Foundations of Computer Science
CS 3133

Pi Fisher

C Term 2015

1 Formal Languages

The alphabet Σ is the set of symbols (a, b, c, ...) that make up strings in our language. Σ^* is the set of all possible strings using Σ. A formal language is a set $L \subseteq \Sigma^*$. The empty string, denoted λ, contains no symbols, and is an element of Σ^*.

Next we give a recursive definition of Σ^*.

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

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

Closure Step: $w \in \Sigma^*$ if and only if we can get it from the basis by applying the recursive step a finite number of times.

For many of our examples, for simplicity, we will use the alphabet $\Sigma = \{a, b\}$. This alphabet can be used to make 2^k unique strings of length k. For example, the 4 strings of length 2 are aa, ab, ba, and bb. The set Σ^* contains infinitely many strings.

Now that we have strings, we can define operations on strings. We will define three operations on strings—concatenation, substring, and reversal.

Concatenation of two strings u and v will produce the string uv. If $u = abbbaa$, and if $v = bbb$, then $uv = abbbaabbb$. We say that v is a substring of u if there exist strings x and y such that $u = xvy$. With the previous definitions of u and v, we would get $x = a$ and $y = aaa$. If $x = \lambda$, we say v is a prefix, and we say

it is a suffix if $y = \lambda$. The reversal of a string is formed by reversing the order of the symbols in the string. $u^R = aabbba$, and $v^R = v$. This lets us define a special language, the palindrome language.

Palindrome Language $= \{w \in \Sigma^* | w^R = w\}$.

We next want ways to describe languages. The first way is using set notation, as we used above for the Palindrome Language. The second way is to enumerate every string in the language, but this doesn't work for infinite languages. The last way is to use operations on languages we already know. The operations we will use on languages are concatenation, powers, and closure.

If we have two languages $L_1, L_2 \in \Sigma^*$, the concatenation $L_1 L_2 = \{uv \in \Sigma^* | u \in L_1, v \in L_2\}$. When we raise languages to powers, it is equivalent to concatenating a language with itself multiple times. $L^4 = LL \oplus \otimes LL$. Note that $L^0 = \{\lambda\}$, and has size 1. The star closure of a language, or the Kleene-star closure, is the union of all the powers of a language. $L^* \coloneqq \bigcup_{n=0}^{\infty} L^n$. Similarly, The plus closure, or positive closure, is the same thing, but starting at 1 rather than at 0. $L^+ \coloneqq \bigcup_{n=1}^{\infty} L^n$.

When a language contains λ, the star closure and positive closure are equal. Also, notice that $(L^*)^* = L^*$. Now we look at some examples.

1.1 Examples

$\{a\}^* = \{\lambda, a, aa, aaa, \dots\}$ is the language of sequences of a's.
$\{a, b\}^* bb \{a, b\}^*$ is the language of strings containing bb as a substring.
$\{bb\}^*$ is the language of strings that are an even number of b's.

2 Regular Sets and Expressions

We define regular sets by recursion.

Basis: $\emptyset, \{\lambda\}, \{a\}\ \forall a \in \Sigma$

Recursive Step: If X and Y are regular sets, then $X \cup Y$, XY, and X^* are all regular sets.

Closure Step: A set X is regular if and only if we can get it from the basis by applying the recursive step a finite number of times.

Regular expressions are the same thing, but we use a shorthand. The following is a regular set:

$$\big((\{a\}\{b\})^* \cup \{b\}\{b\}\big)^*$$

The corresponding regular expression is simply:

$$\big((ab)^* \cup bb\big)^*$$

2.1 Example

$a^*b(a \cup b)^* = (a \cup b)^*ba^*$
Both of these regular expressions represent the strings which contain at least one b. Though these regular expressions look very different, they represent the same set, so they are equivalent. When two regular expressions look different but are equivalent, we call them a regular expression identity. Some more regular expression identities are listed below, followed by some possibly surprising things that are not identities. More examples are in the textbook.

1. $(u^*)^* = u^*$
2. $u(v \cup w) = uv \cup uw$ (This is sometimes called the distributive identity.)
3. $(ab)^* \neq a^*b^*$

To show two regular expressions are not identical, we must find a string in one, but not in the other. It's harder to test if they are equal, and we'll cover this later in the course. To show that the third item is not an identity, notice that $abab \in (ab)^*$, but $abab \notin a^*b^*$.

While it would be nice if we could express every language as a regular language, it is unfortunately impossible to do so. An example is the palindrome language, which we will prove later. While it can be harder to think of non-regular languages than regular languages, in some sense we have more non-regular languages.

3 A Brief Review

A language L is a subset of Σ^*, the set of strings over an alphabet. We can concatenate languages $L_1L_2 = \{uv | u \in L_1, v \in L_2\}$. We can also take the closure of languages as below:

$$L^* \coloneqq \bigcup_{n=0}^{\infty} L^n \qquad L^+ \coloneqq \bigcup_{n=1}^{\infty} L^n$$

Also, regular expressions (regexes) are simply regular sets written without the curly braces, so the regex $a(b \cup a^*)b^+$ is just a shorter way to represent the regular set $\{a\}(\{b\} \cup \{a\}^*)\{b\}^+$. Also, many regular expressions are equivalent to each other, and a good exercise would be to convince yourself that the above expression is equivalent to a^+b^+.

4 Context-Free Grammars (CFGs)

CFGs are things which can generate languages. We'll define them first and then see how they relate to languages. Grammars have two types of symbols, terminal symbols and variables. Terminal symbols are all members of the alphabet, Σ. We denote the set of variables as V. $S \in V$ is the starting variable. Lastly, we have a set P of productions, or "rules". Productions are all of the form $A \to w$, where $A \in V$ and $w \in (V \cup \Sigma)^*$. Thus, we can define a grammar as a tuple by $G = (\Sigma, V, S, P)$.

4.1 Example

Now to look at how a grammar can generate a language. Let $G = (\Sigma, V, S, P)$, let $\Sigma = \{a\}$, $V = \{S\}$, and $P = \{S \to aS, S \to \lambda\}$. We typically write P as a chart as below:

$S \to aS$
$S \to \lambda$

Or, when two productions both have the same variable on the left, we shorten it to

$S \to aS|\lambda$

In fact, usually we don't even bother to state what Σ and V are, since all the terminal symbols and variables that we care about will be shown in our productions. If we ever see a grammar for which not all the terminal symbols and variables appear in the productions, we'll explicitly state Σ and V in advance. Otherwise, assume that uppercase letters are variables and lowercase letters are terminal symbols.

We use the $\Rightarrow$ symbol to denote a derivation. $w_1 \Rightarrow w_2$ means that $w_1 = xAv$, $w_2 = xwy$, and there is a production $A \to w$ which lets us get from w_1 to w_2. The language that can be generated by the above grammar is a^*. As a demonstration of how strings are generated, we'll look at how to get aaa from

the grammar.

$$\begin{aligned} S &\Rightarrow aS && S \to aS \\ &\Rightarrow aaS && S \to aS \\ &\Rightarrow aaaS && S \to aS \\ &\Rightarrow aaa && S \to \lambda \end{aligned}$$

Also, we use a shorthand of $w \overset{*}{\Rightarrow} v$ to mean that $w \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow v$. If we know the number of steps, $w \overset{n}{\Rightarrow} w_n$ means that $w \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$. Also, while it doesn't often show up, we can use $\overset{+}{\Rightarrow}$ to mean that it takes at least one derivation, whereas $\overset{*}{\Rightarrow}$ means it takes at least zero.

Note that we have used two types of arrows. $\to$ is used for productions, whereas $\Rightarrow$ is used for derivations. Also, we call the grammar "context-free" because it doesn't matter the context in which you see variables; regardless of where you see A in a string, if $A \to w$ is a production, you can apply it. If a grammar weren't context-free, you might only be allowed to apply a rule if the variable follows or is followed by a specific string.

5 Language of a Grammar

We say a string $u \in (V \cup \Sigma)^*$ is a sentential form if $S \overset{*}{\Rightarrow} u$. We say a string $v \in \Sigma^*$ is a sentence if $S \overset{*}{\Rightarrow} v$. That is, a sentence is a sentential form consisting only of terminal symbols. For a CFG G, we say the language of the grammar, $L(G)$ is the set of sentences for the grammar. That is, $L(G) = \left\{ w \in \Sigma^* \middle| S \overset{*}{\Rightarrow} w \right\}$.

We say a language L is a context-free language (CFL) if there exists a CFG G such that $L(G) = L$. Note that we can't talk about the grammar of a language, as for every CFL L, there are infinitely many CFGs G such that $L(G) = L$. As a note, we will later show that every regular set is a CFL. We also sometimes use the term 'CFL' to denote the set of every context-free language.

5.1 Example

One might ask the question: Is it possible to have multiple distinct derivations that result in the same string? The answer is yes. We'll see how with the below

grammar.

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

This simple grammar has only one sentence, ab, but there are two distinct derivations of it. They are

$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$
$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$

If a string has multiple derivations in a grammar G, we say that G is ambiguous.

6 Derivation Trees

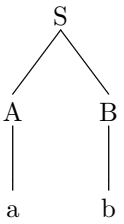
For a derivation $S \overset{*}{\Rightarrow} w$, we sometimes want to look at a derivation tree, for the sake of equating some derivations that might seem identical even though productions are applied in different orders. The derivation tree is a rooted tree, created in the following way.

Step 1. The root of the tree is S.

Step 2. If we apply a production $A \to x_1 x_2 \dots x_n$, then we add n children to A, where each child is an x_i.

Step 3. If we apply a production $A \to \lambda$, then we add a single child, λ to A.

The two derivations from before now have the same derivation tree.



We can try to fix this problem of multiple derivations for the same string by restricting ourselves to leftmost derivations. In a leftmost derivation $S \overset{*}{\underset{L}{\Rightarrow}} w$, we only apply productions to the leftmost variable in our string. We define a rightmost derivation similarly. We now ask the question, is it possible for a grammar to allow multiple leftmost derivations of the same string? The answer

is still yes, but now we can define something. We say a grammar is ambiguous if it can produce at least one string from more than one leftmost derivation. Any other grammar is unambiguous.

6.1 Example

$S \to aS|\lambda|aB$
$B \to aB|\lambda$

With B being a redundant variable, we see it is very easy to create an ambiguous grammar. Our task of trying to remove the ambiguity is harder, and sometimes impossible. Languages for which there is not an unambiguous grammar are ones we call inherently ambiguous.

6.2 Example

$S \to aS|Sa|a$

We ask the questions, what is the language, is the grammar ambiguous, and is it possible to remove any existing ambiguity? In this case, $L(G) = a^+$. Two leftmost derivations of aa exist, so there is ambiguity. If we remove either $S \to aS$ or $S \to Sa$, it will not change $L(G)$, but it will remove the ambiguity. Unfortunately, it is difficult to prove that the resulting language is unambiguous, so we will just have to take educated guesses when asking that question.

7 Languages $\leftrightarrow$ Grammars

Given either a CFG G or a language L, it is often a task to construct the other such that $L(G) = L$. It is often easier to start with a grammar.

7.1 Example

We start with the language defined by

$V = \{S, B\}$
$\Sigma = \{a, b\}$
$S \rightarrow aSa|aBa$
$B \rightarrow bB|b$

Note that the last line is a common construction in a grammar; it will become b^+. The language defined by this CFG has a positive number of a's, then a positive number of b's, and then the first number of a's again. That is, $L(G) = \{a^n, b^m a^n | n, m \geq 1\}$.

7.2 Example

We now look at the differences between two grammars, G_1 and G_2.

G_1 :
$S \rightarrow AB$
$A \rightarrow aA|a$
$B \rightarrow bB|\lambda$
G_2 :
$S \rightarrow aS|aB$
$B \rightarrow bB|\lambda$

For both of these grammars, the language is $L = a^+b^*$. Though these grammars are quite different, we can see by examination that they produce the same language. For now, we don't have a good way to prove this, but we'll cover that later in the term.

7.3 Example

Now we look at the other direction. We would like to construct a CFG to accept the language L which contains only the strings with exactly two b's. We will start with a regular expression, $a^*ba^*ba^*$. We know how to get a^*, so now we can create the CFG.

$S \rightarrow AbAbA$
$A \rightarrow aA|\lambda$

If we want to change the language to one in which strings have at least two b's, we can modify the regex rather elegantly, to $(a \cup b)^* ba^* ba^*$. Of course, we could chage every a^* to $(a \cup b)^*$, but that's unneccesary. For the grammar, we'll use $(a \cup b)^*$ to avoid having too many variables.

$S \rightarrow AbAbA$

$A \rightarrow aA|bA|\lambda$

7.4 Example

Now look at the language of even-length strings. First, we should decide if it is regular. The easiest way to prove that something is regular is to find a regex. Later in the class, we'll examine ways to prove non-regularity. With that, you may guess that this language is regular. It is, and a regex is $((a \cup b)(a \cup b))^*$. Another regex is $(aa \cup ab \cup ba \cup bb)^*$. We will also give two CFGs for this language. The first is rather simple.

$S \rightarrow aaS|abS|baS|bbS|\lambda$

The next construction is, in the opinion of the author, more elegant.

$S \rightarrow aO|bO|\lambda$

$O \rightarrow aS|bS$

In this one, the variables have meaning, which is a useful tool for construction. S means that we have an even number of symbols so far, and O means we have an odd number of symbols so far. Thus, we can only get a λ when we see an S.

7.5 Example

One last example for now is the palindrome language. We know it isn't regular (because it's written in a book, and books never lie), but we now look at a CFG for it.

$S \rightarrow aSa|bSb|a|b|\lambda$

Whenever we add a letter, we either add one to the other end to keep symmetry, or we let it be the middle letter and we stop.

8 Regular Grammars and Languages

Some grammars are regular. Feel free to guess what this means, but for now we will define it by restricting what our productions are allowed to be. In a regular

language, every production must be of one of the following three forms:

- $A \to \lambda$
- $A \to a$
- $A \to aB$

For the above, let $A, B \in V$, and let $a \in \Sigma$. Also, A and B are allowed to be the same. A variable can either go to λ, a symbol in Σ, or a symbol in Σ followed by a variable.

We define that a language L is a Regular Language if there exists a regular grammar G such that $L(G) = L$. Note that regular languages can be created by non-regular grammars, as long as there is at least one grammar that both is regular and creates L. Later in the course, we'll show that regular languages are identical to regular expressions. (If this was your guess, good job!)

Looking back to Example 7.2, notice that G_1 is non-regular, but G_2 is regular. Thus, the language a^+b^* is a regular language. Also, looking at Example 7.1, we see that the more elegant grammar happens to also be regular. Look at the other examples and try to see if the grammars are regular. Remember that, even if the grammar is non-regular, the language might be regular.

One reason we like regular grammars is the derivation trees they produce. In any derivation, there is at most one variable, and any variable is always the last symbol. All derivations thus are both leftmost and rightmost derivations.

A question to consider is whether or not it is possible for a regular grammar to be ambiguous. Unfortunately, the answer is yes. We can add redundant variables that all do the same thing.

$S \to aS|a|aB$
$B \to aS|a|aB$

9 Verifying that $L(G) = L$

The most common way to verify that a language matches a given grammar is induction. Recall that for induction, we have a function $P(n)$ defined for all natural numbers n that is either T or F. We start by showing that $P(0) = T$ (the basis step), and then we show that, if $P(n) = T$ for some n, then $P(n + 1) = T$ (the inductive step). The closure step is just a reminder that all natural numbers can be reached by a finite number of incrementations of 0.

9.1 Example

We want to show that $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. First we have the basis step.

$$\sum_{i=0}^{0} i = \frac{0(1)}{2} = 0$$

Then for the inductive step, we assume that, for some n, the above equality is true.

$$\sum_{i=0}^{n+1} i = n + 1 + \sum_{i=0}^{n} i = n + 1 + \frac{n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

And we're done!

10 Generalised Induction

Unlike previously, we now often want to induce on a recursively defined set, rather than on the natural numbers. The recursively define set will often be of the form $X = \bigcup_{i \geq 0} X_i$. The goal is to show that, $\forall x \in X$, $P(x) = T$. The basis step is to show that $\forall x \in X_0$, $P(x) = T$. The inductive step assumes that $\forall x \in \bigcup_{i=0}^{n} X_i$, $P(x) = T$ and then shows that $\forall x \in X_{n+1}$, $P(x) = T$.

10.1 Example

Given the grammar:

$S \rightarrow aS|aB|\lambda$
$S \rightarrow aB|bS|bC$
$C \rightarrow aC|\lambda$

We want to find a language L and then prove that $L(G) = L$. The language is the language with an even number of b's, and any number of a's. A regular expression for this is $L = a^* \left(a^*ba^*ba^*\right)^*$. To show that $L(G) = L$, we need to show both that $L(G) \subseteq L$ and that $L(G) \supseteq L$. The first one is equivalent to showing $\forall w \in \Sigma^*$, $S \stackrel{*}{\Rightarrow} w$ means that w has an even number of b's. The second direction, which often students forget, is to show that, for any string w with an even number of b's, we can generate it in G.

We'll do the $\supseteq$ direction first. We will consider the string

$$w = a^{n_1}ba^{n_2}b \ldots a^{n_{2k}}ba^{n_{2k+1}} \quad n_i \in \mathbb{N}$$

We want to show $S \stackrel{*}{\Rightarrow} w$.

$$
\begin{aligned}
S &\stackrel{n_1}{\Rightarrow} a^{n_1}S && S \to aS \\
&\Rightarrow a^{n_1}bS && S \to bB \\
&\stackrel{n_2}{\Rightarrow} a^{n_1}ba^{n_2}S && B \to aB \\
&\Rightarrow a^{n_1}ba^{n_2}bS && B \to bS \\
&\text{repeat until the last } b \\
&\Rightarrow a^{n_1}ba^{n_2}b\ldots a^{n_{2k}}bS && B \to bS \\
&\stackrel{n_{2k+1}}{\Rightarrow} a^{n_1}ba^{n_2}b\ldots a^{n_{2k}}ba^{n_{2k+1}}S && S \to aS \\
&\Rightarrow a^{n_1}ba^{n_2}b\ldots a^{n_{2k}}ba^{n_{2k+1}} && S \to \lambda
\end{aligned}
$$

So every string with an even number of b's can be generated in G.

Now for the $\subseteq$ direction. We would like to induce on the length of the derivation. The first thought of what to do is to make sure that the number of b's is always even. Unfortunately, this is not true. However, the sum of the number of b's and B's *is* always even. At the end of the derivation the number of B's is zero, so if we can show that the sum is always even, we'll show that every derived strings has an even number of b's. For more formal notation, we want to prove that, for any string w in any derivation, $n_w(b) + n_w(B)$ is even. For this, we will induct on the length of the derivation.

The basis is rather easy. Every string with a derivation of length zero is S, and $n_w(b) + n_w(B) = 0 + 0$ is even. The inductive step takes more work. Assume that, for every string w such that $S \stackrel{k}{\Rightarrow} w$, $n_w(b) + n_w(B)$ is even. Now we want to show that, for every w such that $S \stackrel{k+1}{\Rightarrow} w$, $n_w(b) + n_w(B)$ is still even. We can break this apart into $S \stackrel{n}{\Rightarrow} w' \Rightarrow w$. By the inductive assumption, we know that $n_{w'}(b) + n_{w'}(B)$ is even. Unfortunately, we don't know which production is used to get to w, so we'll check for each of the 8 productions that the property will be maintained. For that, we will make a small table to check what happens after each of them.

	$n_w(b) + n_w(B)$
$S \to aS$	k
$S \to bB$	$k + 2$
$S \to \lambda$	k
$B \to aB$	k
$B \to bS$	k
$B \to bC$	k
$C \to aC$	k
$C \to \lambda$	k

Since k is even, we know that every production maintains the even-ness of the sum, so we are done.

11 Parsing

Given a CFG G and a terminal string $w \in \Sigma^*$, we want an algorithm that will either prove that $w \notin L(G)$ or give us a derivation $S \stackrel{*}{\Rightarrow} w$. First we will show that the existence of a derivation is equivalent to the existence of a leftmost derivation for the same string. First, if a leftmost derivation $S \underset{L}{\stackrel{*}{\Rightarrow}} w$ exists, a derivation $S \stackrel{*}{\Rightarrow} w$ exists. Now we do the other direction.

First assume that a derivation $S \stackrel{*}{\Rightarrow} w$ exists. If this is leftmost, we're done. Otherwise, we must convert it. Consider the case $S \stackrel{*}{\Rightarrow} w$ is not leftmost. $S \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n = w$. Consider the first non-leftmost rule application $w_k \Rightarrow w_{k+1}$. $w_k = u_1 A u_2 B u_3$, where A is the leftmost variabe and $B \to v$ is the production used between w_k and w_{k+1}. So $w_{k+1} = u_1 A u_2 v u_3$. Later, we have some $j > k$ such that $w_j = u_1 A p$ and $w_{j+1} = u_1 q p$, using the production $A \to q$. Because this is a CFG, we can move this production earlier to any point where this particular A appears without changing the output. We choose to move it so that it is applied to w_k, and now the first non-leftmost rule application happens after the k^{th} application.

11.1 Counterexample

Please note this works only when $w \in \Sigma^*$. Given the grammar

$S \to AB$
$A \to aA|\lambda$
$B \to bB|\lambda$

The language of the grammar is a^*b^*. Note that, while a derivation $S \stackrel{*}{\Rightarrow} A$ exists, $S \underset{L}{\stackrel{*}{\Rightarrow}} A$ does not exist. Also, if G is ambiguous, it is posisble to have two leftmost derivations of the same string. This is something we'd like to fix.

12 Removing Ambiguity

Sometimes it is possible to remove ambiguities. However, some languages are inherently ambiguous (which is defined after some examples). We will consider some examples to try to determine when grammars can be made unambiguous.

12.1 Example

$S \rightarrow aS|Sa|a$

We ask the three questions, what is the langugage, is it ambiguous, and can we make it unambiguous without changing the language?

- $L(G) = a^+$
- $S \Rightarrow aS \Rightarrow aa$ and $S \Rightarrow Sa \Rightarrow aa$, so it is ambiguous.
- By removing either $S \rightarrow aS$ or $S \rightarrow Sa$, we make it unambiguous. Unfortunately, this is an uncomputable problem in general, so we won't prove unambiguity in this class.

12.2 Example

$S \rightarrow bS|Sb|a$

- $L(G) = b^*ab^*$
- $S \Rightarrow bS \Rightarrow bSb \Rightarrow bab$ and $S \Rightarrow Sb \Rightarrow bSb \Rightarrow bab$, so it is ambiguous.
- Just removing rules here, we will change the language of the grammar. However, just adding rules, we don't remove ambiguity. We must completely change the grammar.

  $S \rightarrow bS|aB$
  $B \rightarrow bB|\lambda$

  This is a new grammar, but it preserves the language while being unambiguous.

12.3 Inherent Ambiguity

A language L is inherently ambiguous if, for every CFG G, $[L(G) = L] \Rightarrow [G$ is ambiguous$]$. While this can be hard to prove, we can look at a language that we can convince ourselves is inherently ambiguous.

Let $L = \{a^n b^n c^m d^m | n, m \geq 1\} \cup \{a^n b^m c^m d^n | n, m \geq 1\}$. A grammar for this could look like the one below:

$S \rightarrow AB|C$
$A \rightarrow aAb|ab$
$B \rightarrow cBd|cd$
$C \rightarrow aCd|aDd$
$D \rightarrow bDc|bc$

A and B produce the first and last halves of the first part of L. C and D produce the outside and middle halves of the second part of L. The reason this grammar is ambiguous, and why any grammar for L is ambiguous, is that when $n = m$ (in the intersection of the two parts of L), we can derive the string using either method.

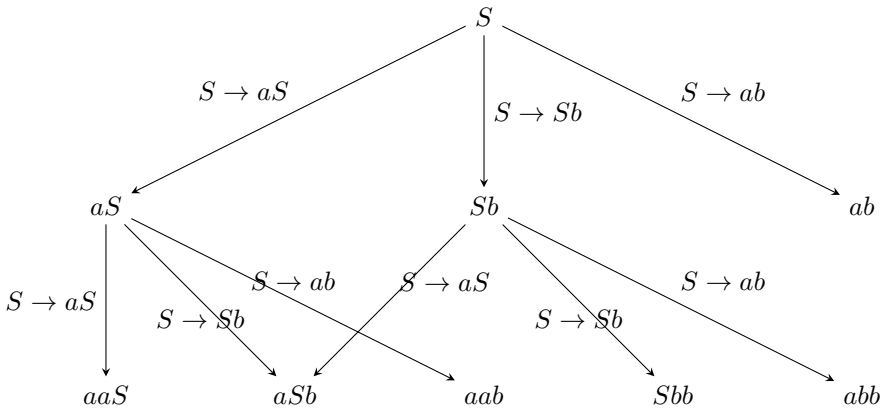
13 Back to the Parsing Problem

We will first define the leftmost graph of a grammar G, $g(G)$, to turn this into a graph problem. The nodes (or vertices) of the graph are exactly the left sentential forms, or strings $w \in (V \cup \Sigma)^*$ such that $S \underset{L}{\overset{*}{\Rightarrow}} w$. We place an arc (v, w) if there is a leftmost rule application $v \underset{L}{\Rightarrow} w$, and we label this arc with the production used. This graph often has an infinite number of nodes.

For the grammar below

$S \rightarrow aS|Sb|ab$

the beginning of the graph looks like



Because the labels often get in the way, we often will leave them out of our diagrams.

14 Quick Review

In the parsing problem, we are given a CFG G, a string $w \in \Sigma^*$, and we must either find a derivation $S \underset{L}{\overset{*}{\Rightarrow}} w$ or determine that $w \notin L(G)$. This is a graph search, where we define the leftmost graph of a grammar $g(G)$ for which the parsing problem becomes a graph search problem in $g(G)$. Our four basic search algorithms are either top-down or bottom-down, and their are either breadth-first or depth-first.

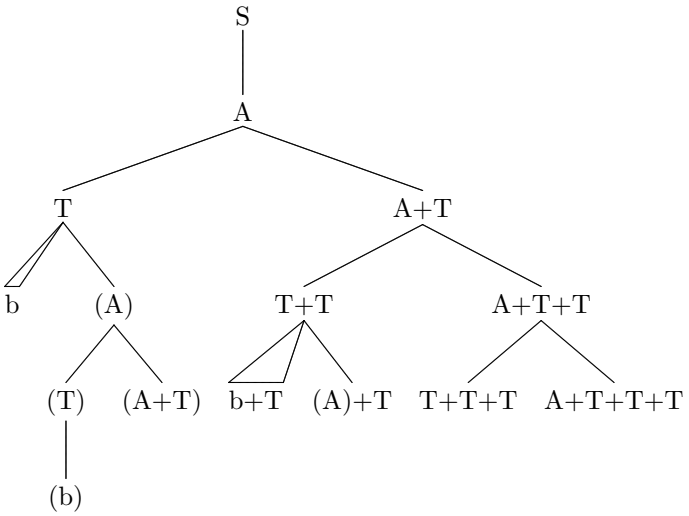
In Top-Down Breadth-First search, we start with the root node S, put it's children into a FIFO queue, and then continue with each node in the queue. To expand a node, we read the string as uAv, where A is the leftmost variable. For each production using A, we have a child. Using the production $A \to p$, we would get the child upv. If we ever reach w, we stop, as we have found a derivation. However, we can be a bit more clever with this. Since we are looking for a particular string w, each time we create a new node, we can test to see if it is possible to reach w. While there are many tests we could perform, the one we will look at (and the one in the book) is ther terminal prefix test. If a node fails the test, we consider it a dead-end, and we do not look at its children. When we get a new node upv, we look at the terminal prefix of upv (the largest prefix not containing a variable), and we determine if it is a prefix of w. Since it is entirely terminal symbols, it will never change. If it is not a prefix of w, we know it is a dead-end. On things you submit, you should label dead-ends with dashed lines as opposed to solid lines. Due to the LaTeX package I'm using being difficult, you'll see triangles used for dead-ends in these notes.

14.1 Example

In the grammar of arithmetic expressions (AE), our alphabet is $\Sigma = \{b, +, (,)\}$. Strings look like $(b + b)$ and $((b) + (((b)) + b))$. The grammar is below.

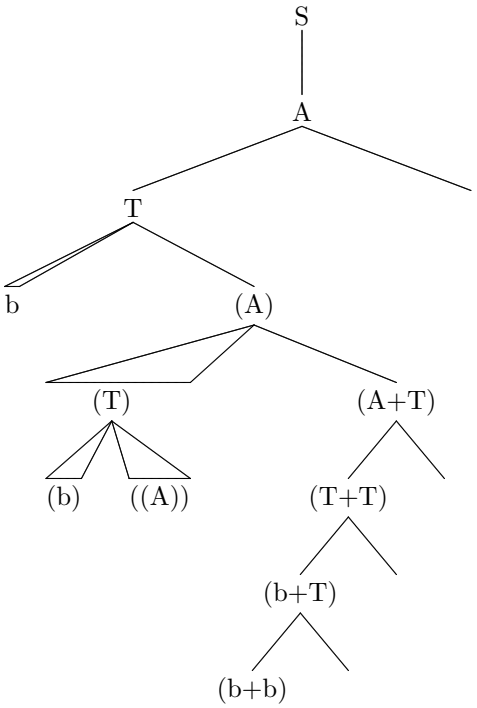
$S \to A$
$A \to T | A + T$
$T \to b | (A)$

For the string $w = (b)$, we use the top-down breadth-first search will the terminal prefix test.

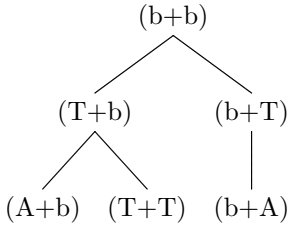


We want to know what can happen when we try to find a derivation. If $w \in L(G)$, we will find a derivation, eventually. Otherwise, either every node will be a dead-end, or the tree will continue on forever. In the above example, the right-most branch of the tree will always have λ be the terminal prefix, so it will never end. The advantage of this algorithm is that if the string is in the language, a derivation *will* be found. The disadvantage is that it is very inefficient, and it might never terminate when $w \notin L(G)$. We can make this less likely by having more rules, such as counting the number of terminal symbols, or seeing if a terminal symbol appears too many times.

To take care of inefficiency, we can use a Depth-First Top-Down search. Here, we maintain only one path. We always try the productions in a preset order. If we ever reach a node which is a dead-end, we go to its parent and try the next production from there. If all children of a node are dead-ends, and if there are no more productions, we call that node a dead-end. Unforuntately, if there is an infinite loop, we might have a case in which we never find a derivation for a string which is in the language. Below is a search for $(b + b)$ in the language.

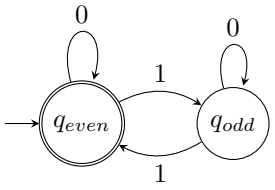


In a Bottom-Up Breadth-First search, we start with w and try to find S. This time, to make things different, let's say we're trying to find a rightmost derivation for the string, $S \underset{R}{\overset{*}{\Rightarrow}} w$. Now we have a more interesting question for how to expand a node. To expand a node, we look at every division $w = uv$ and compare the suffixes of u with the right-hand side of productions. If we have a match, that is $u = u_1 p$ and there is a production $A \to p$, we let $u_1 A v$ be a parent of w. If we have a match, we call this a valid reduction (in part because we usually reduce the length of the string when doing this). However, after the first expansion, when we have variables in our nodes, we don't try every division. We instead look only at those where $w = uv$ and v is a terminal suffix. Starting again with the string $(b + b)$ (and writing the tree upside-down for convenience of the coder), the beginning of the tree looks like this.



15 Finite Automata

To start with, we will look at the language of binary strings with an even number of 1's. (The binary alphabet is $\Sigma = \{0, 1\}$.) Recall from example 10.1 that a regular expression for this is $0^*(0^*10^*10^*)^*$. We would like to model this with a simple automata, or "machine". We'll begin with a deterministic, finite-state automaton, or a DFA. Such a machine M can be represented as a tuple $M = (Q, \Sigma, \delta, q_0, F)$. Q is a finite set of states, Σ is our input alphabet (later machines will have multiple alphabets), $q_0 \in Q$ is our starting state, and $F \subseteq Q$ is the set of "accepting" states. Lastly, we have a transition function $\delta : Q \times \Sigma \to Q$. This function is defined for every state-symbol pair, and tells us how to change between states when we read input symbols. A DFA for the language is below, followed by an explanation of how to interpret it.



The circles around q_{even} and q_{odd} indicate that they are elements of Q—they are our states. The arrow on the left pointing to q_{even} indicates that it is our starting (or initial) state, q_0. The doubled circle around q_{even} says that it is an element of F, an accepting (or final) state. The arrows between states give a pictoral description of δ. An arrow from q_i to q_j labelled with x means that $\delta(q_i, x) = q_j$. Because it is often tedious to list the entire definition of δ, we often represent a machine by a picture as above, rather than by writing out what Q, Σ, δ, q_0, and F are, similar to how we often write out the productions for a grammar rather than also explicitly stating Σ and V.

We will talk about configurations with our machines. A configuration is an ordered pair of a state and a string, or an element $(q, w) \in Q \times \Sigma^*$. One such configuration could be $(q_{even}, 0110100)$. The state is thought of as the "current" state of the machine, and the string is the "unconsumed" portion of the input. If the configuration is (q_{odd}, λ) (or anything where $w = \lambda$), the computation has finished, and the current state tells you if the input is accepted in the language based on whether or not it is an accepting state. In this case, (q_{odd}, λ) means that the input was not accepted.

We look at transitions with the following notation. For $p, q \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$, let $\delta(q, a) = p$. If our configuration is (q, aw), we use the $\vdash$ symbol to say we transition to the configuration (p, w). That is, $(q, aw) \vdash (p, w)$. In the above example, we look at the input string 0110100, and we start in the initial

state, q_{even}.

$$\begin{aligned}(q_{even}, 0110100) &\vdash (q_{even}, 110100)\\ &\vdash (q_{odd}, 10100)\\ &\vdash (q_{even}, 0100)\\ &\vdash (q_{even}, 100)\\ &\vdash (q_{odd}, 00)\\ &\vdash (q_{odd}, 0)\\ &\vdash (q_{odd}, \lambda)\end{aligned}$$

Just as we used $\overset{*}{\Rightarrow}$ to say a derivation of an unknown number of steps, we use $\overset{*}{\vdash}$ to mean any number of transitions. We say that a machine accepts a string w if $(q_0, w) \overset{*}{\vdash} (p, \lambda)$, where $p \in F$.

For any DFA $M = (Q, \Sigma, \delta, q_0, F)$, the language accepted by M is

$$L(M) = \left\{ w \in \Sigma^* \middle| (q_0, w) \overset{*}{\vdash} (p, \lambda), p \in F \right\}$$

Sometimes, it makes it easier to think about a DFA if we allow our transition function to take more than one symbol at a time. We define an extended transition function $\hat{\delta} : Q \times \Sigma^* \to Q$ recursively as follows.

Basis: For all $q \in Q$, $\hat{\delta}(q, \lambda) = q$.

Recursive Step: For all $a \in \Sigma$, $w \in \Sigma^*$, $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$.

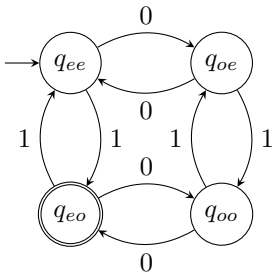
Closure Step: $\hat{\delta}(q, w)$ is defined if and only if we can get it from the basis by applying the recursive step a finite number of times.

This gives us a simpler definition of $L(M)$.

$$L(M) = \left\{ w \in \Sigma^* \middle| \hat{\delta}(q_0, w) \in F \right\}$$

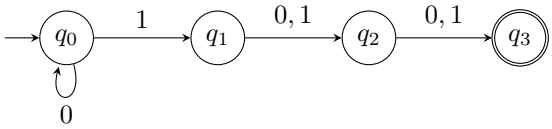
16 Finite Automata Continued

Now we want to try to combine things. What if we want to read an even number of 0's and an odd number of 1's? Well, we can do that with four states.

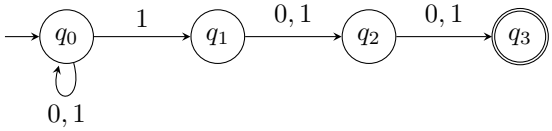


17 Nondeterministic Finite Automata (NFA)

Now we'll look at nondeterminism. We want a machine which will accept strings where the third symbol from the end is a 1. That is, strings with any of the following suffixes $\{100, 101, 110, 111\}$. We'd be able to do this with 8 states (2^3) in a DFA, but it's often easier to design NFAs, as we can do this problem with only 4 states ($3 + 1$). If we were to change from the third symbol from the end to the tenth, for example, we'd only need 11 states as opposed to 1024. Without knowing how this can be, let's examine why a DFA would need to be so large.



This almost seems to work. Anything it accepts will have a 1 as the third to last symbol. However, it fails to accept 01101. We'd like to just replace it with



Which looks like it should accept strings of the form $(0 \cup 1)^*1(0 \cup 1)^2$, which is what we want. However, a DFA requires δ to be a function, which means $\delta(q_0, 1)$ must have only one output, not two. Well, that's exactly the change we make in how we define an NFA. Of course, now there are multiple ways to read strings, only some of which will accept strings we want to accept. We say that an NFA M accepts a string w if *there exists* a way for M to read w such that the computation ends in an accepting state. (These previous two sentences described more formally in the following two paragraphs.)

While a DFA M is defined by $M = (Q, \Sigma, \delta, q_0, F)$, where $\delta : Q \times \Sigma \to Q$, an NFA M' is defined by $M' = (Q', \Sigma', \delta', q_0', F')$, where $\delta' : Q' \times \Sigma' \to P(Q')$, where $P(Q')$ is the power set of Q', or the set containing every possible subset of Q'. Again, because these take a while to write out, we'll often just show the picture rather than writing out what each of the five things is.

For an NFA $M = (Q, \Sigma, \delta, q_0, F)$, and for $q, p \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$, we consider the configuration (q, aw). We say $(q, aw) \vdash (p, w)$ iff $p \in \delta(q, a)$. Because $\delta(q, a)$ is an element of a power set, it is a set, so it makes sense to say that it has elements, such as p. This is where our nondeterminism comes from, since, in the above NFA, $(q_0, 10) \vdash (q_0, 0)$ and $(q_0, 10) \vdash (q_1, 0)$. There are multiple different paths. The language of an NFA M is defined as

$$\left\{ w \in \Sigma^* \middle| \exists p \in F,\ (q_0, w) \overset{*}{\vdash} (p, \lambda) \right\}$$

Looking at the string $w = 01000$, we want to convince ourselves that our NFA will not accept it. To do that, we will have to look through every possible way for M to read w.

$$\begin{aligned}(q_0, 01000) &\vdash (q_0, 1000) \vdash (q_0, 000) \vdash (q_0, 00) \vdash (q_0, 0) \vdash (q_0, \lambda) \\ &\vdash (q_1, 000) \vdash (q_2, 00) \vdash (q_3, 0)\end{aligned}$$

In the first try, we ended in $q_0 \notin F$, so that computation didn't accept the string. In the second, we hit $(q_3, 0)$, and $\delta(q_3, 0) =$ —there is nowhere to go from there, and there is unprocessed input, so we crash, and don't accept. However, if we instead look at $w = 0100$, we'll see that our machine works:

$$\begin{aligned}(q_0, 0100) &\vdash (q_0, 100) \vdash (q_0, 00) \vdash (q_0, 0) \vdash (q_0, \lambda) \\ &\vdash (q_1, 00) \vdash (q_2, 0) \vdash (q_3, \lambda)\end{aligned}$$

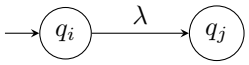
Our first computation still fails, but our second one ends in an accepting configuration, so we accept the string.

It is easy to think that an NFA is "more powerful" than a DFA—that there are languages accepted by NFAs not accepted by any DFA. Later in the course (as with so much, it seems), we'll prove that this is not true. We'll prove for any NFA M, there exists a DFA M' such that $L(M) = L(M')$, and we'll even show a way to construct M'.

We can do something similar to a top-down, breadth-first search on the computations. When we go through our computation, keep track of every possible state you could be in, which is never more than the total number of states (though the number of computations might be more). Then, at each step, you look at everywhere you can transition to from each state you could be in, and you take the union of those sets to see the new set of states you could be in. We'll revisit this idea when we prove the claim about equivalence between NFAs and DFAs.

18 Finite Automata

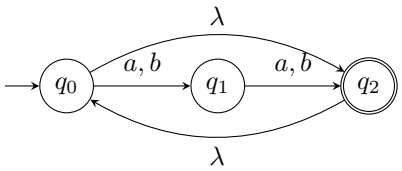
Remember that, in a DFA, our δ returns a single state, and $\delta(q_i, a) = q_j$. In an NFA, it instead can return a set of states, and $\delta(q_i, a) = \{q_{j1}, q_{j2}, \ldots, q_{jn}\}$. Thus, NFAs have nondeterminism, since a single configuration can go to multiple different states. Now we take this a step further, and we introduce the NFA-λ . In an NFA-λ , we allow $\delta(q_i, \lambda)$ to be defined. In these transitions, we don't process any of the input string. In the state diagram, it would look like



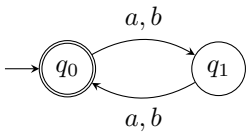
This will make designs of machines even easier.

18.1 Example

We want to design an NFA-λ such that $L(M)$ is the even length strings.

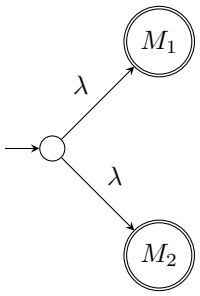


Of course, after seeing what this looks like, we can make a DFA without much trouble.



19 Operations on Languages with λ Transitions

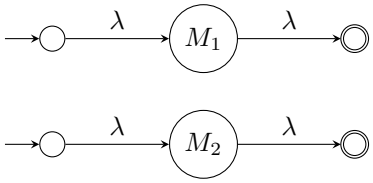
Suppose we have machines M_1 and M_2 with $L(M_1) = L_1$ and $L(M_2) = L_2$. How would we create a new machine M such that $L(M) = L_1 \cup L_2$? λ transitions make this really easy.



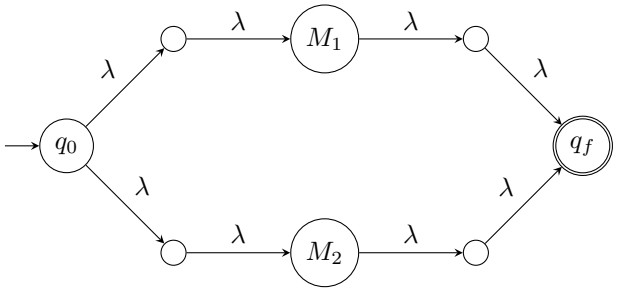
Before we go too much further with this, we want to make our NFA-λ s a bit simpler. We would like three properties to be satisfied. Luckily, with λ transitions, it is simple to see that, for any NFA-λ M, there exists an NFA-λ M' with $L(M) = L(M')$ satisfying

- Only one accepting state q_f
- The indegree of q_0 is 0
- The outdegree of q_f is 0.

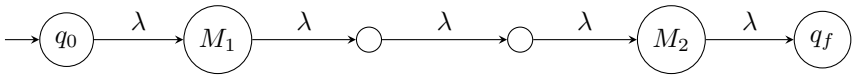
Simply make an artificial starting state which points to the original starting state with a λ transition, and make a new accepting state along with λ transitions from everything which used to be an accepting state. For that purpose, we'll now draw our machines M_1 and M_2 as follows, to show the starting and accepting states in each.



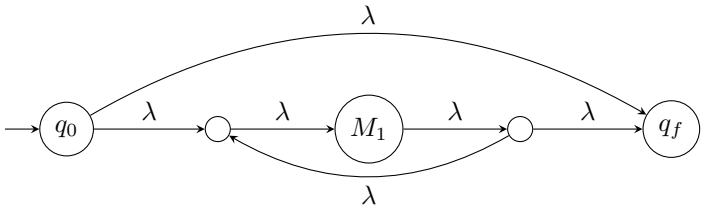
Now when we take the union, we can easily maintain these properties.



Concatenation is similarly simple, and the following machine accepts the languagte L_1L_2.



Lastly, we want to show the closure of the language, L_1^*.

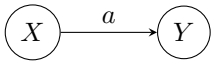


20 Equivalences

We would like to show that these machines do not have any more power than previous machines. We'd like to show that NFA-λ =NFA=DFA=Regular Languages. For now, we'll show that NFA-λ =DFA. Clearly every DFA is just a special case of an NFA-λ , so we want to show that, given an NFA-λ , we can create a DFA accepting the same language. We will do this with the Subset-Construction Method.

Given an NFA-λ M and a string w, we will look at every state we can occupy at each point in the computation. After the first symbol, perhaps there were many transitions using that symbol, and perhaps many λ transitions were available as well, so there are several states we could end up in. After the next symbol, we need to look at where we could get to from each of the previous states. We want to represent this deterministically. The idea is that the states in the DFA will represent subsets of states from the NFA-λ .

Our new starting state is the set containing q_0 and everywhere reachable through λ transitions from q_0, which we will call the λ-closure(q_0). A more formal (recursive) definition will be given later. Once we're in a state X, we want to learn what state Y is such that our DFA can contain the transition



To do this, we will define the input transition function $t(q, a)$ as the set of states reachable from q processing any number of λ and exactly one a. Then

$Y = \bigcup_{q \in X} t(q, a)$. Tomorrow we will go through this with an example.

21 λ-Closure

Last time, we claimed that, for any NFA-λ M, there exists a DFA DM such that $L(DM) = L(M)$. We will show that with the subset construction method. First we need to define the λ-closure of a state and the input transition function t.

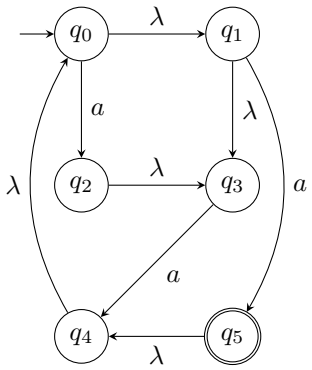
A recursive definition of the λ-closure is as follows.

Basis Step: $q_i \in \lambda\text{-closure}(q_i)$

Recursive Step: $q_j \in \lambda\text{-closure}(q_i)$, $q_k \in \delta(q_j, \lambda) \Rightarrow q_k \in \lambda\text{-closure}(q_i)$

Closure Step: Every state which can be shown in a finite number of iterations to be in the λ-closure of a state is exactly the set of states in the λ-closure of that state.

That is, anything you can reach with a finite number of λ transitions of q_i is in the λ-closure of q_i. In the following NFA-λ ,



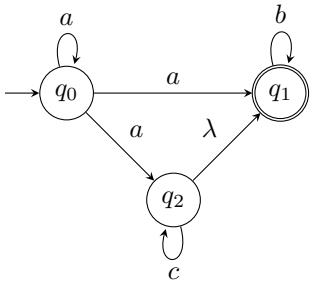
the λ-closure of q_3 is just the set $\{q_3\}$, since you can't reach any state other than itself with only λ transitions. However, the λ-closure of q_4 is the set $\{q_0, q_1, q_3, q_4\}$.

Now we will define the input transition function $t(q, a)$.

$$t(q, a) \coloneqq \bigcup_{q' \in \lambda\text{-closure}(q)} \lambda\text{-closure}(\delta(q', a))$$

Now we can go on to define the subset construction method. First, we let the new starting state be λ-closure(q_0). We then iteratively create transitions $\delta(X, a) = Y$ where $Y = \bigcup_{q \in X} t(q, z)$. Accepting states will be any state in the DFA representing a set containing an accepting state from the NFA-λ .

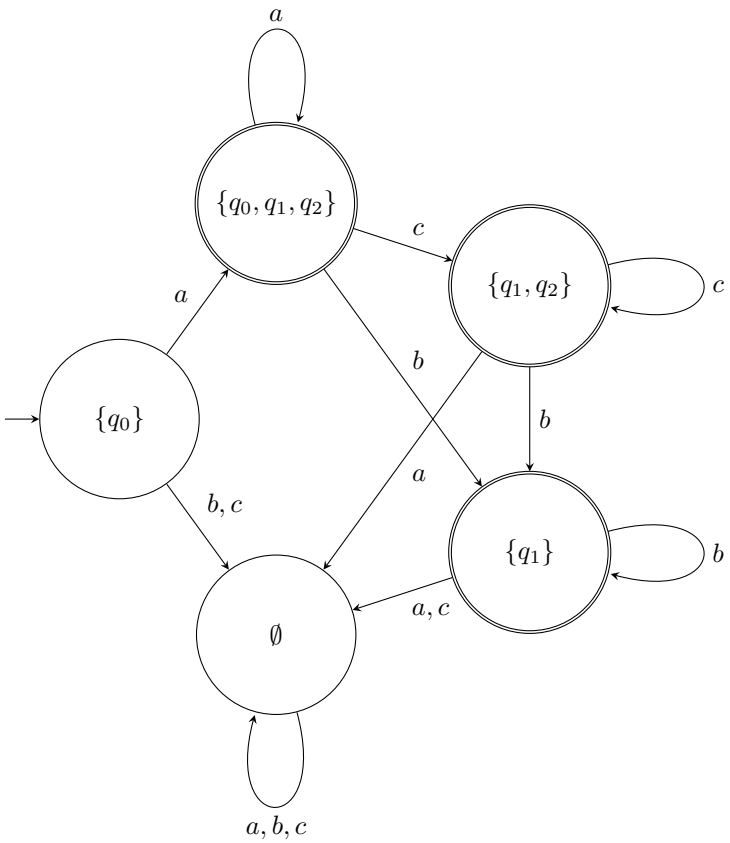
21.1 Example



It is often easiest to construct the subsets in a table. We will now look at that table for the above machine.

t	a	b	c
q_0	$\{q_0, q_1, q_2\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$	$\{q_1, q_2\}$

Now, when we make our DFA, if we are in state $\{q_1, q_2\}$, we will take the union of $t(q_1, x)$ and $t(q_2, x)$ to see where we go to on the symbol x.



Proving that they accept the same languages can be tricky, but the reader is welcome to examine which strings are accepted by the two machines. It should be the case that they both accept $a^+c^*b^*$, but only if you trust the writer's proof-reading.

Because the states in the DFA DM we constructed represent subsets of states in the original NFA-λ M, we can determine how many states DM might have, in the worst case. In the worst case, every subset of the states in M is a state in DM, in which case the set of states in DM is $\mathcal{P}(Q)$, where Q is the set of states in M, and $\mathcal{P}$ represents the power set. From Discrete Mathematics, we recall that $|\mathcal{P}(Q)| = 2^{|Q|}$. Now we see why, for the language accepting strings where the 3$^{\text{rd}}$-to-last symbol is a 1, the DFA takes so many more states than the NFA. We also see why, as the 3 gets bigger, the DFA gains more states so much more quickly than the NFA.

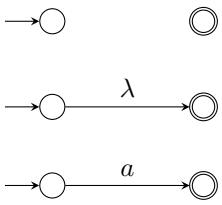
21.2 Automata Summary

We've now shown that the set of languages accepted by DFAs is the same as the set of languages accepted by NFAs and NFA-λ s. Every NFA is a special case of an NFA-λ , and every DFA is a special case of an NFA. All we needed to show was that we could convert an NFA-λ into a DFA. We now claim that this set of languages is the same as the set of regular languages (or regular expressions).

22 RegEx=DFA

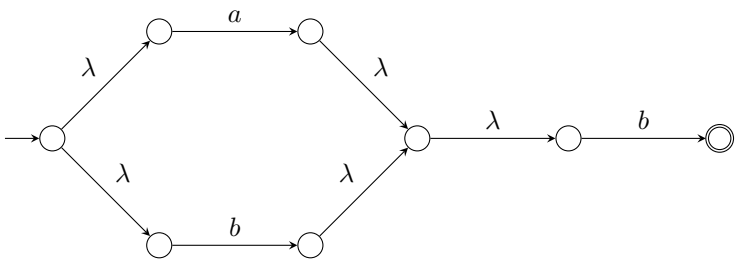
We will use the shorthand that RegEx stands for the set of regular expressions, and DFA stands for the set of languages accepted by some DFA. To prove the equation above, we need to show both that RegEx$\subseteq$DFA and also that RegEx$\supseteq$DFA. It is rather easy to show the first direction with what we did previously with NFA-λ s.

To show RegEx$\subseteq$DFA, we will show how to construct an NFA-λ to accept any regular expression. We will make use of the recursive definition of regular expressions. In section 19, we showed that, given two NFA-λ s, we can construct an NFA-λ to accept the union of their languages, the concatenation of their languages, and the closure of one of their languages. Now all we need to do is show that we can create an NFA-λ for each basis regular expression. There are three of those: $\emptyset$, λ, and a.



22.1 Example

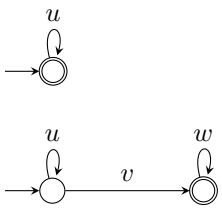
An NFA-λ to accept $(a \cup b)b$ is below:



23 The Other Direction

Now that we've shown that RegEx⊆DFA, or that we can take any regular expression and create an NFA-λ , we want to go the other way. For any NFA-λ M, there is a regular expression u such that $L(M) = u$.

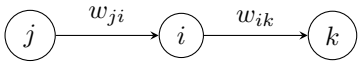
In small examples, this is easy. For example, u^* and u^*vw^* are the languages accepted by the following two machines.



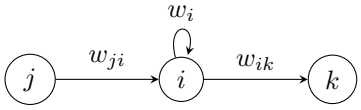
Our goal is to delete states one-by-one to get to something as simple as the above machines. This method will be called the Regular Expression Graph Method. We may assume that each machine has exactly one accepting state. We will number the states 1 through n, and then repeat the following steps.

(1) Pick a state i that is neither the starting state nor the accepting state.

(2) Delete state i with the following procedure. For every pair of states j, k such that neither is i:

Case 1: 

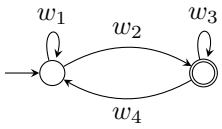
Bypass state i by creating an edge from j to k which processes the regular expression $w_{ji}w_{ik}$.

Case 2: 

Bypass state i by creating an edge from j to k which processes the regular expression $w_{ji}w_i^*w_{ik}$.

(3) Now, for every pair j, k, if there are multiple transitions going from j to k, replace them with a single transition that process the union of what the multiple transitions would process.

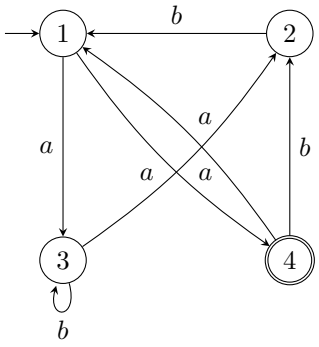
Now when we want to read the language, we will be left with a machine which looks like this.



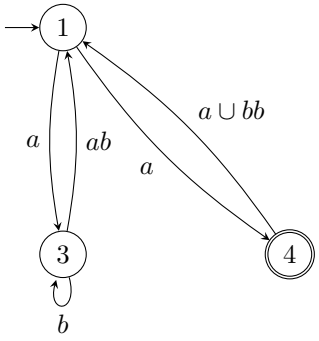
We can read the regular expression as $w_1^*w_2\,(w_3 \cup w_4w_1^*w_2)^*$.

23.1 Example

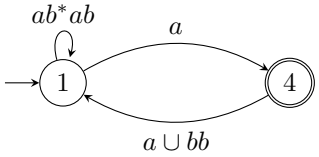
We will try to go through those steps for the following NFA.



First we will delete node 2. This will create an edge from 3 to 1 which processes ab, and will update the edge from 4 to 1 to process $a \cup bb$.



Now we can delete node 3. This will make a loop at node 1 processing ab^*ab.



Now when we read the regular expression, we see that the machine accepts the language $(ab^*ab)^*a\,[(a \cup bb)(ab^*ab)^*a]^*$

24 Tying in Regular Languages

Recall that a regular language can be generated by a regular grammar. We will show that they are the same as the regular expressions, along with all the finite state machines we've looked at so far.

A regular grammar can have only productions of the following form:

$A \to aB$

$A \to a$

$A \to \lambda$

where $A, B \in V$ and $a \in \Sigma$. We allow $A = B$. To show that these are the same languages, we'll show that we can go between NFAs and regular grammars.

24.1 $\subseteq$

Given a language L generated by a regular grammar G, we want to design an NFA M such that $L(G) = L(M) = L$. We'll first look at an example and then

explain how to do it in general.

24.2 Example

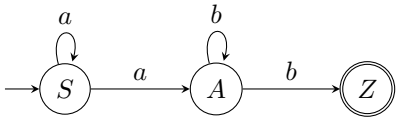
Let G be the grammar

$S \to aS|aA$
$A \to bA|b$

First, we want to determine which language it accepts, so we know what our goal is. The language is a^+b^+. Given $w = aabb$, we can look at the derivation of w in G.

$S \to aS$
$\quad \to aaA$
$\quad \to aabA$
$\quad \to aabb$

We would like there to be a very concrete relation between variables and states. Because a regular grammar allows only one variable at any step in a derivation, we can consider them as the state we're in. If a variable has a production to a terminal symbol, the corresponding state will transition to a special accepting state. The NFA for the above grammar will be



24.3 General Construction

In general, the constrction of the NFA M from a regular grammar G is as follows. Recalling that $M = (Q, \Sigma, q_0, \delta, F)$, we will define what each of those five things is. Let $Q = V \cup \{Z\}$, and notice that we will only use Z if there is a production in the grammar which takes a variable to a terminal symbol. In fact, if no such production exists, we leave our Z. In our machine, Σ will be the same alphabet as the alphabet of the grammar. q_0 is going to be S. Lastly, for each production in P for the grammar, we put in a transition.

$$\delta(A, a) = \begin{cases} B & \text{if } A \to aB \\ Z & \text{if } A \to a \end{cases}$$

This is almost all we need. The only thing remainging is how to deal with λ productions. Z is in F. Also, any variable with a λ production becomes an accepting state (an element of F) in M.

24.4 $\supseteq$

Given a language L generated by an NFA M, we want to design a regular grammar G such that $L(G) = L(M) = L$. Looking at what we did in the other direction, we see that we can do the exact same thing in reverse.

For each state in M, create a corresponding variable in G. S in G is q_0. For every transition in M given by $\delta(A, a) = B$, we create a production in G $A \to aB$. Lastly, for each state $X \in F$, we create a production $X \to \lambda$. Notice that, in the grammar we create, we will never send a variable to a terminal symbol.

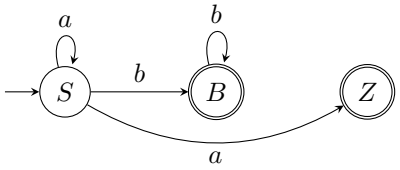
Because we claim this process is reversible, we should be able to do it in one direction and then the other direction and end up with something equivalent. Suppose we start with an NFA M, create a regular grammar G, and from that make an NFA M'. We want to know what we can say about the relation between M and M'. If we remember to leave out Z in M' (as G won't send any variables to terminal symbols), we will end up with two isomorphic (or identical) machines. That is, $M = M'$. Now let's consider what happens if we start with a regular grammar G, create an NFA M, and from that make a regular grammar G'. We would hope that the same thing as above happens here. In this case, G might have terminal productions, but we know that G' will not have terminal productions. If G has no terminal productions, they will be the same grammar. However, if G has a terminal production $A \to a$, G' will have two productions, $A \to aZ$ and $Z \to \lambda$. Aside from that, the two grammars will be identical.

24.5 Example

Starting with the regular grammar G defined by

$S \to aS|bB|a$
$B \to bB|\lambda$

we will create an NFA M accepting the same language.



Lastly we will create a new grammar G' from our NFA M.

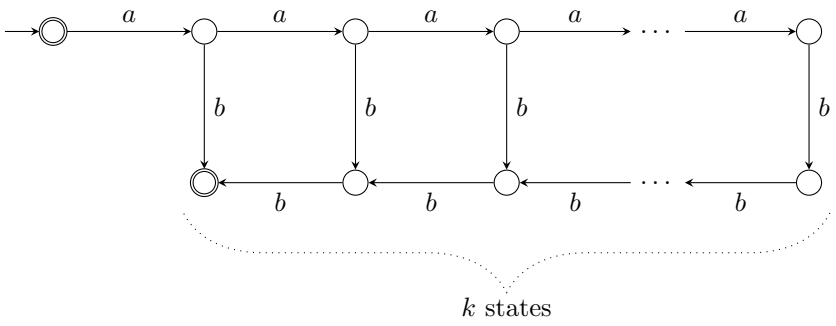
$S \to aS|bB|aZ$
$B \to bB|\lambda$
$Z \to \lambda$

25 Non-Regular Languages?

Now that we've shown that everything we've examined is the same set of languages (regular languages), we ask the question if there are any non-regular languages. Since the course isn't yet over, you may have guessed that the answer is "yes." There are languages which can not be generated by any DFA or any regular grammar. In fact, in some sense, there are more of these than there are regular languages. Two examples which have been mentioned previously in the class are $L = \{a^i b^i | i \geq 0\}$, where we have any number of a's followed by the same number of b's, and the Palindrome Language= $\{w | w^R = w\}$. To show that these languages are not regular, we will introduce the Pumping Lemma, which is probably the hardest part of this course. Before that, try to appreciate the statement that, though infinitely many finite machines exist, there are languages which can not possibly be accepted by such machines. Also, before we get to the Pumping Lemma, we'll try to find out what goes wrong when we try to create NFAs for non-regular languages.

25.1 Example

Consider the language $L = \{a^i b^i | 0 \leq i \leq k\}$, where we put an upper-bound on the size of the strings. A machine to accept this language would look like this:



This machine has $2k + 1$ states and accepts the language. However, if we don't put a bound on the size of the strings, or if we let k go to infinity, we no longer have a finite number of states, and the machine is no longer a finite automaton.

26 Introduction to Pumping Lemma

Suppose we have a DFA M with k states. Suppose we have a string $w \in L(M)$ such that $|w| \geq k$. The pumping lemma claims that, while going through the computational path to process w, we will repeat a state and have a cycle. When we have that cycle, we can "pump" it as many times as we want. We can find a collection of strings in the language by going though that cycle any number of times. The proof of this relies on the pigeonhole principle. The computation of a string of length n will visit $n + 1$ states, including the initial and final states. Because the machine has only k states, processing a string of length k will visit too many states for them to all be unique. That is, at least one state is visited twice. That state is where our loop begins and ends, and we pump it through.

27 Pumping Lemma for Regular Languages

When we show that languages are not regular using the Pumping Lemma, we will use an indirect argument. This is a bit different from how we normally prove things, as we will assume something and then show that our assumption is wrong.

27.1 Formal Statement

Given a language L accepted by a DFA with k states, pick any string $z \in L$ such that $|z| \geq k$. The Pumping Lemma says that we can write $z = uvw$ satisfying

(1) $|v| > 0$ (v will be our cycle)

(2) $|uv| \leq k$ (the cycle will be in the first k characters)

(3) $\forall i \in \mathbb{N}$, $uv^*w \in L$ (we can pump our cycle any number of times)

Notice that the lemma doesn't tell us how to decompose z into u, v, and w, and it certainly doesn't tell us what v is.

27.2 Example

Let $L = \{a^i b^i | i \geq 0\}$. We want to show that L is not regular. We will use the Pumping Lemma and an indirect argument.

Assume indirectly ("for the sake of contradiction") that L is regular. Because of this assumption, there must be some DFA M with k states such that $L(M) = L$. Now we need to pick a string $z \in L$ such that $|z| \geq k$. This can take some cleverness, depending on how complicated our language is. For this language, picking $z = a^k b^k$ will work. Another string $q = a^{\frac{k}{2}} b^{\frac{k}{2}}$ satisfies the length requirement and is in the language, but we will later see why z is better than q. The Pumping Lemma tells us that there is a decomposition $z = xyw$ that satisfies the three points above. Because the length of uv is less than k, we know that u and v consist only of a's, and we know that $v \neq \lambda$. We can write $v = a^i$, where we don't know what i is, but we know $i > 0$. The third condition of the Pumping Lemma tells us that $uv^2w \in L$, but $uv^2w = a^{k+i}b^k$, and $k + i \neq k$, so $uv^2w \notin L$. Here is a contradiction, which proves that we made a false assumption. The only assumption we made (aside from the correctness of the Pumping Lemma, which was partially hand-waved) was that L is regular, so that assumption must be false, and L is non-regular. The reason that q would not have worked as well is that, since the string is only size k, we have no control over where the cycle is. v might be the middle of the string, *aabb*, or it might consist only of b's, and there would be more cases to consider to show that pumping it through would leave our language. Because we chose z, every possible value of v was a^i, for an unknown number i. Thus, there is only one case to consider.

27.3 Example

Let $L = \{w \in \Sigma^* | w^R = w\}$. We want to show that L is not regular. We will use the Pumping Lemma and an indirect argument.

Assume indirectly that the Palindrome Language is regular. Because of this assumption, there must be some DFA M with k states such that $L(M) = L$. Now we need to pick a string $z \in L$ such that $|z| \geq k$. In this case, we will work with the string $z = a^k b a^k$. $|z| = 2k + 1 > k$, and the author (in an all-knowing state of mind) tells you that this string will lead to a contradiction, which is what we want. The Pumping Lemma tells us that there is a decomposition $z = xyw$ that satisfies the three points above. Because the length of uv is less than k, we know that u and v consist only of a's, and we know that $v \neq \lambda$. We can write $v = a^i$, where we don't know what i is, but we know $i > 0$. The third condition of the Pumping Lemma tells us that $uv^2w \in L$, but $uv^2w = a^{k+i}ba^k$, and that isn't a palindrone. Here is a contradiction, which proves that we made a false assumption. The only assumption we made was that L is regular, so that assumption must be false, and L is non-regular.

At this point, you may think that the author is abusing their powers of copy + paste. Don't tell anyone. We'll do one more example, and this will be a bit different.

27.4 Example

Let L be the language of all strings where the length of the string is a perfect square. We want to show that L is not regular. We will use the Pumping Lemma and an indirect argument.

Assume indirectly that the L is regular. Because of this assumption, there must be some DFA M with k states such that $L(M) = L$. Now we need to pick a string $z \in L$ such that $|z| \geq k$. In this case, we will work with the string $z = a^{k^2}$. $|z| = k^2 > k$, and this string will lead to a contradiction. The Pumping Lemma tells us that there is a decomposition $z = xyw$ that satisfies the three points above. Because the length of uv is less than k, we know that $v = a^i$ where $0 < i \leq k$. The third condition of the Pumping Lemma tells us that $uv^2w \in L$, but $uv^2w = a^{k^2+i}$, and we can show that $k^2 + i$ can't be a perfect square. $k^2 < k^2 + i$, and $(k + 1)^2 = k^2 + 2k + 1 > k^2 + i$ (because $i < k < 2k + 1$), so $k^2 + i$ is between two consecutive squares and isn't a square. Here is a contradiction, which proves that we made a false assumption. The only assumption we made was that L is regular, so that assumption must be false, and L is non-regular.

28 More Applications of the Pumping Lemma

So far, we've used the Pumping Lemma to show that certain languages are non-regular. There are some more interesting things we can use it to show. Here are two of them. Suppose we have a DFA M with k states.

(1) $L(M) \neq \emptyset \Leftrightarrow \exists z \in L(M)$ s.t. $|z| < k$

(2) $|L(M)| = \infty \Leftrightarrow \exists z \in L(M)$ s.t. $k \leq |k| < 2k$

28.1 First Property Proof

Notice that the $\Leftarrow$ direction is trivial. If there is a string with length less than k, the language is non-empty. The other direction is more interesting to show.

Assume $L(M) \neq \emptyset$. Choose a string $z \in L(M)$ such that, for any $z' \in L(M)$, $|z| \leq |z'|$. Either $|z| < k$ or $|z| \geq k$. In the first case, we're done. In the second case, by the Pumping Lemma, we can decompose $z = uvw$ to follow the three properties of the Pumping Lemma. By the lemma, $uv^0w \in L(M)$. However, $\left|uv^0W\right| < |uvw|$, so z is not the shortest string in $L(M)$, so this is a contradiction. Therefore, the second case can not happen, and we know that $|z| < k$.

28.2 Second Property Proof

This time, neither direction is trivial, and we need to use the Pumping Lemma for both directions. First we will do the $\Leftarrow$ direction.

Assume $\exists z \in L(M)$ s.t. $k \leq |z| < 2k$. By the Pumping Lemma, since $|z| \geq k$, $z = uvw$ satisfying the necessary properties. Thus, $\forall i \in \mathbb{N}$, $uv^iw \in L(M)$, so $L(M)$ is infinite (has infinitely many strings).

Now for the other direction. Assume $L(M)$ is infinite. Take the shortest $z \in L(M)$ such that $k \leq |z|$. If $|z| < 2k$, we are done. Now we will assume (again for the sake of contradiction) otherwise—assume that $|z| \geq 2k$. The Pumping Lemma implies that $z = uvw$, and $|uv| \leq k$, which further implies that $|v| \leq k$. We know that $uv^0w = uw \in L(M)$. However, $|uw|$ is at most $|z| - k$, but since $|z|$ is strictly greater than $2k$, $|uv| > k$, and z isn't the shortest string with length at least k, so we have a contradiction. Thus, the second case can't happen, and $|z| < 2k$.

29 Closure Properties of Regular Languages

We know that there are some operations we can perform on regular languages to get new regular languages. These will help us to show that more languages are non-regular. Because all regular languages have a regular expression, we can take the union, concatenation, and star-closure of a regular languages to get a regular languages. Because all regular languages can be formed from a DFA M, we can construct a DFA M' which has accepting states only where M doesn't. This will accept the complement of the language we had, $L(M') = \overline{L(M)}$. With the complement and the union, we can use DeMorgan's Law to show that the intersection is good as well. $A \cap B = \overline{\overline{A} \cup \overline{B}}$

Unfortunately, we can't take any subset of a regular language to get a new one. For the most general example, take the regular language $(a \cup b)^* = \Sigma^*$. The Palindrome Language is a subset of Σ^*, but the Palindrome Language is irregular. Also, we can't relax the conditions from the previous examples. For example, given that L_1, L_2 are regular, we know that $L_1 \cap L_2$ is regular. However, if we don't know that L_2 is regular, we can't say anything about the intersection. But, we can use the closure properties from above to show that some languages are non-regular.

29.1 Example

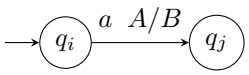
We can show that the language $L = \{a^i b^j | i \neq j\}$ is non-regular. Assume for the sake of contradiction that L is regular. Then the complement $\overline{L}$ is regular. Lastly, we can say that $\overline{L} \cap a^* b^*$ is regular. Unfortunately, that gives us $\{a^i b^i | i \geq 0\}$, which we know is non-regular. This is a contradiction, so we know that L is irregular.

30 More Machines

So far we've shown that the languages of RegExs, DFAs, NFAs, and NFA-λ s are all the same. Now we want to look at Context-Free Languages, CFLs. Context-Free Languages are the languages accepted by CFGs. We will look at machines called Pushdown Automata (PDAs) and then show that the languages are the same.

31 Pushdown Automata

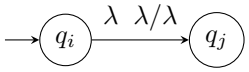
A PDA is essentially an NFA-λ plus a stack. A PDA M has a set Γ, which is the stack alphabet. The δ function now looks like $\delta : Q \times \Sigma \times \Gamma \to \mathcal{P}(Q \times \Gamma)$. Thus, a typical transition would look like $\delta(q_i, a, A) = \{[q_j, B], [q_k, C], \dots\}$. In our machines, we write this as



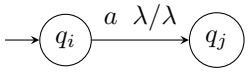
On these transitions, we change the state from q_i to q_j, we process the input symbol a, we pop A from the stack, and lastly we pop B from the stack. For our notation, when we see a lowercase letter, we will assume it is a terminal symbol (an element of Σ). When we see an uppercase letter, we will assume it is for the stack (an element of Γ). λ will appear in both of these, though future versions of the course might use Λ in the stack alphabet for clarification and also consistency.

PDAs are non-deterministic by default. If we were forced to make them deterministic, they would be less powerful. A popular question in this branch of computer science is whether or not adding non-determinism adds power to a type of machine. We saw earlier that, for DFAs, adding it does nothing. For these machines, it does. For Turing Machines, which we'll examine later in the course, it is an open question (and equivalent to the P vs. NP conjecture).

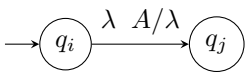
Some interesting transitions in PDAs are below.



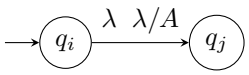
This is a typical λ transition.



This transition is essentially what we do in an NFA.



In this one, we just pop A from the stack.



Lastly, in this one, we push A to the stack.

32 Normal Forms of Context Free Grammars

For CFGs, these will be nicer forms that we can convert grammars into that satisfy properties we like. We will look at the Chomsky and Greibach normal forms.

32.1 Chomsky Normal Form

For every CFG G, there is another CFG G' with $L(G) = L(G')$ such that G' is in Chomsky Normal Form (CNF). Being in this form means that all productions are of the following form:

(1) $S \to \lambda$ (only the starting variable can go to λ, and this only shows up is $\lambda \in L(G)$)

(2) $A \to a$

(3) $A \to BC$, $B, C \neq S$

To see that we can do this, we should show that we can turn any production into a few productions that are equivalent.

32.2 Example

Given the production $A \to bCdF$, we want to convert this into productions allowed in CNF. To start, we'll turn it into:

$A \to BCDF$
$B \to b$
$D \to d$

We can easily see that this is equivalent, as B and D will turn into b and d. Now we need to deal with the fact that A turns into four symbols. A solution

to this is

$A \to BT_1$
$T_1 \to CT_2$
$T_2 \to DF$
$B \to b$
$D \to d$

And this will be in CNF.

32.3 Greibach Normal Form

For every CFG G, there is another CFG G' with $L(G) = L(G')$ such that G' is in Greibach Normal Form (GNF). Being in this form means that all productions are of the following form:

(1) $S \to \lambda$ (only the starting variable can go to λ, and this only shows up is $\lambda \in L(G)$)

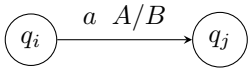
(2) $A \to a$

(3) $A \to aA_1A_2 \dots A_k$

One advantage of this is that, when parsing, the terminal prefix test will be more useful, as every production will increase the length of the terminal prefix by one.

33 Back to PDAs

Recall that, in our PDAs, transitions look like



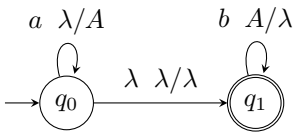
and any number of the symbols can be λ. When tracing our computations, our configurations will need to have the current state and remaining input, and also the content of the stack. That it, our configurations will look like $[q_i, w, \alpha]$, where the three symbols represent the state, input, and stack, respectively. In a PDA M, the language of the machine $L(M)$ is defined by

$$L(M) = \left\{ w \middle| \exists [q_0, w, \lambda] \overset{*}{\vdash} [q_f, \lambda, \lambda], q_f \in F \right\}$$

Note that we only accept if we end in an accepting state with an empty stack. We will later look at variants where we change this.

33.1 Example

We want to design a PDA M to accept $L(M) = \{a^i b^i | i \geq 0\}$. Afterall, we're told that PDAs are more powerful than DFAs, so we should be able to accept an irregular language.



Remember that PDAs are nondeterministic by default. In this machine, We start by reading any number of a's, while adding that many A's to the stack. Then we eventually transition to q_1, where we add as many b's to the stack as it takes to clear the stack. We should be able to trace a computation of $w = aabb$ on this machine, so let's do it.

$$\begin{aligned}[q_0, aabb, \lambda] &\vdash [q_0, abb, A] \\ &\vdash [q_0, bb, AA] \\ &\vdash [q_1, bb, AA] \\ &\vdash [q_1, b, A] \\ &\vdash [q_1, \lambda, \lambda]\end{aligned}$$

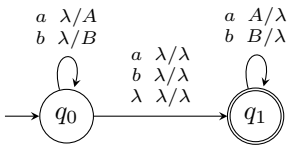
We should also be unable to successfully process $w = aab$.

$$\begin{aligned}[q_0, aab, \lambda] &\vdash [q_0, ab, A] \\ &\vdash [q_0, b, AA] \\ &\vdash [q_1, b, AA] \\ &\vdash [q_1, \lambda, A]\end{aligned}$$

Since we end with no unprocessed string and something in the stack, we reject the computation.

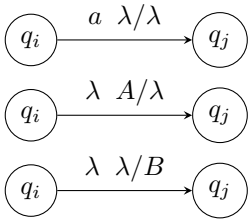
33.2 Example

Now we'll try to construct a PDA to generate the Palindrome language.



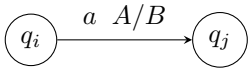
34 Atomic PDA

Just as we talked about variants of the DFA, we will talk about variants of the PDA. The first variant we will talk about is the Atomic PDA. In an Atomic PDA, we only allow the following three types of transitions.

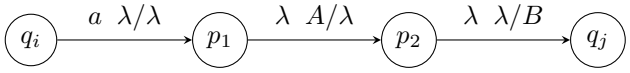


We must separate the transitions into atoms. We want to show that these are exactly as powerful as regular PDAs. Clearly, whenever we have an Atomic PDA, we have a PDA, so that direction is trivial. The other direction is more interesting. Given a PDA M, we want to prove we can construct an Atomic PDA M' such that $L(M) = L(M')$.

Let's examine a single transition in M and see how we make something equivalent in M'. Let's say that, in M, we have the following transition.

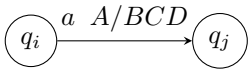


Now in M', we can just have



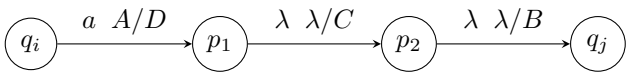
35 Extended PDA

In the extended PDA, we can push as many elements onto the stack as we want per transition. Our transitions look like



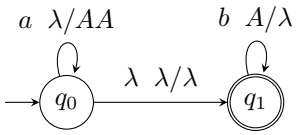
and B will be the top element of the stack after this. Again, one of the directions to show these are equivalent is trivial. Every PDA is an Extended PDA. So now suppose we have an Extended PDA M, and we want to construct a PDA M' such that $L(M) = L(M')$. We will again look at a single transition at a time. Given the transition in M

M' will have the transitions (along with intermediate states)



35.1 Example

Let's say we want to create an Extended PDA to accept the language $L = \{a^i b^{2i} | i \geq 0\}$. While we could do this with a PDA, it's easiest to see how with an Extended PDA.



35.2 Two More Variants

Remember that, in a PDA, we accept only when the computation reaches a final state with an empty stack. In the Final State Only PDA, we accept anything which ends in a final state; we only care about final states. In the Empty Stack Only PDA, we accept anything which ends in an accepting state; we only care

about the result of the stack.For short, we will refer to Final State Only PDAs as FSPDAs and Empty Stack Only PDAs as ESPDAs.

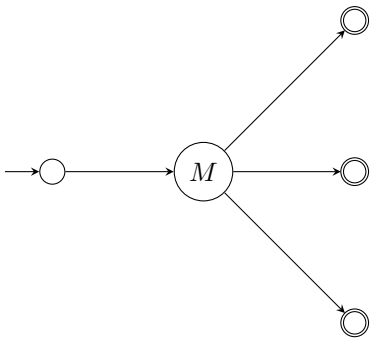
35.3 FSPDAs

Given a FSPDA M, we define the language of M as

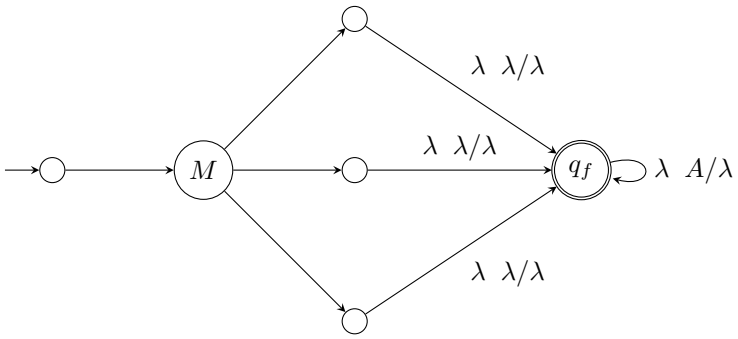
$$L(M) = \left\{ w \middle| \exists [q_0, w, \lambda] \overset{*}{\vdash} [q_f, \lambda, \alpha], q_f \in F \right\}$$

We want to show that this definition lets FSPDAs be exactly as powerful as PDAs. Given either one, we can create the other accepting the same language.

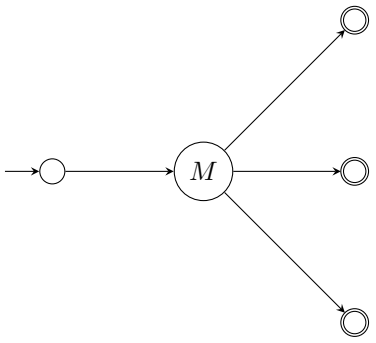
Given a FSPDA M, design a PDA M' such that $L(M') = L(M)$. To do this, we will represent M as follows:



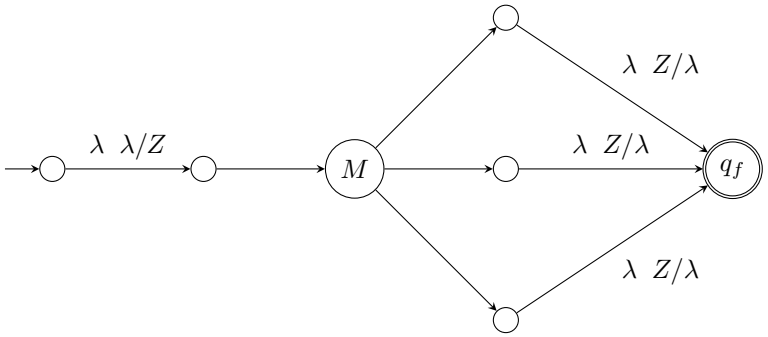
To create M', we will create a new accepting state q_f. All the accepting states in M will no longer be accepting states in M', but they will have a λ transition to q_f. Finally, we add a loop on q_f to empty the stack.



Now we look at the other direction, given a PDA M, we want to make a FSPDA M' with $L(M') = L(M)$. Consider M to be the below machine.



The issue with turning this into a FSPDA is that it will accept too many strings, as it might be possible to get to a final state with symbols on the stack. To solve this, we will make a new starting state, and transition to the old starting state by pushing a marker variable to the stack. Then, from each accepting state in the original machine, we make a transition to a new accepting state which pops that variable. Here, it doesn't matter if we push anything, as we will be in a final state in a FSPDA, but we may as well not push anything.



Thus, we can convert between the two types of machines, and they have the same power.

35.4 ESPDAs

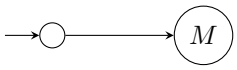
Given a FSPDA M, we define the language of M as

$$L(M) = \left\{ w \middle| \exists [q_0, w, \lambda] \overset{+}{\vdash} [q_i, \lambda, \lambda] \right\}$$

Notice that we require at least one transition, to prevent λ from always being in the language. Also notice that we don't have final states in this model, as we don't care about them. We want to show that this definition lets ESPDAs

be exactly as powerful as PDAs. Given either one, we can create the other accepting the same language.

Given an ESPDA M, design a PDA M' such that $L(M') = L(M)$. To do this, we will represent M as follows:



First, we will make every state accepting. However, then we have the same problem with always accepting λ, so we need a new starting state from which will have every transition that our original starting state had.

36 CFL=PDA

Now that we know that all of these variants of the PDA are equivalent, we want to show that they can accept any context free language.

36.1 $\subseteq$

Let $L \in$ CFL. There is a CFG G in Greibach Normal Form such that $L(G) = L$. Remember that every grammar in GNF has only the following transitions:

(1) $S \to \lambda$ (only the starting variable can go to λ, and this only shows up is $\lambda \in L(G)$)

(2) $A \to a$

(3) $A \to aA_1A_2 \dots A_k$

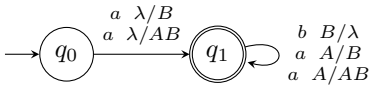
Notice that, for every string w in a derivation, $w = ab$, where $a \in \Sigma^*$ and $b \in V^*$.

36.2 Example

Consider the grammar below

$S \to aAB|aB$
$A \to aAB|aB$
$B \to b$

We can make an extended PDA to capture the same language (which in this case is $L(G) = \{a^i b^i | i > 0\}$). The variables will be the contents of the stack. Everything S goes to will be a transition from the initial state, and everything else will be a loop on the accepting state. Once the stack is empty, the derivation is completed.



36.3 $\supseteq$

36.4 More Languages

From here, the natural question is, are there languages even more powerful than CFLs? It turns out that the answer is still "yes", and an example is the language $L = \{a^i b^i c^i | i > 0\}$. To show things like this, we will want a Pumping Lemma for CFL.

37 A New Pumping Lemma

Given a language $L \in$ CFL, $\exists$ a CFG G in Chomsky Normal Form s.t. $L(G) = L$. Recall that CNF allows only the following productions.

(1) $S \to \lambda$ (only the starting variable can go to λ, and this only shows up is $\lambda \in L(G)$)

(2) $A \to a$

(3) $A \to BC$, $B, C \neq S$

Derivation trees will always be binary trees (sort of... nodes have either 0, 1, or 2 children). Given a derivation $S \stackrel{*}{\Rightarrow} w$, $w \in \Sigma^*$, construct the derivation tree. If the DT has depth n, then $|w| \leq 2^{n-1}$. This also means that, if $|w| \geq 2^n$, then the depth of the DT is at least $n+1$.

37.1 Pumping Lemma for Context Free Languages

Let $L \in$ CFL. There is a number k (depending on L) such that: if $z \in L$, $|z| \geq k$, then $z = uvwxy$ satisfying

(1) $|v| + |x| > 0$

(2) $|vwx| \leq k$

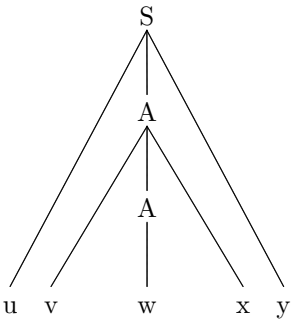
(3) $uv^iwx^iy \in L \ \forall i \in \mathbb{N}$

This is very similar to the Pumping Lemma for Regular Languages. However, there are some key differences. In the one for regular languages, there is an underlying DFA with k states. In this one, our k depends on the language, and not on any underlying machine.

37.2 How to Pick k

When we dealt with regular languages, we picked our k based on the number of states in a DFA accepting L. For CFLs, we instead look towards grammars. Specifically those in CNF. If L is a CFL, there is a grammar G in CNF s.t. $L(G) = L$. G has a set V of variables. Let $k = 2^{|V|}$. Now we look at how we can use the new Pumping Lemma.

Given a CFL L, pick a $z \in L$ with $|z| \geq k$. The derivation tree for $S \stackrel{*}{\Rightarrow} z$ has depth at least $|V| + 1$. Because the depth is greater than $|V|$, there is a variable $A \in V$ such that A appears at least twice in the tree. The (simplified) derivation tree will look like the following.



With the PL for regular languages, we could require the loop to be in the first k symbols of the string. Here, we will define a "last repitition" of A such that the subtree formed by its descendants have no more occurrences of A. Then, we will look at $S \stackrel{*}{\Rightarrow} uAy$ to have a next-to-last repitition of A, and the subderivation $A \stackrel{+}{\Rightarrow} vAx$ gives us the last repitition. This subderivation can be pumped any number of times, and that's what we will pump. From this, we get that $uv^iwx^iy \in L$. Because we required the A's that we show in the tree to be the last A's, $|vwx|$ must be less than k, otherwise it would be able to be repeated. We also are guaranteed that $|vx| > 0$, because λ productions and variable-to-variable productions are not allowed in CNF.

37.3 Example

Prove that $L = \{a^ib^ic^i | i \geq 0\} \notin$CFL.

First, assume for the sake of contradiction that L is context free. Thus, there exists a k for the L we can use for the PL. Consider $z = a^kb^kc^k \in L$. $|z| = 3k \geq k$, so $z = uvxyw$ exists to satisfy the three requirements of the PL. We now examine possible cases based on what v and x are.

Case 1: If either v or x contains two different symbols (e.g. $v = a^ib^j$), then $uv^2wx^2y \notin L$, because it will use a symbol it has previously used (in the given example, $a^kb^ja^ib^k$ is a prefix of uv^2wx^2y).

Case 2: If each contains exactly one symbol, then $uwy \notin L$, since the number of a's, b's, and c's can't all decrease by the same amount (at least one of them doesn't decrease, and at least one decreases by at least 1).

38 Closure Properties of CFLs

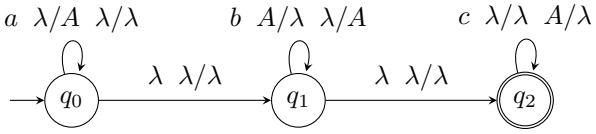
Similar to regular languages, CFLs are closed under union ($S \to S_1|S_2$), concatenation ($S \to S_1S_2$), and Kleene star ($S \to S_1S|\lambda$). Also, given languages $R \in$REG and $C \in$CFL, $R \cap C \in$CFL.

39 Power-Adding Variants

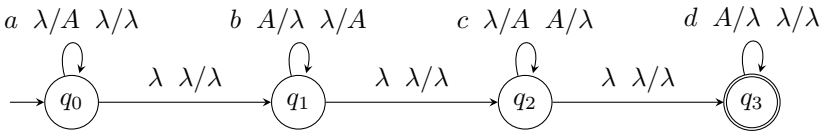
Now we talk about 2-stack PDAs, which can accept languages such as $\{a^ib^ic^i|i \geq 0\}$, which is not context free. We will see that these are as powerful as Turing Machines. 2-stack PDAs accept languages according to the criteria

$$L(M) = \left\{ w \middle| \exists [q_0, w, \lambda, \lambda] \overset{*}{\vdash} [q_f, \lambda, \lambda, \lambda], q_f \in F \right\}$$

A machine to accept that language is as follows:



Now, of course, the obvious language to consider is $\{a^ib^ic^id^i|i \geq 0\}$. This turns out to not be too difficult.



It turns out that 2-stack PDAs accept everything that a finite machine can accept, and they are equivalent to Turing Machines.

40 Chomsky Types of Languages

There is a heirarchy of languages we've been talking about. There's a small chart to describe these languages.

Languages	Machines	Chomsky Type
RE	2-stack PDA, TM	0
CSL	LBA	1
CFL	PDA	2
REG	DFA, NFA, NFA-λ	3

RE languages are Recursively Enumerable. This course won't talk about type 1 languages, so called context-sensitive languages, which can be accepted by linearly-bounded automata.

41 Turing Machines

In a Turing Machine (TM), we have an input tape. The tape is something we can write to, and we can go both to the right and to the left, which is something that we couldn't do with previous machines. The input will start at position 1 of the input tape, and the machine will start reading from position 0. A TM has Σ, an input alphabet; Γ, a tape alphabet; $B \in \Gamma$, a "blank symbol"; Q, a set of states; $q_0 \in Q$, a starting state; sometimes $F \subseteq Q$, a set of accepting states; and a function $\delta : Q \times (\Sigma \cup \Gamma) \to Q \times (\Sigma \cup \Gamma) \times \{L, R\}$. The reason we only sometimes have F is that we have two types of TMs: some TMs accept languages, and some TMs compute things. A typical transition will look like $\delta(q_i, x) = [q_j, y, d]$, where we go from state q_i to q_j, we read x from the tape, write y to the tape, and move in direction d on the tape. An initial configuration will look like this:

B	a	b	b	a	a	B	B	$\cdots$

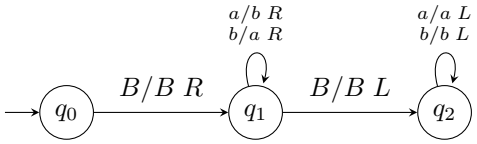
We want to decide what lets us terminate with this sort of machine. There are a few ways for this to happen. We will say that the TM halts (good, normal termination) if $\delta(q_i, x)$ is **NOT** specified. For NFAs, this was a bad way to terminate, but it is a way to accept things in TMs. If we have a computing TM, we are trying to compute $f(w)$, where w is our input. When this happens, we are trying to halt with $f(w)$ written on the tape. If we are accepting a language, we want to halt in an accepting state to say that the input is part of the language. Sometimes, a TM will not halt. One possibility is an infinite loop. A computation could move left and right on the tape without changing anything, and will never terminate. We could also try to read to the left of position 0 on the tape which is abnormal termination. **This is not considered halting.** In this case, our transition was defined and told us to move left, but we could not move left from position 0. Halting is only when $\delta(q_i, x)$ is not defined.

41.1 Example

We can again write things out with either a chart or a state machine. We'll first look at a chart for a TM that computes a function.

δ	B	a	b
q_0	$[q_1, B, R]$		
q_1	$[q_2, B, L]$	$[q_1, b, R]$	$[q_1, a, R]$
q_2		$[q_2, a, L]$	$[q_2, b, L]$

This machine will interchange the symbols a and b and then return to position 0. This is an example of a TM that computes a function The state machine looks like



To trace a computation of a TM we use configurations. Our configurations look like uq_ivB, where uv is a portion of the tape, starting at the left, such that everything to the right of it is blank. The machine at that configuration is reading the first symbol in v, that is, the tape head is at the position of the first symbol in v. The computation of aba for the above machine is

q_0BabaB
$\vdash Bq_1abaB$
$\vdash Bbq_1baB$
$\vdash Bbaq_1aB$
$\vdash Bbabq_1B$
$\vdash Bbaq_2bB$
$\vdash Bbq_2abB$
$\vdash Bq_2babB$
$\vdash q_2BbabB$
$\vdash$HALT

The language of an accepting TM is the set of strings for which the computation halts in an accepting state. A language L is RE if $\exists M$ such that $L(M) = L$. If $w \notin L(M)$, any of the following can happen.

(1) We halt in an accepting state.

(2) We do not halt.

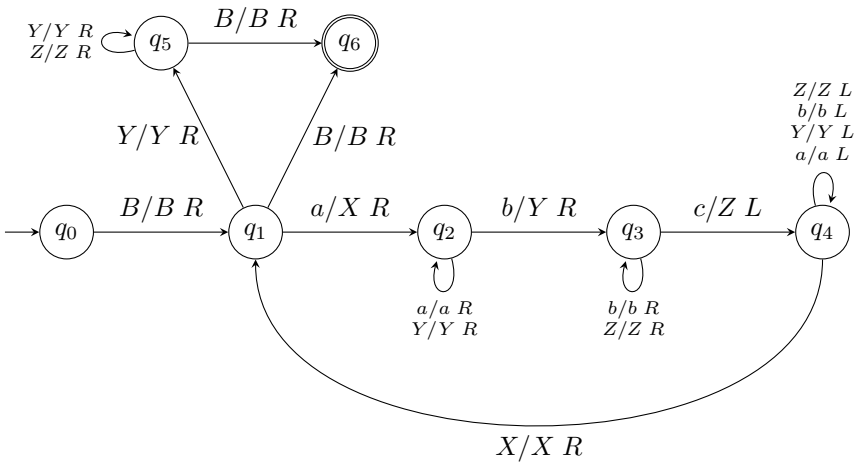
(2a) We never terminate

(2b) We have an abnormal termination

We say L is recursive if there exists a TM M such that M always halts and $L(M) = M$. We sometimes also call this "decidable". Remember that some TMs won't satisfy the requirement to always halt, but that doesn't mean that the languages they accept aren't recursive. Those languages might have other TMs which do always halt which accept them.

41.2 Example

Let's show that $L = \{a^i b^i c^i | i \geq 0\}$ is recursive. A natural thing to try is to read an a, then a b, then a c, and then go back to the beginning. To do that, we will mark an a, skip ahead to the next b, mark it, then skip ahead to the next c, and mark it. We'll mark these letters by replacing them on the tape with X, Y, and Z, respectively. After we mark a c, we run to the left on the tape until we find an X (the most recently marked a), at which point we go one spot to the right and repeat. To accept, once we get back to the beginning of the loop, if we can't mark an a, we make certain nothing is unmarked, and then accept. Lastly, we need to be sure to accept λ. Here's the machine.



42 Variants of the Standard Turing Machine

The first variation is the accept by halting only TM (HOTM). This variant has no accepting states, and accepts any string if the computation of that string halts. For all other strings, the computation either has an abnormal termination or a loop.

TODO: Flesh out these notes, as they're pretty suck right now.

HOTM$\Leftrightarrow$STM.

$\Rightarrow$: make each state accepting. $\Leftarrow$: for each non-accepting state, make transitions for all undefined symbols to a new state q_f. At q_f, make transitions to read each symbol, go to the left on the tape, and stay at the same state. Now let all the states no longer be accepting states.

43 Multitrack TM

This has multiple tracks, all read at the same time. $\delta(q_i, [a, b]) = \big[q_i, [c, d], D\big]$ (where $D \in \{L, R\}$). The extra track adds no power.

$\Leftarrow$: ignore the second track. $\Rightarrow$: have a larger tape alphabet such that $[c, d] \in \Gamma$. Each symbol in the tape alphabet is a vector of symbols from the original tape alphabet.

44 Two-Way TM

This TM isn't bounded to the left, and can't have abnormal terminations. It's also equally powerful.

$\Leftarrow$: put a special symbol X at position -1. For each state, define $\delta(q_i, X) = q_{REJECT}$. $\Rightarrow$: make a two-track TM and let the second track be negative positions. Have two copies of each state to "keep track" of which track to read. Have a special symbol on track 2 at position 0.

END TODO.

45 Hey Look! A Missing Day!

I missed this class. It was Friday, February 27.

46 Multitape TM

Similar to the multitrack TM, but our tapes can move in different directions. This lets us be at different positions in both. A transition looks like

$$\delta(q_i, x_1, x_2, \ldots, x_k) = [q_j, y_1, d_1, y_2, d_2, \ldots, y_k, d_k], \quad d \in \{L, R, S\}$$

The same languages are accepted by Multitape TMs and standard TMs. Starting with a Multitape TM with k tracks, we can construct a Multitrack TM with $2k + 1$ tracks. k of the tracks will have whatever the k tapes have. Another k tracks will be blank everywhere except for a marking symbol (X) to represent where the tape head is on that track. The last track marks position 0. A transition from a multitape TM becomes the following transitions in the multitrack TM

(i) Move right until find marker for first tape.

(ii) Keep track of first symbol by changing states.

(iii) Move back to position 0.

(iv) Repeat above three steps for each remaining tape.

(v) "Do" the transition.

(vi) Move right until find marker for first tape.

(vii) Change the symbol on the tape.

(viii) Move the marker if needed.

(ix) Move back to position 0.

(x) Repeat above four steps for each remaining tape.

47 Hey Look! A Missing Day!

Tuesday, March 3's notes are currently written by hand, but students want what I have so far. This day covered time complexity and P vs. NP. Also NP-hard

and NP-complete. We know (trivially) that P$\subseteq$NP. Most people think P$\neq$NP, and all of our online security resides in NP. However, lots of people think that NP$\cap$coNP=P, and all of our online security is in NP$\cap$coNP. Seems like not a big deal. Afterall, proving that one isn't worth a million bucks. It's just probably easier and equally worrying.