some A with $n < s$ states, $L = L(A)$. Apply the Pumping Lemma where player R chooses $n = |Q^A|$. Let w be a word of shortest length in L. Note that $|w| > s > n$, which makes this a legal choice.

Given a decomposition x, y, z, choose $k = 0$. Since $xyz = w$, $|xy^0z| < |w|$. This is a contradiction since we assumed we selected the smallest word in L, and then we made it shorter. □

4.2 Equivalent Descriptions of Languages

Earlier in the course, we stated a basic theorem that we didn't prove fully. We'd like to complete that proof now.

Theorem. The following conditions on a language are equivalent:

1. $L = L(r)$ for some regular expression r.
2. $L = L(A)$ for some DFA A.
3. $L = L(N)$ for some NFA N.
4. $L = L(N)$ for some ϵ-NFA N.

We've shown (1) $\implies$ (2) and (3) $\implies$ (2) in class previously. (2) $\implies$ (3) $\implies$ (4) is trivial. We're still missing (2) $\implies$ (1), so we'll prove that now.

Lemma. For every DFA, A, there exists a regular expression r, such that $L(r) = L(A)$.

Proof. Let $A = (\Sigma, Q, q_0, \delta, F)$. Without loss of generality, redefine $Q = \{q_1, \ldots, q_n\}$ for some n.

By induction on k, we will show that for every $i, j \leq n$, there exists a regular expression R_{ij}^k such that $L(R_{ij}^k) = \{w : \text{ on } A, w \text{ takes us from } q_i \text{ to } q_j \text{ without visiting any state of index } > k\}$.

Note that

$$L(A) = \bigcup_{q_j \in F} L(R_{ij}^n)$$

Base case: $k = 0$. $L_{ij}^0 = \{w :$ go from q_i to q_j without touching any other state$\}$. We have two cases.

1. $i \neq j$. If there is an edge between the two states, let $a_1, \ldots, a_k$ be all letters on this edge, and $R_{ij}^0 = a_1 + \ldots + a_k$. If there is no direct edge from q_i to q_j, then $R_{ij}^0 = \emptyset$ (i.e. there is no way to go from q_i to q_j without touching any other state).
2. $i = j$. If there is a self-edge labeled $a_1, \ldots, a_k$, then $R_{ii}^0 = a_1 + \ldots + a_k + \epsilon$. Note that ϵ is added here because you can also go from a state to itself without reading any letters. If there is no self loop on q_i, then $R_{ii}^0 = \epsilon$.

Induction step: assume we have regular expressions R_{ij}^l for $i, j \leq n$, and $l < k$. Construct R_{ij}^k for all such i, j. We then have:

$$R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^\star R_{kj}^{k-1} + R_{ij}^{k-1}$$

R_{ik}^{k-1} and R_{kj}^{k-1} indicate the cases where we aren't stepping on state k. $(R_{kk}^{k-1})^\star$ indicates that we can step on state k as many times as we want (or not at all). Finally, R_{ij}^{k-1} represents the case where we go from state q_i to q_j without stepping on q_k at all. □

4.3 Algorithms on Languages and Finite Automata

← October 10, 2013

Let's look at a number of algorithms that you should be able to carry out on languages and automata.

1. **Given w and DFA (or NFA, or ϵ-NFA) M, decide if $w \in L(M)$.**
2. **Given a regular expression r, construct a (ϵ-)NFA, M, such that $L(M) = L(r)$.**

   Example 4.8. Let $r = (a + b)a^\star + b(a + \epsilon)$.

   Start by constructing an automaton for atoms a, b, and ϵ, then move on to more complex expressions like $(a + b)$ or $(a + \epsilon)$ by combining the automata created previously.

3. **Given w and r, is $w \in L(r)$?** Note that this is difficult to do in general without converting the regular expression to an automaton.
4. **Given an NFA, M, construct a DFA A such that $L(M) = L(A)$.** Recall that states in the equivalent DFA each represent a set of states from the NFA.
5. **Given a DFA, A, construct a regular expression r such that $L(r) = L(A)$.** We briefly touched on this before, so we'll do a more in-depth example now.

   General idea:

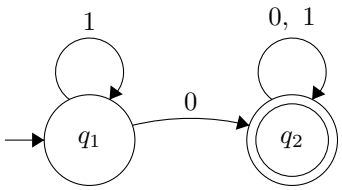
(a) Rename the states of A as $\{q_1, \ldots, q_n\}$ for some n.

(b) For every $k, i, j \leq n$, let $R_{ij}^k = \{w :$ if we read w starting in state i, we will end in state j, and, along the path, we step on no state whose index is $> k\}$.

(c) By induction on k, construct regular expressions r_{ij}^k such that $L(r_{ij}^k) = R_{ij}^k$ (for all i, j)

(d) Finally, to get a regular expression for $L(A)$, let $r = \sum\{r_{ij}^n : i \in F\}$.

Example 4.9. Let A be the following DFA:



Notice that $L(A) = \{w : w$ contains a $0\}$. We can also see that a regular expression for this language is $r = 1^\star 0(0+1)^\star$, but determining this was only possible because this automaton is simple.

If we couldn't determine the regular expression like this, we could alternatively follow the translation procedure. The translation procedure requires much more work, but no creativity or luck is necessary.

We need the following tables in order to carry out the translation procedure:

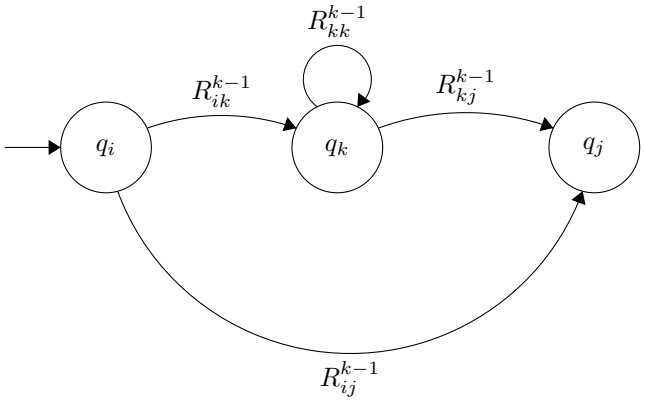
0	regular expression	1	regular expression	2	regular expression
R_{11}^0	$(1+\epsilon)$	R_{11}^1	$r_{11}^0(r_{11}^0)^\star r_{11}^0 + r_{11}^0$	R_{11}^0	
R_{12}^0	0	R_{12}^1	$r_{11}^0(r_{11}^0)^\star r_{12}^0 + r_{12}^0$	R_{12}^0	
R_{21}^0	$\emptyset$	R_{21}^1		R_{21}^0	
R_{22}^0	$0+1+\epsilon$	R_{22}^1		R_{22}^0	

The 0, 1, and 2 in the headers of these tables represent the superscript. You need to fill out these tables fully, although we have omitted some of that here.

How do we construct r_{ij}^k from the expressions $\{r_{ij}^{k-1} : i, j \leq m\}$? You can use the recursive formula:

$$r_{ij}^k = r_{ik}^{k-1}(r_{kk}^{k-1})^\star r_{kj}^{k-1} + r_{ij}^{k-1}$$

Here's a visualization:



6. **Given a DFA $A = (\Sigma, Q, q_0, \delta, F)$, decide whether $L(A) = \emptyset$ or not.**

   We can't just use a regular expression representation of A to determine this, since $L(\emptyset) = \emptyset$, but there are also other regular expressions r such that $L(r) = \emptyset$. For example, $\emptyset + \emptyset$, $(\emptyset + \emptyset)(\emptyset + \emptyset)$, or $((a + b)b^\star + b)\emptyset$.

   Algorithm 1: recall that if $L(A)$ contains no strings of length $\leq |Q|$ then $L(A) = \emptyset$, by the Pumping Lemma.

   Now, list all strings of length $\leq |Q|$: say, $w_0, w_1, \ldots, w_{2|Q|}$. For each such w_i, check whether $w_i \in L(A)$. If all these words are rejected, then we can conclude that $L(A)$ is empty.

   Algorithm 2: check if there's a path that starts in q_0 and ends in an accepting state. This uses the concept of **reachability**. Construct:

$$\begin{aligned} \{q_0\} &= \text{ all states reachable from } q_0 \text{ with 0 edges} \\ &\vdots \\ A_k &= \text{ all states reachable from } q_0 \text{ traversing at most } k \text{ edges} \\ A_{k+1} &= A_k \cup \{\text{all neighbors}\} \end{aligned}$$

   Finally, check $A_n \cap F$. If this set is empty, then $L(A) = \emptyset$. Otherwise, a path exists which means $L(A)$ is non-empty.

7. **Given two DFAs, A_1, A_2, decide whether $L(A_1) = L(A_2)$.** Note that these languages could be arbitrarily large. Also, keep in mind that there could be different regular expressions, graphs, and automata that all compute the same language.

   Claim: $L(A_1) = L(A_2)$ if and only if $L(A_1) \backslash L(A_2) = \emptyset$ and $L(A_2) \backslash L(A_1) = \emptyset$.

   We need to construct A_3 such that $L(A_3) = L(A_1) \backslash L(A_2)$ (using the product automaton). Similarly, we need to construct A_4 such that $L(A_4) = L(A_2) \backslash L(A_1)$.

   We then check for emptiness in both $L(A_3)$ and $L(A_4)$. If both are empty, then

we can conclude that $L(A_1) = L(A_2)$. Otherwise, $L(A_1) \neq L(A_2)$.

Recall that if $A_1 = (\Sigma, Q^1, q_0^1, \delta^1, F^1), A_2 = (\Sigma, Q^2, q_0^2, \delta^2, F^2)$, then $A_3 = (\Sigma, Q^1 \times Q^2, (q_0^1, q_0^2), \delta^1 \times \delta^2, \{(q,p) : q \in F_1, p \notin F_2\})$. The construction of A_4 is similar.

Alternatively, we could check all strings of length at most $|Q^1| \cdot |Q^2|$. If the machines agree on all of those strings, then they will always agree and are therefore equivalent machines.

8. **Given a DFA A, decide whether $L(A)$ is infinite.**

   A possible algorithm for this is to check the graph, looking for cycles that are reachable from q_0 and can eventually reach an accepting state.

   Claim: let $A = (\Sigma, Q, q_0, \delta, F)$. $L(A)$ is finite if and only if it contains no w of length $|Q| < |w| \leq 2|Q|$.

   The Pumping Lemma could be used to prove this claim.

5 Context-Free Grammars

← October 15, 2013

There are some languages that are more complex than regular languages. Let's look at a few examples of such languages.

Example 5.1. Let $\Sigma = \{a, b, +, (,)\}$. Let $L = \{w \in \Sigma^\star : w$ has the same number of left brackets as right brackets$\}$.

Claim: L is not a regular language. This is intuitively clear because a finite amount of memory cannot be used to count the number of left parentheses to compare to the number of right parentheses.

Proof. Apply the Pumping Lemma to expressions of the form

$$\underbrace{((((}_{n}\underbrace{))))}_{n}$$

□

Example 5.2. Let $\Sigma = \{0, 1\}$ and $L_p = \{w \in \{0,1\}^\star : w$ is a palindrome$\}$. Namely, $w = w^R$ (w is equal to w reversed). For example, "MALAYALAM" and "MADAM" are palindromes.

Claim: L_p is not a regular language.

Proof. We will apply the Pumping Lemma. Given n, consider $w = 0^n 10^n \in L_p$. Note that $|w| \geq n$ and xyz exist such that $|xy| \leq n$, so $xy = 0^k$ for some $k \leq n$. So, $xy^2z = 0^m 10^n$ for some $m > n$, so $w \notin L_p$. □

We will introduce the notion of a **context-free grammar** (**CFG**). A CFG is a set of rules of a particular form for defining languages.

Example 5.3. Let's consider L_p again – the language of all $\{0,1\}$-palindromes. A context-free grammar for this language could have the following rules:

$$S \tag{1}$$
$$S \to 0 \tag{2}$$
$$S \to 1 \tag{3}$$
$$S \to \epsilon \tag{4}$$
$$S \to 0S0 \tag{5}$$
$$S \to 1S1 \tag{6}$$

An example construction of the palindrome "011110" is as follows:

$$\begin{aligned} S &\text{ by (1)} \\ 0S0 &\text{ by (5)} \\ 01S10 &\text{ by (6)} \\ 011S110 &\text{ by (6)} \\ 011110 &\text{ by (4)} \end{aligned}$$

Claim: the set of words in $\{0,1\}^\star$ that can be obtained by starting with S and applying the rules a finite number of times equals the set of all palindromes over $\{0,1\}^\star$.

In order to prove this claim, we must prove two separate claims:

1. Every word that these rules generate is of the form wSw^R or $w0w^R$ or $w1w^R$ or ww^R, for some $w \in \{0,1\}^\star$.

   The set of all expressions that can be generated by these rules is $I(A,P)$ for $A = \{S\}$, and $P = \{S \to 0, S \to 1, S \to \epsilon, S \to 0S0, S \to 1S1\}$. Therefore, we can prove this claim by structural induction.

2. These rules will generate <u>all</u> palindromes.

   This claim could also be proven by structural induction. Let $A = \{\epsilon, 0, 1\}$, and let $P = \{\frac{\text{w}}{\text{0w0}}, \frac{\text{w}}{\text{1w1}}\}$. Then, $L_p = I(A,P)$.

5.1 Defining Context-Free Grammars

Definition. A **context-free grammar** (or **CFG**) is a tuple $G = (V, T, P, S)$, where

- V is a finite set of variables.
- T is a finite set of terminal symbols, disjoint from V.
- P is a finite set of rules. Each rule of P has the form $X \to \sigma$, where $X \in V$ and $\sigma \in (V \cup T)^\star$.
- $S \in V$ is the start symbol.

In the example of generating palindromes, $V = \{S\},\ T = \{0, 1\},\ S = S$, and $P = \{S \to \epsilon, S \to 0, S \to 1, S \to 0S0, S \to 1S1\}$.

You might be wondering why we call these languages "context-free." The rules in a CFG can only consider the variable at hand – not the context of its neighbours in a larger string. Context-free grammars are limited by requiring that a variable generates the same expression regardless of the word the variable is embedded in.

5.1.1 Defining The Languages Generated By Such Grammars

Given a CFG $G = (V, T, P, S)$, we wish to define $L(G)$, the language described by G. But first, we have a couple of preliminary technical definitions of some notation.

Definition. For a variable $X \in V$ and some $\sigma \in (V \cup T)^\star$, we say that $X \overset{\star}{\underset{G}{\Longrightarrow}} \sigma$ if σ can be obtained from X by applying a finite sequence of rules from P of G.

For instance, in the grammar for L_p, we can say $S \overset{\star}{\underset{G}{\Longrightarrow}} 0$, $S \overset{\star}{\underset{G}{\Longrightarrow}} 0S0$, $S \overset{\star}{\underset{G}{\Longrightarrow}} 01S10$, etc.

Definition. $r \overset{n}{\underset{G}{\Longrightarrow}} u$ denotes that r entails u in n steps. $r \overset{1}{\underset{G}{\Longrightarrow}} u$ if and only if $r = u$. By induction, we say $r \overset{n+1}{\underset{G}{\Longrightarrow}} u$ if and only if for some v, $r \overset{n}{\underset{G}{\Longrightarrow}} v$ and $v \overset{1}{\underset{G}{\Longrightarrow}} u$.

This leads to a reworded definition of $r \overset{\star}{\underset{G}{\Longrightarrow}} u$.

Definition. $r \overset{\star}{\underset{G}{\Longrightarrow}} u$ if and only if for some n, $r \overset{n}{\underset{G}{\Longrightarrow}} u$.

That's enough technical definitions. Let's now define the language described by a context-free grammar, G.

Definition. $L(G)$, the language described by G, is defined as

$$L(G) := \{w \in T^\star : S \overset{\star}{\underset{G}{\Longrightarrow}} w\}$$

Note that the words w included in the language $L(G)$ do not contain any variables – only terminal symbols.

Example 5.4. Let $L_< = \{0^m 1^k : m \leq k\}$. Note that $L_<$ is clearly not a regular language. So, we wish to define $G = (V, T, P, S)$ to describe $L_<$. Let's look at two different ways of defining this grammar.

Our first grammar for $L_<$ is:

- $V = \{S\}$
- $T = \{0, 1\}$
- $P = \{S \mapsto \epsilon, S \mapsto 0S1, S \mapsto S1\}$
- $S = S$

Note that including $S \mapsto 1$ in P would be redundant because we could use our last rule, $S \mapsto S1$, in conjunction with our first rule, $S \mapsto \epsilon$.

Let's now look at another grammar that describes the same language:

- $V = \{S, A, B\}$
- $T = \{0, 1\}$
- $P = \{S \mapsto \epsilon, S \mapsto ASB, A \mapsto 0, A \mapsto \epsilon, B \mapsto 1\}$
- $S = S$

With this grammar, we always generate a string in the form

$$\underbrace{AAA}_{n} S \underbrace{BBB}_{n}$$

However, some of those As will map to ϵ, effectively eliminating them.

5.2 Parse Trees

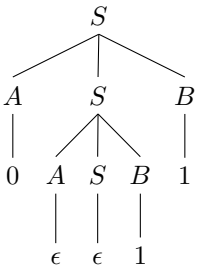
Definition. A parse tree for a grammar $G = (V, T, P, S)$ is a tree where

- Each vertex is labeled by a member of V,
- The root is labeled S, and,
- For every internal node, the set of its immediate children read (from left-to-right) σ, for σ such that $X \to \sigma \in P$, where X is the label of the parent node.

Definition. Given a parse tree G for some grammar G, the **sentential** described by T is the sequence σ of its leaf nodes, read from left-to-right.

We distinguish between sententials and words because a sentential is a sequence that *may* contain variables, and valid words cannot contain variables.

Example 5.5. Consider the previous grammar we described. Let's look at a parse tree for $w = 011$ under this grammar.



Theorem. $\sigma \in (V \cup T)^\star$ has a parse tree with respect to some grammar G if and only if $S \underset{G}{\overset{\star}{\Longrightarrow}} \sigma$.

Proof. We need to prove two claims to show that this theorem indeed holds.

1. If $S \overset{\star}{\underset{G}{\Longrightarrow}} \sigma$, then σ has a parse tree with respect to G.

   Proof. We will use structural induction on the generation of σ.

   Base case: If $\sigma = S$, then the parse tree is just "S" (with no children).

   Induction step: note that $S \to \sigma_1 \to \sigma_2 \to \ldots \to \sigma$. Assume σ was generated by applying a rule from P to a variable in σ'.

   By the induction hypothesis, there exists a parse tree T' for σ'. Since σ was generated by applying some rule $X \to \tau$ in G, we can append the sequence τ as children to the node corresponding to X in T', and the new tree, T, is a tree for σ. □

2. If $\sigma \in (V \cup T)^\star$ has a parse tree with respect to G, then $S \overset{\star}{\underset{G}{\Longrightarrow}} \sigma$.

   Proof. We can prove this claim by induction on the depth of T. (The depth of a tree is the number of nodes in the longest path from the root to a leaf.)

   This proof is left as an exercise. □

□

← October 17, 2013

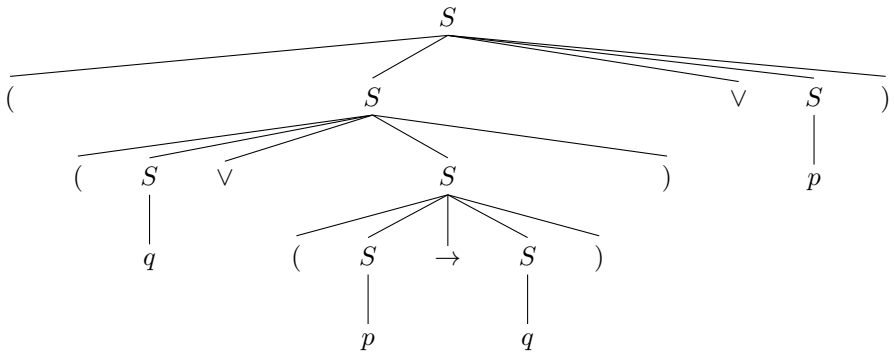
Parse trees do not concern themselves with the order of application of the rules at each step. However, they do let you see the history of how the rules were applied to each part to derive a particular string (by looking at the levels of the tree above the leaves).

Corollary. $w \in L(G)$ if and only if there exists a parse tree for G whose leaves are labelled by the symbols of w.

Example 5.6. We want a grammar to represent propositional logic over the variables p and q. For example, $p \vee q$ and $((q \vee (p \to q)) \vee p)$. We will define our grammar as $G = (V, T, P, S)$ where

- $V = \{S\}$
- $T = \{p, q, \to, \vee, \wedge, \neg, (,)\}$
- $P = \{S \vdash p,\ S \vdash q,\ S \vdash (S \to S),\ S \vdash (S \vee S),\ S \vdash (S \wedge S),\ (\neg S)\}$
- $S = S$

Let's look at a sample parse tree for $((q \vee (p \to q)) \vee p)$:



Given a grammar G, we have a way of verifying that some $w \in L(G)$, by constructing a generating sequence or a parse tree for w. We do not have a way of checking if a given w is in $L(G)$ or not. We also don't have a tool for proving that some $w \notin L(G)$. Finally, we have no way of showing that some language L is *not* context-free.

These are some languages that are context-free:

- $\{0^n 1^n : n \in \mathbb{N}\}$
- $\{0^n 10^n : n \in \mathbb{N}\}$
- $\{w : \#_1(w) = \#_0(w)\}$
- $\{w : w \text{ is of the form } w'w'^R \text{ for some } w'\}$

However, $\{w : w \text{ is of the form } w'w' \text{ for some } w' \}$ is *not* a context-free language.

5.3 Pushdown Automata

The machines corresponding to context-free languages are called **pushdown automata** (also referred to as **PDA**s). They are finite automata but with stack memory.

Intuitively, it is trivial for a pushdown automaton to compute a language of words like $w = w'w'^R$. You can form a non-deterministic machine that pushes each letter it reads onto the stack, while simultaneously guessing that the top of the stack currently represents the center character of the string. We don't know where the center of the string is, but the nondeterminism will eventually "guess" the center correctly. Before we reach the center of the string, we want to push each character we read onto the stack. After the center, we want to compare the next character of the input with the top of the stack. If they're equal, continue, otherwise lead to a blackhole state.

These machines make it obvious that a language of all words of the form $w'w'$ cannot possibly be context-free.

Creating a machine with stack memory seems arbitrary. However, it's equally arbitrary as the definition of context-free languages. Every context-free language can be computed by a pushdown automaton.

Let's discuss pushdown automata more formally.

Definition. A **pushdown automaton** (**PDA**) is a tuple $D = (\Sigma, \Gamma, Q, q_0, z_0, \delta, F)$, where

- Σ is the alphabet of the language we want to accept.
- Γ is the set of characters we could store in memory (on the stack) – and this isn't necessarily equal to Σ.
- Q is the set of states.
- q_0 is the start state.
- $z_0 \in \Gamma$ is the initial content of the stack (in order for us to be able to tell when we've reached the bottom of the stack).
- δ is a set of rules, of the form $(p, a, z) \to (q, \gamma)$, where $\gamma \in \Gamma^\star$. In particular, p is the current state, a is the character read from the input, z is the letter at the top of the stack, and q is the destination state.
- $F \subseteq Q$ is the set of accepting states.

Γ and z_0 are new to us, and the meaning of δ is different than before. This definition of δ means "if you are in state p, you see z at the top of the stack, and read input letter a, then move to state q, and replace z by γ at the top of the stack."

What is $L(D)$ for such a machine D? Intuitively, it's the set of all words w that can lead from (q_0, z_0) to some accepting state.

Definition. (p, γ, w) is an **instantaneous description** (**ID**), meaning at this point we are in state p, we have γ in the stack, and ahead of us, the input is w.

← October 22, 2013

The lecture from October 22, 2013 is omitted. If you have notes from this lecture, feel free to email me (chris@cthomson.ca) and I'll include them here.

← October 24, 2013

Example 5.7. Consider $L = \{a^n b^n : n \in \mathbb{N}\}$. The context-free grammar for L would have rules $S \to aSb$ and $S \to \epsilon$.

The corresponding machine that could accept this language L is a pushdown automaton with these transitions:

$$\delta(q_0, a, z_0) = (q_1, A)$$
$$\delta(q_1, a, A) = (q_1, AA)$$

We aim to show three things:

- The equivalence of a context-free grammar (CFG) to a pushdown automaton (PDA).
- A Pumping Lemma for context-free languages.
- Closure properties of context-free languages.

Last time, we considered two ways of defining the language of a pushdown automaton:

- $L_E(D)$, with the emptying of the stack leading to acceptance.
- $L_f(D)$, with finite acceptance states.

Last time, we showed that for every PDA D_1, there exists a PDA D_2 such that $L_f(D_1) = L_E(D_2)$.

Claim: for every PDA D_1, there exists a PDA D_2 such that $L_E(D_1) = L_f(D_2)$.

Proof. Given $D_1 = (\Sigma, \Gamma, Q, q_0, z_0, \delta, F)$, we wish to define D_2 such that for every $w \in \Sigma^\star$, D_1 empties the stack on w if and only if D_2 on w reaches a final state.

Let $D_2 = (\Sigma, \Gamma \cup \{z^\star\}, Q \cup \{q_0', q_F\}, \delta', \{q_F\})$. The new start state will be q_0'. We will define a transition with our new transition function δ':

$$\delta'(q_0, \epsilon, z_0) = \delta(q_0, z_0, z^\star)$$

Now, for every rule in D_1, $\delta(q, a, z) = \{(p, \gamma), \ldots\}$, we copy the rule into D_2. We also define another rule:

$$\delta'(p, \epsilon, z^\star) = (q_F, \epsilon)$$

The intuition here is when the old machine empties the stack, it will push $z^\star$ onto the stack of D_2. Then, whenever we see $z^\star$, we accept.

If P_1 was the set of instructions of D_1, the set of instructions for D_2 will be

$$P_2 = P_1 \cup \{\delta'(q_0', \epsilon, z_0) = (q_0, z_0 z^\star), \delta'(p, \epsilon, z^\star) = (q_F, \epsilon)\ \forall p \in Q\}$$

□

Theorem. For every language L, there exists a PDA D such that $L_E(D) = L$ if and only if there exists a context-free grammar G such that $L = L(G)$.

In order to prove this theorem, we need to show two directions:

- Given a grammar G, we can construct a PDA D such that $L(D) = L(G)$.
- Given a PDA D, we can construct a grammar G for $L(D)$.

Proof. Recall that a context-free grammar is $G = (V, T, P, S)$ with each rule of P having the form $A \to w$ for some $w \in (V \cup T)^\star$.

$$L(G) = \{w \in T^\star : s \overset{\star}{\underset{G}{\Longrightarrow}} w\}$$

How should we construct a PDA for $L(G)$? Essentially, we want to replace a variable A by w on the stack.

Construct $D_G = (\Sigma, \Gamma, Q, \delta, q_0, z_0)$, where

- $\Sigma = T_G$
- $Q = \{q_0, q_1\}$

- $\Gamma = T \cup V \cup \{z_0\}$
- δ has the rules:
  - $\delta(q_0, \epsilon, z_0) = \{(q, S)\}$
  - $\delta(q_1, \epsilon, A) = \{(q, w) : A \to w \in P_G\}$
  - $\delta(q, a, a) = (q, \epsilon)$ for every $a \in \Sigma$

We have now proven one direction of this theorem. We will accept, without proof, the claim that for every PDA, there is a grammar for the same language. □

5.4 Ogden's Lemma: Pumping Lemma for Context-Free Languages

We need a tool for showing that a given language is *not* context-free. First, we will state a pumping lemma for such languages, followed by a proof of the lemma. After, we will apply it to show that some languages are not context-free.

Lemma (Ogden's Lemma)**.** For every context-free language L, there exists some $n \in \mathbb{N}$ such that for every $w \in L$, if $|w| \geq n$, then there exists a decomposition $w = X_1X_2X_3X_4X_5$ such that

- $|X_2X_3X_4| \leq n$,
- $X_2X_4 \neq \epsilon$ (at least one of X_2, X_4 is not empty), and,
- For every k, $X_1X_2^kX_3X_4^kX_5 \in L$

Let's look at an application of this new Pumping Lemma to show that a language is not context-free.

Example 5.8. Consider $L = \{a^nb^nc^n : n \in \mathbb{N}\}$.

Claim: L is not context-free. Intuitively, a^nb^n would be context-free (since we could use a stack to keep track of n), but that is not possible for $n > 2$.

Proof. Let us show that L fails the new Pumping Lemma.

Given n as the lemma requires, pick $w = a^nb^nc^n$ (for the same n). Clearly, $|w| \geq n$, and $w \in L$.

Now, the first player decomposes my w as

$$a^nb^nc^n = X_1X_2X_3X_4X_5$$

such that the three decomposition requirements are satisfied.

By the requirement that $|X_2X_3X_4| \leq n$, $X_2X_3X_4$ can either contain only a's and b's, or only b's and c's. In particular, $X_2X_3X_4$ can contain one or two different letters, but not a and c at the same time.

Regardless of what the decomposition was, pick any $k \neq 1$. Consider $X_1X_2^kX_3X_4^kX_5$ compared to $X_1X_2X_3X_4X_5 = a^nb^nc^n$. k has changed the number of occurrences of at least one

letter and at most two letters (the letters in X_2 and X_4).

Therefore, $X_1X_2^kX_3X_4^kX_5$ no longer has the same number of occurrences of a, b, and c. So, $X_1X_2^kX_3X_4^kX_5 \notin L$. L fails the Pumping Lemma, which shows that L is not context-free. □

← October 29, 2013

Example 5.9. Let $L_{\text{exp}} = \{a^{(2^n)} : n \in \mathbb{N}\}$. In particular, $L_{\text{exp}} = \{a, aa, \underbrace{aa\ldots a}_{8}, \underbrace{aa\ldots a}_{16}, \ldots\}$.

(This is what we call a **unary language** – a language over one letter.)

Claim: L_{exp} is not context-free.

Proof. We'll show that L_{exp} violates the Pumping Lemma for context-free languages.

We're given some n, we pick $w = a^{2^n} = \overbrace{a\ldots a}^{2^n} \in L$, and given any legal decomposition (satisfying the first two conditions of the Pumping Lemma), $w = X_1X_2X_3X_4X_5$.

Pick $k = 2$. Let's examine $X_1X_2^2X_3X_4^2X_5 = a^m$ for some m. First, note that $m = 2^n + |X_2X_4| > 2^n$ since $|X_2X_4| \geq 1$. Also, $m < 2^n + n$. This implies:

$$2^n < m \leq 2^n + n \leq 2^n + 2^n = 2^{n+1}$$

Therefore, m is not a power of two, and consequently, $X_1X_2^2X_3X_4^2X_5 \notin L$. □

We could follow a similar argument with the Pumping Lemma for regular languages to show that L_{exp} is not regular either.

Example 5.10. Let $\Sigma = \{a, b\}$. Earlier, we saw that $L_p = \{ww^R : w \in \Sigma^\star\}$ is context-free, but not regular. Now, let's take a look at a similar language, L_{double}, which is not context-free or regular.

Claim: $L_{\text{double}} = \{ww : w \in \Sigma^\star\}$ is not context-free.

Intuitively, L_p and L_{double} should be equally complex languages to compute. However, due to the somewhat arbitrary definition of context-free languages and pushdown automata, we cannot compute L_{double}, but we can compute L_p.

Proof. Let us show that L_{double} violates the Pumping Lemma for context-free languages.

Given some n, pick $w = a^nb^na^nb^n \in L_{\text{double}}$. For any decomposition $w = X_1X_2X_3X_4X_5$, if $|X_2X_3X_4| \leq n$, this part intersects at most two of the four blocks, so $X_2X_3X_4 = a^ib^j$ or a^i or b^ia^j or b^i.

Pick $k = 2$. In any case, $X_1X_2^2X_3X_4^2X_5$ is not of the form $w'w'$, so it is not in L_{double}. □

The preceding proof is rough. A complete proof would go over the four cases one by one.

We want to prove the Pumping Lemma for context-free languages (Ogden's Lemma). But first, we have to introduce a number of additional lemmas.

Lemma (Noam Chomsky's Generative Grammar)**.** Every context-free language can be generated by a grammar of the following restricted form. Every generation rule in the grammar is of one of these forms:

- $A \to BC$ for some $A, B, C \in V$
- $A \to a$ for some $A \in V$ and $a \in T$
- $A \to \epsilon$ for some $A \in V$

Let's think about the intuition behind this lemma. An arbitrary generation rule looks similar to $A \to abAaBCd \in (V \cup T)^\star$. How do we get from this arbitrary form to the restricted Chomsky form?

Analyzing the rule from left to right, we would introduce the rule $A_1 \to a$ and replace the arbitrary rule with $A \to A_1 bAaBCd$. Then, we would replace this rule (again) by $A \to A_1 A_2$ and introduce $A_2 \to bAaBCd$. You continue to make these replacements until all rules are legitimate rules in Chomsky form.

Lemma. Let G be a grammar satisfying the Chomsky restrictions. For every $w \in L(G)$ and every generating G-tree T for w, T has a branch of length at least $\log_2(|w|)$.

This lemma could be restated and proven combinatorially.

Lemma. Let T be a tree in which each internal node has at most two children. Then, the number of leaves in T is at most $2^{\text{depth}(T)}$, where depth(T) is the number of nodes in the longest root-to-leaf path.

Proof. We will prove this lemma by induction on the depth of T.

Base case: If depth$(T) = 1$, then the number of leaves is $1 \leq 2$.

Inductive step: assume the claim holds for depth$(T) = n$. Consider a tree T of depth $n + 1$, with child trees T_1 and T_2.

Applying the inductive hypothesis, |leaves of T| = |leaves of T_1| + |leaves of T_2| $\leq 2^n + 2^n \leq 2^{n+1}$. □

We now have the tools we need to prove Ogden's Lemma (the Pumping Lemma for context-free languages).

Proof. Let L be any context-free language and let $G = (V, T, P, S)$ be a Chomsky-type grammar such that L is the language of G ($L = L(G)$).

Pick $n = 2^{|V|+2}$. Let $w \in L$ be of length $|w| \geq n$. Since $w \in L(G)$, there is some G-tree that generates w.

Since depth$(T) \geq \log_2(|w|)$ (by the second lemma) and $|w| \geq n = 2^{|V|+2}$, depth$(T) > |V|+1$.

Therefore, for some branch (root-to-leaf) in the tree, some $A \in V$ occurs at least twice along the branch. Notice:

$$A \underset{G}{\overset{\star}{\Longrightarrow}} X_2X_3X_4 \text{ and } A \underset{G}{\overset{\star}{\Longrightarrow}} X_2AX_4 \text{ and } A \underset{G}{\overset{\star}{\Longrightarrow}} X_2X_2AX_4X_4 \text{ etc.}$$

So, for every k, $A \overset{\star}{\underset{G}{\Longrightarrow}} X_2^k A X_4^k$, and $A \overset{\star}{\underset{G}{\Longrightarrow}} X_3$. Therefore, $S \overset{\star}{\underset{G}{\Longrightarrow}} X_1 X_2^k X_3 X_4^k X_5$ for all k.

We will conclude this proof once we establish that indeed $|X_2 X_4| \geq 1$ and $|X_2 X_3 X_4| \leq n$. We establish this by picking the lowest repetition of a variable along the longest branch in T. □

5.5 Deciding Membership in a Context-Free Language

← October 31, 2013

Recall: for a regular language L, there is a corresponding DFA D, and to check if a particular word w is in L or not, we simply run w through D and see if D accepts w.

Can we repeat this idea for a context-free language? Not really. The problem is that pushdown automata are inherently non-deterministic. There is no clear way to simulate the run of a non-deterministic pushdown automaton. Unlike finite automata, there is a potentially unbounded number of possible states for the stack, due to the PDA's non-determinism.

We can still find a way to decide membership in context-free languages though! We can do this by using a grammar for L, rather than using an automaton for L.

5.5.1 Method 1: Brute Force

Assume, without loss of generality, that G is in Chomsky form.

Each $w \in L$ has a parse tree, which is a binary tree, and all its internal nodes are labeled by variables in V. The number of internal nodes in a binary tree with n leaves is $\leq 2n - 1$.

Decision algorithm: list all binary trees with at most $2|w| - 1$ internal nodes, and check if any of them is a legal G-parse tree for w. There are at most $|V|^{2|w|-1}$ trees to check.

The downside to this approach is that's a lot of trees to check. For example, if we have just three variables and the word is of length 10, we would need to check up to $3^{19} = 1,162,261,467$ trees. Luckily, there are more efficient algorithms.

5.5.2 Method 2: Tableau Method

Let $w = a_1 \dots a_n$. We're going to build an $n \times n$ table. Let i represent the column number and j represent the row number.

Filling out the table: in each X_{ij} entry, for $i \leq j$, we aim to have all variables $A \in V$ (that is, a finite set of variables) such that $A \overset{\star}{\underset{G}{\Longrightarrow}} a_i \dots a_j$. Then, $w \in L(G)$ if and only if $S \in X_{1n}$.

Bottom row: for each X_{ii}, we need to find all $A \in V$ such that $A \overset{\star}{\underset{G}{\Longrightarrow}} a_i$.

Higher rows: assume the i-th row has been filled in already. Given any X_{ij} in the $i+1$-th row, find all As such that $A \overset{\star}{\underset{G}{\Longrightarrow}} \underbrace{a_k \dots a_j}_{i+1}$. For every l such that $k \leq l \leq j$, check all

combinations A, B, C such that $A \to BC \in P$, $B \in X_{kl}$ and $C \in X_{(l+1)j}$. (We are looking for all ways to cut a string $a_k \dots a_j$ in two.)

This algorithm runs in $\frac{n^2}{2} \times n = \theta(n^3)$ time.

5.6 Closure Properties of Context-Free Languages

For regular languages, we could use almost any operation on a language (or multiple regular languages) and the result would still be regular. Is the story similar for context-free languages?

Assume L_1 and L_2 are context-free languages. Is it guaranteed that so are $L_1 \cup L_2, L_1 \cap L_2, L_1^\star, L_1 \backslash L_2$, $L_1 L_2$, etc?

5.6.1 Non-closure of Context-Free Languages Under Intersection

Claim: context-free languages are not closed under the intersection operation.

Proof. Let $L_1 = \{a^n b^n c^m : n, m \in \mathbb{N}\}$. L_1 is context-free because we can come up with the following context-free grammar for L:

- $S \to aS'bA$
- $S \to \epsilon$
- $S' \to aS'b$
- $A \to cA$

It's also easy to see that a pushdown automaton would exist that can compute L_1. We would simply push onto the stack for every a, and pop for every b.

Let $L_2 = \{a^m b^n c^n : n, m \in \mathbb{N}\}$. This is also clearly a context-free language.

What is $L_1 \cap L_2$? $L_1 \cap L_2 = \{a^n b^n c^n : n \in \mathbb{N}\}$, which is not context-free. This serves as a counterexample. □

What went wrong with the argument we used to show closure of intersection for regular languages? For regular languages, we used the product automaton to run both DFAs in parallel. If both accepted a particular word, then the product automaton would also accept the word.

Each state in our product DFA was a set of two states – one for each machine. However, for context-free languages, we cannot combine our stacks in a similar way. We can't simply simulate a double stack with one stack, the same way we simulated two states with one.

Claim: if L_1 is a context-free language and L_2 is regular, then $L_1 \cap L_2$ is context-free.

This claim could be proven by building a product automaton containing a PDA for L_1 and a DFA for L_2. This is possible since only one stack is needed (for L_2).

5.7 Substitution of Languages

Given some $w = a_1 \dots a_n \in \Sigma^\star$, and languages $L_{a_1} \dots L_{a_n}$, let $L(w)$ denote $L_{a_1} L_{a_2} \dots L_{a_n}$.

Given a language L_a for every $a \in \Sigma$, and some language $R \subseteq \Sigma^\star$, let $L(R) = \bigcup_{w \in R} L(w)$.

Example 5.11. Let $L_1 = a^\star$, $L_0 = b^\star$, $R = \{0, 1, 01\}$.

We say $L(R) = a^\star \cup b^\star \cup a^\star b^\star = \{w : w = a^i b^j \text{ for some } i, j\}$.

Claim: for every context-free R and $L(A)$ for all $a \in \Sigma$, the language $L(R)$ is also context-free.

Proof. Let $G = (V, T, P, S)$ be a grammar for R, and for each a, let $G_a = (V_a, T_a, P_a, S_a)$ be a grammar for L_a.

We define a new set of variables $\{S_a : a \in \Sigma\}$. Replace every rule of a form similar to $A \to V_1 V_2 a_1 a_2 a_3 V_3 \dots$ by a corresponding rule similar to $A \to V_1 V_2 S_{a_1} S_{a_2} S_{a_3} V_3 \dots$. Finally, add in all the rules from G_a's with S_a replacing S. □

6 Turing Machines

← November 12, 2013

Thesis (Church Thesis)**.** Every task that can be computed can be computed by a Turing machine.

We call this a thesis rather than a theorem because it's not particularly clear what it means for a task to be "computed." This thesis is more of a philosophical statement than it is a factual, concrete theorem.

The intent behind this thesis is that any task that can be carried out by a computer (any task you can write a program for) can be carried out by a Turing machine.

Turing machines were introduced by Alan Turing in 1936 and have not changed since. Even if we consider futuristic models of computing, such as quantum computing, we do not need to look at machines any more complex than Turing machines.

6.1 Defining Turing Machines

Intuitively, the Turing machine is an extension to what we have seen before. In particular, Turing machines are similar to pushdown automata but the memory of Turing machines looks a bit different. Like pushdown automata and finite automata, Turing machines are purely theoretical constructs.

A Turing machine has a finite set of states and a memory in the form of a one-sided tape, with a read head. The read head can read or write any arbitrary position in memory, but can only be moved left or right in sequence.

Definition. A **Turing machine** is defined as $T = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where

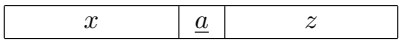
- Q is a finite set of states.

- Σ is a finite set of characters, called the input alphabet.
- Γ is a finite set of characters, called the tape alphabet. $\Sigma \subseteq \Gamma$ and conventionally there also exists some special symbol $\Delta \in \Gamma \backslash \Sigma$. This special symbol is useful for various purposes, such as denoting the beginning, end, or middle of a string – it depends on the use case. Other special symbols are also permitted.
- $q_0 \in Q$ is the initial state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, defined by $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$, where $\{L, R, S\}$ are possible moves for the tape head (move left, move right, or stay).

Recall that the definition of a pushdown automaton was arbitrary in some ways. For instance, why could the language of palindromes be computed by a PDA but not the language of all strings containing the same word twice in a row? We could've changed many things about the definition of pushdown automata, but any of those changes would change the types of languages that PDAs could accept. That is not the case here. Making arbitrary changes to the definition of Turing machines, such as making the tape two-sided or adding multiple tapes, would not change the languages computable by Turing machines. We'll look into this in more detail later.

Definition. The **instantaneous description** (or **ID** or **configuration**) of a Turing machine is $(q, x\underline{a}z)$, where

- q is the current state.
- $\underline{a}$ is the character under the reading head, as indicated by an underline.
- x is the string to the left of a on the tape.
- z is the string to the right of a on the tape.



Next, we're interested in defining the "legal next ID" relation.

Definition. $(q, x\underline{a}z) \underset{T}{\vdash} (p, y\underline{b}w)$ if some instruction in δ caused the move from the first ID, $(q, x\underline{a}z)$, to the second, $(p, y\underline{b}w)$.

For example, if $\delta(q, a) = (p, b, L)$, then $(q, x\underline{a}z) \underset{T}{\vdash} (p, x'\underline{d}bz)$ where d is the rightmost character in x and $x = x'd$.

Next, we want to extend our "legal next ID" relation to be an accessibility relation.

Definition. $(q, x\underline{a}z) \overset{\star}{\underset{T}{\vdash}} (p, y\underline{b}w)$. The ID on the right is said to be **accessible** from the ID on the left if there is some finite sequence of instantaneous descriptions $I_0, I_1, \ldots, I_n$ such that $I_0 = (q, x\underline{a}z)$ and for each i, $I_i \underset{T}{\Longrightarrow} I_{i+1}$, and $I_n = (p, y\underline{b}w)$.

Let's now look at how to define the language that a particular Turing machine accepts.

Definition. Given a Turing machine $T = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, we define the language it accepts as

$$L(T) = \{w \in \Sigma^\star : (q_0, \epsilon\underline{\Delta}w) \overset{\star}{\underset{T}{\vdash}} (q_a, x\underline{a}z) \text{ for any } x, z \in \Gamma^\star \text{ and } a \in \Gamma\}$$

A run of our machine will start with the tape looking like this, while in state q_0:

$\underline{\Delta}$	w

When our machine accepts, the tape will look like this, while in state q_a:

x	$\underline{a}$	z

Example 6.1. Define a Turing machine T such that $L(T)$ is the set of all palindromes over $\{a, b\}^\star$.

A Turing machine can compute this in a way that is very similar to how a human would do it. We check the first letter, go to the end of the word, and check the last letter to see if it's the same as the first letter or not.

Let's construct our Turing machine so that the tape begins with our special symbol Δ, followed by our input word w, and then followed by Δ again:

$\underline{\Delta}$	w	Δ	$\ldots$

We will now define our transitions.

$$\begin{aligned}
\delta(q_0, \Delta) &= (q_1, \Delta, R) \\
\delta(q_1, a) &= (q_2, \Delta, R) \\
\delta(q_2, a) &= (q_2, a, R) \\
\delta(q_2, b) &= (q_2, b, R) \\
\delta(q_2, \Delta) &= (q_3, \Delta, L) \\
\delta(q_3, a) &= (q_4, \Delta, L) \\
\delta(q_3, b) &= (q_r, \ldots) \\
\delta(q_4, a) &= (q_4, a, L) \\
\delta(q_4, b) &= (q_4, b, L) \\
\delta(q_4, \Delta) &= (q_1, \Delta, R) \\
\delta(q_1, b) &= (q_5, \Delta, R) \\
&\vdots = \vdots \\
\delta(q_3, \Delta) &= (q_a, \ldots) \\
\delta(q_6, \Delta) &= (q_a, \ldots)
\end{aligned}$$

The q_2 state indicates that we have just read an a on the left, and are currently searching for the right end of the string. The transitions on q_2 are moving right until we reach our end-of-string symbol, Δ.

The q_3 state indicates that the last symbol we read on the left end was an a and we have

now reached the right end of the string on the tape.

The q_4 state indicates that we are moving left, looking for the beginning of the string, indicated by running into Δ on the left.

The q_5 state indicates that we saw b on the left end and are moving to the right, looking for the end of the string. q_5 is similar to q_2. We need to define additional transitions for b using $q_5, \ldots$ that match the corresponding states q_2, q_3, q_4.

The instantaneous descriptions for states q_a and q_r do not need to specify the character to write to the tape, or the direction of the tape head's travel, since the computation has finished.

Example 6.2. Construct a Turing machine T such that $L(T) = \{ww : w \in \{a, b\}^\star\}$. Recall that this language is not computable by a pushdown automaton.

T must use some sort of trick to mark the middle of the string. We could swap each letter in the string, starting at the two ends, with its uppercase counterpart. We know we reach the middle when we have just changed a letter to its uppercase equivalent, and then move only to immediately find another uppercase letter.

Once we locate the middle character, we need to mark it in some way. Note that we can't arbitrarily insert a *new* element into the tape without losing any data. We could move to some set of states that collectively indicate where the middle point is in the string. Alternatively, we could leave the middle character in a different case (upper/lowercase) than the rest of the string. There are lots of possibilities for how we could mark the middle character.

6.2 Decidability of Turing Machines

← November 14, 2013

For $w \notin L(T)$, it could either be the case the upon reading w, T reaches the reject state q_r, or T loops indefinitely.

Definition. L is **recognizable** if there exists a Turing machine T such that $L(T) = L$.

Definition. L is **decidable** if there exists a Turing machine T such that $L(T) = L$ and T halts on every input.

Example 6.3. Let $L_{\text{exp}} = \{a^{2^n} : n \geq 0\}$. Recall that L_{exp} is not context-free, and is therefore not regular either.

Claim: L_{exp} is a decidable language.

The general idea for a Turing machine that can compute this language is to repeatedly divide the number of a's by two, and then check if the result is 1 (in which case, accept) or an odd number that is not 1 (in which case, reject).

We can use the capital letter strategy as we have discussed before, to split the input in half each time. Or, we could remove every second symbol by replacing it with some new symbol, #.

Example 6.4. Let $L_{\text{times}} = \{a^i b^j c^k : k = i \cdot j\}$.

Claim: L_{times} is decidable.

Proof. Let us give a high-level description of a Turing machine that decides this language.

First step: before we start counting the a's, b's, and c's, we should check the input w to ensure it is a member of $a^\star b^\star c^\star$. If not, we want to reject immediately.

We can check the basic format of w using only a finite automaton. We would define states q_1, q_2, etc., and transitions like $\delta(q_1, a) = (q_1, a, R)$, $\delta(q_1, b) = (q_2, b, R)$.

Next step: we want to check the counts of a's, b's, and c's.

We will mark an a by changing it to A, and then proceed to the end of the string and remove a c. When all A's are uppercase, we capitalize one b and make all A's lowercase again and re-run this algorithm. Once all b's have been replaced by B's, check to see if there are any c's left. If so, reject. Otherwise, we can accept. □

6.3 Computing Functions with Turing Machines

So far, we have used Turing machines as language recognizers. Our output was just "accept" or "reject." However, Turing machines can be readily used to compute functions.

Definition. Let $r : \Sigma^\star \to \Sigma^\star$. A Turing machine T is said to compute f if for every w in the domain of f,

$$(q_0, \epsilon \underline{\Delta} w) \overset{\star}{\underset{T}{\vdash}} (q_a, \epsilon \underline{\Delta} f(w))$$

That is, once the Turing machine T accepts the word w, it replaces w on the input tape with the result of applying the function f to w, $f(w)$.

Example 6.5. f_{double} is defined by $f_{\text{double}}(w) = ww$.

Claim: for every $w \in \Sigma^\star$, f_{double} is computable. That is, we claim that there exists some Turing machine T such that for every $w \in \Sigma^\star$, T can compute f_{double}.

We can construct a Turing machine that computes f_{double}. The machine would make the first letter in the input string uppercase, then proceed to the end of the string and write the letter in lowercase, after some delimiter #. Once all letters in the original string are uppercase, we can move the second w over by removing the #. Alternatively, we could copy the second word in uppercase and then transform it into lowercase once we're done, eliminating the need for the # symbol.

6.3.1 Computing Functions over Natural Numbers

Definition. The natural number n will be represented by the word a^n, for simplicity.

To compute a function $f : \mathbb{N}^k \to \mathbb{N}$, we construct a Turing machine that on input $a^{n_1} \# a^{n_2} \# \ldots a^{n_k}$ outputs the string $a^{f(n_1, n_2, \ldots, n_k)}$. This allows us to accept multiple natural numbers as input.

It is easy to verify that each of the following functions are computable: $f(n) = n$, $f(n, m) = n + m$, $f(n, m) = n \cdot m$, and $f(n, m) = n^m$.

We could use capital letters to represent negative numbers, if desired.

6.3.2 Computing the Composition of Computable Functions

Let T_1 be a Turing machine that computes some function f_1, and let T_2 be a Turing machine that computes some function f_2. For simplicity, we'll assume that f_1 and f_2 are both functions from $\mathbb{N} \to \mathbb{N}$. There exists a Turing machine T_3 that computes their decomposition, $f_2(f_1(n))$.

It's easy to see that once $f_1(n)$ has finished computing, it leaves itself as input for f_2. T_3 will contain both the finite states of T_1 and T_2, and without loss of generality we can assume they are distinct. Upon reaching q_a^1, we will switch to q_0^2, and run T_2.

6.3.3 The Relationship Between Computing Functions & Accepting Languages

Definition. Given a language $L \subseteq \Sigma^\star$, let $\chi(w) = 1$ if $w \in L$ or 0 otherwise.

This gives us a way to translate accepting a language into a function that produces either zero or one. Analogously, given any function $f : \Sigma^\star \to \{0, 1\}$, we can define a language $L_f = \{w : f(w) = 1\}$. So, we have a way of converting languages to functions, and a way of converting functions into languages.

Note that for every language L, $L_{\chi_L} = L$. Also, note that whenever a function $f : \Sigma^\star \to \{0, 1\}$ is computable, the language L_f is decidable.

Let T_1 be a Turing machine that computes f. Let T_2 accept w if $T_1(w) = 1$, and reject if $T_1(w) = 0$. So, T_2 decides the language L_f.

In the other direction, if L is decidable, then χ_L is computable. Importantly, note that L must be *decidable* – being recognizable is not sufficient. If T recognizes L but does not necessarily halt on every input w, it is not clear how to compute the function χ_L. Indeed, some such functions are not computable.

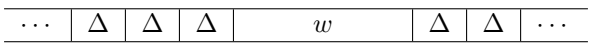
6.4 The Power of Variations of Turing Machines

← November 19, 2013

The Turing machine model of computation is robust to variations in the sense that any *reasonable* change in its definition does not effect the scope of tasks that can be computed by such machines.

6.4.1 A Machine With a Two-Sided Tape

The definition of Turing machines states that the machine's tape is one-sided, and the other side is not bounded. But what if we consider a two-sided tape which is unbounded on both ends?



The two-sided tape variation is clearly at least as potent as the first, since we can simply ignore the unboundedness of the left end. But the other direction holds as well. Namely, any computation of a machine with a two-sided tape can be simulated by a regular Turing machine with a one-sided tape.

With our one-sided tape, whenever we want to move off the left bound of the tape, we can enter a different finite state that indicates that we want to shift everything over to the right by one position. We can then perform that shift, then finally return to accessing the newly-allocated location on the left end of the tape.

We are not concerned with how wasteful this approach may seem. Above all, we care to describe any procedure that works, preferably one that makes the most logical sense, even if it is wasteful.

6.4.2 A Machine With Multiple Tapes

Suppose we modified Turing machines to support multiple tapes. These tapes could all store different information, and we would have a tape head for each tape.

We might want to use a machine like this because in some circumstances it is more natural. For example, we may want to keep the input intact on one of the tapes. We could also simulate multiple programs running in parallel with a machine like this.

We would define a multi-tape machine as $M = (\Sigma, \Gamma, Q, q_0, q_a, q_r, \delta)$ (the same as before), but where δ has a new definition:

$$\delta : Q \times \underbrace{\Gamma \times \Gamma \times \ldots \times \Gamma}_{k \text{ times}} \to Q \times \underbrace{\Gamma \times \Gamma \times \ldots \times \Gamma}_{k \text{ times}} \times \underbrace{\{L, R, S\} \times \ldots \times \{L, R, S\}}_{k \text{ times}}$$

Claim: any computation on a multi-tape machine can be carried out by a standard one-sided, one-tape Turing machine.

Proof. The idea of this proof is similar to that of the machine with the two-sided tape.

Δ	a	b	a	b	A	#	a	a	A	#	⋯	⋯

We store all of the tapes on a single tape, delimited by a special symbol such as #. Whenever we want to write into a tape, we must first shift everything to the right of that position over. To keep track of where each tape head is, we could capitalize the letter that each tape head is currently on. □

6.4.3 A Non-Deterministic Machine

Modify a Turing machine so that at each step, if you're at some state $q \in Q$ and see some input $\gamma \in \Gamma$, you have finitely many options of your next step, but you are not dictated which option to choose.

We define a machine like this as $M = (\Sigma, \Gamma, Q, q_0, q_a, q_r, \delta)$ (as before). The only difference with respect to standard, deterministic Turing machines is that now, for every state $q \in Q$

and tape letter $\gamma \in \Gamma$, $\delta(q, \gamma)$ is a finite set of "options," each a member of $Q \times \Gamma \times \{L, R, S\}$. In other words,

$$\delta : Q \times \Gamma \to \{A : A \text{ is a finite subset of } Q \times \Gamma \times \{L, R, S\}\}$$

The languages accepted by such a machine T are $L(T) = \{w :$ there is a finite sequence of legal moves leading from $(q_0, \epsilon \underline{\Delta} w)$ to some $(q_a, x\underline{y}z)\}$.

We say $(q, x\underline{y}z) \underset{T}{\vdash} (p, x'\underline{y'}z')$ if $(p, x'\underline{y'}z')$ is the result of applying one of the transition instructions in $\delta(q, y)$ to $(q, x\underline{y}z)$.

7 Turing-Recognizable Languages

Recall that we mentioned two types of tasks:

- **Decision problems**. Given some input $w \in \Sigma^\star$ and some language $L \subseteq \Sigma^\star$, decide if $w \in L$ or not.
- **Function computations**. Given some input $w \in \Sigma^\star$ and some function $f : \Sigma^\star \to \Sigma^\star$, output the string $f(w)$.

We have focused mainly on decision problems so far, and will continue to do so.

We will define the notion of a decidable language and the notion of a recognizable language in different terms than before.

Definition. L is **decidable** if there exists a Turing machine M such that on every input $w \in \Sigma^\star$, M halts, and if $w \in L$ it halts in state q_a, and if $w \notin L$, it halts in state q_r. $L(M) = L$.

Definition. L is **recognizable** if there exists a Turing machine M such that for every $w \in L$, M halts on input w in state q_a, but if $w \notin L$, M either halts in state q_r or loops indefinitely. $L(M) = L$.

There are different levels of complexity when it comes to languages:

1. Finite languages are simplest.
2. Regular languages.
3. Context-free languages.
4. Turing-decidable languages.
5. Turing-recognizable languages.
6. Other languages that are not even Turing-recognizable.

At each of these levels, there exists a language that is not included in the previous level. Our next goal is to examine Turing-recognizable languages, and languages that are not even recognizable by a Turing machine.

7.1 The Equivalence of Infinite Sets

Cantor asked: "are all infinite sets equivalent?"

Definition. Sets A and B are **equivalent** if there exists a function $f : A \to B$ that is one-to-one and onto. We denote this equivalent as $A \sim B$. We call such an f a **matching**.

We're interested in knowing if such a matching *exists* between the two sets. We don't necessarily need to exhibit a specific mapping.

Example 7.1. Let $A = \mathbb{N} = \{0, 1, 2, 3, \ldots\}$ and let $B = \{0, 2, 4, 6, \ldots\}$ be the set of even numbers.

These two sets are equivalent. We can use the mapping $n \mapsto 2n$ to show that $A \sim B$. That is, every number in the natural numbers can be mapped to its double, and since both sets are infinite, we will never run out of elements to match with.

Example 7.2. Is $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ equivalent to $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$?

Yes, these two sets are equivalent. You could map the even numbers to the positive integers in $\mathbb{Z}$, and the odd numbers with the negative integers in $\mathbb{Z}$.

In fact, it's the case that any infinite subset of $\mathbb{N}$ is equivalent to $\mathbb{N}$. It also turns out that $\mathbb{N} \sim \mathbb{Q}$ (the set of rational numbers).

Example 7.3. Let $A = (0, 1)$ and $B = (0, 100)$ be intervals. Are A and B equivalent?

Yes, these two intervals are equivalent, by the function $f(x) = 100x$.

Furthermore, both are equivalent to the full real line $\mathbb{R}$. Notice that the tangent function $tg(x)$ maps $(\frac{-\pi}{2}, \frac{+\pi}{2}) \to \mathbb{R}$. We've seen that any two intervals are equivalent, so therefore all intervals are equivalent to $\mathbb{R}$.

7.1.1 Cantor's Diagonalization Argument

← November 21, 2013

It's important to note that it is not the case that all infinite sets are equivalent to each other. In order to show this, we must show that every trick (every mapping) is bound to fail for $\mathbb{N} \to \mathbb{R}$.

Theorem (Cantor's Theorem (1905))**.** $\mathbb{N} \not\sim \mathbb{R}$.

Proof. Let f be any function from $\mathbb{N}$ to $\mathbb{R}$. We will show that f is not onto. Namely, there exists some number $r \in [0, 1]$ such that for any $n \in \mathbb{N}$, $f(n) = r$.

Without loss of generality, assume that we are only trying to cover the unit interval $[0, 1]$. (If we can't even cover $[0, 1]$ with our mapping, then we can't cover all of $\mathbb{R}$.)

Given any $f : \mathbb{N} \to [0, 1]$, construct a table for f:

$$\begin{array}{lccccccc}
f(0) = & 0 & . & \mathbf{1} & 7 & 2 & 3 & \ldots \\
f(1) = & 0 & . & 1 & \mathbf{1} & 1 & 2 & \ldots \\
f(2) = & 0 & . & 2 & 3 & \mathbf{4} & 5 & \ldots \\
f(3) = & 0 & . & 9 & 8 & 7 & \mathbf{0} & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

We want to construct a number that is not in this list. Consider the number on the diagonal of the table (bolded). In our case, it is $d = 0.1140\ldots$. Let d' be such that for every i, $d'(i) \neq d(i)$. For example, we could let $d'(i)$ be 2 when $d(i) = 1$ and 1 otherwise, so we would have $d' = 0.2211\ldots$.

Claim: d' is not in the range of f.

Assume by way of contradiction that for some $n \in \mathbb{N}$, $f(n) = d'$. Examine the n-th digit of $f(n)$, which is where the diagonal intersects $f(n)$.

By our construction of the number d, the n-th digit of $f(n)$ is $d(n)$. However, $d'(n) \neq d(n)$ by our definition of d'.

So, $d'(n)$ is not the n-th digit of $f(n)$. So, $d' \neq f(n)$. □

We could also prove Cantor's theorem without the diagonalization argument, if that confuses you.

Proof. Recall:

$$\sum_{n=2}^{\infty} \frac{1}{2^n} = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \ldots = \frac{1}{2}$$

Given any $f : \mathbb{N} \to [0, 1]$, define a sequence of intervals $I_0, I_1, I_2, \ldots, I_n$, where

$$I_n = \left[f(n) - \frac{1}{2^{n+3}}, f(n) + \frac{1}{2^{n+3}} \right]$$

I_n is an interval centered at $f(n)$, and $|I_n| = \frac{1}{2^{n+2}}$. Clearly, $\bigcup_{n=0}^{\infty} I_n$ covers the entire range of f. But:

$$\left| \bigcup_{n=0}^{\infty} I_n \right| \leq \sum_{n=0}^{\infty} \frac{1}{2^{n+2}} = \frac{1}{2}$$

In particular, we (supposedly) used length $\frac{1}{2}$ to cover the entire unit interval, which has length 1. This is not possible. □

7.1.2 Countable and Uncountable Sets

There are many classes of infinite sets. The infinite sets $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{Z}$, the set of all even numbers, etc., are **countable** sets. The infinite sets $\mathbb{R}$, $[0, 1]$, $\mathbb{R}^2$, $\mathbb{R}^n$ are **uncountable** sets.

Not only are there countable and uncountable sets, but there are many different classes of uncountable sets. The set of uncountable classes is itself an uncountable infinite set. We enumerate these classes as $\aleph_i$.

In 1960, Paul Cohen made an interesting observation:

Theorem. $\mathbb{R} = \aleph_1$ cannot be proven nor refuted.

For every finite Σ, $\Sigma^\star$ is countable. The set of all Turing machines is countable, since every Turing machine (and therefore every program) can be coded as some finite string. However, the set of all languages $\{L : L \subseteq \Sigma^\star\}$ is uncountable.

It's easy to see that the set of all languages is uncountable if you list off every word $\sigma_i \in \Sigma^\star$, then for each language, you assign "0" or "1" indicating whether that word σ_i is included in the language or not. This is a set of infinite strings, which cannot be counted.

Notice that the set of all Turing machines is countable, but the set of all languages is not. This implies that there exists some language L that is not recognizable by a Turing machine.

Example 7.4. Consider $L_{\text{verify}} = \{(P, S) : P \text{ is a C program, } S \text{ is a specification, and } P \text{ satisfies } S\}$.

Input: A program P and a specification S.

Output: does $(P, S) \in L_{\text{verify}}$?

L_{verify} is not recognizable by a Turing machine.

7.2 Undecidability: The Halting Problem

As input, you're given $\langle M \rangle$, which is a description of a Turing machine (as a string of characters), and $w \in \Sigma^\star$. The machine must decide if M halts on the input w. We could equivalently state this as problem $H = \{(\langle M \rangle, \text{w}) : M \text{ halts on input } w\}$.

Theorem. The halting problem H is not a decidable language.

In other words, there is no Turing machine that halts on every input of the form $(\langle M \rangle, w)$ and accepts if and only if $(\langle M \rangle, w) \in H$.

Proof. We will prove that H is undecidable by way of contradiction.

Assume that some Turing machine decides H. Now, define a new Turing machine D. D takes a description of a Turing machine M as input, and:

$$D(\langle M \rangle) = \begin{cases} \text{accept} & \text{if } (\langle M \rangle, \langle M \rangle) \notin H \\ \text{reject} & \text{if } (\langle M \rangle, \langle M \rangle) \in H \end{cases}$$

If H was decidable, then there would also be a Turing machine that implements D.

Let us examine what this last machine will do on the input $\langle D \rangle$. If D accepts $\langle D \rangle$, then $(\langle D \rangle, \langle D \rangle) \in H$, but then $D(\langle D \rangle)$ should reject. Similarly, if D rejects $\langle D \rangle$ then $(\langle D \rangle, \langle D \rangle) \notin H$, so $D(\langle D \rangle)$ should accept. This is a contradiction. □

We could also use a diagonalization argument to prove this theorem. The machine D constantly flips the diagonal, which similarly results in a contradiction when we consider the machine D and its intersection with the diagonal.

So, we have established one example of a concrete language (H) that is not decidable.

7.2.1 Recognizability of the Halting Problem

In order to determine if H is recognizable or not, we need a Turing machine that can simulate the run of every Turing machine M on any input w. Such a machine is called a **universal Turing machine**, and it just reads the code and runs it. This is what a regular computer does.

Construct a universal Turing machine U that simulates the run of M on w for every input $(\langle M \rangle, w)$.

Note that if $(\langle M \rangle, w) \in H$, then $U(\langle M \rangle, w)$ will accept, and if $(\langle M \rangle, w) \notin H$, then will either halt and reject, if M does so, or U will not halt. In any case, U will not accept such $(\langle M \rangle, w)$.

Observe that $L(U) = H$. Therefore, U is a recognizer for H, so H is recognizable.

Recall that we have discussed how there *are* actually languages that are not recognizable. We came to this conclusion by a counting argument – there are uncountably-many languages but only countably-many Turing machines (programs). The mapping from the set of all Turing machines to the set of all languages is a function, but it cannot be onto.

Theorem. A language L is decidable if and only if both L and its complement $\Sigma^\star \backslash L = \overline{L}$ are recognizable.

Earlier in the course, we discussed how if a language is regular, its complement was also regular. But it is not the case that if L is finite, then $\overline{L}$ is also finite.

Claim: If L is decidable, then so is $\overline{L}$.

Proof. If M accepts L and halts on every input, we can define $\overline{M}$ that accepts if and only if M rejects. So, $L(\overline{M}) = \overline{L}$.

This argument *relies* on M halting on every input, because for $\overline{M}$ to accept or reject some input, it must simulate M, wait for accept/reject, and then make the opposite decision.

To prove the other direction of the theorem, we need to show that if some M recognizes L and some $\overline{M}$ recognizes $\overline{L}$, then we can construct some $M^\star$ that decides L. That is, we can construct some $M^\star$ such that $L(M^\star) = L$ where $M^\star$ halts on every input.

$M^\star$ will run both M and $\overline{M}$ in parallel, on each input w. Since every w either is in L or is in $\overline{L}$, we have a guarantee that for every w, at least one of M or $\overline{M}$ will halt. If M halts first, $M^\star$ copies its accept/reject decision. If $\overline{M}$ halts first, $M^\star$ flips its decision. □

Corollary. $\overline{H}$ is not recognizable.

We know this corollary to be true because we know H is recognizable but not decidable. $\overline{H}$ not being recognizable is also very logical since in some ways, $\overline{H}$ is a more complex language to reason about than H, since $\overline{H} = \{(\langle M \rangle, w) : M \text{ does not accept } w\}$.

7.3 Proving Undecidability: The Reduction Technique

We can use the diagonalization trick to show that a language is undecidable. In many cases though, we can more simply use **the reduction technique**. A reduction is a proof by contradiction, which aims to show that *if* we had a black box that decides an undecidable problem, we could use it to decide another problem that we know to be undecidable (such as the halting problem), which would be a contradiction.

Example 7.5. Let $A_{\text{Halt}} = \{(\langle M \rangle, w) : M \text{ halts on } w\}$.

Claim: A_{Halt} is not decidable.

Proof. Assume by way of contradiction that some Turing machine M decides A_{Halt}, and use M to construct $M^\star$ that decides H.

We feed the input of H, $(\langle D \rangle, w)$, into M and if $(\langle D \rangle, w) \in L(M)$, then we can run w on the *actual* machine D and check that decision, since we know D will halt on input w. Otherwise, if $(\langle D \rangle, w) \notin L(M)$, we can reject immediately. □

Example 7.6. Let $E_{\text{mpt}} = \{\langle M \rangle : L(M) = \emptyset\}$.

Claim: E_{mpt} is not decidable. This is intuitive when you consider a question like "will I ever fall in love?", and we define a language of all people you will fall in love with. But clearly, the question of whether or not that set is empty is not easily answered.

Proof. We wish to construct some $M^\star$, that uses any hypothetical M that decides E_{mpt}, to solve H.

Given any input $(\langle M \rangle, w)$ for H, construct a new Turing machine M_w such that:

$$M_w(x) = \begin{cases} \text{run } M \text{ on } x & \text{if } x = w \\ \text{reject} & \text{otherwise} \end{cases}$$

Note that if $(\langle M \rangle, w) \in H$, then $L(M_w) = \{w\}$ and if $(\langle M \rangle, w) \notin H$, then $L(M_w) = \emptyset$. □

Example 7.7. Let $EQ_{\text{TM}} = \{(\langle M_1 \rangle, \langle M_2 \rangle) : L(M_1) = L(M_2)\}$.

Claim: EQ_{TM} is not decidable. Note that $\overline{EQ_{\text{TM}}}$ is recognizable, since we can just start running the machines on various inputs, then the moment they disagree on an input, we can instantly say that they are not equivalent.

Proof. We will reduce the undecidable emptiness problem to EQ_{TM}. Recall that E_{mpt} takes one machine, M, as input and accepts if the language is empty, or rejects otherwise.

Pick some simple M_0 that rejects every input. $(\langle M \rangle, \langle M_0 \rangle)$ as input into EQ_{TM}. If EQ_{TM} accepts, then $L(M)$ is empty, so we should accept. Otherwise, if EQ_{TM} rejects, then $L(M)$ is not empty, so we should reject.

We know E_{mpt} is undecidable, so this is a contradiction. Therefore, EQ_{TM} is not decidable. □

← November 28, 2013

Example 7.8. Let $\text{Regular}_{\text{TM}} = \{\langle M \rangle : L(M) \text{ is a regular language}\}$.

Claim: $\text{Regular}_{\text{TM}}$ is not decidable.

Proof. Assume by way of contradiction that some Turing machine M_{Regular} decides the language $\text{Regular}_{\text{TM}}$. We will use this hypothetical M_{Regular} to decide H.

On input $(\langle M \rangle, w)$, construct a machine M_w, defined as:

$$M_w(x) = \begin{cases} \text{accept} & \text{if } x = a^n b^n \text{ for some } n \\ \text{run } M \text{ on } W & \text{otherwise} \end{cases}$$

Note that if M accepts w, $L(M_w) = \Sigma^\star$ is regular, and we can accept this input for H. On the other hand, if M does not accept w, then $L(M_w) = \{a^n b^n : n \in \mathbb{N}\}$ is not regular, and we can reject this input for H. □

Similarly, let $\text{CF}_{\text{TM}} = \{\langle M \rangle : L(M) \text{ is a context-free language}\}$. We can use the same idea that we used for $\text{Regular}_{\text{TM}}$, except instead of using $\Sigma^\star$ and $\{a^n b^n : n \in \mathbb{N}\}$, we should use $\Sigma^\star$ and $\{a^n b^n c^n : n \in \mathbb{N}\}$, since the latter is not context-free.

7.4 Some Well-Known Undecidable Problems

We will take a brief look at some well-known undecidable problems, but we won't prove that they're undecidable.

7.4.1 Post Correspondence Problem (PCP)

Given a finite set of dominoes, with a string on the top and bottom of each, can you arrange them such that the word on the top equals the word on the bottom?

If you can use each of the dominoes exactly once, this is decidable. However, if you're allowed to use each of them as a template, meaning you can use each domino as many times as you'd like, then this is an undecidable problem. This undecidable problem is known as the **post correspondence problem** (or **PCP** for short).

7.4.2 Hilbert's 10th Problem

Given a polynomial equation, where $a_i \in \mathbb{Z}$ for all $0 \leq i \leq n$:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 = 0$$

Hilbert's 10th problem asks if there exists an integer solution to a given polynomial equation.

To determine if *any* solution (including non-integer solutions) exists, you just have to look for a point where the polynomial is positive and a point where it is negative. That's easy – and decidable. But to find an integer solution is undecidable.

For example, can you determine if there exists an integer solution to $2x^4 - 4x^3 + 3x^2 - 2x + 7$? It's not easy.

7.5 Rice's Theorem

Theorem (Rice's Theorem)**.** Every nontrivial problem about Turing machines is undecidable.

Definition. A set P of Turing machines is **nontrivial** if:

- For all M_1, M_2, if $M_1 \in P$ and $L(M_1) = L(M_2)$, then $M_2 \in P$, and
- There is some M such that $M \notin P$, and
- There is some M such that $M \in P$.

Note that we're only talking about problems involving the language that the Turing machine computes – not the Turing machine's code itself.

You cannot cite Rice's Theorem on assignments or on exams in this course. It would trivialize many questions.

7.6 Decidable Languages

We have spent a lot of time discussing undecidable languages. It's important to remember that there are many useful languages that are decidable. Let's look at two last examples of relevant languages that are decidable.

Example 7.9. Let $E_{\text{DFA}} = \{D : D \text{ is a DFA and } L(D) = \emptyset\}$. That is, E_{DFA} represents the language of deterministic finite automata that compute the empty language.

Claim: E_{DFA} is decidable.

Decision algorithm: we could perform a reachability analysis on the graph of D to see if any element of F can be reached from q_0 by following valid transitions in δ.

Alternatively, we could use the Pumping Lemma to show this.

Example 7.10. Let $E_{\text{CFG}} = \{G : G \text{ is a context-free grammar and } L(G) = \emptyset\}$.

Claim: E_{CFG} is decidable.

Decision algorithm: mark all members of T. Let M_i be the set of all members of $T \cup V$, marked at step i. Then let $M_{i+1} = M_i \cup \{A \in V : \text{some rule in } P \text{ starting with } A \text{ generates a string in } M_i^\star\}$. Repeat this process until we no longer can mark anything, at which point we check if the last marked set contains S or not.

For example, consider the following context-free grammar:

$$S \to AB$$
$$A \to aB$$
$$B \to b$$

We would have:

$$M_1 = \{a, b\}$$
$$M_2 = \{a, b\} \cup \{B\}$$
$$M_3 = \{a, b, B\} \cup \{A\}$$
$$M_4 = \{a, b, B, A\} \cup \{S\}$$