CS 154 Class Notes

David Taylor

Fall, 2021

Contents

Chapter 1

The Basics: Definitions and Notation

1.1 Strings

This class will deal extensively with strings, and sets.

We use Σ to denote the set of characters in the strings. So, for instance, we can consider strings made from $\Sigma = \{a, b, c\}$, strings made of those three characters. In this case, there are exactly three strings of length 1: a, b, and c. So, a represents either a character, or a string of length 1. In most cases, either the context should make it clear which of the two it represents, or else maybe it can be used either way.

λ is used to denote the empty string, the one-and-only string of length 0. It is nota character in Σ, it is not a character at all. It is used because, in many (most?) cases, it would be very ambiguous to just not write anything at all, and expect the reader to understand that there is a string of length 0 there. It would be confusing to write "With $\Sigma = \{a, b, c\}$, there are four strings of length at most 1: , a, b, and c." Instead, we write "...length at most 1: λ, a, b, and c." (Note: in many textbooks for this topic, they use ε instead of λ for the empty string. I use λ, because it matches the software that we will use during the course.)

For strings, concatenation can be denoted in two different ways. The string ab represents a concatenated with b. In this case, it doesn't matter too much if you consider a and b to be strings or characters: in either case, you end up with a length 2 string ab. Alternatively, the $\cdot$ symbol is used to denote concatenation: $a \cdot b = ab$. **Note:** $\cdot$ is commonly used as the product symbol, and it is reused here for good reason, some of which we will see. However, **for strings, concatenation ($\cdot$) is not commutative**. $a \cdot b = ab \neq ba = b \cdot a$. a and b are characters here, not variables, and this is concatenation, not multiplication. In thinking about strings, this should be clear: strings change if you change the order of their characters, so "dormitory" and "dirtyroom" are not equal strings. Okay, maybe that example is a bad one for some of you. "dormitory" and "dioomrrty" are not equal strings.

If we concatenate two strings, the resulting string has length equal to the sum of the lengths of the original strings. For example, $aba \cdot bbac = ababbac$, and $|aba| + |bbac| = 3 + 4 = 7 = |ababbac|$. When used around a strings, the $|\ |$ notation denotes the length (or size) of the string, the number of characters that it contains. Notice: when we concatenate λ with a non-empty string, we generally do not write λ in the result. So, $\lambda \cdot abc = abc$. **We do not write $\lambda \cdot abc = \lambda abc$, which makes it look like λ is a character from Σ.** Our length rule still holds: $|\lambda \cdot abc| = |\lambda| + |abc| = 0 + 3 = 3 = |abc|$. When we concatenate two (or more) strings? $\lambda \cdot \lambda = \lambda$. **We do not write $\lambda \cdot \lambda = \lambda\lambda$.**

1.2 Conventions

1. I will generally use alphabetically early letters of the English alphabet to symbolize characters of whatever alphabet we are considering, like $\Sigma = \{a, b, c\}$. For the class, we will rarely (never?) need to use more than 5 characters for our language alphabet.

2. I will generally use alphabetically late letter of the English alphabet, in particular u, v, w, x, y, z, as variables, usually representing a string.

3. I will generally use alphabetically middle letters of the English alphabet, in particular i, j, k, l, to symbolize integer variables.

4. I will generally use Greek, or upper case letters, to represent sets.

1.3 Basic Logic Symbols and Definitions

1. $\top$ means true. I will assume that you know what true means. I might also just write true, or T.

2. $\perp$ means false. I will assume that you know what false means. I might also just write false, or F.

3. $\vee$ is the symbol for "or", and can be read as "or". You can also use $|$. $\top \vee \top = \top \vee \perp = \perp \vee \top = \top$. $\perp \vee \perp = \perp$. (In some sources, $+$ is also used. If you interpret $\perp$ as 0, and $\top$ as 1, then $a + b = 0$ in the cases when $a \vee b = \perp$, and $a + b \neq 0$ if $a \vee b = \top$. I will avoid this notation in this class, because the $+$ symbol will be overloaded as is.)

4. $\wedge$ is the symbol for "and", and can be read as "and". You can also use &. $\top \wedge \top = \top$. $\top \wedge \perp = \perp \wedge \top = \perp \wedge \perp = \perp$. (In some sources, $\cdot$ is also used. If you interpret $\perp$ as 0, and $\top$ as 1, then $a \cdot b = 0$ in the cases when $a \wedge b = \perp$, and $a \cdot b = 1$ if $a \wedge b = \top$. I will avoid this notation, because the $\cdot$ symbol will be overloaded as is.)

5. $\Rightarrow$ is the symbol for "implies", and can be read as "implies". You can also read $x \Rightarrow y$ as "If x then y." $(\top \Rightarrow \top) = (\perp \Rightarrow \top) = (\perp \Rightarrow \perp) = \top$. $(\top \Rightarrow \perp) = \perp$. The non-intuitive rules here are the two of the form, for $x \in \{\perp, \top\}$, $\perp \Rightarrow x = \top$: if the conditional part of the if/then statement is false, then the if/then statement is true.

6. $\Leftrightarrow$ is the symbol for "if and only if" or "iff" or "means the same as". $(\top \Leftrightarrow \top) = (\perp \Leftrightarrow \perp) = \top$. $(\top \Leftrightarrow \perp) = (\perp \Leftrightarrow \top) = \perp$.

7. $\neg$ is the symbol for "not". $\neg\top = \perp$. $\neg\perp = \top$. This is a unary operator. I will also sometimes write it as a bar over a statement. (This is the set complement notation.) So, $\overline{\top} = \perp$ and $\overline{\perp} = \top$.

Notice the way these symbols are used: if I write $x \Rightarrow y$, it is implied that I am writing a true statement. (Generally, for every statement that I am writing, it is implied that it is a true statement. Otherwise, why would you take the time to read it?) So, if I write $x \Rightarrow y$, you can infer from that that the simultaneous values $x = \top$ and $y = \perp$ do not hold, because otherwise $x \Rightarrow y$ would be a false statement. Similarly, if I write $x \Leftrightarrow y$, you can infer that x and y are either both true, or both false. For a simple statement like this, $x = y$ might suffice.

1.4 Basic Set Symbols and Definitions

You should all have some intuitive idea of what a set is. We will talk about sets and their members: elements in the set. There is exactly one set with no members, the empty set, which I will write as $\emptyset$ or $\{\}$. (Here, you can really see why we need to use λ for the empty string. Otherwise, $\{\}$ would be ambiguous: is it the empty set, or a set with an empty string in it?)

Formally, we define a set recursively:

1. $\emptyset$ or $\{\}$ is the set of size 0, with no members. $\forall x, x \notin \emptyset$.
   - $\forall$ is the symbol for "for all", and it implies that we will consider each member in a set. It looks like an A, rotated 180 degrees.
   - in the above statement, $\forall x$ implies for each and every x in our set of every possible thing.
   - if we want to talk more specifically about, let's say, the set of integers, $\mathbb{Z}$, we could write $\forall x \in \mathbb{Z}, x \notin \emptyset$. We could also write $x \in \mathbb{Z} \Rightarrow x \notin \emptyset$.
2. {x} represents a set of size 1, with one member. $x \in \{x\}$. $\forall y \neq x, y \notin \{x\}$.
3. $X \cup Y$ represents union. ($\cup$ looks like the U from Union.) $\forall z, z \in X \cup Y \Leftrightarrow z \in X \vee z \in Y$. $\cup$ and $\vee$ both open to the top. We can also use $X + Y$.
4. $X \cap Y$ represents intersection. ($\cap$ doesn't look like the U from Union.) $\forall z, z \in X \cap Y \Leftrightarrow z \in X \wedge z \in Y$. $\cap$ and $\wedge$ both open to the bottom.
5. $X - Y$ represents set subtraction, items in X but not in Y. $\forall z, z \in X - Y \Leftrightarrow z \in X \wedge z \notin Y$. The symbol \ is used in some texts, but I will not use it.
6. $X \oplus Y$ represents exclusive or, or xor. $(\forall z, z \in X \oplus Y \Leftrightarrow (z \in X \wedge z \notin Y) \cup (z \notin X \wedge z \in Y)$. This is also known as the symmetric difference, and sometimes $\triangle$ or $\ominus$ is used, but I will use $\oplus$.
7. $\overline{X}$ represents the complement of X, also written as X^C or X'. It represents all elements *not* in X, and none that are. To define $\overline{X}$, it helps to know what your total universe of possible set members are. If we only care about sets of whole numbers, if X is the set of all even whole numbers, Y would be the set of all positive odd integers.
8. $|X|$ represents the *size* of set X, the number of elements in it.

Order of precedence: complement has high precedence. The four binary operators listed before it have equal precedence, consider them left to right. $\in$ has low precedence. So, $z \in X \cup Y$ states that item z is a member of the set made from the union of X and Y.

Even though they have the same order of precedence, for us, $\cup$ will generally be more important than $\cap$, $-$, or $\oplus$.: $\cup$ can be used to create larger sets, from small ones, and our base case sets ($\emptyset$ and size 1 sets) are small, so it is generally more important to be able to create larger sets from the small ones we start with.

For finite sets, rather than showing how the set is build from the base case building blocks, we can just list the set. So, if we want a set with the elements 1, 3, and 5, we can just write that set as $\{1, 3, 5\}$, instead of using the more complex recursive set notation: $(\{1\} \cup \{3\}) \cup \{5\}$. Parenthesis work as expected., although the $\cup$ operation is associative (and also commutative): $(\{1\} \cup \{3\}) \cup \{5\} = \{1\} \cup (\{3\} \cup \{5\}) = \{1\} \cup (\{5\} \cup \{3\})$.

Also:

- The order that elements are listed in a set doesn't matter. $\{1,3,5\} = \{3,5,1\} = \{5,3,1\}$.
- An element is in a set (or, and element is a member of a set), or it is not. It cannot be in the set more than once. So, $\{1,5,1,5,3\} = \{1,3,5\}$. (This wouldn't be true of multi-sets, but we are interested in sets.)
- We can also define strings by giving a rule within the set notation. $\{x : x \in \mathbb{W} \wedge x < 6\}$ would be read as "x such that x is in the whole numbers and $x < 6$." That is another way to define the set $\{0,1,2,3,4,5\}$.

To make this concrete, let $X = \{1,2,3,4\}$ and $Y = \{3,4,5,6\}$

- $X \cup Y = \{1,2,3,4,5,6\}$
- $X \cap Y = \{3,4\}$
- $X \oplus Y = X \triangle Y = X \ominus Y = \{1,2,5,6\}$
- $X - Y = A \backslash B = \{1,2\}$

1.5 Set Quantifiers

1. $\forall$, "for all", mentioned above in the $\emptyset$ discription. $\forall x \in Y, S(x)$ means that for each member in the set Y, plugging that member into S, it will be true. For example, $\forall x \in \{2,4,6,12\}$, x is even, because each and every one of those values is even
2. $\exists$, "there exists". $\exists x \in Y, S(x)$ for some statement S means that for at least one member in the set Y, plugging that member into S, it will be true. For example, $\exists x \in \{2,5,6,12,13,16\}$, x is odd, because there is at least one of the values which is odd. (In this case exactly one of the values is odd, but it can be more.)

These definitions seem straightforward, but they start to get a little bit confusing when we consider negations of logical statements, otherwise known as De Morgan's laws:

1. $\neg(x \vee y) \Leftrightarrow (\neg x \wedge \neg y)$
2. $\neg(x \wedge y) \Leftrightarrow (\neg x \vee \neg y)$
3. $\neg \forall x, S(x) \Leftrightarrow \exists x, \neg S(x)$
4. $\neg \exists x, S(x) \Leftrightarrow \forall x, \neg S(x)$

For the intuition behind, let's say, the last item, if you want to prove that it is not true that bigfoot exists, you could show that, for all things, none of them are bigfoot. Or, for the second to last item: if you want to prove that it is not true that every pumpkin weighs less than 1200 lbs, it would suffice to show that there exists one pumpkin that weighs at least 1200 lbs.

Notice, for *any* statement S, the following hold:

- "$\exists x \in \emptyset$ such that $S(x)$" is false. (It is vacuously false.) In order for "$\exists x \in X$ such that $S(x)$" to be true, you need to have two things: first, for some x, $S(x)$ must be true. Second, $x \in X$ for that value that makes $S(x)$ true. Clearly, the second part of this cannot be true, so the statement is false.

- "$\forall x \in \emptyset$, $S(x)$" is true. (It is vacuously true.) Using the third of the De Morgan's laws above, what would it mean for "$\forall x \in \emptyset$ such that S" to be false? It would mean that "$\exists x \in \emptyset$ such that $\neg S(x)$ is true. But we know from the previous item that "$\exists x \in \emptyset$ such that *any statement you want to put here*" is false, so its negation (our original statement) is true.

Notice, in the above, $S(x)$ can be any statement that evaluates to true or false. So, if our domain is integers, $S(x)$ could be "x is odd", or it could be the statement "x is *not* odd (x is even). The statement "x is odd" is true about some integers, but false about other integers. However, the statement "$\exists x \in \emptyset$ such that x is odd" is false, and the statement "$\forall x \in \emptyset, x$ is odd" is a true.

1.5.1 Quantifier Confusion

On their own, quantifiers seem relatively straightforward, but they get more confusing if you start nesting quantifiers. For example: $\neg\forall x\exists y\forall z, S(x,y,z) \Leftrightarrow \exists x\neg\exists y\forall z, S(x,y,z) \Leftrightarrow \exists x\forall y\neg\forall z, S(x,y,z) \Leftrightarrow \exists x\forall y\exists z, \neg S(x,y,z$
They get even worse when you start trying to negate them, as you might during proof by contradiction.

Consider the following simple theorem, stated without proof: Given a set S such that $|S| \geq 4$ and the members of S are all positive integers, $\exists x \in X$ such that $x > \pi$. How could we use that theorem, constructively?

One way is that, if the conditions of the theorem are met ($|S| \geq 4$ and S members are all positive integers), you can assume that at least one of those integers $> \pi$. Another is that you can show, for any set without a member with value $> \pi$, it must not be that $|S| \geq 4$ *and* $|S|$ members are all positive integers. So, for $S = \{-15, -8, -2, 0, 2\}$, the members are not all positive. For $S = \{1, 1.1, 1.2, 1.3, 1.4, 1.5, 2\}$, the members are not all integers. For $S = \{1, 2, 3\}$, $|S| < 4$. In all three of these cases, because the conclusion of the theorem doesn't hold $\nexists x \in S$ such that $x > \pi$, we know that at least one of the requirements of the theorem must not hold.

Consider next an incorrect application of this logic. Consider the set $S = \{1, 2, 3, 47\}$. $1 \in S$, and $1 \not> \pi$. Similarly, $2 \in S$, and $2 \not> \pi$. And, $3 \in S$, and $3 \not> \pi$. Can we now conclude that at least one of $|S| \geq 4$ or S members are all positive integers must be false? Of course not. The theorem says only one member $x \in S$ such that $x > \pi$. Showing that other members from S are $\not> \pi$ does not negate the fact that one of them, 47, is.

In this context, with this theorem, those results may seem obvious. We will revisit this later in the semester, in a less familiar context, and the results will be... less obvious.

1.6 Infinite Set Sizes[1]

Are sets $\{a, b, 7, \emptyset\}$ vs {1, 2, 3, 4} the same size? Yes, each has size 4. But what if we don't know how to count more than 1 item? Well, we could...cross off one member from each set, and if they both run out at the same time, they are the same size.

How about the set of all primes between 17 and 10^{12} vs the squares of those same primes? Hey, we don't really need to count those, or cross members off. It is pretty easy to notice that each member from the first set, squared, gives a different member from the second set. If you can pair members from each set up with a member from the other set, the sets are the same size.

How about the set of all non-negative even integers vs. the set of all non-negative odd integers? Again, let's pair numbers from the first set up with numbers from the second set, by adding 1 to each number in the first set. {0, 2, 4, 6...} + 1 gives {1, 3, 5, 7...}

[1]We will return to this material while covering Turing Machines, it will not be needed during the first part of the class. It was covered pretty nicely in that first Veritasium video anyway.

How about the set of all non-negative even integers vs. the set of all non-negative integers? {0, 2, 4, 6, 8...} vs {0, 1, 2, 3, 4, 5..}? **This is probably the first one where our intuition fails us.** Our intuition tells us: if a set is a struct subset of another, the subset is smaller. But, our intuition is build on finite sized sets. In our everyday experiences, it is rare to come across infinite sized sets, and intuition built there serves us poorly for infinite sized sets. Using the same argument as before, $\frac{1}{2}\cdot$\{0, 2, 4, 6, 8...\} vs {0, 1, 2, 3, 4, 5..}. The sets are the same size.

These sets (all even integers, all odd integers, all integers, all non-negative integers, and all even non-negative integers) are all known as countably infinite sets. Rather than mapping the sets to each other, we can map each to a standard set: the counting numbers. If you can take an infinite set and order it so that you know which item comes first, second, third, etc., and every item in the set gets assigned a position? The set is countably infinite.

Let's consider a tougher one: the set of pairs of positive integers. If we want to order those items, we cannot just write something like (1,1), (1,2), (1,3), . . . , (2,1), (2,2), (2,3), . . . , (3,1), . . . , ... Why? Because, while I could tell you that the pair (1, 65) is the 65th number in that list, what position does (2,1) have? Any specific pair must have a finite position in the list.

Instead, let's consider the numbers in a well-defined order that defines a position for each. In this order, I will first consider all pairs where the two numbers sum to 2, then to 3, then 4, etc. When considering pairs of a given sum, I will start with the first number at 1, and count up to the sum-1. So:

(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3), (3,2), (4,1), (1,5), ...

Here, after we have considered, $\Sigma_{i=1}^{n-1} i$ pairs, we will have considered every pair where the integers sum to n or less. So, the number (7, 13) will come in the first $\Sigma_{i=1}^{20} i = 210$ items. (I think it comes at position 198.) So, the pairs of positive integers is also countably infinite.

This also gives us a quick proof that the positive rational numbers are countable: for each reduced fraction, it will map back into a pair of positive integers. This will map all positive rationals into a (strict) subset of the positive integer pairs. The integer pairs are countably infinite, this mapping shows that the positive rationals are not a larger set than that. (The set of all rationals is also countable.)

1.6.1 Uncountable Sets

Consider the set of reals between 0 and 1. Each real can be written as an infinite string over the digits 0 to 9. We will prove that this set is not countable by contradiction: assume that it is countable, then there exists some ordering in which we can write down each and every real between 0 and 1. Fix one such ordering, with the numbers listed as $x_1, x_2, x_3 \ldots$, where each x_1 is an infinitely long number in decimal.

Next, we define a function on that ordering. Let $f(i) = \lfloor x_i \cdot 10^i \rfloor$. That is, it is the ith digit of x_i as an integer.

Define another function: $g(i) = 5$ if $i = 7$, otherwise $g(i) = 7$.

Finally, we construct a number, given the list: let $s = \sum_{i=1}^{\infty} g(f(i))/10^i$. What is special about this number? It cannot be in the ordering, anywhere. Why not? Consider that it is in position j. What is the jth digit of s? It depends on what the jth digit of x_j is. If the jth digit of x_j is 7, the jth digit of s is 5, otherwise it is 5. So, it differs from the x_jth number in the jth digit, and the two numbers cannot have the same digit at that position.

Note, there isn't very much special about the g function: it maps each digit to a different digit, but doesn't map 9 to 0 or 0 to 9. Why is that important? Real numbers ending in $\overline{9}$ or $\overline{0}$ can be written in non-unique ways, because $.\overline{999} = 1.\overline{000}$. So, if we simply say that two numbers are different in their third digit, it doesn't force them to have different values, as with the number $.956 = .955\overline{9}$. But, if that digit is off by 2, the two numbers cannot be equal.

So, for any fixed ordering of real numbers, we can construct one that is not in the ordering. The reals between 0 and 1 are not countably infinite. This is known as a *diagonalization argument*, because if you make a table for the numbers, and you can grab the digits along the diagonal to change them.

Next, consider the set of infinite binary strings. Similar to above, we can consider a diagonalization argument to construct a binary string not on any list of infinite binary strings: Given a list, construct a binary string that, as its ith bit, flips the ith bit of the ith number on the list. There is nothing surprising that this works: we can consider infinite binary strings to be binary representations of the real numbers between 0 and 1, just as we used decimal representations above, except we no longer need to worry about two different strings having "equal" numeric values, because we are explicitly asking if the strings are equal, instead of asking about the real numbers represented. Why switch to binary? If we interpret 0 as false, and 1 as true, we can consider any infinite binary string as functions, mapping all positive (or even non-negative) integers to true or false. There are more true/false functions of the counting numbers, or subsets of the counting numbers, than there are counting numbers.

1.7 Relations

In this class, we will primarily be concerned with binary relations. Given two sets (or the same set twice), a binary relation can be considered a subset of the Cartesian product symbolized by $\times$) of those sets. $\{a, b, c, \ldots y, z\} \times \{\text{vowel, consonant}\} = \{(a,\text{vowel}), (a,\text{consonant}), (b,\text{vowel}), (b,\text{consonant}), (c,\text{vowel}), \ldots$ $(y,\text{vowel}), (y,\text{consonant}), (z,\text{vowel}), (z,\text{consonant})\}$. We can specify a relation on those sets by specifying any arbitrary subset of that product. But, usually, it will have some kind of meaning. The relation $\{(a,\text{vowel}), (b,\text{consonant}), (c,\text{consonant}), \ldots (y,\text{vowel}), (y,\text{consonant}), (z,\text{consonant})\}$, specifies how each letter can be used.

More familiarly, we can consider binary relations like $<$ over the reals, or over the integers. In those cases, both sets for the relation are the same set, and instead of listing each pair for which the relation holds, we understand that, for integers x and y, $x < y$ holds exactly for those cases when integer x is smaller than integer y in value. $<, \leq, =, >, \geq, \neq$ are all very familiar relations over reals, while $\subset, \subseteq, =, \neq$ are familiar relations over sets. Notice, for these relations, both sets in the product are the same. So, we ask if one set is a subset of another set, and both arguments are sets, unlike the example from the previous paragraph. The relations we care about in this class will be like this, binary relations where each of the two sets are the same domain.

1.8 Equivalence Relations

The equality relation has certain properties. For instance, over the integers, $\mathbb{Z}$, equality is:

1. reflexive. $\forall x \in \mathbb{Z}, x = x$
2. symmetric $\forall x, y \in \mathbb{Z}, (x = y) \to (y = x)$
3. transitive $\forall x, y, z \in \mathbb{Z}, (x = y) \wedge (y = z) \to (x = z)$

Over any given domain $\mathbb{D}$, any binary relation that has these same properties is an *equivalence relation*, breaking its domain into *equivalence classes*. So, over domain $\mathbb{D}$, R is an equivalence relation if it is:

1. reflexive. $\forall x \in \mathbb{D}, xRx$
2. symmetric $\forall x, y \in \mathbb{D}, (xRy) \to (yRx)$
3. transitive $\forall x, y, z \in \mathbb{D}, (xRy) \wedge (yRz) \to (xRz)$

While equivalence classes for $=$ over $\mathbb{Z}$ are not very interesting, because each integer is in its own class, consider the following relation R over complex numbers: given real numbers a, b, c, d, let $(a+bi)R(c+di)$ if and only if $a^2+b^2 = c^2+d^2$. That is, R holds over two complex numbers if they have the same magnitude. You can check that all three properties hold, so this R is an equivalence relation. It breaks the complex numbers into equivalence classes, values of the same magnitude. That is, considering each complex number as a point in a plane, each equivalence class is made of a circle of all points, centered at the origin. All points on that circle will be an equal distance from the origin.

Besides equality itself, probably the best known example of this is to consider integers modulus a positive integer, such as 12. $27 \equiv_{\%12} 51$ or $(27\%12) = (51\%12)$. That is, if you consider all integers modulus 12, 51 and 27 are equivalent. If we do summations and products using numbers modulus 12, they are interchangeable:
$51 \cdot 18 = 918$, $918\%12 = 6$.
$(51\%12) \cdot (18\%12) = 3 \cdot 6 = 18$, $18\%12 = 6$.

Over the domain of integers, "%12" is an equivalence relation, and so it breaks that domain of integers into equivalence classes. With that relation, $3 \equiv 27 \equiv 51 \equiv -9$.

The $\leq$ relation is *not* an equivalence relation: it is reflexive and transitive, but not symmetric.

The $<$ relation is *not* an equivalence relation: it is transitive, but $a < b \to b \not< a$ (it is anti-reflexive), and it is *never* the case that $a < a$ ($a \not< a$) (it is anti-symmetric). This is a stronger statement than just saying that those relations don't always hold.

1.9 Closure

A set is closed under an operation if applying the operation to members of the set always gives another member of the set. For example:

- The integers $\mathbb{Z}$ are closed under the operations of addition, subtraction, and multiplication. They are not closed under division, because there exists members of the set, such as 7, 9, where 7/9 is not an integer.
- The set $\{0\}$ (a size 1 set, containing only the integer 0) is closed also closed under addition, subtraction, and multiplication, but not division.
- The set $\{-1, 1\}$ is closed under multiplication and division, but not addition or subtraction. (There exist set members, such as 1, 1, where $1 + 1 = 2 \notin \{-1, 1\}$.)

1.10 Induction

Through this section, all of my writing assumes that you have previously seen material for the topics being discussed, otherwise my writings are likely too brief to serve as a useful introduction. This section, in particular, assumes that you have seen induction and learned it previously. If you haven't, this section definitely lacks enough detail to describe it, and parts of it might not even be fully accurate. If you really want to learn induction well, first learn an easy induction example. Next, given any induction example, learn a more complex one.

Induction is a proof method whereby you show that some statement $S(i)$ holds for some specific value of i, known as the *base case*. Next, you assume that the statement is true for an arbitrary (variable) size (the *inductive hypothesis*), and show that that implies that the statement is true for some larger size. Once both of these statements are proven, you have shown that the statement holds for some infinite set of sizes.

Most commonly, you will prove that the base case holds for some simple example, of size 0, 1, or 2. Lets assume that you are able to prove that $S(0)$ holds. Next, you will show that $S(n) \Rightarrow S(n+1)$. What does this do for you? Well, you have alread shown that $S(1)$, and for $i = 1$, we can use the $S(n) \Rightarrow S(n+1)$ rule to show that $S(1) \Rightarrow S(2)$. But then, for $i = 2$, we can use the $S(n) \Rightarrow S(n+1)$ rule to show that $S(2) \Rightarrow S(3)$. Next, $S(3) \Rightarrow S(4)$, which leads to $S(4) \Rightarrow S(5)$...etc. For any positive integer k, if we apply our base case of $n = 1$, and then $k-1$ times, we apply the rule $S(n) \Rightarrow S(n+1)$, we prove $S(k)$. Some examples follow.

1. Prove by induction that $\sum_{i=1}^{n} i = n(n+1)/2$.

   Base case: prove it holds for $n = 1$: $\sum_{i=1}^{1} i = 1 \overset{?}{=} 1(1+1)/2 = 1$, it is true. (We could also have used $n = 0$ as a base case, assuming that we interpret $\sum_{i=1}^{0} i$ to be a summation over 0 items, giving a sum of 0.)

   Inductive Hypothesis: assume that $\sum_{i=1}^{n} i = n(n+1)/2$. Can we use that to prove that $\sum_{i=1}^{n+1} i = (n+1)((n+1)+1)/2$?

   $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + \sum_{i=n+1}^{n+1} i = (\sum_{i=1}^{n} i) + n + 1$. By the inductive hypothesis, that $= n(n+1)/2 + n + 1 = (n^2+n)/2 + (2n+2)/2 = (n^2+3n+2)/2 = (n+1)(n+2)/2 = (n+1)((n+1)+1)/2$, done.

   This proves that the statement holds for all positive integers. (If we had proven it using base case of 0, it would be proven for all non-negative integers. Or, if we had chosen a base case of 2 for some reason, the inductive step would then prove that the statement holds for all integers 2 and larger. It *is* true for 0 and 1, but the 2 base case wouldn't have *proven* it to be true.)

2. Prove by induction that for all n, $\sum_{i=1}^{n}(2i-1) = n^2$ positive integers n.

   Base case: prove for $n = 1$: $\sum_{i=1}^{1}(2i-1) = (2 \cdot 1 - 1) = 1 = 1^2$

   If it is true for a given n, does that imply it is true for $n+1$?

   Given, $\sum_{i=1}^{n}(2i-1) = n^2$. Is $\sum_{i=1}^{n+1}(2i-1) = (n+1)^2$?

   $\sum_{i=1}^{n+1}(2i-1) = \sum_{i=1}^{n}(2i-1) + 2(n+1) - 1$
   $= n^2 + 2n + 1 =$
   $(n+1)^2$. Hey, that is what we are trying to prove, great.

3. You are given a 1000 mile racetrack, and a car that gets 100 miles per gallon, with a 10 gallon tank. Unfortunately, the tank is empty. Fortunately, there are a total of 10 gallons of gas, spread across n different stops around the track. Prove that there is a starting point that can drive around then entire track without ever running out of gas.

   First, we defined some variables: Let g_i be the amount of gas at stop i, and let x_i be the distance to the next stop.

   The problem tells us that $\sum_{i=1}^{n} x_i = 1000$ miles, $\sum_{i=1}^{n} g_i = 10$ gallons.

   Base case, $n = 1$. With only 1 station, $g_1 = 10$, $x_1 = 1000$, start at g_1 and drive, good.

   Assume it works for an arbitrary $n \geq 1$, prove that that implies it works for $n+1$

   First, for $n+1$ stations there must always be at least one gas stop with enough gas to reach the next gas stop. Why? Otherwise, for each i, $100g_i < x_i$. But then, $1000 = \sum_{i=1}^{n+1} 100g_i < \sum_{i=1}^{n+1} x_i = 1000$, a contradiction. So, there exists at least one station that can reach the next station.

   Let's renumber stations to make one such station station n. No matter where you start, as long as it isn't between station n and $n+1$, you will run out of gas between station n and station $n+1$: if you get to station n with y gallons and refuel there, you will have $y + g_n$ gallons. Driving to station

$n+1$ uses $x_n/100 \leq g_n$ gallons, because station n had enough gas to make it to station $n+1$, or $100g_n \geq x_n$.

We have assumed that for any and every possible scenario with n stations, there is a solution. I am interested in one of those scenarios in particular, I will define it with g' and x' values, based on our current $n+1$ stations: For $1 \leq i \leq n-1$, $g'_i = g_i$ and $x'_i = x_i$. And, $g'_n = g_n + g_{n+1}$, and $x'_n = x_n + x_{n+1}$. The new g' and x' values sum to 10 and 1000 respectively, and there are n stations, so by our induction hypothesis, there is a starting station to make it around the track for that set of n stations. Consider that station s works as a starting station, where you begin by fueling with g'_s gallons of gas to drive the first x'_s miles. The claim is that station s will also work for the $n+1$ station scenario. Why?

For cars driving in either scenario, things look identical until you get to station n: the gas stops and distances are identical. Once we get to station n, they differ: the n station scenario has an extra g_{n+1} gas in its tank for the next x_n miles. But, the $n+1$ station car won't run out of gas during that stretch anyway. Then, once that car has driven x_n from the nth station, it gets to the $n+1$st stop, fuels g_{n+1} extra gas, and then once again exactly matches the state of the car in the n station scenario. That car will make it around to the starting point, as will this car, with identical fuel readings for the rest of their journeys. The solution for *this particular* n station scenario, that we constructed, implies a solution for the *arbitrary* $n+1$ station scenario that we were given. (We used the $n+1$ station scenario data to construct an n station scenario of interest to us, such that if it had a solution, the $n+1$ station scenario would too.)

4. Our last item was to find the flaw in the following inductive proof. Prove: consider all monochromatic horses in the world. (That is, ignore spotted horses. If a horse is basically all black, we call that a black horse. If a horse is basically all brown, we call that a brown horse.)

   In each and every possible set of horses, where each individual horse is just one color, all horses in the set are the same color as each other.

   Prove by induction: base case: in any set of 1 horses, they are all the same color.

   Induction hypothesis: assume in any set of n horses, they are all the same color. Prove that that implies it also holds for $n+1$

   You give me any set of $n+1$ horses, and I remove horse $n+1$, leaving a set of n horses, all of which are the same color by induction hypothesis.

   Similarly, if I remove only horse 1, the remaining n horses must also be the same color by induction.

   If horse a is the same color as b, and b is the same color as c, then a is the same color as c. Horse color transitivity property.

   When I removed horse $n+1$, horse 1 was left in the set, it is the same color as all other horses in the set. When I removed horse 1, horse $n+1$ was left in the set, it is the same color as all other horses in the set. By transitivity, horse 1 is the same color as the other horses in the set which are the same color as horse $n+1$, all are the same color.

   For example, if we assume it is true for all sets of 74 horses, prove true for 75. Give me 75 horses, remove horse 75, horses 1 through 74 are a set of 74 horses, all are the same color by our inductive hypothesis. Similarly, horses 2 through 75 are a set of 74 horses, all are the same color by our inductive hypothesis.

   So, all horses in the set are the same color. The only problem is that the thing we proved isn't true. We know the proof must be flawed, because it proves a false statement, but what is wrong with it.

But, knowing the proof is flawed is different than knowing what the flaw is. There is a flip side to this: if you give a proof of a statement, the statement being true does not imply that the proof is correct. So, what is wrong with the above proof?

Answer 1 *Only thing broken is that the transitivity statement assumes that $n + 1 \geq 3$, otherwise there is no "middle horse" to complete the transitivity chain. So, our inductive hypothesis implicitly assumes that $n \geq 2$. Put another way, if you could prove the statement was true with a base case of $n = 2$, the rest of the proof is correct, and it would indeed prove that there was only one color horse.*

1.11 Sets of Strings, and Radix Order

For much of this course, we will consider several different ways to define sets of strings. Before the course started, you probably previously saw most of the basic set notation covered earlier in this chapter. The remainder of the chapter will explicitly cover notation commonly used for sets of strings.

Just as I defined a possible alphabet $\Sigma = \{a, b, c\}$ by explicit listing its characters, of course we can specify a set of strings by explicitly listing the elements of the set. For example, over alphabet $\Sigma = \{a, b, c\}$, define $X = \{a, aa, ab, ac, aaa, aab, aac, aba, abb, abc, aca, acb, acc\}$. Here, X is the set of all strings, length 3 or less, which start with a. The set X has size 13, denoted by $|X| = 13$. (The size of a set and the size/length of a string both use the same notation.)

Even though sets have no order for their elements, for X, I have given the elements in an order where shorter strings are listed before longer strings, and strings of the same length are listed alphabetically. This ordering (radix ordered, or length-lexicographic order) will be the most common order used in the class, if an order is needed. The underlying set itself has no ordering, only its presentation is ordered. That presentation ordering can be useful. Consider: $\{a, aa, ab, ac\} = \{aa, ac, a, ab\} = \{aa, ab, aa, ac, a, aa\}$. That last set looks different, because one element from the set has been repeated, even though each element can only be in a set once. So, this shows three different ways to represent the same set, and that set has 4 elements. You can clearly see how unintuitive the last method is for writing the set, because the set looks like it has six elements, not four. For simplicity of presentation, try to avoid writing sets with repeated elements, because it obscures how large the set is. (There may be other reasons as well, but there will certainly be times when it is easier to allow duplicates in the presentation.)

Why bother to explain this? Because if we create a set of strings through operations on other sets, it may create some elements more than once. Any string, no matter how many times it is included into a set, makes up at most one member of the set.

1.12 Common Operations and Notation on Sets of Strings

Specifically for sets of strings, we will frequently **use the + symbol to represent union**. This isn't any different than the standard union operation, but for general sets, I would more commonly use the $\cup$ notation, whereas in this class, for sets of strings, + will be more common. Additionally, we will sometimes leave out the set notation altogether. So: $ab + ac \cdot b$ represents the set $\{ab, acb\}$. Notice here, the $\cdot$ operator has higher precedence than the + operator, which is what we are used to.

If we consider the **concatenation** of two sets of strings, that gives a new set: for each element in the first set, we concatenate it with each element of the second set, and each of those concatenations gives us an element in the resulting set. So:
$\{\lambda, a, b, ab, abb, abc\} \cdot \{bc, bbc\} = \{bc, bbc, abc, abbc, bbc, bbbc, abbc, abbbc, abbbc, abbbbc, abcbc, abcbbc\}$

$= \{bc, abc, bbc, abbc, bbbc, abbbc, abcbc, abbbbc, abcbbc\}$.
Notice, the middle way of writing that set has some repeats. They aren't particularly easy to see. Putting the set into radix order makes those duplicates easier to find. We can be guaranteed that $|X \cdot Y| \leq |X| \cdot |Y|$, where left side of the inequality concatenates the set X with the set Y, and then gets the size of that set, while on the right side of the inequality, the $\cdot$ represents multiplication of the two integer set sizes. The two sides will be equal if there are no two different ways to get any member in the result, such as $a \cdot bbc = ab \cdot bc$ here. You can interpret $\cdot$ between a set and a single string as being between two sets: $a \cdot X = \{a\} \cdot X$ and $Y \cdot b = Y \cdot \{b\}$.

For strings, we may use **exponents** as short-hand: if I want to concatenate a string 5 with itself times, instead of writing $x \cdot x \cdot x \cdot x \cdot x$, we can write x^5. So, if $x = abc$, $x \cdot x \cdot x \cdot x \cdot x = x^5 = abcabcabcabcabc$. **This is not the same thing as $aaaaabbbbbccccc$.** Repeating myself, string concatenation does not have the commutative property.

When dealing with strings, the **parenthesis** are not part of the string[2]. If I use the alphabet $\Sigma = \{a, b, c\}$, I use parenthesis to show that the exponent applies to that whole string. So, $(ab)^4cbc^3a(abc)^2a = abababab cbcccaabcabca$. (Parenthesis are assumed to also work in the previous general discussion of sets, but I wanted to explicitly point them out in the context of strings, as that is where we will be concentrating most of our time for the semester.)

The **order of precedence** for the operators is a familiar one: ( ) has the highest precedence, followed by exponents, then $\cdot$ concatenation, and then $+$ union. So, $ab + b(ca)^2b \cdot (b + c)$ represents the set $\{ab, bcacabb, bcacabc\}$. The $\in$ operator has even lower precedence, so for $x \in ab + b(ca)^2b \cdot (b + c)$, x could represent any of the strings in $\{ab, bcacabb, bcacabc\}$.

As another example, $(a + bb + ca)^2 = \{aa, abb, aca, bba, caa, bbbb, bbca, cabb, caca\}$, because any of the three strings inside of the parenthesis can be chosen as the first and/or second string repetition caused by the ^{2}. The order I have given is probably not the one that you would naturally write down on your own, but it is the radix ordering. To further explain how this example follows the notation we have already seen, $(a + bb + ca)^2 = (a + bb + ca) \cdot (a + bb + ca) = \{a, bb, ca\} \cdot \{a, bb, ca\}$.

$a(bc)^ia$ such that $3 \leq i \leq 6$ represents the set $\{abcbcbca, abcbcbcbca, abcbcbcbcbca, abcbcbcbcbcbca\}$. The exponents are simply shorthand, they give us an easier way to write those 4 strings than to put them explicitly. It is assumed that the exponent here is a non-negative integer; ab^{-2} is undefined as a string.

If, instead of a definite number of repetitions, if I want to repeat a string an arbitrary integer number of times, I will use x^*. This represents the set of strings $\{x^i : i \in \mathbb{W}\}$, or $\{x^0 = \lambda, x, x^2, x^3, \ldots\}$. (Here, $\mathbb{W}$ is the set of whole numbers, the non-negative integers. I do not have a definition for $(abc)^{-1}$.) So, for for $x = abc$, $x^* = \{(abc)^i : i \in \mathbb{W}\}$, or $\{\lambda, abc, abcabc, abcabcabc, \ldots\}$. Notice: for any string x, $x^0 = \lambda$. This operation * is known as the Kleene star operation, and like other exponents, it has high precedence.

When using this notation, when we have something that looks like a simple string, it is ambiguous: $a + b$ represents $\{a, b\}$, but does ab represent a set with a single string like $\{ab\}$, or does it just represent the string ab of length 2? The assumption is that, if it isn't clear from context, it should be specified. So, if I write $ab \cdot (a + b)$, because $(a + b)$ is a set, you should interpret ab as a set as well. The result would be $aba + abb$. But, if I were to ask the size of ab, that would be rough: am I asking for the size of the set? (Probably.) Or am I asking for the length of the string? (Probably not, I generally use length for a string.) But, I should really specify it here.

There are some **special edge cases** you should be aware of: the following logically follow from the rules we have seen so far, but are confusing enough that I want to list them explicitly:

- For any string x, $\lambda \cdot x = x \cdot \lambda = x$.

[2]An exception is when the '(' and ')' characters are part of the alphabet we are considering for our strings, in which case some care must be taken in order to distinguish which parenthesis are being used as characters, and which are being used as parenthesis.

- For any set of strings X, $\emptyset + X = X + \emptyset = X$. There are no members in $\emptyset$, adding those 0 members to another set doesn't change the other set.
- For any set of strings X, $\emptyset \cdot X = X \cdot \emptyset = \emptyset$. For each of the 0 members of $\emptyset$, we are trying to concatenate them with members of X. There are 0 ways to do that.
- For any set of strings X, $\{\lambda\} \cdot X = X \cdot \{\lambda\} = X$. There is exactly one member in $\{\lambda\}$, and it is λ. For any string x, $\lambda \cdot x = x \cdot \lambda = x$.
- $\emptyset^* = \{\lambda\}$. There are no elements to take in $\emptyset$, so it is impossible to take 1 or more strings from the set. We can still take 0 elements. Forming a string out of no elements, that is λ.

If you consider the above rules closely, it helps to explain some of the notation used for sets: concatenation has some similarities with multiplication, union with addition, $\emptyset$ with 0, and $\{\lambda\}$ with 1. (Notation doesn't give any clues for that last one.) You can see some analogous behavior:

- $\forall x \in \mathbb{Z}, 0 + x = x + 0 = x.$ $\forall$ sets of strings $X, \emptyset + X = X + \emptyset = X.$
- $\forall x \in \mathbb{Z}, 0 \cdot x = x \cdot 0 = 0.$ $\forall$ sets of strings $X, \emptyset \cdot X = X \cdot \emptyset = \emptyset.$
- $\forall x \in \mathbb{Z}, 1 \cdot x = x \cdot 1 = x.$ $\forall$ sets of strings $X, \{\lambda\} \cdot X = X \cdot \{\lambda\} = X.$

Relations like these are explored in abstract algebra classes, which would follow the Discrete Math course which is a prerequisite of CS154. (Some of you may have seen the study of groups, fields, or rings in your Math 42 or equivalent class.)

The First Exam Covers Through Here, minus the Infinite Set section.

Chapter 2

Regular Formal Languages

In Computer Science, a Formal Language is a set of strings. That is it. That is the whole definition. Different sets of rules for how we are allowed to define that set of strings allows for languages of different complexity levels. This chapter starts with several different sets of rules.

2.1 Regular Expressions

If you haven't heard of regular expressions before, they are a specific way to describe sets of strings. We recursively define *regular expressions* over alphabet Σ

- Base Case:
  - $\emptyset$ is a regular expression representing the empty set, no strings at all.
  - λ is a regular expression representing the set $\{\lambda\}$ with just one member, the 0 length string λ.
  - $\forall x \in \Sigma, x$ is a regular expression representing the set $\{x\}$, a single string of length one.
- If x, y are regular expressions, so is $x + y$. It represents the strings in x's language plus the strings in y's language (union)
- If x, y are regular expressions, so is $x \cdot y$ or xy. It represents the concatenation of x's language with y's language (concatenation): the set of strings that can be achieved by taking one string from x's language and concatenating it with a string from y's language.
- If x is a regular expression x^* is a regular expression. It represents the concatenation of as many strings from x's language as you want. (Any finite, non-negative integer).
- If x is a regular expression (x) is also a regular expression. It represents the same language as x.

Please note for these regular expression operations, the order of operations should be familiar from the notation: parenthesis are highest, Kleene star is next (exponentiation), concatenation next (multiplication), and union is lowest (addition).

Can you give me a regular expression for strings over a,b,c, starting with a, ending with a, and has an even number of b's and c's combined? (Any string starting with an a, and ending with an a, which also happens to have an even total number of b's and c's, should be in your set. aaababcbaa is good.)

2.2 Regular Grammars

There are two main types of regular grammars.

2.2.1 Right-Linear Grammars

A right-linear grammar is defined by a set of variables V, a set of terminals Σ (the alphabet), set of production rules R, and a start variable $S \in V$. I will use upper case letters to denote the variables in V, and we will literally use the variable S as the start symbol, so $S \in V$.

Each production rule is in one of two formats:

1. $A \to x$ for $A \in V$ and $x \in \Sigma^*$.
2. $A \to xB$ for $A \in V$, $B \in V$, and $x \in \Sigma^*$.

That is, each production rule has a single variable on the left of the $\to$, and on the right it has a string of variables, followed by at most one variable. If, starting with the S variable, we can use production rules from a grammar to get to a particular string of (only) terminals, that string is in the language of the grammar. The grammar *generates* the set of all strings in its language.

For example, consider the following right-linear grammar:
$V = \{S, A\}$
$\Sigma = \{a, b\}$
S is the starting variable
$R = \{S \to a,\ S \to bbS,\ S \to aA,\ S \to babA,\ A \to bbA,\ A \to aS,\ A \to babS\}$
We can see that the string *ababbba* is in the language of this grammar by the following *parsing*:
$S \to aA \to ababS \to ababbbS \to ababbba$.

The above format is not a particularly elegant one in which to give the grammar. Let's first space those rules out a little bit:
$R = \{$
$S \to a,$
$S \to bbS,$
$S \to aA,$
$S \to babA,$
$A \to bbA,$
$A \to aS,$
$A \to babS\}$
Notice, given these rules, we might infer what V and Σ are. The symbols to the left of $\to$ are each variables. Additionally, with the convention that we will use lower case letters as members of Σ, and upper case members as members of V, we can also determine which symbols to the right of the $\to$ symbol are terminals vs. variables. While it is true that there could be additional, unused members of both V and Σ, if a grammar has a variable that isn't used in any rules, it isn't useful. And, if a grammar has terminals that don't show up in any rule, those terminals will not be part of any string in the grammar's language. (We would care about them if we consider the language's complement, for which all of Σ must be known.)

Additionally, I plan to use S as our start variable by default. (In cases in which there is no S variable, my default will be that the first production rule has the start variable on the left.) With those conventions, and dropping the set notation, I might specify exactly the same grammar by just saying: "Consider a grammar with the rules:
$S \to a$
$S \to bbS$

$S \to aA$
$S \to babA$
$A \to bbA$
$A \to aS$
$A \to babS$

Finally, even that can be abbreviated. In this grammar, we have four production rules for S, and three for A. We can shorten the rule description as:
$S \to a|bbS|aA|babA$
$A \to bbA|aS|babS$
where, in this context, the | symbol means "or".

Finally: some references will use a stricter definition of right-linear grammars, allowing for at most one terminal in any production rule, rather than an arbitrary number, but notice: given a longer rule rule like:
$A \to abcB$
we could convert that into a set of equivalent rules of that restricted form:
$A \to aX_1$
$X_1 \to bX_2$
$X_2 \to cB$
where X_1 and X_2 are variables that don't appear in any other production rules. For this class, we will generally use the less restrictive definition.

2.2.2 Left-Linear Grammars

A left-linear grammar is defined identically to a right-linear grammar, except that when a production rule has a variable on the right hand side, it will be the leftmost symbol. So, each production rule is in one of two formats:

1. $A \to x$ for $A \in V$ and $x \in \Sigma^*$.
2. $A \to Bx$ for $A \in V$, $B \in V$, and $x \in \Sigma^*$.

Other than that minor difference, all other statements about right-linear grammars hold for left-linear grammars.

2.2.3 Regular Grammars

Regular grammars are right-linear or left-linear grammars. **Note:** the grammar cannot mix both right- and left-linear rules. Either (*every* production rule needs to be right-linear), or (*every* production rule needs to be left-linear). This is **not the same** as saying every production rule needs to be (right-linear or left-linear).

We will mostly deal with right linear grammars here.

2.2.4 Linear Grammars

Linear grammars are grammars in which each production rule has at most one variable on the right hand side of the $\to$ symbol. It need not be the leftmost or rightmost symbol, it can have terminals both to its left *and* to its right. Other than knowing this definition, we will not deal with these for the course, but I thought it was worth mentioning them, in case you look at other references and the term comes up.

2.3 Regular Languages

If a language can be defined by a regular grammar, we will call it a *regular language*. (Without proof (for now), right-linear and left-linear grammars can define exactly the same languages.)

Some questions to think about:

1. Are regular languages closed under union? Intersection? Concatenation?
2. Are the languages of regular expressions closed under union? Intersection? Concatenation?
3. Can regular expression be used to describe each and every possible regular language? Can the describe languages that aren't regular?

Can you prove each of your answers?

2.4 Distinguishing Extensions

Given a language L over alphabet Σ (the language does not need to be regular), two strings $x, y \in \Sigma^*$ can be *distinguished* from each other if there exists a string $z \in \Sigma^*$ such that only one of xz or yz is $\in L$. If two strings *can not* be distinguished from each other, let's call this relation R_L. So, $xR_Ly \Leftrightarrow \forall z \in \Sigma^*, (xz \in L) = (yz \in L)$.

For any language L, R_L (indistinguishability) is an equivalence relation. It is:

1. reflexive: for any string $x, \forall z \in \Sigma^*, (xz \in L) = (xz \in L)$.
2. symmetric: for any strings $x, y, (\forall z \in \Sigma^*, (xz \in L) = (yz \in L)) \Leftrightarrow (\forall z \in \Sigma^*, (yz \in L) = (xz \in L))$.
3. transitive: for any strings w, x, y:
$(\forall z \in \Sigma^*, (wz \in L) = (xz \in L)) \wedge (\forall z \in \Sigma^*, (xz \in L) = (yz \in L)) \Rightarrow (\forall z \in \Sigma^*, (wz \in L) = (yz \in L))$.

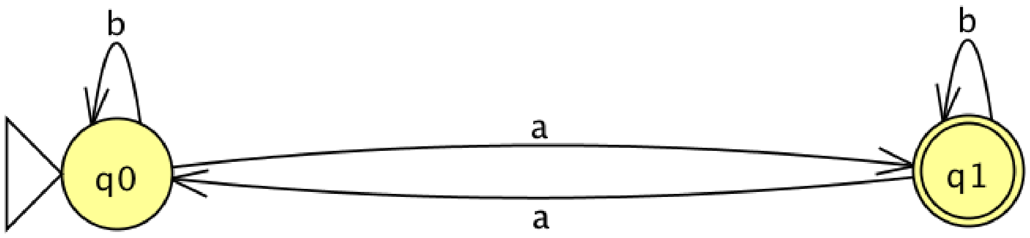
Chapter 3

Finite Automata

3.1 Deterministic Finite Automata

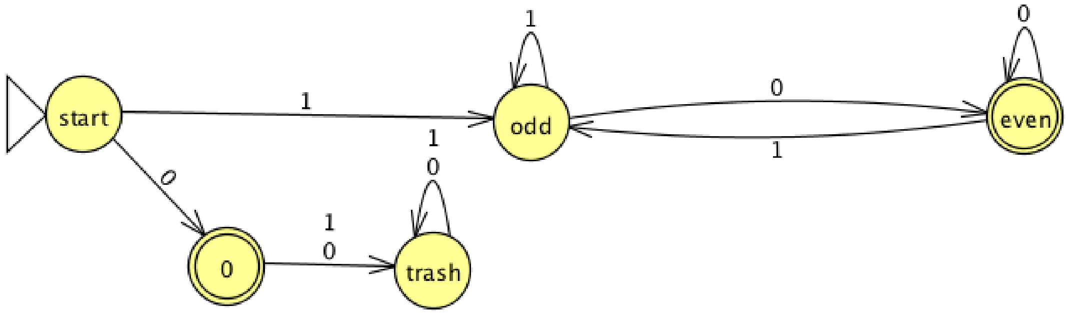
In class, we saw some examples of DFAs. Five different things are needed to describe a DFA: a set of states Q, a single start state $q \in Q$, a set of accepting or final states $F \subseteq Q$, and alphabet Σ, and a transition function $\delta : Q \times \Sigma \to Q$. Furthermore, we specified that the transition function must have exactly one transition for each $Q \times \Sigma$ pair.

1. Our first machine had $\Sigma = \{a, b\}$, and recognized strings with an odd number of as in them. We could list the machine in text form: $Q = \{q0, q1\}$, start state $q0$, $F = \{q1\}$, and $\delta(q0, a) = q1, \delta(q0, b) = q0, \delta(q1, a) = q0, \delta(q1, b) = q1$. I think it is fair to say that that text is not terribly intuitive. Instead, we draw the machine with a picture:



This is drawn with JFLAP, which can be downloaded from jflap.org. The big triangle pointing to $q0$ denotes the start state.

2. Next, we consider strings over the alphabet $\Sigma = \{0, 1\}$, where we want to accept the string if it represents an even number, in binary, most significant bit first. We eventually got to this machine:



We didn't start with that machine. We started with a 2 state machine, the states now named "even" and "odd", where "even" was the start state. However, that machine admits some strings that aren't properly formatted. For instance, it accepts λ, which isn't really a number. It also accepts strings with leading 0s, like 0010. So, we added the "start" state so that we wouldn't accept λ. We added the "0" state to accept the string 0. And, we added the "trash" state (or trap state) so that, if we start any string *other than* 0 with a 0, we are guaranteed to reject that string.

3. Finally, considering binary numbers as even or odd seems easy. How about if we take binary strings, and interpret them as binary numbers, but then ask: is the number, modulus $3 = 2$? To do it quickly, don't worry about empty strings or leading zeros, just interpret those as binary numbers anyway. In the following machine, as you process the string, you will be in state $= i$ if the string you have seen so far, interpreted as a binary number modulus 3, is i. You can check that for individual numbers: if the number you have seen so far, modulus 3, is 1, then adding a 0 at the end multiplies the value by two, and you will have remainder 2. But, if you add a 1 at the end, it multiplies by two and adds 1, leaving remainder 0.