III. Supplementary Materials

The Three Hour Tour Through Automata Theory

Analyzing Problems as Opposed to Algorithms

In CS336 you learned to analyze ***algorithms***. This semester, we're going to analyze ***problems***.

We're going to see that there is a hierarchy of problems: easy, hard, impossible.

For each of the first two, we'll see that for any problem, there are infinitely many programs to solve the problem.

> By the way, this is trivial. Take one program and add any number of junk steps. We'll formalize this later once we have some formalisms to work with.

Some may be better than others. But in some cases, we can show some sort of boundary on how good an algorithm we can attempt to find for a particular problem.

Let's Look at Some Problems

Let's look at a collection of problems, all which could arise in considering one piece of C++ code. **[slide - Let's Look at Some Problems]**

As we move from problem 1 to problem 5, things get harder in two key ways. The first is that it seems we'll need more complicated, harder to write and design programs. In other words, it's going to take more time to write the programs. The second is that the programs are going to take a lot longer to run. As we get to problems 4 and 5, it's not even clear there is a program that will do the job. In fact, in general for problem 4 there isn't. For problem 5, in general we can't even get a formal statement of the problem, much less an algorithmic solution.

Languages

Characterizing Problems as Language Recognition Tasks

In order to create a formal theory of problems (as opposed to algorithms), we need a single, relatively straightforward framework that we can use to describe any kind of possibly computable function. The one we'll use is language recognition.

What is a Language?

A language is a set of strings over an alphabet, which can be any *finite* collection of symbols. **[Slide - Languages]**

Defining a Problem as a Language Recognition Task

We can define any problem as a language recognition task. In other words, we can output just a boolean, True or False. Some problems seem naturally to be described as recognition tasks. For example, accept grammatical English sentences and reject bad ones. (Although the truth is that English is so squishy it's nearly impossible to formalize this. So let's pick another example -- accept the syntactically valid C programs and reject the others.)

Problems that you think of more naturally as functions can also be described this way. We define the set of input strings to consist of strings that are formed by concatenating an input to an output. Then we only accept strings that have the correct output concatenated to each input. **[Slide - Encoding Output]**

Branching Out -- Allowing for Actual Output

Although it is simpler to characterize problems simply as recognition problems and it is possible to reformulate functional problems as recognition problems, we will see that we can augment the formalisms we'll develop to allow for output as well.

Defining Languages Using Grammars

Now what we need is a general mechanism for defining languages. Of course, if we have a finite language, we can just enumerate all the strings in it. But most interesting languages are infinite. What we need is a *finite* mechanism for specifying *infinite* languages. Grammars can do this.

The standard way to write a grammar is as a production system, composed of rules with a left hand side and a right hand side. Each side contains a sequence of symbols. Some of these symbols are terminal symbols, i.e., symbols in the language we're defining. Others are drawn from a finite set of nonterminal symbols, which are internal symbols that we use just to help us define the language. Of these nonterminal symbols, one is special -- we'll call it the start symbol. **[Slide - Grammars 1]**

If there is a grammar that defines a language, then there is an infinite number of such grammars. Some may be better, from various points of view than others. Consider the grammar for odd integers. What different grammars could we write? One thing we could do would be to introduce the idea of odd and even digits. **[Slide - Grammars 2]**

Sometimes we use single characters, disjoint from the characters of the target language, in our rules. But sometimes we need more symbols. Then we often use < and > to mark multiple character nonterminal symbols. **[Slide - Grammars 3]**

Notice that we've also introduced a notation for OR so that we don't have to write as many separate rules. By the way, there are lots of ways of writing a grammar of arithmetic expressions. This one is simple but it's not very good. It doesn't help us at all to determine the precedence of operators. Later we'll see other grammars that do that.

Grammars as Generators and as Acceptors

So far, we've defined problems as language recognition tasks. But when you look at the grammars we've considered, you see that there's a sense in which they seem more naturally to be generators than recognizers. If you start with S, you can generate all the strings in the language defined by the grammar. We'll see later that we'll use the idea of a grammar as a generator (or an enumerator) as one way to define some interesting classes of languages.

But you can also use grammars as acceptors, as we've suggested. There are two ways to do that. One is *top-down*. By that we mean that you start with S, and apply rules. **[work this out for a simple expression for the Language of Simple Arithmetic Expressions]** At some point, you'll generate a string without any nonterminals (i.e., a string in the language). Check and see if it's the one you want. If so accept. If not, try again. If you do this systematically, then if the string is in the language, you'll eventually generate it. If it isn't, you may or may not know when you should give up. More on that later.

The other approach is bottom up. In this approach, we simply apply the rules sort of backwards, i.e., we run them from right to left, matching the string to the right hand sides of the rules and continuing until we generate S and nothing else. **[work this out for a simple expression for the Language of Simple Arithmetic Expressions]** Again, there are lots of possibilities to consider and there's no guarantee that you'll know when to stop if the string isn't in the language. Actually, for this simple grammar there is, but we can't assure that for all kinds of grammars.

The Language Hierarchy

Remember that our whole goal in this exercise is to describe classes of problems, characterize them as easy or hard, and define computational mechanisms for solving them. Since we've decided to characterize problems as languages to be recognized, what we need to do is to create a language hierarchy, in which we start with very simple languages and move toward more complex ones.

Regular Languages

Regular languages are very simple languages that can be defined by a very restricted kind of grammar. In these grammars, the left side of every rule is a single nonterminal and the right side is a single terminal optionally followed by a single nonterminal. **[slide - Regular Grammars]** If you look at what's going on with these simple grammars, you can see that as you apply rules, starting with S, you generate a terminal symbol and (optionally) have a new nonterminal to work with. But you can never end up with multiple nonterminals at once. (Recall our first grammar for Odd Integers [**slide - Grammars 1**]).

Of course, we also had another grammar for that same language that didn't satisfy this restriction. But that's okay. If it is possible to define the language using the restricted formalism, then it falls into the restricted class. The fact that there are other, less restricted ways to define it doesn't matter.

It turns out that there is an equivalent, often useful, way to describe this same class of languages, using regular expressions. **[Slide Regular Expressions and Languages]** Regular expressions don't look like grammars, in the sense that there are no production rules, but they can be used to define exactly the same set of languages that the restricted class of regular grammars can define. Here's a regular expression for the language that consists of odd integers, and one for the language of identifiers. We can try to write a regular expression for the language of matched parenthesis, but we won't succeed.

Intuitively, regular languages are ones that can be defined without keeping track of more than a finite number of things at once. So, looking back at some of our example languages **[slide Languages]**, the first is regular and none of the others is.

Context Free Languages

To get more power in how we define languages, we need to return to the more general production rule structure.

Suppose we allow rules where the left hand side is composed of a single symbol and the right hand side can be anything. We then have the class of context-free grammars. We define the class of context-free languages to include any language that can be generated by a context-free grammar.

The context-free grammar formalism allows us to define many useful languages, including the languages of matched parentheses and of equal numbers of parentheses but in any order **[slide - Context-Free Grammars]**. We can also describe the language of simple arithmetic expressions **[slide - Grammars 3]**.

Although this system is a lot more powerful (and useful) than regular languages are, it is not adequate for everything. We'll see some quite simple artificial language it won't work for in a minute. But it's also inadequate for things like ordinary English. [**slide - English Isn't Context-Free]**.

Recursively Enumerable Languages

Now suppose we remove all restrictions from the form of our grammars. Any combination of symbols can appear on the left hand side and any combination of symbols can appear on the right. The only real restriction is that there can be only a finite number of rules. For example, we can write a grammar for the language that contains strings of the form $a^n b^n c^n$. **[slide - Unrestricted Grammars]**

Once we remove all restrictions, we clearly have the largest set of languages that can be generated by any finite grammar. We'll call the languages that can be generated in this way the class of recursively enumerable languages. This means that, for any recursively enumerable language, it is possible, using the associated grammar, to generate all the strings in the language. Of course, it may take an infinite amount of time if the language contains an infinite number of strings. But any given string, if it is enumerated at all, will be enumerated in a finite amount of time. So I guess we could sit and wait. Unfortunately, of course, we don't know how long to wait, which is a problem if we're trying to decide whether a string is in the language by generating all the strings and seeing if the one we care about shows up.

Recursive Languages

There is one remaining set of languages that it is useful to consider. What about the recursively enumerable languages where we could guarantee that, after a finite amount of time, either a given string would be generated or we would know that it isn't going to be. For example, if we could generate all the strings of length 1, then all the strings of length 2, and so forth, we'd either generate the string we want or we'd just wait until we'd gone past the length we cared about and then report failure. From a practical point of view, this class is very useful since we like to deal with solutions to problems that are guaranteed to halt. We'll call this class of languages the recursive languages. This means that we can not only generate the strings in the language, we can actually, via some algorithm, decide whether a string is in the language and halt, with an answer, either way.

Clearly the class of recursive languages is a subset of the class of recursively enumerable ones. But, unfortunately, this time we're not going to be able to define our new class by placing syntactic restrictions on the form of the grammars we use. There are some useful languages, such, as $a^n b^n c^n$, that are recursive. There are some others, unfortunately, that are not.

The Whole Picture

[Slide - The Language Hierarchy]

Computational Devices

Formal Models of Computational Devices

If we want to make formal statements about the kinds of computing power required to solve various kinds of problems, then we need simple, precise models of computation.

We're looking for models that make it easy to talk about what can be computed -- we're not worrying about efficiency at this point.

When we described languages and grammars, we saw that we could introduce several different structures, each with different computational power. We can do the same thing with machines. Let's start with really simple devices and see what they can do. When we find limitations, we can expand their power.

Finite State Machines

The only memory consists of the ability to be in one of a finite number of states. The machine operates by reading an input symbol and moving to a new state that is determined solely by the state it is in and the input that it reads. There is a unique start state and one or more final states. If the input is exhausted and the machine is in a final state, then it accepts the input. Otherwise it rejects it.

Example: An FSM to accept odd integers. **[Slide - Finite State Machines 1]**

Example: An FSM to accept valid identifiers. **[Slide - Finite State Machines 2**]

Example: How about an FSM to accept strings with balanced parentheses?

Notice one nice feature of every finite state machine -- it will always halt and it will always provide an answer, one way or another. As we'll see later, not all computational systems offer these guarantees.

But we've got this at a price. There is only a finite amount of memory. So, for example, we can't count anything. We need a stronger device.

Push Down Automata

Deterministic PDAs

Add a single stack to an FSM. Now the action of the machine is a function of its state, the input it reads, and the values at the top of the stack.

Example: A PDA to accept strings with balanced parentheses. **[Slide - Push Down Automata]** Notice that this really simple machine only has one state. It's not using the states to remember anything. All its memory is in the stack.

Example: A PDA to accept strings of the form $w\#w^R$, where $w \in \{a,b\}^*$ **[slide - Pushdown Automaton 2]**.

Example: How about a PDA to accept strings with some number of a's, followed by the same number of b's, followed by the same number of c's? **[slide - PDA 3]** It turns out that this isn't possible. The problem is that we could count the a's on the stack, then pop for b's. But how could we tell if there is the right number of c's. We'll see in a bit that we shouldn't be too surprised about this result. We can create PDA's to accept precisely those languages that we can generate with CFGs. And remember that we had to use an unrestricted grammar to generate this language.

Nondeterministic PDAs

Example: How about a PDA to accept strings of the form $w\, w^R$? We can do this one if we expand our notion of a PDA to allow it to be nondeterministic. The problem is that we don't know when to imagine that the reversal starts. What we need to do is to guess. In particular, we need to try it at every point. We can do this by adding an epsilon transition from the start state (in which we're pushing w) to the final state in which we're popping as we read w^R.**[slide - A Nondeterministic PDA]** Adding this kind of nondeterminism actually adds power to the PDA notion. And actually, it is the class of nondeterministic PDA's that is equivalent to the class of context-free languages. No surprise, since we were able to write a context free grammar for this language

By the way, it also makes sense to talk about nondeterministic finite state machines. But it turns out that adding nondeterminism to finite state machines doesn't increase the class of things they can compute. It just makes it easier to describe some machines. Intuitively, the reason that nondeterminism doesn't buy you anything with finite state machines is that we can simulate a nondeterministic machine with a deterministic machine. We just make states that represent sets of states in the nondeterministic machine. So in essence, we follow all paths. If one of them accepts, we accept.

Then why can't we do that with PDA's? For finite state machines, there must be a finite number of states. DAH. So there is a finite number of subsets of states and we can just make them the states of our new machine. Clunky but finite. Once we add the stack, however, there is no longer a finite number of states of the total machine. So there is not a finite number of subsets of states. So we can't simulate being in several states at once just using states. And we only have one stack. Which branch would get it? That's why adding nondeterminism actually adds power for PDAs.

Turing Machines

Clearly there are still some things we cannot do with PDAs. All we have is a single stack. We can count one thing. If we need to count more than one thing (such as a's and b's in the case of languages defined by $a^n b^n c^n$), we're in trouble.

So we need to define a more powerful computing device. The formalism we'll use is called the Turing Machine, after its inventor, Alan Turing. There are many different (and equivalent) ways to write descriptions of Turing Machines, but the basic idea is the same for all of them **[slide - Turing Machines]**. In this new formalism, we allow our machines to write onto the input tape. They can write on top of the input. They can also write past the input. This makes it easier to define computation that actually outputs something besides yes or no if we want to. But, most importantly, because we view the tape as being of infinite length, all limitations of finiteness or limited storage have been removed, even though we continue to retain the core idea of a finite number of states in the controller itself.

Notice, though, that Turing Machines are not guaranteed to halt. Our example one always does. But we could certainly build one that scans right until it finds a blank (writing nothing) and then scans left until it finds the start symbol and then scans right again and so forth. That's a legal (if stupid) Turing Machine. Unfortunately, (see below) it's not always possible to tell, given a Turning Machine, whether it is guaranteed to halt. This is the biggest difference between Turing Machines and the FSMs and PDAs, both of which will always halt.

Extensions to Turing Machines

You may be thinking, wow, this Turing Machine idea sure is restrictive. For example, suppose we want to accept all strings in the simple language {w # w^R}. We saw that this was easy to do in one pass with a pushdown automaton. But to do this with the sort of Turing Machine we've got so far would be really clunky. **[work this out on a slide]** We'd have to start at the left of the string, mark a character, move all the way to the right to find the corresponding character, mark it, scan back left, do it again, and so forth. We've just transformed a linear process into an n^2 one.

But suppose we had a Turing Machine with 2 tapes. The first thing we'll do is to copy the input onto the second tape. Now start the read head of the first tape at the left end of the input and the read head of the second tape at the right end. At each step in the operation of the machine, we check to make sure that the characters being read on the two tapes are the same. And we move the head on tape 1 right and the head on tape 2 to the left. We run out of input on both machines at the same time, we accept. **[slide -A Two Head Turing Machine]**

The big question now is, "Have we created a new notational device, one that makes it easier to describe how a machine will operate, or have we actually created a new kind of device with more power than the old one? The answer is the former. We can prove that by showing that we can simulate a Turing Machine with any finite number of tapes by a machine that computes the same thing but only has one tape. **[slide - Simulating k Heads with One]** The key idea here is to use the one tape but to think of it has having some larger number of tracks. Since there is a finite tape alphabet, we know that we can encode any finite number of symbols in a finite (but larger) symbol alphabet. For example, to simulate our two headed machine with a tape alphabet of 3 symbols plus start and blank, we will need 2*2*5*5 or 100 tape symbols. So to do this simulation, we must do two main things: Encode all the information from the old, multi-tape machine on the new, single tape machine and redesign the finite state controller so that it simulates, in several moves, each move of the old machine.

It turns out that any "reasonable" addition you can think of to our idea of a Turing Machine is implementable with the simple machine we already have. For example, any nondeterministic Turing Machine can be simulated by a deterministic one. This is really significant. In this, in some ways trivial, machine, we have captured the idea of computability.

Okay, so our Turing Machines can do everything any other machine can do. It also goes the other way. We can propose alternative structures that can do everything our Turing Machines can do. For example, we can simulate any Turing Machine with a deterministic PDA that has two stacks rather than one. What this machine will do is read its input tape once, copying onto the first stack all the nonblank symbols. Then it will pop all those symbols off, one at a time, and move them to the second stack. Now it can move along its simulated tape by transfering symbols from one stack to the other. **[slide - Simulating a Turing Machine with Two Stacks]**

The Universal Turing Machine

So now, having shown that we can simulate anything on a simple Turing Machine, it should come as no surprise that we can design a Turing Machine that takes as its input the definition of another Turing Machine, along with an input for that machine. What our machine does is to simulate the behavior of the machine it is given, on the given input.

Remember that to simulate a k-tape machine by a 1 tape machine we had first to state how to encode the multiple tapes. Then we had to state how the machine would operate on the encoding. We have to do the same thing here. First we need to decide how to encode the states and the tape symbols of the input machine, which we'll call M. There's no upper bound on how many states or tape symbols there will be. So we can't encode them with single symbols. Instead we'll encode states as strings that start with a "q" and then have a binary encoding of the state number (with enough leading zeros so all such encodings take the same number of digits). We'll encode tape symbols as an "a" followed by a binary encoding of the count of the symbol. And we'll encode "move left" as 10, "move right" as 01, and stay put as 00. We'll use # as a delimiter between transitions. **[slide - Encoding States, Symbols, and Transitions]**

Next, we need a way to encode the simulation of the operation of M. We'll use a three tape machine as our Universal Turing Machine. (Remember, we can always implement it on a one tape machine, but this is a lot easier to describe.) We'll use one tape to encode the tape of M, the second tape contains the encoding of M, and the third tape encodes the current state of M during the simulation. **[slide - The Universal Turing Machine]**

A Hierarchy of Computational Devices

These various machines that we have just defined, fall into an inclusion hierarchy, in the sense that the simpler machines can always be simulate by the more powerful ones. **[Slide - A Machine Hierarchy]**

The Equivalence of the Language Hierarchy and the Computational Hierarchy

Okay, this probably comes as no surprise. The machine hierarchy we've just examined exactly mirrors the language hierarchy. **[Slide - Languages and Machines]**

Actually, this is an amazing result. It seems to suggest that there's something quite natural about these categories.

Church's Thesis

If we really want to talk about naturalness, can we say anything about whether we've captured what it means to be computable? Church's Thesis (also sometimes called the Church-Turing Thesis) asserts that the precise concept of the Turing Machine that halts on all inputs corresponds to the intuitive notion of an algorithm. Think about it. Clearly a Turing Machine that halts defines an algorithm. But what about the other way around? Could there be something that is computable by some kind of algorithm that is not computable by a Turing Machine that halts? From what we've seen so far, it may seem unlikely, since every extension we can propose to the Turing Machine model turns out possibly to make things more convenient, but it never extends the formal power. It turns out that people have proposed various other formalisms over the last 50 years or so, and they also turn out to be no more powerful than the Turing Machine. Of course, something could turn up, but it seems unlikely.

Techniques for Showing that a Problem (or Language) Is Not in a Particular Circle in the Hierarchy

Counting

$a^n b^n$ is not regular.

Closure Properties

L = { $a^n b^m c^p$: $m \neq n$ or $m \neq p$ } is not deterministic context-free. **[slide - Using Closure Properties]** Notice that L' contains all strings that violate at least one of the requirements for L. So they may be strings that aren't composed of a string of a's, followed by a string of b's, followed by a string of c's. Or they may have that property but they violate the rule that m, n, and p cannot all be the same. In other words, they are all the same. So if we intersect L' with the regular expression a*b*c*, we throw away everything that isn't a string of a's then b's then c's, and we're left with strings of n a's, followed, by n b's, followed by n c's.

Diagonalization

Remember the proof that the power set of the integers isn't countable. If it were, there would be a way of enumerating the sets, thus setting them in one to one correspondence with the integers. But suppose there is such a way **[slide - Diagonalization]**. Then we could represent it in a table where element (i, j) is 1 precisely in case the number j is present in the set i. But now construct a new set, represented as a new row of the table. In this new row, element i will be 1 if element (i,i) of the original table was 0, and vice versa. This row represents a new set that couldn't have been in the previous enumeration. Thus we get a contradiction and the power set of the integers must not be countable.

We can use this technique for perhaps the most important result in the theory of computing.

The Unsolvability of the Halting Problem

There are recursively enumerable languages that are not recursive. In other words, there are sets that can be enumerated, but there is no decision procedure for them. Any program that attempts to decide whether a string is in the language may not halt.

One of the most interesting such sets is the following. Consider sets of ordered pairs where the first element is a description of a Turing Machine. The second element is an input to the machine. We want to include only those ordered pairs where the machine halts on the input. This set is not recursively enumerable. In other words, there's no way to write an algorithm that, given a machine and an input, determines whether or not the machine halts on the input. **[slide - The Unsolvability of the Halting Problem]**

Suppose there were such a machine. Let's call it HALTS. HALTS(M,x) returns true if Turing machine M halts on input x. Otherwise it returns false. Now we write a Turing Machine program that implements the TROUBLE algorithm. Now what happens if we invoke HALTS(TROUBLE,TROUBLE)? If HALTS says true, namely that TROUBLE will halt on itself, then TROUBLE loops (i.e., it doesn't halt, thus contradicting our assumption that HALTS could do the job). But if HALTS says FALSE, namely that TROUBLE will not halt on itself, then TROUBLE promptly halts, thus again proving our supposed oracle HALTS wrong. Thus HALTS cannot exist.

We've used a sort of stripped down version of diagonalization here **[slide - Viewing the Halting Problem as Diagonalization]** in which we don't care about the whole row of the item that creates the contradiction. We're only invoking HALTS with two identical inputs. It's just the single element that we care about and that causes the problem.

Let's Revisit Some Problems

Let's look again at the collection of problems that we started this whole process with. **[slide - Let's Revisit Some Problems]**

Problem 1 can be solved with a finite state machine.

Problem 2 can be solved with a PDA.

Problem 3 can be solved with a Turing Machine.

Problem 4 can be semi solved with a Turning Machine, but it isn't guaranteed to halt.

Problem 5 can't even be stated.

So What's Left?

Review of Mathematical Concepts

1 Sets

1.1 What is a Set?

A ***set*** is simply a collection of objects. The objects (which we call the ***elements*** or ***members*** of the set) can be anything: numbers, people, fruits, whatever. For example, all of the following are sets:

A = {13, 11, 8, 23}
B = {8, 23, 11, 13}
C = {8, 8, 23, 23, 11, 11, 13, 13}
D = {apple, pear, banana, grape}
E = {January, February, March, April, May, June, July, August, September, October, November, December}
F = {x : x $\in$ E and x has 31 days}
G = {January, March, May, July, August, October, December}
N = the nonnegative integers (We will generally call this set N, the ***natural numbers***.)
H = {i : $\exists$x $\in$ N and i = 2x}
I = {0, 2, 4, 6, 8, …}
J = the even natural numbers
K = the syntactically valid C programs
L = {x : x $\in$ K and x never gets into an infinite loop}
Z = the integers (… -3, -2, -1, 0, 1, 2, 3, …)

In the definitions of F and H, we have used the colon notation. Read it as "such that". We've also used the standard symbol $\in$ for "element of". We will also use $\notin$ for "not an element of". So, for example, 17 $\notin$ A is true.

Remember that a set is simply a collection of elements. So if two sets contain precisely the same elements (regardless of the way we actually defined the sets), then they are identical. Thus F and G are the same set, as are H, I, and J.

An important aside: **Zero is an even number.** This falls out of any reasonable definition we can give for even numbers. For example, the one we used to define set H above. Or consider: 2 is even and any number that can be derived by adding or subtracting 2 from an even number is also even. In order to construct a definition for even numbers that does not include zero, we'd have to make a special case. That would make for an inelegant definition, which we hate. And, as we'll see down the road, we'd also have to make corresponding special cases for zero in a wide variety of algorithms.

Since a set is defined only by what elements it contains, it does not matter what order we list the elements in. Thus A and B are the same set.

Our definition of a set considers only whether or not an element is contained within the set. It does not consider how many times the element is mentioned. In other words, duplicates don't count. So A, B, and C are all equal.

Whenever we define a set, it would be useful if we could also specify a decision procedure for it. A ***decision procedure*** for a set S is an algorithm that, when presented with an object O, returns True if O $\in$ S and False otherwise. Consider set K above (the set of all syntactically valid C programs). We can easily decide whether or not an object is an element of K. First, of course, it has to be a string. If you bring me an apple, I immediately say no. If it is a string, then I can feed it to a C compiler and let it tell me whether or not the object is in K. But now consider the set L (C programs that are guaranteed to halt on all inputs). Again, I can reject apples and anything else that isn't even in K. I can also reject some programs that clearly do loop forever. And I can accept some C programs, for example ones that don't contain any loops at all. But what about the general problem. Can I find a way to look at an arbitrary C program and tell whether or not it belongs in L. It turns out, as we'll see later, that the answer to this is no. We can prove that no program to solve this problem can exist. But that doesn't mean that the set L doesn't exist. It's a perfectly fine set. There just isn't a decision procedure for it.

The smallest set is the set that contains no elements. It is called the ***empty set***, and is written ∅ or {}.

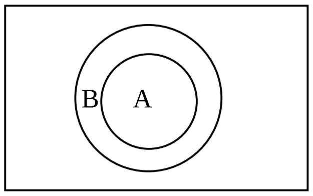
When you are working with sets, it is very important to keep in mind the difference between a set and the elements of a set. Given a set that contains more than one element, this not usually tricky. It's clear that {1, 2} is distinct from either the number 1 or the number 2. It sometimes becomes a bit trickier though with ***singleton sets*** (sets that contain only a single element). But it is equally true here. So, for example, {1} is distinct from the number 1. As another example, consider {∅}. This is a set that contains one element. That element is in turn a set that contains no elements (i.e., the empty set).

1.2 Relating Sets to Each Other

We say that A is a ***subset*** of B (which we write as $A \subseteq B$) if every element of A is also an element of B. The symbol we use for subset ($\subseteq$) looks somewhat like $\leq$. This is no accident. If $A \subseteq B$, then there is a sense in which the set A is "less than or equal to" the set B, since all the elements of A must be in B, but there may be elements of B that are not in A.

Given this definition, notice that every set is a subset of itself. This fact turns out to offer us a useful way to prove that two sets A and B are equal: First prove that A is a subset of B. Then prove that B is a subset of A. We'll have more to say about this later in Section 6.2.

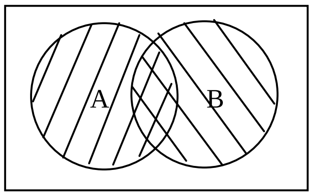
We say that A is ***proper subset*** of B (written $A \subset B$) if $A \subseteq B$ and $A \neq B$. The following Venn diagram illustrates the proper subset relationship between A and B:



Notice that the empty set is a subset of every set (since, trivially, every element of ∅, all none of them, is also an element of every other set). And the empty set is a *proper* subset of every set other than itself.

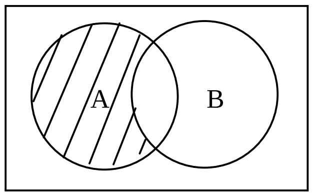
It is useful to define some basic operations that can be performed on sets:

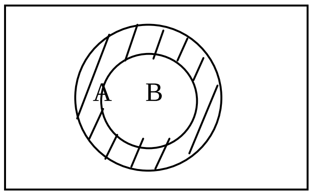
The ***union*** of two sets A and B (written $A \cup B$) contains all elements that are contained in A or B (or both). We can easily visualize union using a Venn diagram. The union of sets A and B is the entire hatched area:



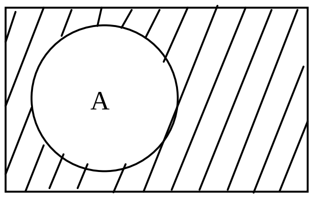
The ***intersection*** of two sets A and B (written $A \cap B$) contains all elements that are contained in both A and B. In the Venn diagram shown above, the intersection of A and B is the double hatched area in the middle.

The ***difference*** of two sets A and B (written A - B) contains all elements that are contained in A but not in B. In both of the following Venn diagrams, the hatched region represents A - B.

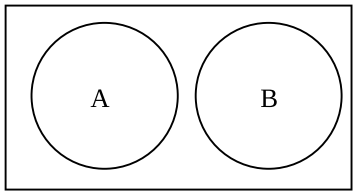




The ***complement*** of a set A with respect to a specific domain D (written as $\overline{A}$ or ¬A) contains all elements of D that are not contained in A (i.e, $\overline{A}$ = D - A). For example, if D is the set of residents of Austin and A is the set of Austin residents who like barbeque, then $\overline{A}$ is the set of Austin residents who don't like barbeque. The complement of A is shown as the hatched region of the following Venn diagram:



Two sets are ***disjoint*** if they have no elements in common (i.e., their intersection is empty). In the following Venn diagram, A and B are disjoint:



So far, we've talked about operations on pairs of sets. But just as we can extend binary addition and sum up a whole set of numbers, we can extend the binary operations on sets and perform then on sets of sets. Recall that for summation, we have the notation

$$\sum A_i$$

Similarly, we'll introduce

$$\bigcup A_i \qquad \text{and} \qquad \bigcap A_i$$

to indicate the union of a set of sets and the intersection of a set of sets, respectively.

Now consider a set A. For example, let A = {1, 2, 3}. Next, let's enumerate the set of all subsets of A:

{∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

We call this set the ***power set*** of A, and we write it 2^A. The power set of A is interesting because, if we're working with the elements of A, we may well care about all the ways in which we can combine those elements.

Now for one final property of sets. Again consider the set A above. But this time, rather than looking for all possible subsets, let's just look for a single way to carve A up into subsets such that each element of A is in precisely one subset. For example, we might choose any of the following sets of subsets:

{{1}, {2, 3}} or {{1, 3}, 2} or {{1, 2, 3}}

We call any such set of subsets a partition of A. Partitions are very useful. For example, suppose we have a set S of students in a school. We need for every student to be assigned to precisely one lunch period. Thus we must construct a partition of S: a set of subsets, one for each lunch period, such that each student is in precisely one subset. More formally, we say that Π is a ***partition*** of a set A if and only if (a) no element of Π is empty; (b) all members of Π are disjoint (alternatively, each element of A is in only one element of Π); and (c) $\bigcup \Pi = A$ (alternatively, each element of A is in some element of Π and no element not in A is in any element of Π).

This notion of partitioning a set is fundamental to programming. Every time you analyze the set of possible inputs to your program and consider the various cases that must be dealt with, you're forming a partition of the set of inputs: each input must fall through precisely one path in your program. So it should come as no surprise that, as we build formal models of computational devices, we'll rely heavily on the idea of a partition on a set of inputs as an analytical technique.

2 Relations and Functions

In the last section, we introduced some simple relations that can hold between sets (subset and proper subset) and we defined some operations (functions) on sets (union, intersection, difference, and complement). But we haven't yet defined formally what we mean by a relation or a function. Let's do that now. (By the way, the reason we introduced relations and functions on sets in the last section is that we're going to use sets as the basis for our formal definitions of relations and functions and we will need the simple operations we just described as part of our definitions.)

2.1 Relations

An ***ordered pair*** is a sequence of two objects. Given any two objects, x and y, there are two ordered pairs that can be formed. We write them as (x, y) and (y, x). As the name implies, in an ordered pair (as opposed to in a set), order matters (unless x and y happen to be equal).

The ***Cartesian product*** of two sets A and B (written A × B) is the set of all ordered pairs (a, b) such that a ∈ A and b ∈ B. For example, let A be a set of people {Dave, Sue, Billy} and let B be a set of desserts {cake, pie, ice cream}. Then

A × B = { (Dave, cake), (Dave, pie), (Dave, ice cream),
(Sue, cake), (Sue, pie), (Sue, ice cream),
(Billy, cake), (Billy, pie), (Billy, ice cream)}

As you can see from this example, the Cartesian product of two sets contains elements that represent all the ways of pairing someone from the first set with someone from the second. Note that A × B is not the same as B × A. In our example,

B × A = { (cake, Dave), (pie, Dave), (ice cream, Dave),
(cake, Sue), (pie, Sue), (ice cream, Sue),
(cake, Billy), (pie, Billy), (ice cream, Billy)}

We'll have more to say about the cardinality (size) of sets later, but for now, let's make one simple observation about the cardinality of a Cartesian product. If A and B are finite and if there are p elements in A and q elements in B, then there are p*q elements in A×B (and in B×A).

We're going to use Cartesian product a lot. It's our basic tool for constructing complex objects out of simpler ones. For example, we 're going to define the class of Finite State Machines as the Cartesian product of five sets. Each individual finite state machine then will be a five tuple (K, Σ , δ, s, F) drawn from that Cartesian product. The sets will be:

1. The set of all possible sets of states: {{q1}, {q1, q2}, {q1, q2, q3}, …}. We must draw K from this set.
2. The set of all possible input alphabets: {{a}, {a, b, c}, {α, β, γ}, {1, 2, 3, 4}, {1, w, h, j, k}, {q, a, f}, {a, β, 3, j, f}…}. We must draw Σ from this set.
3. The set of all possible transition functions, which tell us how to move from one state to the next. We must draw δ from this set.
4. The set of all possible start states. We must draw s from this set.
5. The set of all possible sets of final states. (If we land in one of these when we've finished processing an input string, then we accept the string, otherwise we reject.) We must draw F from this set.

Let's return now to the simpler problem of choosing dessert. Suppose we want to define a relation that tells us, for each person, what desserts he or she likes. We might write the Dessert relation, for example as

{(Dave, cake), (Dave, ice cream), (Sue, pie), (Sue, ice cream)}

In other words, Dave likes cake and ice cream, Sue likes pie and ice cream, and Billy hates desserts.

We can now define formally what a relation is. A ***binary relation*** over two sets A and B is a subset of A × B. Our dessert relation clearly satisfies this definition. So do lots of other relations, including common ones defined on the integers. For example, Less than (written <) is a binary relation on the integers. It contains an infinite number of elements drawn from the Cartesian product of the set of integers with itself. It includes, for example:

$\{(1,2), (2,3), (3, 4), \ldots\}$

Notice several important properties of relations as we have defined them. First, a relation may be equal to the empty set. For example, if Dave, Sue, and Billy all hate dessert, then the dessert relation would be {} or $\varnothing$.

Second, there are no constraints on how many times a particular element of A or B may occur in the relation. In the dessert example, Dave occurs twice, Sue occurs twice, Billy doesn't occur at all, cake occurs once, pie occurs once, and ice cream occurs twice.

If we have two or more binary relations, we may be able combine them via an operation we'll call composition. For example, if we knew the number of fat grams in a serving of each kind of dessert, we could ask for the number of fat grams in a particular person's dessert choices. To compute this, we first use the Dessert relation to find all the desserts each person likes. Next we get the bad news from the FatGrams relation, which probably looks something like this:

{(cake, 25), (pie, 15), (ice cream, 20)

Finally, we see that the composed relation that relates people to fat grams is {(Dave, 25), (Dave, 20), (Sue, 15), (Sue, 20)}. Of course, this only worked because when we applied the first relation, we got back desserts, and our second relation has desserts as its first component. We couldn't have composed Dessert with Less than, for example.

Formally, we say that the ***composition*** of two relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, written $R_2 \circ R_1$ is $\{(a,c) : \exists\ (a, b) \in R_1 \text{ and } (b, c) \in R_2\}$. Note that in this definition, we've said that to compute $R_2 \circ R_1$, we first apply R_1, then R_2. In other words we go right to left. Some definitions go the other way. Obviously we can define it either way, but it's important to check carefully what definition people are using and to be consistent in what you do. Using this notation, we'd represent the people to fat grams composition described above as FatGrams ° Dessert.

Now let's generalize a bit. An ordered pair is a sequence (where order counts) of two elements. We could also define an ordered triple as a sequence of three elements, an ordered quadruple as a sequence of four elements, and so forth. More generally, if n is any positive integer, then an ***ordered n-tuple*** is a sequence of n elements. For example, (Ann, Joe, Mark) is a 3-tuple.

We defined binary relation using our definition of an ordered pair. Now that we've extended our definition of an ordered pair to an ordered n-tuple, we can extend our notion of a relation to allow for an arbitrary number of elements to be related. We define an ***n-ary relation*** over sets $A_1, A_2, \ldots A_n$ as a subset of $A_1 \times A_2 \times \ldots \times A_n$. The n sets may be different, or they may be the same. For example, let A be a set of people:

A = {Dave, Sue, Billy, Ann, Joe, Mark, Cathy, Pete}

Now suppose that Ann and Dave are the parents of Billy, Ann and Joe are the parents of Mark, and Mark and Sue are the parents of Cathy. Then we could define a 3-ary (or ***ternary***) relation Child-of as the following subset of $A \times A \times A$:

{(Ann, Dave, Billy), (Ann, Joe, Mark), (Mark, Sue, Cathy)}

2.2 Functions

Relations are very general. They allow an object to be related to any number of other objects at the same time (as we did in the dessert example above). Sometimes, we want a more restricted notion, in which each object is related to a unique other object. For example, (at least in an ideal world without criminals or incompetent bureaucrats) each American resident is related to a unique social security number. To capture this idea we need functions. A ***function*** from a set A to a set B is a special kind of a binary relation over A and B in which each element of A occurs precisely once. The dessert relation we defined earlier is not a function since Dave and Sue each occur twice and Billy doesn't occur at all. We haven't restricted each person to precisely one dessert. A simple relation that *is* a function is the successor function Succ defined on the integers:

$\text{Succ}(n) = n + 1.$

Of course, we cannot write out all the elements of Succ (since there are an infinite number of them), but Succ includes:

$\{\ldots, (-3, -2), (-2, -1), (-1, 0), (0, 1), (1, 2), (2, 3), \ldots\}$

It's useful to define some additional terms to make it easy to talk about functions. We start by writing

$$f: A \rightarrow B,$$

which means that f is a function from the set A to the set B. We call A the ***domain*** of f and B the ***codomain*** or ***range***. We may also say that f is a function from A to B. If $a \in A$, then we write

$$f(a),$$

which we read as "f of a" to indicate the element of B to which a is related. We call this element the ***image*** of a under f or the ***value*** of f for a. Note that, given our definition of a function, there must be exactly one such element. We'll also call a the ***argument*** of f. For example we have that

$$\text{Succ}(1) = 2,\ \text{Succ}(2) = 3, \text{ and so forth.}$$

Thus 2 is the image (or the value) of the argument 1 under Succ.

Succ is a ***unary function***. It maps from a single element (a number) to another number. But there are lots of interesting functions that map from ordered pairs of elements to a value. We call such functions ***binary functions***. For example, integer addition is a binary function:

$$+: (Z \times Z) \rightarrow Z$$

Thus + includes elements such as ((2, 3), 5), since 2 + 3 is 5. We could also write

$$+((2,3)) = 5$$

We have double parentheses here because we're using the outer set to indicate function application (as we did above without confusion for Succ) and the inner set to define the ordered pair to which the function is being applied. But this is confusing. So, generally, when the domain of a function is the Cartesian product of two or more sets, as it is here, we drop the inner set of parentheses and simply write

$$+(2,3) = 5.$$

Alternatively, many common binary functions are written in infix notation rather than the prefix notation that is standard for all kinds of function. This allows us to write

$$2+3 = 5$$

So far, we've had unary functions and binary functions. But just as we could define n-ary relations for arbitrary values of n, we can define n-ary functions. For any positive integer n, an ***n-ary function*** f is a function is defined as

$$F: (D_1 \times D_2 \ldots \times D_n) \rightarrow R$$

For example, let Z be the set of integers. Then

$$\text{QuadraticEquation}: (Z \times Z \times Z) \rightarrow F$$

is a function whose domain is an ordered triple of integers and whose domain is a set of functions. The definition of Quadratic Equation is:

$$\text{QuadraticEquation}(a, b, c)\ (x) = ax^2 + bx + c$$

What we did here is typical of function definition. First we specify the domain and the range of the function. Then we define how the function is to compute its value (an element of the range) given its arguments (an element of the domain). QuadraticEquation may seem a bit unusual since its range is a set of functions, but both the domain and the range of a function can be any set of objects, so sets of functions qualify.

Recall that in the last section we said that we could compose binary relations to derive new relations. Clearly, since functions are just special kinds of binary relations, if we can compose binary relations we can certainly compose binary functions. Because a function returns a unique value for each argument, it generally makes a lot more sense to compose functions than it does relations, and you'll see that although we rarely compose relations that aren't functions, we compose functions all the time. So, following our definition above for relations, we define the ***composition of two functions*** $F_1 \subseteq A \times B$ and $F_2 \subseteq B \times C$, written $F_2 \circ F_1$ is $\{(a,c) : \exists b\ (a, b) \in F_1 \text{ and } (b, c) \in F_2\}$. Notice that the composition of two functions must necessarily also be a function. We mentioned above that there is sometimes confusion about the order in which relations (and now functions) should be applied when they are composed. To avoid this problem, let's introduce a new notation F(G(x)). We use the parentheses here to indicate function application, just as we did above. So this notation is clear. Apply F to the result of first applying G to x. This notation reads right to left as does our definition of the $\circ$ notation.

A function is a special kind of a relation (one in which each element of the domain occurs precisely once). There are also special kinds of functions:

A function f : D → R is ***total*** if it is defined for every element of D (i.e., every element of D is related to some element of R). The standard mathematical definition of a function requires totality. The reason we haven't done that here is that, as we pursue the idea of "computable functions", we'll see that there are total functions whose domains cannot be effectively defined (for example, the set of C programs that always halt). Thus it is useful to expand the definition of the function's domain (e.g., to the set of all C programs) and acknowledge that if the function is applied to certain elements of the domain (e.g., programs that don't halt), its value will be undefined. We call this broader class of functions (which does include the total functions as a subset) the set of ***partial*** functions. For the rest of our discussion in this introductory unit, we will consider only total functions, but be prepared for the introduction of partial functions later.

A function f : D → R is ***one to one*** if no element of the range occurs more than once. In other words, no two elements of the domain map to the same element of the range. Succ is one to one. For example, the only number to which we can apply Succ and derive 2 is 1. QuadraticEquation is also one to one. But + isn't. For example, both +(2,3) and +(4,1) equal 5.

A function f : D → R is ***onto*** if every element of R is the value of some element of D. Another way to think of this is that a function is onto if all of the elements of the range are "covered" by the function. As we defined it above, Succ is onto. But let's define a different function Succ' on the natural numbers (rather than the integers). So we define

Succ' : N → N.

Succ' is not onto because there is no natural number i such that Succ'(i) = 0.

The easiest way to envision the differences between an arbitrary relation, a function, a one to one function and an onto function is to make two columns (the first for the domain and the second for the range) and think about the sort of matching problems you probably had on tests in elementary school.

Let's consider the following five matching problems and let's look at various ways of relating the elements of column 1 (the domain) to the elements of column 2 (the range):

1

$$\begin{tikzcd} A \arrow[r] \arrow[dr] \arrow[ddr] & x \\ B & y \\ C & z \end{tikzcd}$$

2

$$\begin{tikzcd} A \arrow[r] & x \\ B \arrow[ur] & y \\ C \arrow[ur] & z \end{tikzcd}$$

3

$$\begin{tikzcd} A \arrow[r] & x \\ B \arrow[r] & y \\ C \arrow[dr] & z \\ & q \end{tikzcd}$$

4

$$\begin{tikzcd} A \arrow[dr] & x \\ B \arrow[ur] & y \\ C \arrow[r] & z \\ D \arrow[ur] & \end{tikzcd}$$

5

$$\begin{tikzcd} A \arrow[dr] & x \\ B \arrow[ur] & y \\ C \arrow[r] & z \end{tikzcd}$$

The relationship in example 1 is a relation but it is not a function, since there are three values associated A. The second example is a function since, for each object in the first column, there is a single value in the second column. But this function is neither one to one (because x is derived from both A and B) nor onto (because z can't be derived from anything). The third example is a function that is one to one (because no element of the second column is related to more than one element of the first column). But it still isn't onto because z has been skipped: nothing in the first column derives it. The fourth example is a function that is onto (since every element of column two has an arrow coming into it), but it isn't one to one, since z is derived from both C and D. The fifth and final example is a function that is both one to one and onto. By the way, see if you can modify either example 3 or example 4 to make them both one to one and onto. You're not allowed to change the number of elements in either column, just the arrows. You'll notice that you can't do it. In order for a function to be both one to one and onto, there must be equal numbers of elements in the domain and the range.

The ***inverse*** of a binary relation R is simply the set of ordered pairs in R with the elements of each pair reversed. Formally, if $R \subseteq A \times B$, then $R^{-1} \subseteq B \times A = \{(b, a): (a, b) \in R\}$. If a relation is a way of associating with each element of A a corresponding element of B, then think of its inverse as a way of associating with elements of B their corresponding elements in A. Every relation has an inverse. Every function also has an inverse, but that inverse may not also be a function. For example, look again at example two of the matching problems above. Although it is a function, its inverse is not. Given the argument x, should we return the value A or B? Now consider example 3. Its inverse is also not a (total) function, since there is no value to be returned for the argument z. Example four has the same problem example two does. Now look at example five. Its inverse is a function. Whenever a function is both one to one and onto, its inverse will also be a function and that function will be both one to one and onto.

Inverses are useful. When a function has an inverse, it means that we can move back and forth between columns one and two without loss of information. Look again at example five. We can think of ourselves as operating in the {A, B, C} universe or in the {x, y, z} universe interchangeably since we have a well defined way to move from one to the other. And if we move from column one to column two and then back, we'll be exactly where we started. Functions with inverses (alternatively, functions that are both one to one and onto) are called ***bijections***. And they may be used to define ***isomorphisms*** between sets, i.e., formal correspondences between the elements of two sets, often with the additional requirement that some key structure be preserved. We'll use this idea a lot. For example, there exists an isomorphism between the set of states of a finite state machine and a particular set of sets of input strings that could be fed to the machine. In this isomorphism, each state is associated with precisely the set of strings that drive the machine to that state.

3 Binary Relations on a Single Set

Although it makes sense to talk about n-ary relations, for arbitrary values of n (and we have), it turns out that the most useful relations are often binary ones -- ones where n is two. In fact, we can make a further claim about some of the most useful relations we'll work with: they involve just a single set. So instead of being subsets of $A \times B$, for arbitrary values of A and B, they are subsets of $A \times A$, for some particular set of A of interest. So let's spend some additional time looking at this restricted class of relations.

3.1 Representing Binary Relations on a Single Set

If we're going to work with some binary relation R, and, in particular, if we are going to compute with it, we need some way to represent the relation. We have several choices. We could:
1) List the elements of R.
2) Encode R as a computational procedure. There are at least two ways in which a computational procedure can define R. It may:
   a) enumerate the elements of R, or
   b) return a boolean value, when given an ordered pair. True means that the pair is in R; False means it is not.
3) Encode R as a directed graph.
4) Encode R as an ***incidence matrix***.

For example, consider the mother-of relation M in a family in which Doreen is the mother of Ann, Ann is the mother of Catherine, and Catherine is the mother of Allison. To exploit approach 1, we just write

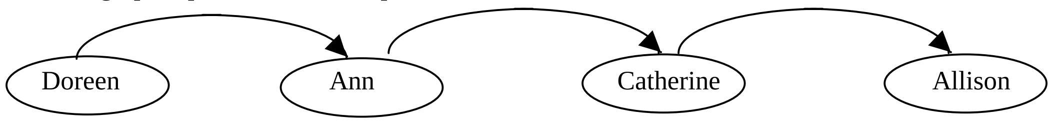
M = {(Doreen, Ann), (Ann, Catherine), (Catherine, Allison)}.

Clearly, this approach only works for finite relations.

The second approach simply requires code appropriate to the particular relation we're dealing with. One appeal of this approach is that it works for both finite and infinite relations, although, of course, a program that enumerates elements of an infinite relation will never halt.

Next we consider approach 3. Assuming that we are working with a finite relation $R \subseteq A \times A$, we can build a directed graph to represent R as follows:
1) Construct a set of nodes, one for each element of A that appears in any element of R.
2) For each ordered pair in R, draw an edge from the first element of the pair to the second.

The following directed graph represents our example relation M defined above:



And, finally, approach 4: Again assuming a finite relation R ⊆ A × A, we can build an incidence matrix to represent R as follows:

1) Construct a square boolean matrix S whose number of rows and columns equals the number of elements of A that appear in any element of R.
2) Label one row and one column for each such element of A.
3) For each element (p, q) of R, set S(p, q) to 1(or True). Set all other elements of S to 0 (or False).

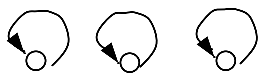
The following boolean matrix represents our example relation M defined above:

	Doreen	Ann	Catherine	Allison
Doreen	0	1	0	0
Ann	0	0	1	0
Catherine	0	0	0	1
Allison	0	0	0	0

3.2 Properties of Relations

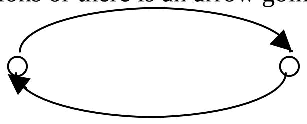
Many useful binary relations have some kind of structure. For example, it might be the case that every element of the underlying set is related to itself. Or it might happen that if a is related to b, then b must necessarily be related to a. There's one special kind of relation, called an equivalence relation that is particularly useful. But before we can define it, we need first to define each of the individual properties that equivalence relations possess.

A relation R ⊆ A × A is ***reflexive*** if, for each a ∈ A, (a, a) ∈ R. In other words a relation R on the set A is reflexive if every element of A is related to itself. For example, consider the relation Address defined as "lives at same address as". Address is a relation over a set of people. Clearly every person lives at the same address as him or herself, so Address is reflexive. So is the Less than or equal to relation on the integers. Every integer is Less than or equal to itself. But the Less than relation is not reflexive: in fact no number is Less than itself. Both the directed graph and the matrix representations make it easy to tell if a relation is reflexive. In the graph representation, every node will have an edge looping back to itself. In the graph representation, there will be ones along the major diagonal:



1		
	1	
		1

A relation R ⊆ A × A is ***symmetric*** if, whenever (a, b) ∈ R, so is (b, a). In other words, if a is related to b, then b is related to a. The Address relation we described above is symmetric. If Joe lives with Ann, then Ann lives with Joe. The Less than or equal relation is not symmetric (since, for example, 2 ≤ 3, but it is not true that 3 ≤ 2). The graph representation of a symmetric relation has the property that between any two nodes, either there is an arrow going in both directions or there is an arrow going in neither direction. So we get graphs with components that look like this:



If we choose the matrix representation, we will end up with a symmetric matrix (i.e., if you flip it on its major diagonal, you'll get the same matrix back again). In other words, if we have a matrix with 1's wherever there is a number in the following matrix, then there must also be 1's in all the squares marked with an *:

	*	*		
1				3
2				
	*			

A relation $R \subseteq A \times A$ is ***antisymmetric*** if, whenever $(a, b) \in R$ and $a \neq b$, then $(b, a) \notin R$ The Mother-of relation we described above is antisymmetric: if Ann is the mother of Catherine, then one thing we know for sure is that Catherine is not also the mother of Ann. Our Address relation is clearly not antisymmetric, since it is symmetric. There are, however, relations that are neither symmetric nor antisymmetric. For example, the Likes relation on the set of people: If Joe likes Bob, then it is possible that Bob likes Joe, but it is also possible that he doesn't.

A relation $R \subseteq A \times A$ is ***transitive*** if, whenever $(a, b) \in R$ and $(b, c) \in R$, $(a, c) \in R$. A simple example of a transitive relation is Less than. Address is another one: if Joe lives with Ann and Ann lives with Mark, then Joe lives with Mark. Mother-of is not transitive. But if we change it slightly to Ancestor-of, then we get a transitive relation. If Doreen is an ancestor of Ann and Ann is an ancestor of Catherine, then Doreen is an ancestor of Catherine.

The three properties of reflexivity, symmetry, and transitivity are almost logically independent of each other. We can find simple, possibly useful relationships with seven of the eight possible combinations of these properties:

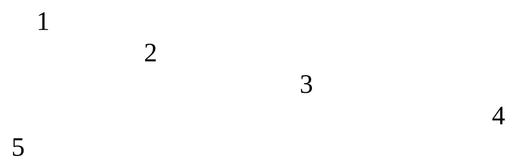
	Domain	Example
None of the properties	people	Mother-of
Just reflexive	people	Would-recognize-picture-of
Just symmetric	people	Has-ever-been-married-to
Just transitive	people	Ancestor-of
Reflexive and symmetric	people	Hangs-out-with (assuming we can say one hangs out with oneself)
Reflexive and transitive	numbers	Less than or equal to
Symmetric and transitive		
All three	numbers	Equality
	people	Address

To see why we can't find a good example of a relation that is symmetric and transitive but not reflexive, consider a simple relation R on {1, 2, 3, 4}. As soon as R contains a single element that relates two unequal objects (e.g., (1, 2)), it must, for symmetry, contain the matching element (2, 1). So now we have R = {(1, 2), (2, 1)}. To make R transitive, we must add (1, 1). But that also makes R reflexive.

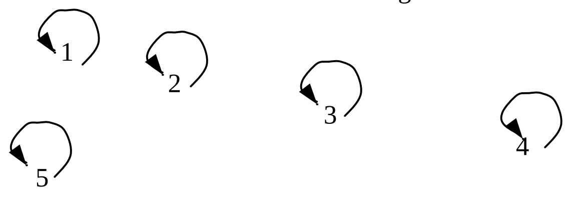
3.3 Equivalence Relations

Although all the combinations we just described are possible, one combination is of such great importance that we give it a special name. A relation is an ***equivalence relation*** if it is reflexive, symmetric and transitive. Equality (for numbers, strings, or whatever) is an equivalence relation (what a surprise, given the name). So is our Address (lives at same address) relation.

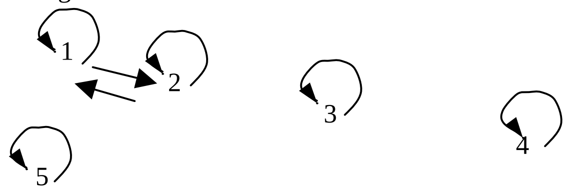
Equality is a very special sort of equivalence relation because it relates an object only to itself. It doesn't help us much to carve up a large set into useful subsets. But in fact, equivalence relations are an incredibly useful way to carve up a set. Why? Let's look at a set P, with five elements, which we can draw as a set of nodes as follows:



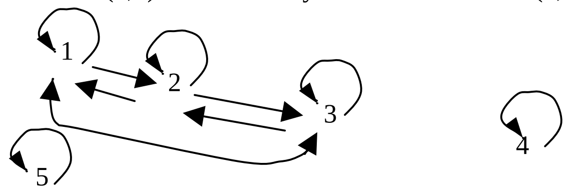
Now let's build an equivalence relation E on P. The first thing we have to do is to relate each node to itself, in order to make the relation reflexive. So we've now got:



Now let's add one additional element (1,2). As soon as we do that, we must also add (2,1), since E must be symmetric. So now we've got:



Suppose we now add (2,3). We must also add (3,2) to maintain symmetry. In addition, because we have (1, 2) and (2, 3), we must create (1,3) for transitivity. And then we need (3, 1) to restore symmetry. That gives us



Notice what happened here. As soon as we related 3 to 2, we were also forced to relate 3 to 1. If we hadn't, we would no longer have had an equivalence relation. See what happens now if you add (3, 4) to E.

What we've seen in this example is that an equivalence relation R on a set S carves S up into a set of clusters, which we'll call ***equivalence classes***. This set of equivalence classes has the following key property:

For any s, t $\in$ S, if s $\in$ Class_i and (s, t) $\in$ R, then t $\in$ Class_i.

In other words, all elements of S that are related under R are in the same equivalence class. To describe equivalence classes, we'll use the notation [a] to mean the equivalence class to which a belongs. Or we may just write [description], where description is some clear property shared by all the members of the class. Notice that in general there may be lots of different ways to describe the same equivalence class. In our example, for instance, [1], [2], and [3] are different names for the same equivalence class, which includes the elements 1, 2, and 3. In this example, there are two other equivalence classes as well: [4] and [5].

It is possible to prove that if R is an equivalence relation on a nonempty set A then the equivalence classes of R constitute a partition of A. Recall that Π is a partition of a set A if and only if (a) no element of Π is empty; (b) all members of Π are disjoint; and (c) $\bigcup \Pi = A$. In other words, if we want to take a set A and carve it up into a set of subsets, an equivalence relation is a good way to do it.

For example, our Address relation carves up a set of people into subsets of people who live together. Let's look at some more examples:

- Let A be the set of all strings of letters. Let SameLength $\subseteq A \times A$ relate strings whose lengths are the same. SameLength is an equivalence relation that carves up the universe of all strings into a collection of subsets, one for each natural number (i.e., strings of length 0, strings of length 1, etc.).

- Let Z be the set of integers. Let EqualMod3 $\subseteq Z \times Z$ relate integers that have the same remainder when divided by 3. EqualMod3 has three equivalence classes, [0], [1], and [2]. [0] includes 0, 3, 6, etc.

- Let CP be the set of C programs, each of which accepts an input of variable length. We'll call the length of any specific input n. Let SameComplexity $\subseteq CP \times CP$ relate two programs if their running-time complexity is the same. More specifically, $(c_1, c_2) \in$ SameComplexity precisely in case:

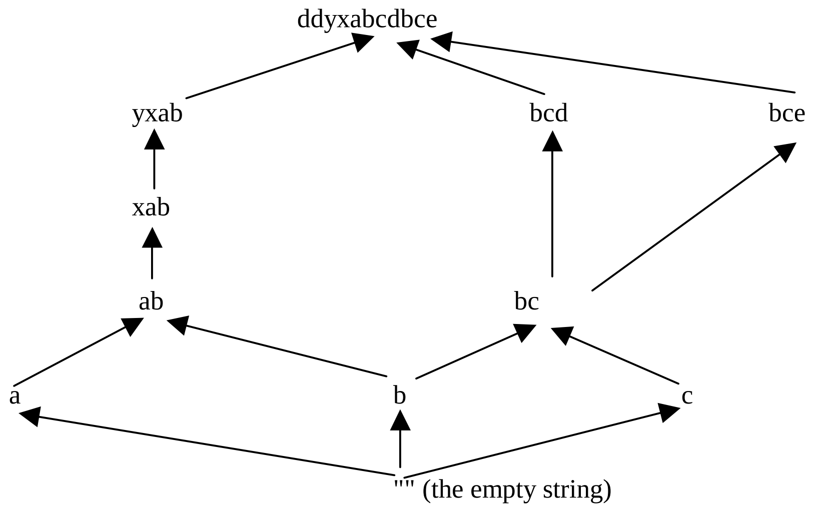
  $$\exists m_1, m_2, k\ [\forall n > k,\ \text{RunningTime}(c_1) \leq m_1*\text{RunningTime}(c_2)\ \text{AND}\ \text{RunningTime}(c_2) \leq m_2*\text{RunningTime}(c_1)]$$

Not every relation that connects "similar" things is an equivalence relation. For example, consider SimilarCost(x, y), which holds if the price of x is within $1 of the price of y. Suppose A costs $10, B costs $10.50, and C costs $11.25. Then SimilarCost(A, B) and SimilarCost(B, C), but not SimilarCost(A, C). So SimilarCost is not transitive, although it is reflexive and symmetric.

3.4 Orderings

Important as equivalence relations are, they're not the only special kind of relation worth mentioning. Let's consider two more.

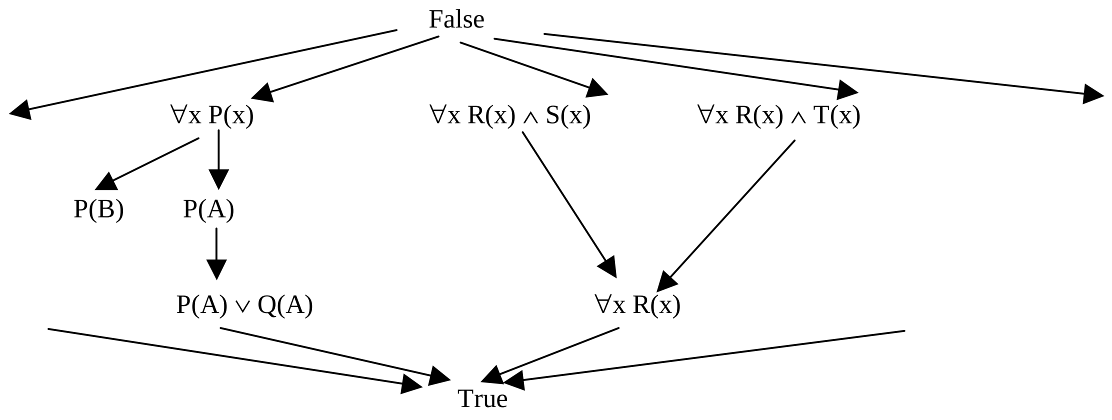
A ***partial order*** is a relation that is reflexive, antisymmetric, and transitive. If we write out any partial order as a graph, we'll see a structure like the following one for the relation SubstringOf. Notice that in order to make the graph relatively easy to read, we'll adopt the convention that we don't write in the links that are required by reflexivity and transitivity. But, of course, they are there in the relation itself:



Read an arrow from x to y as meaning that (x, y) is an element of the relation. So, in this example, "a" is a substring of "ab", which is a substring of "xab", and so forth. Note that in a partial order, it is often the case that there are some elements (such as "ab" and "bc") that are not related to each other at all (since neither is a substring of the other).

Sometimes a partial order on a domain D defines a mininal and/or a maximal element. In this example, there is a minimal element, the empty string, which is a substring of every string. There appears from this picture to be a maximal element, but that's just because we drew only a tiny piece of the graph. Every string is a substring of some longer string, so there is in fact no maximal element.

Let's consider another example based on the ***subsumption*** relation between pairs of logical expressions. A logical expression A subsumes another expression B iff (if and only if), whenever A is true B must be true regardless of the values assigned to the variables and functions of A and B. For example: ∀x P(x) subsumes P(A), since, regardless of what the predicate P is and independently of any axioms we have about it, and regardless of what object A represents, if ∀x P(x) is true, then P(A) must be true. Why is this a useful notion? Suppose we're building a theorem proving or reasoning program. If we already know ∀x P(x), and we are then told P(A), we can throw away this new fact. It doesn't add to our knowledge (except perhaps to focus our attention on the object A) since it is subsumed by something we already knew. A small piece of the subsumption relation on logical expressions is shown in the following graph. Notice that now there is a maximal element, False, which subsumes everything (in other words, if we have the assertion False in our knowledge base, we have a contradiction even if we know nothing else). There is also a minimal element, True, which tells us nothing.



A ***total order*** $R \subseteq A \times A$ is a partial order that has the additional property that $\forall a, b \in A$, either (a, b) or $(b, a) \in R$. In other words, every pair of elements must be related to each other one way or another. The classic example of a total order is ≤ (or ≥, if you prefer) on the integers. The ≤ relation is reflexive since every integer is equal to itself. It's antisymmetric since if $a \le b$ and $a \ne b$, then for sure it is not also true that $b \le a$. It's transitive: if $a \le b$ and $b \le c$, then $a \le c$. And, given any two integers a and b, either $a \le b$ or $b \le a$. If we draw any total order as a graph, we'll get something that looks like this (again without the reflexive and transitive links shown):

$$\begin{tikzcd}6 \\ 5 \arrow[u] \\ 4 \arrow[u] \\ 3 \arrow[u]\end{tikzcd}$$

This is only a tiny piece of the graph, of course. It continues infinitely in both directions. But notice that, unlike our earlier examples of partial orders, there is no splitting in this graph. For every pair of elements, one is above and one is below.

4 Important Properties of Binary Functions

Any relation that uniquely maps from all elements of its domain to elements of its range is a function. The two sets involved can be anything and the mapping can be arbitrary. However, most of the functions we actually care about behave in some sort of regular fashion. It is useful to articulate a set of properties that many of the functions that we'll study have. When these properties are true of a function, or a set of functions, they give us techniques for proving additional properties of the objects involved. In the following definitions, we'll consider an arbitrary binary function # defined over a set we'll call A with elements we'll call a, b, and c. As examples, we'll consider functions whose actual domains are sets, integers, strings, and boolean expressions.

A binary function # is ***commutative*** iff $\forall$a,b a # b = b # a

Examples:	a + b = b + a	integer addition
	a ∩ b = b ∩ a	set intersection
	a AND b = b AND a	boolean and

A binary function # is ***associative*** iff $\forall$a,b,c (a # b) # c = a # (b # c)

Examples:	(a + b) + c = a + (b + c)	integer addition
	(a ∩ b) ∩ c = a ∩ (b ∩ c)	set intersection
	(a AND b) AND c = a AND (b AND c)	boolean and
	(a ‖ b) ‖ c = a ‖ (b ‖ c)	string concatenation

A binary function # is ***idempotent*** iff $\forall$a a # a = a.

Examples:	min(a, a) = a	integer min
	a ∩ a = a	set intersection
	a AND a = a	boolean and

The ***distributivity*** property relates two binary functions: A function # distributes over another function ! iff
$\forall$a,b,c a # (b ! c) = (a # b) ! (a # c) and (b ! c) # a = (b # a) ! (c # a)

Examples:	a * (b + c) = (a * b) + (a * c)	integer multiplication over addition
	a ∪ (b ∩ c) = (a ∪ b) ∩ (a ∪ c)	set union over intersection
	a AND (b OR c)=(a AND b) OR (a AND c)	boolean AND over OR

The ***absorption laws*** also relate two binary functions to each other: A function # absorbs another function ! iff
$\forall$a,b a # (a ! b) = a

Examples:	a ∩ (a ∪ b) = a	set intersection absorbs union
	a OR (a AND b) = a	boolean OR absorbs AND

It is often the case that when a function is defined over some set A, there are special elements of A that have particular properties with respect to that function. In particular, it is worth defining what it means to be an identity and to be a zero:

An element a is an ***identity*** for the function # iff $\forall$x ∈ A, x # a = x and a # x = x

Examples:	b * 1 = b	1 is an identity for integer multiplication
	b + 0 = b	0 is an identity for integer addition
	b ∪ ∅ = b	∅ is an identity for set union
	b OR False = b	False is an identity for boolean OR
	b ‖ "" = b	"" is an identity for string concatenation

Sometimes it is useful to differentiate between a right identity (one that satisfies the first requirement above) and a left identity (one that satisfies the second requirement above). But for all the functions we'll be concerned with, if there is a left identity, it is also a right identity and vice versa, so we will talk simply about an identity.

An element a is a ***zero*** for the function # iff $\forall$x ∈ A, x # a = a and a # x = a

Examples:	b * 0 = 0	0 is a zero for integer multiplication
	b ∩ ∅ = ∅	∅ is a zero for set intersection
	b AND FALSE = FALSE	FALSE is a zero for boolean AND

Just as with identities, it is sometimes useful to distinguish between left and right zeros, but we won't need to.

Although we're focusing here on binary functions, there's one important property that unary functions may have that is worth mentioning here:

A unary function % is a ***self inverse*** iff $\forall x\ \ \%(\%x)) = x$. In other words, if we compose the function with itself (apply it twice), we get back the original argument. Note that this is not the same as saying that the function is its own inverse. In most of the cases we'll consider (including the examples given here), it is not. A single application of the function produces a new value, but if we apply the function a second time, we get back to where we started.

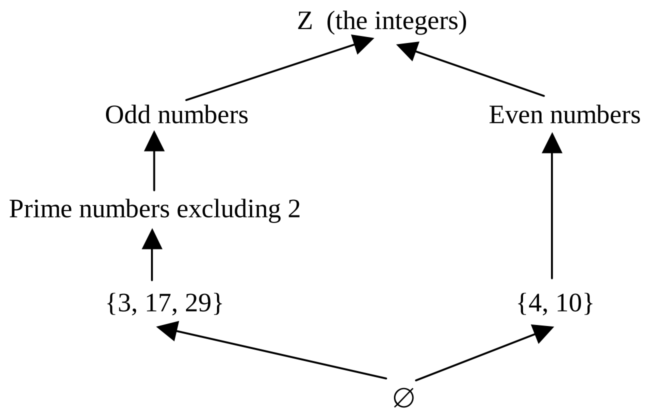
Examples:	$-(-(a)) = a$	Multiplying by -1 is a self inverse for integers
	$1/(1/a)$	Dividing into 1 is a self inverse for integers
	$\overline{\overline{a}} = a$	Complement is a self inverse for sets
	$\neg(\neg\, a) = a$	Negation is a self inverse for booleans
	$(a^R)^R = a$	Reversal is a self inverse for strings

5 Relations and Functions on Sets

In the last two sections, we explored various useful properties of relations and functions. With those tools in hand, let's revisit the basic relations and functions on sets.

5.1 Relations

We have defined two relations on sets: subset and proper subset. What can we say about them? Subset is a partial order, since it is reflexive (every set is a subset of itself), transitive (if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$) and antisymmetric (if $A \subseteq B$ and $A \neq B$, then it must not be true that $B \subseteq A$). For example, we see that the subset relation imposes the following partial order if you read each arrow as "is a subset of":



What about proper subset? It is not a partial order since it is not reflexive.

5.2 Functions

All of the functional properties we defined above apply in one way or another to the functions we have defined on sets. Further, as we saw above, there some set functions have a zero or an identity. We'll summarize here (without proof) the most useful properties that hold for the functions we have defined on sets:

Commutativity $A \cup B = B \cup A$
$A \cap B = B \cap A$

Associativity $(A \cup B) \cup C = A \cup (B \cup C)$
$(A \cap B) \cap C = A \cap (B \cap C)$

Idempotency	$A \cup A = A$ $A \cap A = A$
Distributivity	$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$ $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
Absorption	$(A \cup B) \cap A = A$ $(A \cap B) \cup A = A$
Identity	$A \cup \varnothing = A$
Zero	$A \cap \varnothing = \varnothing$
Self Inverse	$\overline{\overline{A}} = A$

In addition, we will want to make use of the following theorems that can be proven to apply specifically to sets and their operations (as well as to boolean expressions):

De Morgan's laws	$\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\overline{A \cap B} = \overline{A} \cup \overline{B}$

6 Proving Properties of Sets

A great deal of what we do when we build a theory about some domain is to prove that various sets of objects in that domain are equal. For example, in our study of automata theory, we are going to want to prove assertions such as these:

- The set of strings defined by some regular expression E is identical to the set of strings defined by some second regular expression E'.
- The set of strings that will be accepted by some given finite state automaton M is the same as the set of strings that will be accepted by some new finite state automaton M' that is smaller than M.
- The set of languages that can be defined using regular expressions is the same as the set of languages that can be accepted by a finite state automaton.
- The set of problems that can be solved by a Turing Machine with a single tape is the same as the set of problems that can be solved by a Turing Machine with any finite number of tapes.

So we become very interested in the question, "How does one prove that two sets are identical"? There are lots of ways and many of them require special techniques that apply in specific domains. But it's worth mentioning two very general approaches here.

<u>6.1 Using Set Identities and the Definitions of the Functions on Sets</u>

Sometimes we want to compare apples to apples. We may, for example, want to prove that two sets of strings are identical, even though they may have been derived differently. In this case, one approach is to use the set identity theorems that we enumerated in the last section. Suppose, for example, that we want to prove that

$$A \cup (B \cap (A \cap C)) = A$$

We can prove this as follows:

$$\begin{aligned} A \cup (B \cap (A \cap C)) &= (A \cup B) \cap (A \cup (A \cap C)) && \text{Distributivity} \\ &= (A \cup B) \cap ((A \cap C) \cup A) && \text{Commutativity} \\ &= (A \cup B) \cap A && \text{Absorption} \\ &= A && \text{Absorption} \end{aligned}$$

Sometimes, even when we're comparing apples to apples, the theorems we've listed aren't enough. In these cases, we need to use the definitions of the operators. Suppose, for example, that we want to prove that

$$A - B = A \cap \overline{B}$$

We can prove this as follows (where U stands for the Universe with respect to which we take complement):

$$\begin{aligned}A - B &= \{x : x \in A \text{ and } x \notin B\} \\ &= \{x : x \in A \text{ and } (x \in U \text{ and } x \notin B)\} \\ &= \{x : x \in A \text{ and } x \in U - B\} \\ &= \{x : x \in A \text{ and } x \in \bar{B}\} \\ &= A \cap \bar{B}\end{aligned}$$

6.2 Showing Two Sets are the Same by Showing that Each is a Subset of the Other

Sometimes, though, our problem is more complex. We may need to compare apples to oranges, by which I mean that we are comparing sets that aren't even defined in the same terms. For example, we will want to be able to prove that A: the set of languages that can be defined using regular expressions is the same as B: the set of languages that can be accepted by a finite state automaton. This seems very hard: Regular expressions look like

$a^* (b \cup ba)^*$

Finite state machines are a collection of states and rules for moving from one state to another. How can we possibly prove that these A and B are the same set? The answer is that we can show that the two sets are equal by showing that each is a subset of the other. For example, in the case of the regular expressions and the finite state machines, we will show first that, given a regular expression, we can construct a finite state machine that accepts exactly the strings that the regular expression describes. That gives us $A \subseteq B$. But there might still be some finite state machines that don't correspond to any regular expressions. So we then show that, given a finite state machine, we can construct a regular expression that defines exactly the same strings that the machine accepts. That gives us $B \subseteq A$. The final step is to exploit the fact that

$A \subseteq B \text{ and } B \subseteq A \Rightarrow A = B$

7 Cardinality of Sets

It seems natural to ask, given some set A, "What is the size of A?" or "How many elements does A contain?" In fact, we've been doing that informally. We'll now introduce formal techniques for discussing exactly what we mean by the size of a set. We'll use the term ***cardinality*** to describe the way we answer such questions. So we'll reply that the cardinality of A is X, for some appropriate value of X. For simple cases, determining the value of X is straightforward. In other cases, it can get quite complicated. For our purposes, however, we can get by with three different kinds of answers: a natural number (if A is finite), "countably infinite" (if A has the same number of elements as there are integers), and "uncountably infinite" (if A has more elements than there are integers).

We write the cardinality of a set A as $|A|$.

A set A is ***finite*** and has cardinality $n \in N$ (the natural numbers) if either $A = \varnothing$ or there is a bijection from A to $\{1, 2, \ldots n\}$, for some value of n. In other words, a set is finite if either it is empty or there exists a one-to-one and onto mapping from it to a subset of the positive integers. Or, alternatively, a set is finite if we can count its elements and finish. The cardinality of a finite set is simply a natural number whose value is the number of elements in the set.

A set is ***infinite*** if it is not finite. The question now is, "Are all infinite sets the same size?" The answer is no. And we don't have to venture far to find examples of infinite sets that are not the same size. So we need some way to describe the cardinality of infinite sets. To do this, we need to define a set of numbers we'll call the cardinal numbers. We'll use these numbers as our measure of the size of sets. Initially, we'll define all the natural numbers to be cardinal numbers. That lets us describe the cardinality of finite sets. Now we need to add new cardinal numbers to describe the cardinality of infinite sets.

Let's start with a simple infinite set N, the natural numbers. We need a new cardinal number to describe the (infinite) number of natural numbers that there are. Following Cantor, we'll call this number $\aleph_0$. (Read this as "aleph null". Aleph is the first symbol of the Hebrew alphabet.)

Next, we'll say that any other set that contains the same number of members as N does also has cardinality $\aleph_0$. We'll also call a set with cardinality $\aleph_0$ ***countably infinite***. And one more definition: A set is ***countable*** if it is either finite or countably infinite.

To show that a set has cardinality $\aleph_0$, we need to show that there is a bijection between it and N. The existence of such a bijection proves that the two sets have the same number of elements. For example, the set E of even natural numbers has cardinality $\aleph_0$. To prove this, we offer the bijection:

Even : E → N
Even(x) = x/2

So we have the following mapping from E to N:

E	N
0	0
2	1
4	2
6	3
…	…

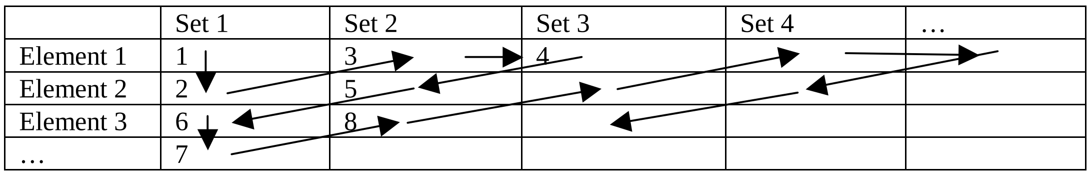
This one was easy. The bijection was obvious. Sometimes it's less so. In harder cases, a good way to think about the problem of finding a bijection from some set A to N is that we need to find an enumeration of A. An ***enumeration*** of A is simply a list of the elements of A in some order. Of course, if A is infinite, the list will be infinite, but as long as we can guarantee that every element of A will show up eventually, we have an enumeration. But what is an enumeration? It is in fact a bijection from A to the positive integers, since there is a first element, a second one, a third one, and so forth. Of course, what we need is a bijection to N, so we just subtract one. Thus if we can devise a technique for enumerating the elements of A, then our bijection to N is simply

Enum : A → N
Enum(x) = x's position in the enumeration - 1

Let's consider an example of this technique:

Theorem: The union of a countably infinite number of countably infinite sets is countably infinite.

To prove this theorem, we need a way to enumerate all the elements of the union. The simplest thing to do would be to start by dumping in all the elements of the first set, then all the elements of the second, etc. But, since the first set is infinite, we'll never get around to considering any of the elements of the other sets. So we need another technique. If we had a finite number of sets to consider, we could take the first element from each, then the second element from each, and so forth. But we also have an infinite number of sets, so if we try that approach, we'll never get to the second element of any of the sets. So we follow the arrows as shown below. The numbers in the squares indicate the order in which we select elements for the enumeration. This process goes on forever, but it is systematic and it guarantees that, if we wait long enough, any element of any of the sets will eventually be enumerated.



	Set 1	Set 2	Set 3	Set 4	…
Element 1	1	3	4		
Element 2	2	5			
Element 3	6	8			
…	7				

It turns out that a lot of sets have cardinality $\aleph_0$. Some of them, like the even natural numbers, appear at first to contain fewer elements. Some of them, like the union of a countable number of countable sets, appear at first to be bigger. But in both cases there is a bijection between the elements of the set and the natural numbers, so the cardinality is $\aleph_0$.

However, this isn't true for every set. There are sets with more than $\aleph_0$ elements. There are more than $\aleph_0$ real numbers, for example. As another case, consider an arbitrary set S with cardinality $\aleph_0$. Now consider the power set of S (the set of all subsets of S). This set has cardinality greater than $\aleph_0$. To prove this, we need to show that there exists no bijection between the power set of S and the integers. To do this, we will use a technique called ***diagonalization***. Diagonalization is a kind of proof by contradiction. It works as follows:

Let's start with the original countably infinite set S. We can enumerate the elements of S (since it's countable), so there's a first one, a second one, etc. Now we can represent each subset SS of S as a binary vector that contains one element for each element of the original set S. If SS contains element 1 of S, then the first element of its vector will be 1, otherwise 0. Similarly for all the other elements of S. Of course, since S is countably infinite, the length of each vector will also be countably infinite. Thus we might represent a particular subset SS of S as the vector:

Elem 1 of S	Elem 2 of S	Elem 3 of S	Elem 4 of S	Elem 5 of S	Elem 6 of S	……
1	0	0	1	1	0	……

Next, we observe that if the power set P of S were countably infinite, then there would be an enumeration of it that put its elements in one to one correspondence with the natural numbers. Suppose that enumeration were the following (where each row represents one element of P as described above. Ignore for the moment the numbers enclosed in parentheses.):

	Elem 1 of S	Elem 2 of S	Elem 3 of S	Elem 4 of S	Elem 5 of S	Elem 6 of S	…….
Elem 1 of P	1 (1)	0	0	0	0	0	…..
Elem 2 of P	0	1 (2)	0	0	0	0	…..
Elem 3 of P	1	1	0 (3)	0	0	0	…..
Elem 4 of P	0	0	1	0 (4)	0	0	…..
Elem 5 of P	1	0	1	0	0 (5)	0	…..
Elem 6 of P	1	1	1	0	0	0 (6)	…..

●
●
●

If this really is an enumeration of P, then it must contain all elements of P. But it doesn't. To prove that it doesn't, we will construct an element $L \in P$ that is not on the list. To do this, consider the numbers in parentheses in the matrix above. Using them, we can construct L:

¬(1)	¬(2)	¬(3)	¬(4)	¬(5)	¬(6)	…..

What we mean by ¬(1) is that if (1) is a 1 then 0; if (1) is a 0, then 1. So we've constructed the representation for an element of P. It must be an element of P since it describes a possible subset of S. But we've built it so that it differs from the first element in the list above by whether or not it includes element 1 of S. It differs from the second element in the list above by whether or not it includes element 2 of S. And so forth. In the end, it must differ from every element in the list above in at least one place. Yet it is clearly an element of P. Thus we have a contradiction. The list above was not an enumeration of P. But since we made no assumptions about it, no enumeration of P can exist. In particular, if we try to fix the problem by simply adding our new element to the list, we can just turn around and do the same thing again and create yet another element that 's not on the list. Thus there are more than $\aleph_0$ elements in P. We'll say that sets with more than $\aleph_0$ elements are ***uncountably infinite***.

The real numbers are uncountably infinite. The proof that they are is very similar to the one we just did for the power set except that it's a bit tricky because, when we write out each number as an infinite sequence of digits (like we wrote out each set above as an infinite sequence of 0's and 1's), we have to consider the fact that several distinct sequences may represent the same number.

Not all uncountably infinite sets have the same cardinality. There is an infinite number of cardinal numbers. But we won't need any more. All the uncountably infinite sets we'll deal with (and probably all the ones you can even think of unless you keep taking power sets) have the same cardinality as the reals and the power set of a countably infinite set.

Thus to describe the cardinality of all the sets we'll consider, we will use one of the following:

- The natural numbers, which we'll use to count the number of elements of finite sets,

- $\aleph_0$, which is the cardinality of all countably infinite sets, and
- uncountable, which is the cardinality of any set with more than $\aleph_0$ members.

8 Closures

Imagine some set A and some property P. If we care about making sure that A has property P, we are likely to do the following:

1. Examine A for P. If it has property P, we're happy and we quit.
2. If it doesn't, then add to A the smallest number of additional elements required to satisfy P.

Let's consider some examples:

- Let A be a set of friends you're planning to invite to a party. Let P be "A should include everyone who is likely to find out about the party" (since we don't want to offend anyone). Let's assume that if you invite Bill and Bill has a friend Bob, then Bill may tell Bob about the party. This means that if you want A to satisfy P, then you have to invite not only your friends, but your friends' friends, and their friends, and so forth. If you move in a fairly closed circle, you may be able to satisfy P by adding a few people to the guest list. On the other hand, it's possible that you'd have to invite the whole city before P would be satisfied. It depends on the connectivity of the FriendsOf relation in your social setting. The problem is that whenever you add a new person to A, you have to turn around and look at that person's friends and consider whether there are any of them who are not already in A. If there are, they must be added, and so forth. There's one positive feature of this problem, however. Notice that there is a unique set that does satisfy P, given the initial set A. There aren't any choices to be made.
- Let A be a set of 6 people. Let P be "A can enter a baseball tournament". This problem is different from the last in two important ways. First, there is a clear limit to how many elements we have to add to A in order to satisfy P. We need 9 people and when we've got them we can stop. But notice that there is not a unique way to satisfy P (assuming that we know more than 9 people). Any way of adding 3 people to the set will work.
- Let A be the Address relation (which we defined earlier as "lives at same address as"). Since relations are sets, we should be able to treat Address just as we've treated the sets of people in our last two examples. We know that Address is an equivalence relation. So we'll let P be the property of being an equivalence relation (i.e., reflexive, symmetric, and transitive). But suppose we are only able to collect facts about living arrangements in a piecemeal fashion. For example, we may learn that Address contains {(Dave, Mary), (Sue, Pete), (John, Bill)}. Immediately we know, because Address must be reflexive, that it must also contain {(Dave, Dave), (Mary, Mary), (Sue, Sue), (Pete, Pete), (John, John), (Bill, Bill)}. And, since Address must also be symmetric it must contain {(Mary, Dave), (Pete, Sue), (Bill, John)}. Now suppose that we discover that Mary lives with Sue. We add {(Mary, Sue)}. To make Address symmetric again, we must add {(Sue, Mary)}. But now we also have to make it transitive by adding {(Dave, Sue), (Sue, Dave)}.
- Let A be the set of natural numbers. Let P be "the sum of any two elements of A is also in A." Now we've got a property that is already satisfied. The sum of any two natural numbers is a natural number. This time, we don't have to add anything to A to establish P.
- Let A be the set of natural numbers. Let P be "the quotient of any two elements of A is also in A." This time we have a problem. 3/5 is not a natural number. We can add elements to A to satisfy P. If we do, we end up with exactly the rational numbers.

In all of these cases, we're going to want to say that A is ***closed*** with respect to P if it possesses P. And, if we have to add elements to A in order to satisfy P, we'll call a smallest such expanded A that does satisfy P a ***closure*** of A with respect to P. What we need to do next is to define both of these terms more precisely .

8.1 Defining Closure

The first set of definitions of closure that we'll present is very general, although it does require that we can describe property P as an n-ary relation (for some value of n). We can use it to describe what we did in all but one of the examples above, although in a few cases it will be quite cumbersome to do so.

Let n be an integer greater than or equal to 1. Let R be an n-ary relation on a set D. Thus elements of R are of the form $(d_1, d_2, \ldots, d_n)$. We say that a subset S of D is ***closed under*** R if, whenever:

1. $d_1, d_2, \ldots d_{n-1} \in S$, (all of the first n-1 elements are already in the set S) and
2. $(d_1, d_2, \ldots d_{n-1}, d_n) \in R$ (the last element is related to the n-1 other elements via R)

it is also true that $d_n \in S$.

A set S' is a ***closure*** of S with respect to R (defined on D) iff:

1. $S \subseteq S'$,
2. S' is closed under R, and
3. $\forall T$ ($T \subseteq D$ and T is closed under R) $\Rightarrow |S'| \leq |T|$.

In other words, S' is a closure of S with respect to R if it is an extension of S that is closed under R and if there is no smaller set that also meets both of those requirements. Note that we can't say that S' must be the smallest set that will do the job, since we do not yet have any guarantee that there is a unique such smaller set (recall the softball example above).

These definitions of closure are a very natural way to describe our first example above. Drawing from a set A of people, you start with S equal to your friends. Then, to compute your invitee list, you simply take the closure of S with respect to the relation FriendOf, which will force you to add to A your friends' friends, their friends, and so forth.

Now consider our second example, the case of the baseball team. Here there is no relation R that specifies, if one or more people are already on the team, then some specific other person must also be on. The property we care about is a property of the team (set) as a whole and not a property of patterns of individuals (elements). Thus this example, although similar, is not formally an instance of closure as we have just defined it. This turns out to be significant and leads us to the following definition:

Any property that asserts that a set S is closed under some relation R is a ***closure property*** of S. It is possible to prove that if P is a closure property, as just defined, on a set A and S is a subset of A, then the closure of S with respect to R exists and is unique. In other words, there exists a unique minimal set S' that contains S and is closed under R. Of all of our examples above, the baseball example is the only one that cannot be described in the terms of our definition of a closure property. The theorem that we have just stated (without proof) guarantees, therefore, that it will be the only one that does not have a unique minimal solution.

The definitions that we have just provided also work to describe our third example, in which we want to compute the closure of a relation (since, after all, a relation is a set). All we have to do is to come up with relations that describe the properties of being reflexive, symmetric, and transitive. To help us see what those relations need to be, let's recall our definitions of symmetry, reflexivity, and transitivity:

- A relation $R \subseteq A \times A$ is ***reflexive*** if, for each $a \in A$, $(a, a) \in R$.
- A relation $R \subseteq A \times A$ is ***symmetric*** if, whenever $(a, b) \in R$, so is (b, a).
- A relation $R \subseteq A \times A$ is ***transitive*** if, whenever $(a, b) \in R$ and $(b, c) \in R$, $(a, c) \in R$.

Looking at these definitions, we can come up with three relations, Reflexivity, Symmetry, and Transitivity. All three are relations on relations, and they will enable us to define these three properties using the closure definitions we've given so far. All three definitions assume a base set A on which the relation we are interested is defined:

- $\forall a \in A$, $((a, a)) \in$ Reflexivity. Notice the double parentheses here. Reflexivity is a unary relation, where each element is itself an ordered pair. It doesn't really "relate" two elements. It is simply a list of ordered pairs. To see how it works to define reflexive closure, imagine a set A = {x, y}. Now suppose we start with a relation R on A = {(x, y)}. Clearly R isn't reflexive. And the Reflexivity relation tells us that it isn't because the reflexivity relation on A contains {((x, x)), ((y, y))}. This is a unary relation. So n, in the definition of closure, is 1. Consider the first element ((x, x)). We consider all the components before the nth (i.e., first) and see if they're in A. This means we consider the first zero components. Trivially, all zero of them are in A. So the nth (the first) must also be. This means that (x, x) must be in R. But it isn't. So to compute the closure of R under Reflexivity, we add it. Similarly for (y, y).
- $\forall a,b \in A$, $a \neq b \Rightarrow [((a, b), (b, a)) \in$ Symmetry]. This one is a lot easier. Again, suppose we start with a set A = {x, y} and a relation R on A = {(x, y)}. Clearly R isn't symmetric. And Symmetry tells us that. Symmetry on A = {((x, y), (y, x)), ((y, x), (x, y))}. But look at the first element of Symmetry. It tells us that for R to be closed, whenever (x, y) is in R, (y, x) must also be. But it isn't. To compute the closure of R under Symmetry, we must add it.

- $\forall$a,b,c $\in$ A, [a $\neq$ b $\wedge$ b $\neq$ c] $\Rightarrow$ [((a, b), (b, c), (a, c)) $\in$ Transitivity]. Now we will exploit a ternary relation. Whenever the first two elements of it are present in some relation R, then the third must also be if R is transitive. This time, let's start with a set A = {x, y, z} and a relation R on A = {(x, y), (y, z)}. Clearly R is not transitive. The Transitivity relation on A is {((x, y), (y, z), (x, z)), ((x, z), (z, y), (x, y)), ((y, x), (x, z), (y, z)), ((y, z), (z, x), (y, x)), ((z, x), (x, y), (z, y)), ((z, y), (y, x), (z, x))}. Look at the first element of it. Both of the first two components of it are in R. But the third isn't. To make R transitive, we must add it.

These definitions also work to enable us to describe the closure of the integers under division as the rationals. Following the definition, A is the set of rationals. S (a subset of A) is the integers and R is QuotientClosure, defined as:

- $\forall$a,b,c $\in$ A, [a/b = c] $\Rightarrow$ [(a, b, c) $\in$ QuotientClosure].

So we've got a quite general definition of closure. And it makes it possible to prove the existence of a unique closure for any set and any relation R. The only constraint is that this definition works only if we can define the property we care about as an n-ary relation for some finite n. There are cases of closure where this isn't possible, as we saw above, but we won't need to worry about them in this class.

So we don't really need any new definitions. We've offered a general definition of closure of a set (any set) under some relation (which is the way we use to define a property). But most of the cases of closure that we'll care about involve the special case of the closure of a binary relation given some property that may or may not be naturally describable as a relation. For example, one could argue that the relations we just described to define the properties of being reflexive, symmetric, and transitive are far from natural. Thus it will be useful to offer the following alternative definitions. Don't get confused though by the presence of two definitions. Except when we cannot specify our property P as a relation (and we won't need to deal with any such cases), these new definitions are simply special cases of the one we already have.

We say that a binary relation B on a set T is ***closed under*** property P if B possesses P. For example, LessThanOrEqual is closed under transitivity since it is transitive. Simple enough. Next:

Let B be a binary relation on a set T. A relation B' is a ***closure*** of B with respect to some property P iff:

1. B $\subseteq$ B',
2. B' is closed under P, and
3. There is no smaller relation B" that contains B and is closed under P.

So, for example, the transitive closure of B = {(1, 2), (2, 3)} is the smallest new relation B' that contains B but is transitive. So B' = {(1, 2), (2, 3), (1, 3)}.

You'll generally find it easier to use these definitions than our earlier ones. But keep in mind that, with the earlier definitions it is possible to prove the existence of a unique closure. Since we went through the process of defining reflexivity, symmetry, and transitivity using those definitions, we know that there always exists a unique reflexive, symmetric, and transitive closure for any binary relation. We can exploit that fact that the same time that we use the simpler definitions to help us find algorithms for computing those closures.

8.2 Computing Closures

Suppose we are given a set and a property and we want to compute the closure of the set with respect to the property. Let's consider two examples:

- Compute the symmetric closure of a binary relation B on a set A. This is trivial. Simply make sure that, for all elements x of A, (x, x) $\in$ B.
- Compute the transitive closure of a binary relation B on a set A. This is harder. We can't just add a fixed number of elements to B and then quit. Every time we add a new element, such as (x, y), we have to look to see whether there's some element (y, z) so now we also have to add (x, z). And, similarly, we must check for any element (w, x) that would force us to add (w, y). If A is infinite, there is no guarantee that this process will ever terminate. Our theorem that guaranteed that a unique closure exists did not guarantee that it would contain a finite number of elements and thus be computable in a finite amount of time.

We can, however, guarantee that the transitive closure of any binary relation on a *finite* set is computable. How? A very simple approach is the following algorithm for computing the transitive closure of a binary relation B with N elements on a set A:

```
Set Trans = B;                              /* Initially Trans is just the original relation.
/* We need to find all cases where (x, y) and (y, z) are in Trans. Then we must insert (x, z) into Trans if
/*    it isn't already there.
Boolean AddedSomething = True;              /* We'll keep going until we make one whole pass through
                                               without adding any new elements to Trans.
while AddedSomething = True do
        AddedSomething = False;
        Xcounter := 0;
        Foreach element of Trans do
                xcounter := xcounter + 1;
                x = Trans[xcounter][1]      /* Pull out the first element of the current element of Trans
                y = Trans[xcounter][2]      /* Pull out the second element of the i'th element of Trans
                                            /* So if the first element of Trans is (p, q), then
                                            /* x = p and y = q the first time through.
                zcounter := 0;
                foreach element of Trans do
                        zcounter := zcounter + 1;
                        if Trans[zcounter][1] = y then do    /* We've found another element (y, ?) and we may need to
                                z = Trans[zcounter][2];                    /* add (x, ?) to Trans.
                                if (x, z) ∉ Trans then do    /* we have to add it
                                        Insert(Trans, (x, z))
                                        AddedSomething = True;
        end;    end;    end;    end;    end;
```

This algorithm works. Try it on some simple examples. But it's very inefficient. There are much more efficient algorithms. In particular, if we represent a relation as an incidence matrix, we can do a lot better. Using Warshall's algorithm, for example, we can find the transitive closure of a relation of n elements using $2n^3$ bit operations. For a description of that algorithm, see, for example, Kenneth Rosen, *Discrete Mathematics and its Applications*, McGraw-Hill.

9 Proof by Mathematical Induction

In the last section but one, as a sideline to our main discussion of cardinality, we presented diagonalization as a proof technique. In this section, we'll present one other very useful proof technique.

Mathematical induction is a technique for proving assertions about the set of positive integers. A proof by induction of assertion A about some set of positive integers greater than or equal to some specific value has two steps:

1. Prove that A holds for the smallest value we're concerned with. We'll call this value v. Generally v = 0 or 1, but sometimes A may hold only once we get past some initial unusual cases.
2. Prove that $\forall n \geq v, A(n) \Rightarrow A(n+1)$

We'll call A(n) the ***induction hypothesis***. Since we're trying to prove that $A(n) \Rightarrow A(n+1)$, we can assume the induction hypothesis as an axiom in our proof.

Let's do a simple example and use induction to prove that the sum of the first n odd positive integers is n^2. Notice that this appears to be true:

(n = 1) $1 = 1 = 1^2$
(n = 2) $1 + 3 = 4 = 2^2$
(n = 3) $1 + 3 + 5 = 9 = 3^2$
(n = 4) $1 + 3 + 5 + 7 = 16 = 4^2$, and so forth.

To prove it, we need to follow the two steps that we just described:

1. Let v = 1: $1 = 1^2$
2. Prove that, $\forall n \geq 0$,

   $$(\sum_{i=1}^{n} \text{Odd}_i = n^2) \Rightarrow (\sum_{i=1}^{n+1} \text{Odd}_i = (n+1)^2)$$

   To do this, we observe that the sum of the first n+1 odd integers is the sum of the first n of them plus the n+1'st, i.e.,

   $$\begin{aligned}\sum_{i=1}^{n+1} \text{Odd}_i &= \sum_{i=1}^{n} \text{Odd}_i + \text{Odd}_{n+1} \\ &= n^2 + \text{Odd}_{n+1} && \text{(Using the induction hypothesis)} \\ &= n^2 + 2n + 1 && (\text{Odd}_{n+1} \text{ is } 2n + 1) \\ &= (n + 1)^2\end{aligned}$$

Thus we have shown that the sum of the first n+1 odd integers must be equivalent to $(n+1)^2$ if it is known that the sum of the first n of them is equivalent to n^2.

Mathematical induction lets us prove properties of positive integers. But it also lets us prove properties of other things if the properties are described in terms of integers. For example, we could talk about the size of finite sets, or the length of finite strings. Let's do one with sets: For any finite set A, $|2^A| = 2^{|A|}$. In other words, the cardinality of the power set of A is 2 raised to the power of the cardinality of A. We'll prove this by induction on the number of elements of A (|A|). We follow the same two steps:

1. Let v = 0. So A is $\varnothing$, |A| = 0, and A's power set is $\{\varnothing\}$, whose cardinality is $1 = 2^0 = 2^{|A|}$.
2. Prove that, $\forall n \geq 0$, if $|2^A| = 2^{|A|}$ is true for all sets A of cardinality n, then it is also true for all sets S of cardinality n+1. We do this as follows. Since $n \geq 0$, and any such S has n + 1 elements, S must have at least one element. Pick one and call it a. Now consider the set T that we get by removing a from S. |T| must be n. So, by the induction hypothesis (namely that $|2^A| = 2^{|A|}$ if |A| = n), our claim is true for T and we have $|2^T| = 2^{|T|}$. Now let's return to the power set of the original set S. It has two parts: those subsets that include a and those that don't. The second part is exactly 2^T, so we know that it has $2^{|T|} = 2^n$ elements. The first part (all the subsets that include a) is exactly all the subsets that don't include a with a added in). Since there are 2^n subsets that don't include a and there are the same number of them once we add a to each, we have that the total number of subsets of our original set S is 2^n (for the ones that don't include a) plus 2^n (for the ones that do include a), for a total of $2(2n) = 2^{n+1}$, which is exactly $2^{|S|}$.

Why does mathematical induction work? It relies on the ***well-ordering property*** of the integers, which states that every nonempty set of nonnegative integers has a least element. Let's see how that property assures us that mathematical induction is valid as a proof technique. We'll use the technique of proof by contradiction to show that, given the well-ordering property, mathematical induction must be valid. Once we have done an induction proof, we know that A(v) (where v is 0 or 1 or some other starting value) is true and we know that $\forall n \geq 0$, A(n) $\Rightarrow$ A(n+1). What we're using the technique to enable us to claim is that, therefore, $\forall n \geq v$, A(n). Suppose the technique were not valid and there was a set S of nonnegative integers $\geq$ v for which A(n) is False. Then, by the well-ordering property, there is a smallest element in this set. Call it x. By definition, x must be equal to or greater than v. But it cannot actually be v because we proved A(v). So it must be greater than v. But now consider x - 1. Since x - 1 is less than x, it cannot be in S (since we chose x to be the smallest value in S). If it's not in S, then we know A(x - 1). But we proved that $\forall n \geq 0$, A(n) $\Rightarrow$ A(n+1), so A(x - 1) $\Rightarrow$ A(x). But we assumed $\neg$ A(x). So that assumption led us to a contradiction; thus it must be false.

Sometimes the principle of mathematical induction is stated in a slightly different but formally equivalent way:

1. Prove that A holds for the smallest value v with which we're concerned.
2. State the ***induction hypothesis*** H, which must be of the form, "There is some integer $n \geq v$ such that A is true for all integers k where $v \leq k \leq n$."
3. Prove that ($\forall n \geq v$, A(n)) $\Rightarrow$ A(n + 1). In other words prove that whenever A holds for all nonnegative integers starting with v, up to an including n, it must also hold for n + 1.

You can use whichever form of the technique is easiest for a particular problem.

Regular Languages and Finite State Machines

1 Regular Languages

The first class of languages that we will consider is the regular languages. As we'll see later, there are several quite different ways in which regular languages can be defined, but we'll start with one simple technique, regular expressions.

A ***regular expression*** is an expression (string) whose "value" is a language. Just as 3 + 4 and 14/2 are two arithmetic expressions whose values are equal, so are $(a \cup b)^*$ and $a^* \cup (a \cup b)^*b(a \cup b)^*$ two regular expressions whose values are equal. We will use regular expressions to denote languages just as we use arithmetic expressions to denote numbers. Just as there are some numbers, like Π, that cannot be expressed by arithmetic expressions of integers, so too there are some languages, like a^nb^n, that cannot be expressed by regular expressions. In fact, we will define the class of ***regular languages*** to be precisely those that *can* be described with regular expressions.

Let's continue with the analogy between arithmetic expressions and regular expressions. The syntax of arithmetic expressions is defined recursively:

1. Any numeral in {0, 1,2, ...} is an arithmetic expression.
2. If α and β are expressions, so is $(\alpha + \beta)$.
3. If α and β are expressions, so is $(\alpha * \beta)$.
4. If α and β are expressions, so is $(\alpha - \beta)$.
5. If α and β are expressions, so is (α/β).
6. If α is an expression, so is $-\alpha$.

These operators that we've just defined have associated with them a set of precedence rules, so we can write -3 + 4*5 instead of (-3 + (4*5)).

Now let's return to regular expressions. The syntax of regular expressions is also defined recursively:

1. $\varnothing$ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is $\alpha\beta$
3. If α , β are regular expressions, then so is $\alpha\cup\beta$.
4. If α is a regular expression, then so is α^*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

Similarly there are precedence rules for regular expressions, so we can write $a^* \cup bc$ instead of $(a^* \cup (bc))$. Note that * binds more tightly than does concatenation, which binds more tightly than $\cup$.

In both cases (arithmetic expressions and regular expressions) there is a distinction between the expression and its value. 5 + 7 is not the same expression as 3 * 4, even though they both have the same value. (You might imagine the expressions being in quotation marks: "5 + 7" is clearly not the same as "3 * 4". Similarly, "John's a good guy" is a different sentence from "John is nice", even though they both have the same meaning, more or less.) The rules that determine the value of an arithmetic expression are quite familiar to us, so much so that we are usually not consciously aware of them. But regular expressions are less familiar, so we will explicitly specify the rules that define the values of regular expressions. We do this by recursion on the structure of the expression. Just remember that the regular expression itself is a syntactic object made of parentheses and other symbols, whereas its value is a language. We define L() to be the function that maps regular expressions to their values. We might analogously define L() for arithmetic expressions and say that L(5 + 7) = 12 = L(3 * 4). L() is an example of a ***semantic interpretation function***, i. e., it is a function that maps a string in some language to its meaning. In our case, of course, the language from which the arguments to L() will be drawn is the language of regular expressions as defined above.

L() for regular expressions is defined as follows:

1. $L(\varnothing) = \varnothing$ and $L(a) = \{a\}$ for each $a \in \Sigma$
2. If α, β are regular expressions, then

$L((\alpha\beta)) = L(\alpha)\, L(\beta)$

= all strings that can be formed by concatenating to some string from α some string from β.

Note that if either α or β is $\varnothing$, then its language is $\varnothing$, so there is nothing to concatenate and the result is $\varnothing$.

3. If α, β are regular expressions, then $L((\alpha\cup\beta)) = L(\alpha) \cup L(\beta)$
4. If α is a regular expression, then $L(\alpha^*) = L(\alpha)^*$
5. $L((\alpha)) = L(\alpha)$

So, for example, let's find the meaning of the regular expression $(a \cup b)^*b$:

$$\begin{aligned} & L((a \cup b)^*b) \\ =\ & L((a \cup b)^*)\, L(b) \\ =\ & L(a \cup b)^*\, L(b) \\ =\ & (L(a) \cup L(b))^*\, L(b) \\ =\ & (\{a\} \cup \{b\})^*\, \{b\} \\ =\ & \{a, b\}^*\, \{b\} \end{aligned}$$

which is just the set of all strings ending in b. Another example is $L(((a \cup b)(a \cup b))a(a \cup b)^*) = \{xay\text{: x and y are strings of a's and b's and } |x| = 2\}$. The distinction between an expression and its meaning is somewhat pedantic, but you should try to understand it. We will usually not actually write L() because it is generally clear from context whether we mean the regular expression or the language denoted by it. For example, $a \in (a \cup b)^*$ is technically meaningless since $(a \cup b)^*$ is a regular expression, not a set. Nonetheless, we use it as a reasonable abbreviation for $a \in L((a \cup b)^*)$, just as we write 3 + 4 = 4 + 3 to mean that the values of "3 + 4" and "4 + 3", not the expressions themselves, are identical.

Here are some useful facts about regular expressions and the languages they describe:

- $(a \cup b)^* = (a^*b^*)^* = (b^*a^*)^*$ = set of all strings composed exclusively of a's and b's (including the empty string)
- $(a \cup b)c = (ac \cup bc)$ Concatenation distributes over union
- $c(a \cup b) = (ca \cup cb)$ "
- $a^* \cup b^* \neq (a \cup b)^*$ The right-hand expression denotes a set containing strings of mixed a's and b's, while the left-hand expression does not.
- $(ab)^* \neq a^*b^*$ In the right-hand expression, all a's must precede all b's. That's not true for the left-hand expression.
- $a^* \cup \varnothing^* = a^* \cup \varepsilon = a^*$

There is an algebra of regular expressions, but it is rather complex and not worth the effort to learn it. Therefore, we will rely primarily on our knowledge of what the expressions mean to determine the equivalence (or non-equivalence) or regular expressions.

We are now in a position to state formally our definition of the class of ***regular languages***: Given an alphabet Σ, the set of regular languages over Σ is precisely the set of languages that can be defined using regular expressions with respect to Σ. Another equivalent definition (given our definition of regular expressions and L()), is that the set of regular languages over an alphabet Σ is the smallest set that contains $\varnothing$ and each of the elements of Σ, and that is closed under the operations of concatenation, union, and Kleene star (rules 2, 3, and 4 above).

2 Proof of the Equivalence of Nondeterministic and Deterministic FSAs

In the lecture notes, we stated the following:

> ***Theorem***: For each NDFSA, there is an equivalent DFSA.

This is an extremely significant theorem. It says that, for finite state automata, nondeterminism doesn't add power. It adds convenience, but not power. As we'll see later, this is not true for all other classes of automata.

In the notes, we provided the first step of a proof of this theorem, namely an algorithm to construct, from any NDFSA, an equivalent DFSA. Recall that the algorithm we presented was the following: Given a nondeterministic FSA M (K, Σ, Δ, s, F), we derive an equivalent deterministic FSA M' = (K', Σ, δ', s', F') as follows:

1. Compute E(q) for each q in K. $\forall q \in K$, $E(q) = \{p \in K : (q, \varepsilon) \mid\!-^*_M (p, \varepsilon)\}$. In other words, E(q) is the set of states reachable from q without consuming any input.
2. Compute s' = E(s).
3. Compute δ', which is defined as follows: $\forall Q \in 2^K$ and $\forall a \in \Sigma$, $\delta'(Q, a) = \cup\{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$. Recall that the elements of 2^K are sets of states from the original machine M. So what we've just said is that to compute the transition out of one of these "set" states, find all the transitions out of the component states in the original machine, then find all the states reachable from them via epsilon transitions. The new state is the set of all states reachable in this way. We'll actually compute δ' by first computing it for the new start state s' and each of the elements of Σ. Each state thus created becomes an element of K', the set of states of M'. Then we compute δ' for any new states that were just created. We continue until there are no additional reachable states. (so although δ' is defined for all possible subsets of K, we'll only bother to compute it for the reachable such subsets and in fact we'll define K' to include just the reachable configurations.)
4. Compute K' = that subset of 2^K that is reachable, via δ', as defined in step 3, from s'.
5. Compute $F' = \{Q \in K' : Q \cap F \neq \emptyset\}$. In other words, each constructed "set" state that contains at least one final state from the original machine M becomes a final state in M'.

However, to complete the proof of the theorem that asserts that there is an *equivalent* DFSA for every NDFSA, we need next to prove that the algorithm we have defined does in fact produce a machine that is (1) deterministic, and (2) equivalent to the original machine.

Proving (1) is trivial. By the definition in step 3 of δ', we are guaranteed that δ' is defined for all reachable elements of K' and that it is single valued.

Next we must prove (2). In other words, we must prove that M' accepts a string w if and only if M accepts w. We constructed the transition function δ' of M' so that each step of the operation of M' mimics an "all paths" simulation of M. So we believe that the two machines are identical, but how can we prove that they are? Suppose we could prove the following:

> ***Lemma***: For any string $w \in \Sigma^*$ and any states $p, q \in K$, $(q, w) \mid\!-^*_M (p, \varepsilon)$ iff $(E(q), w) \mid\!-^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p. In other words, if the original machine M starts in state q and, after reading the string w, can land in state p, then the new machine M' must behave as follows: when started in the state that corresponds to the set of states the original machine M could get to from q without consuming any input, M' reads the string w and lands in one of its new "set" states that contains p. Furthermore, because of the only-if part of the lemma, M' must end up in a "set" state that contains only states that M could get to from q after reading w and following any available epsilon transitions.

If we assume that this lemma is true, then the proof that M' is equivalent to M is straightforward: Consider any string $w \in \Sigma^*$. If $w \in L(M)$ (i.e., the original machine M accepts w) then the following two statements must be true:

1. The original machine M, when started in its start state, can consume w and end up in a final state. This must be true given the definition of what it means for a machine to accept a string.

2. $(E(s), w) \vdash^*_{M'} (Q, \varepsilon)$ for some Q containing some $f \in F$. In other words, the new machine, when started in its start state, can consume w and end up in one of its final states. This follows from the lemma, which is more general and describes a computation from any state to any other. But if we use the lemma and let q equal s (i.e., M begins in its start state) and p = f for some $f \in F$ (i.e., M ends in a final state), then we have that the new machine M', when started in its start state, E(s), will consume w and end in a state that contains f. But if M' does that, then it has ended up in one of its final states (by the definition of K' in step 5 of the algorithm). So M' accepts w (by the definition of what it means for a machine to accept a string). Thus M' accepts precisely the same set of strings that M does.

Now all we have to do is to prove the lemma. What the lemma is saying is that we've built M' from M in such a way that the computations of M' mirror those of M and guarantee that the two machines accept the same strings. But of course we didn't build M' to perform an entire computation. All we did was to describe how to construct δ'. In other words, we defined how individual steps of the computation work. What we need to do now is to show that the individual steps, when taken together, do the right thing for strings of any length. The obvious way to do that, since we know what happens one step at a time, is to prove the lemma by induction on |w|.

We must first prove that the lemma is true for the base case, where |w| = 0 (i.e., w = ε). To do this, we actually have to do two proofs, one to establish the *if* part of the lemma, and the other to establish the *only if* part:

Basis step, *if* part: Prove $(q, w) \vdash^*_M (p, \varepsilon)$ if $(E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p. Or, turning it around to make it a little clearer,

$[\ (E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p $] \Rightarrow (q, w) \vdash^*_M (p, \varepsilon)$

If |w| = 0, then M' makes no moves. So it must end in the same state it started in, namely E(q). If we're told that it ends in some state that contains p, then $p \in E(q)$. But, given our definition of E(x), that means exactly that, in the original machine M, p is reachable from q just be following ε transitions, which is exactly what we need to show.

Basis step, *only if* part: Recall that *only if* is equivalent to implies. So now we need to show:

$[\ (q, w) \vdash^*_M (p, \varepsilon)\ ] \Rightarrow (E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

If |w| = 0, and the original machine M goes from q to p with only w as input, it must go from q to p following just ε transitions. In other words $p \in E(q)$. Now consider the new machine M'. It starts in E(q), the set state that includes all the states that are reachable from q via ε transitions. Since the new machine is deterministic, it will make no moves at all if its input is ε. So it will halt in exactly the same state it started in, namely E(q). Since we know that $p \in E(q)$, we know that M' has halted in a set state that includes p, which is exactly what we needed to show.

Next we'll prove that if the lemma is true for all strings w of length k, $k \geq 0$, then it is true for all strings of length k + 1. Considering strings of length k + 1, we know that we are dealing with strings of a least one character. So we can rewrite any such string as zx, where x is a single character and z is a string of length k. The way that M and M' process z will thus be covered by the induction hypothesis. We'll use our definition of δ', which specifies how each individual step of M' operates, to show that, assuming that the machines behave correctly for the first k characters, they behave correctly for the last character also and thus for the entire string of length k + 1. Recall our definition of δ':

$\delta'(Q, a) = \cup\{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$.

To prove the lemma, we must show a relationship between the behavior of:

M: $(q, w) \vdash^*_M (p, \varepsilon)$, and
M': $(E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

Rewriting w as zx, we have
M: $(q, zx) \vdash^*_M (p, \varepsilon)$
M': $(E(q), zx) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

Breaking each of these computations into two pieces, the processing of z followed by the processing of x, we have:
M: $(q, zx) \vdash^*_M (s_i, x) \vdash^*_M (p, \varepsilon)$
M': $(E(q), zx) \vdash^*_{M'} (S, x) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

In other words, after processing z, M will be in some set of states s_i, and M' will be in some state, which we'll call S. Again, we'll split the proof into two parts:

<u>Induction step, *if* part</u>:

$[\ (E(q), zx) \vdash^*_{M'} (S, x)\ \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains $p\ ] \Rightarrow (q, zx) \vdash^*_{M} (s_i, x) \vdash^*_{M} (p, \varepsilon)$

If, after reading z, M' is in state S, we know, from the induction hypothesis, that the original machine M, after reading z, must be in some set of states s_i and that S is precisely that set. Now we just have to describe what happens at the last step when the two machines read x. If we have that M', starting in S and reading x lands in P, then we know, from the definition of δ' above, that P contains precisely the states that M could land in after starting in any s_i and reading x. Thus if $p \in P$, p must be a state that M could land in.

<u>Induction step, *only if* part</u>:

$(q, zx) \vdash^*_{M} (s_i, x) \vdash^*_{M} (p, \varepsilon) \Rightarrow (E(q), zx) \vdash^*_{M'} (S, x)\ \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

By the induction hypothesis, we know that if M, after processing z, can reach some set of states s_i, then S (the state M' is in after processing z) must contain precisely all the s_i's. Knowing that, and our definition of δ', we know that from S, reading x, M' must be in some set state P that contains precisely the states that M can reach starting in any of the s_i's, reading x, and then following all ε transitions. So, after consuming w (zx), M', when started in E(q) must end up in a state P that contains all and only the states p that M, when started in q, could end up in.

This theorem is a very useful result when we're dealing with FSAs. It's true that, by and large, we want to build deterministic machines. But, because we have provided a constructive proof of this theorem, we know that we can design a deterministic FSA by first describing a nondeterministic one (which is sometimes a much simpler task), and then applying our algorithm to construct a corresponding deterministic one.

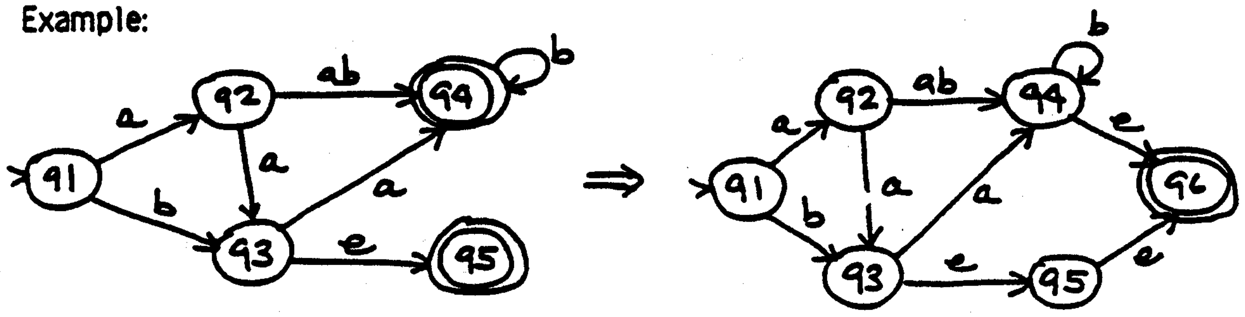
3 Generating Regular Expressions from Finite State Machines

1. Preparations (Note: FA may be non-deterministic in general)

   If a) there is more than one final state
   or b) there is just one final state but it lies on a loop,

   then a) create a new final state
   b) run e-transitions from the old final state(s) to the new one
   and c) make the old final state(s) non-final

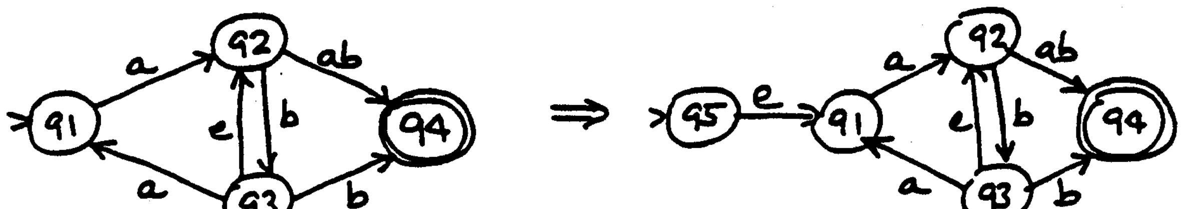
   Example:



   If the initial state is part of a loop,

   then a) create a new initial state
   b) run an e-transition from the new initial state to the old one
   and c) make the old initial state non-initial

   Example:



   (Note: nothing needs to be done to the final state here because it does not lie on a loop. Similarly, nothing needs to be done to the initial state in the first example.)

II. Eliminating states

One by one, remove states which are neither initial nor final, replacing the connections between remaining states in such a way that the transitions are preserved. In general, the reconstructed transitions may be labelled with <u>regular expressions</u> rather than just by strings.

1. Example:

$$\begin{tikzcd} q1 \arrow[r, "a"] & q2 \arrow[r, "ba"] & q3 \end{tikzcd}$$

q2 can be eliminated and the connection between q1 and q3 becomes:

$$\begin{tikzcd} q1 \arrow[r, "aba"] & q3 \end{tikzcd}$$

In general, if r1 and r2 are any regular expressions, produced perhaps by other steps in the algorithm, and occur in the configuration:

$$\begin{tikzcd} q_i \arrow[r, "r1"] & q_j \arrow[r, "r2"] & q_k \end{tikzcd}$$

then this can be replaced by

$$\begin{tikzcd} q_i \arrow[r, "r1 \cdot r2"] & q_k \end{tikzcd}$$

2) If the eliminated state happens to contain a "simple" loop, e.g.:

$$\begin{tikzcd} q1 \arrow[r, "a"] & q2 \arrow[loop above, "bb"] \arrow[r, "ba"] & q3 \end{tikzcd}$$

when q2 is eliminated, the transition becomes:

$$\begin{tikzcd} q1 \arrow[r, "a(bb)^* ba"] & q3 \end{tikzcd}$$

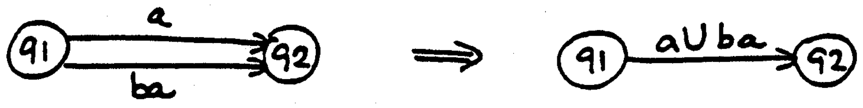
In general,

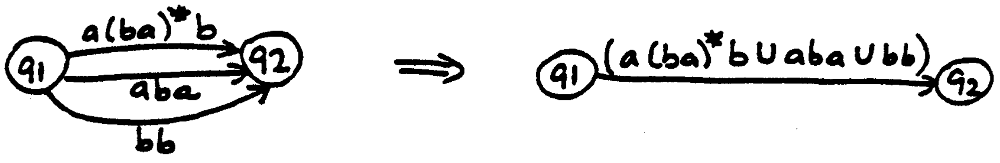
$$\begin{tikzcd} q_i \arrow[r, "r1"] & q_j \arrow[loop above, "r2"] \arrow[r, "r3"] & q_k \end{tikzcd}$$

becomes

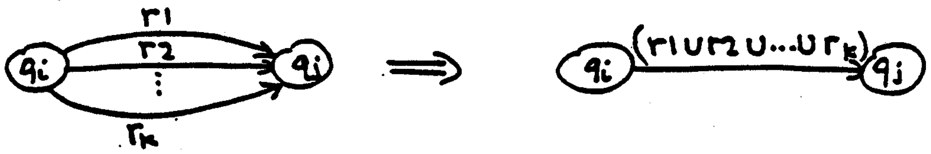
$$\begin{tikzcd} q_i \arrow[r, "r1(r2)^* r3"] & q_k \end{tikzcd}$$

3) Parallel transitions can be coalesced into a single transition labelled by an expression which is the union of the originals:





In general:

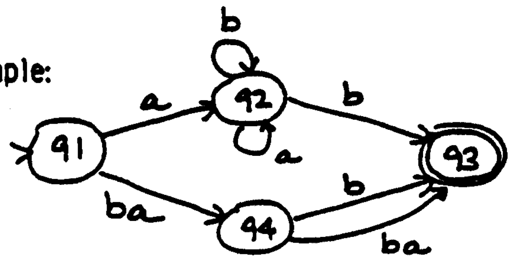


One proceeds in this manner, eliminating states one by one, until only a single transition remains connecting the initial and the one final state. The label on this transition is a regular expression for the original automaton.

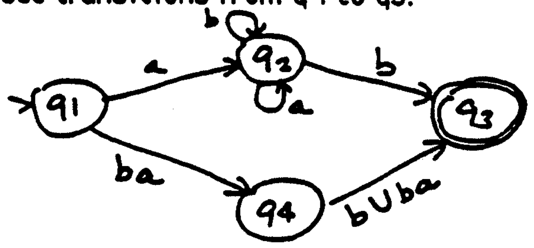
The order of elimination of states doesn't matter; equivalent regular expressions will be obtained so long as the procedure is done correctly.

Note that coalescing parallel transitions doesn't eliminate a state but reduces the number of transitions. In general, one should perform this step before eliminating a state that has parallel transitions into or out of it.

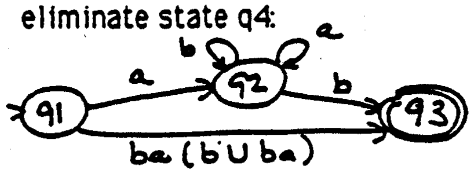
Example:



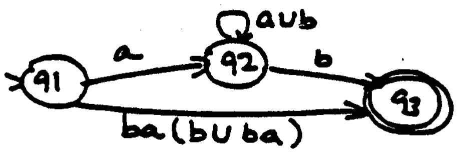
coalesce transitions from q4 to q3:



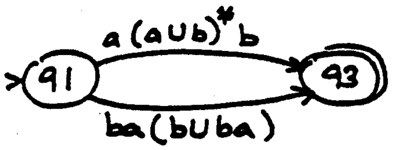
eliminate state q4:



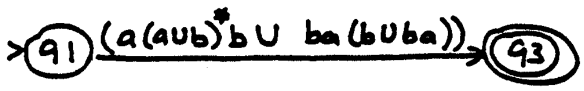
coalesce transitions from q2 to q2:



eliminate q2:

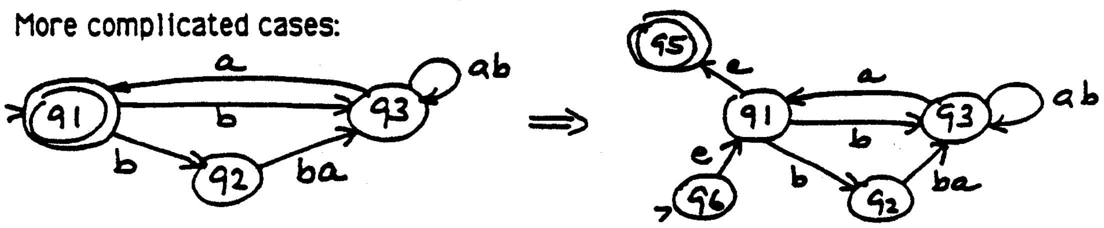


coalesce parallel transitions:

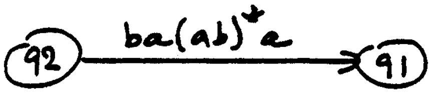


This is a regular expression for the FA. Check against the original.

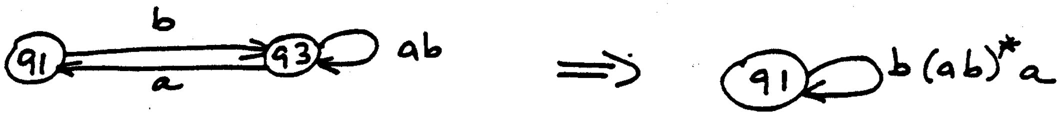
More complicated cases:



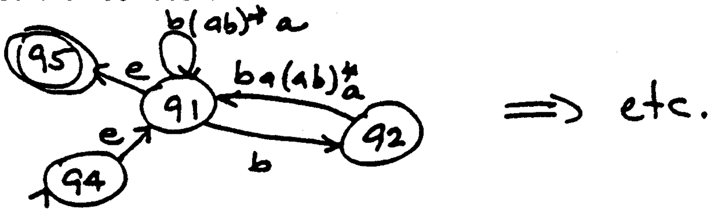
suppose we eliminate q3. Then the path from q2 to q1 becomes:



but we have also destroyed a path from q1 back to q1, namely:



so the net result is:



4 The Pumping Lemma for Regular Languages

4.1 What Does the Pumping Lemma Tell Us?

The pumping lemma is a powerful technique for showing that a language is **not** regular. The lemma describes a property that must be true of any language if it is regular. Thus, if we can show that some language L does not possess this property, then we know that L is not regular.

The key idea behind the pumping lemma derives from the fact that every regular language is recognizable by some FSM M (actually an infinite number of finite state machines, but we can just pick any one of them for our purposes here). So we know that, if a language L is regular, then every string s in L must drive M from the start state to some final state. There are only two ways this can happen:

1. s could be short and thus drive M from the start state to a final state without traversing any loops. By short we mean that if M has N states, then $|s| < N$. If s were any longer, M would have to visit some state more than once while processing s; in other words it would loop.
2. s could be long and M could traverse at least one loop. If it does that, then observe that another way to take M from the start state to the same final state would be to skip the loop. Some shorter string w in L would do exactly that. Still further ways to take M from the start state to the final state would be to traverse the loop two times, or three times, or any number of times. An infinite set of longer strings in L would do this.

Given some regular language L, we know that there exists an infinite number of FSMs that accept L, and the pumping lemma relies on the existence of at least one such machine M. In fact, it needs to know the number of states in M. But what if we don't have M handy? No problem. All we need is to know that M exists and that it has some number of states, which we'll call N. As long as we make no assumptions about what N is, other than that it is at least 1, we can forge ahead without figuring out what M is or what N is.

The pumping lemma tells us that if a language L is regular, then any sufficiently long (as defined above) string s in L must be "pumpable". In other words, there is some substring t of s that corresponds to a loop in the recognizing machine M. Any string that can be derived from s either by pumping t out once (to get a shorter string that will go through the loop one fewer times) or by pumping t in any number of times (to get longer strings that will go through the loop additional times) must also be in L since they will drive M from its start state to its final state. If we can show that there is even one sufficiently long string s that isn't pumpable, then we know that L must not be regular.

You may be wondering why we can only guarantee that we can pump out once, yet we must be able to pump in an arbitrary number of times. Clearly we must be able to pump in an arbitrary number of times. If there's a loop in our machine M, there is no limit to the number of times we can traverse it and still get to a final state. But why only once for pumping out? Sometimes it may in fact be possible to pump out more. But the lemma doesn't require that we be able to. Why? When we pick a string s that is "sufficiently long", all we know is that it is long enough that M must visit at least one state more than once. In other words, it must traverse at least one loop of length one at least once. It may do more, but we can't be sure of it. So the only thing we're guaranteed is that we can pump out the one pass through the loop that we're sure must exist.

Pumping Lemma:

If L is regular, then
  $\exists\, N \geq 1$, such that
    $\forall$ strings w, where $|w| \geq N$,
      $\exists$ x, y, z, such that $w = xyz$, and
        $|xy| \leq N$, and
        $y \neq \varepsilon$, and
        $\forall\, q \geq 0$, xy^qz is in L.

The lemma we've just stated is sometimes referred to as the Strong Pumping Lemma. That's because there is a weaker version that is much less easy to use, yet no easier to prove. We won't say anything more about it, but at least now you know what it means if someone refers to the Strong Pumping Lemma.

4.2 Using the Pumping Lemma

The key to using the pumping lemma correctly to prove that a language L is not regular is to understand the nested quantifiers in the lemma. Remember, our goal is to show that our language L fails to satisfy the requirements of the lemma (and thus is not regular). In other words, we're looking for a counterexample. But when do we get to pick any example we want and when do we have to show that there is no example? The lemma contains both universal and existential quantifiers, so we'd expect some of each. But the key is that we want to show that the lemma does not apply to our language L. So we're essentially taking the not of it. What happens when we do that? Remember two key results from logic:

$\neg\forall x\ P(x) = \exists x\ \neg P(x)$

$\neg\exists x\ P(x) = \forall x\ \neg P(x)$

So if the lemma says something must be true for all strings, we show that there exists at least one string for which it's false. If the lemma says that there exists some object with some properties, we show that all possible objects fail to have those properties. More specifically:

At the top level, the pumping lemma states

L regular $\Rightarrow$ $\exists N \geq 1$, P(L, N), where P is the rest of the lemma (think of P as the Pumpable property).

To show that the lemma does not correctly describe our language L, we must show

$\neg(\exists N\ N \geq 1, P(L, N))$, or, equivalently,

$\forall N\ \neg P(L, N)$

The lemma asserts first that there exists some magic number N, which defines what me mean by "sufficiently long." The lemma doesn’t tell us what N is (although we know it derives from the number of states in some machine M that accepts L). We need to show that no such N exists. So we don't get to pick a specific N to work with. Instead:

We must carry N through the rest of what we do as a variable and make no assumptions about it.

Next, we must look inside the pumpable property. The lemma states that every string that is longer than N must be pumpable. To prove that that isn't true of L, all we have to do is to find a single long string in L that isn't pumpable. So:

We get to pick a string w.

Next, the lemma asserts that there is a way to carve up our string into substrings x, y, and z such that pumping works. So we must show that there is no such x, y, z triple. This is the tricky part. To show this, we must enumerate all logically possible ways of carving up w into x, y, and z. For each such possibility, we must show that at least one of the pumping requirements is not satisfied. So:

We don't get to pick x, y, and z. We must show that pumping fails for all possible x, y, z triples.

Sometimes, we can show this easily without actually enumerating ways of carving up w into x, y, and z. But in other cases, it may be necessary to enumerate two or more possibilities.

Let's look at an example. The classic one is L = $a^x b^x$ is not regular. Intuitively, we knew it couldn't be. To decide whether a string is in L, we have to count the a's, and then compare that number to the number of b's. Clearly no machine with a finite number of states can do that. Now we're actually in a position to prove this. We show that for any value of N we'll get a contradiction. We get to choose any w we want as long as it's length is greater than or equal to N. Let's choose w to be be $a^N b^N$. Next, we must show that there is no x, y, z triple with the required properties:

$|xy| \leq N$,

$y \neq \varepsilon$,

$\forall\ q \geq 0$, xy^qz is in L.

Suppose there were. Then it might look something like this (this is just for illustration):

$$\begin{array}{c} 1 \qquad | \qquad 2 \\ \underline{a\,a\,a\,a\,a\,a\,a}\ \underline{a\,a}\ \underline{a\,b\,b\,b\,b\,b\,b\,b\,b\,b\,b} \end{array}$$

x y z

Don't take this picture to be making any claim about what x, y, and z are. But what the picture does show is that w is composed of two regions:

1. The initial segment, which is all a's.
2. The final segment, which is all b's.

Typically, as we attempt to show that there is no x, y, z triple that satisfies all the conditions of the pumping lemma, what we'll do is to consider the ways that y can be spread within the regions of w. In this example, we observe immediately that since $|xy| \leq N$, y must be a^g for some $g \geq 1$. (In other words, y must lie completely within the first region.) Now there's just one case to consider. Clearly we'll add just a's as we pump, so there will be more a's than b's, so we'll generate strings that are not in L. Thus w is not pumpable, we've found a contradiction, and L is not regular.

The most common mistake people make in applying the pumping lemma is to show a particular x, y, z triple that isn't pumpable. Remember, you must show that all such triples fail to be pumpable.

Suppose you try to apply the pumping lemma to a language L and you fail to find a counterexample. In other words, every string w that you examine is pumpable. What does that mean? Does it mean that L is regular? No. It's true that if L is regular, you will be unable to find a counterexample. But if L isn't regular, you may fail if you just don't find the right w. In other words, even in a non regular language, there may be plenty of strings that are pumpable. For example, consider $L = a^x b^x \cup a^y$. In other words, if there are any b's there must be the same number of them as there are a's, but it's also okay just to have a's. We can prove that this language is not pumpable by choosing $w = a^N b^N$, just as we did for our previous example, and the proof will work just as it did above. But suppose that were less clever. Let's choose $w = a^N$. Now again we know that y must be a^g for some $g \geq 1$. But now, if we pump y either in or out, we still get strings in L, since all strings that just contain a's are in L. We haven't proved anything.

Remember that, when you go to apply the pumping lemma, the one thing that is in your control is the choice of w. As you get experience with this, you'll notice a few useful heuristics that will help you find a w that is easy to work with:

1. Choose w so that there are distinct regions, each of which contains only one element of Σ. When we considered $L = a^x b^x$, we had no choice about this, since every element of L (except ε) must have a region of a's followed by a region of b's. But suppose we were interested in $L' = \{w \in \{a, b\}^*$: w contains an equal number of a's and b's$\}$. We might consider choosing $w = (ab)^N$. But now there are not clear cut regions. We won't be able to use pumping successfully because if y = ab, then we can pump to our hearts delight and we'll keep getting strings in L. What we need to do is to choose $a^N b^N$, just as we did when we were working on L. Sure, L' doesn't require that all the a's come first. But strings in which all the a's do come first are fine elements of L', and they produce clear cut regions that make the pumping lemma useful.
2. Choose w so that the regions are big enough that there is a minimal number of configurations for y across the regions. In particular, you must pick w so that it has length at least N. But there's no reason to be parsimonious. For example, when we were working on $a^x b^x$, we could have chosen $w = a^{N/2} b^{N/2}$. That would have been long enough. But then we couldn't have known that y would be a string of a's. We would have had to consider several different possibilities for y. (You might want to work this one out to see what happens.) It will generally help to choose w so that each region is of length at least N.
3. Whenever possible, choose w so that there are at least two regions with a clear boundary between them. In particular, you want to choose w so that there are at least two regions that must be related in some way (e.g., the a region must be the same length as the b region). If you follow this rule and rule 2 at the same time, then you'll be assured that as you pump y, you'll change one of the regions without changing the other, thus producing strings that aren't in your language.

The pumping lemma is a very powerful tool for showing that a language isn't regular. But it does take practice to use it right. As you're doing the homework problems for this section, you may find it helpful to use the worksheet that appears on the next page.

Using the Pumping Lemma for Regular Languages

If L is regular, then

There exists an $N \geq 1$, (Just **call it N**) such that

for all strings w, where $|w| \geq N$,

(Since true for all w, it must be true for any particular one, so you **pick w**)

(Hint: describe w in terms of N)

there exist x, y, z, such that w = xyz

and $|xy| \leq N$,

and $y \neq \varepsilon$,

and for all $q \geq 0$, xy^qz is in L.

(Since must hold for all y, we **show that it can't hold for any y that meets the requirements: $|xy| \leq N$, and $y \neq \varepsilon$. To do this:**

Write out w:

List all the possibilities for y:

[1]

[2]

[3]

[4]

For each possibility for y, xy^qz must be in L, for all q. So:

For each possibility for y, find some value of q such that xy^qz is not in L. Generally q will be either 0 or 2.

	y	**q**
[1]		
[2]		
[3]		
[4]		

Q.E.D.

Context-Free Languages and Pushdown Automata

1 Context-Free Grammars

Suppose we want to generate a set of strings (a language) L over an alphabet Σ**.** How shall we specify our language? One very useful way is to write a grammar for L. A ***grammar*** is composed of a set of rules. Each rule may make use of the elements of Σ (which we'll call the ***terminal alphabet*** or ***terminal vocabulary)***, as well as an additional alphabet, the ***non-terminal alphabet*** or ***vocabulary***. To distinguish between the terminal alphabet Σ and the non-terminal alphabet, we will use lower-case letters: a, b, c, etc. for the terminal alphabet and upper-case letters: A, B, C, S, etc. for the non-terminal alphabet. (But this is just a convention. Any character can be in either alphabet. The only requirement is that the two alphabets be disjoint.)

A grammar generates strings in a language using ***rules***, which are instructions, or better, licenses, to replace some non-terminal symbol by some string. Typical rules look like this:

$S \rightarrow ASa$, $B \rightarrow aB$, $A \rightarrow SaSSbB$.

In context-free grammars, rules have a single non-terminal symbol (upper-case letter) on the left, and any string of terminal and/or non-terminal symbols on the right. So even things like $A \rightarrow A$ and $B \rightarrow \varepsilon$ are perfectly good context-free grammar rules. What's not allowed is something with more than one symbol to the left of the arrow: $AB \rightarrow a$, or a single terminal symbol: $a \rightarrow Ba$, or no symbols at all on the left: $\varepsilon \rightarrow Aab$. The idea is that each rule allows the replacement of the symbol on its left by the string on its right. We call these grammars context free because every rule has just a single nonterminal on its left. We can't add any contextual restrictions (such as aAa). So each replacement is done independently of all the others.

To generate strings we start with a designated ***start symbol*** often S (for "sentence"), and apply the rules as many times as we please whenever any one is applicable. To get this process going, there will clearly have to be at least one rule in the grammar with the start symbol on the left-hand side. (If there isn't, then the grammar won't generate any strings and will therefore generate $\varnothing$, the empty language.) Suppose, however, that the start symbol is S and the grammar contains both the rules $S \rightarrow AB$ and $S \rightarrow aBaa$. We may apply either one, producing AB as the "working string" in the first case and aBaa in the second.

Next we need to look for rules that allow further rewriting of our working string. In the first case (where the working string is AB), we want rules with either A or B on the left (any non-terminal symbol of the working string may be rewritten by rule at any time); in the latter case, we will need a rule rewriting B. If, for example, there is a rule $B \rightarrow aBb$, then our first working string could be rewritten as AaBb (the A stays, of course, awaiting its chance to be replaced), and the second would become aaBbaa.

How long does this process continue? It will necessarily stop when the working string has no symbols that can be replaced. This would happen if either:

(1) the working string consists entirely of terminal symbols (including, as a special case, when the working string is ε, the empty string), or
(2) there are non-terminal symbols in the working string but none appears on the left-hand side of any rule in the grammar (e.g., if the working string were AaBb, but no rule had A or B on the left).

In the first case, but not the second, we say that the working string is ***generated by the grammar***. Thus, a grammar generates, in the technical sense, only strings over the terminal alphabet, i.e., strings in Σ^*. In the second case, we have a ***blocked*** or ***non-terminated derivation*** but no generated string.

It is also possible that in a particular case neither (1) nor (2) is achieved. Suppose, for example, the grammar contained only the rules $S \rightarrow Ba$ and $B \rightarrow bB$, with S the start symbol. Then using the symbol $\Rightarrow$ to connect the steps in the rewriting process, all derivations proceed in the following way:

$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow ...$

The working string is always rewriteable (in only one way, as it happens), and so this grammar would not produce any terminated derivations, let alone any terminated derivations consisting entirely of terminal symbols (i.e., generated strings). Thus this grammar generates the language $\varnothing$.

Now let us look at our definition of a context-free grammar in a somewhat more formal way. A context-free grammar (CFG) G consists of four things:

(1) V, a finite set (the total alphabet or vocabulary), which contains two subsets, Σ (the ***terminal symbols***, i.e., the ones that will occur in strings of the language) and $V - \Sigma$ (the ***nonterminal symbols***, which are just working symbols within the grammar).

(2) Σ, a finite set (the terminal alphabet or terminal vocabulary).

(3) R, a finite subset of $(V - \Sigma) \times V^*$, the set of ***rules.*** Although each rule is an ordered pair (nonterminal, string), we'll generally use the notion nonterminal $\rightarrow$ string to describe our rules.

(4) S, the ***start symbol*** or ***initial symbol***, which can be any member of $V - \Sigma$.

For example, suppose $G = (V, \Sigma, R, S)$, where

$V = \{S, A, B, a, b\}$, $\Sigma = \{a, b\}$, and $R = \{S \rightarrow AB, A \rightarrow aAa, A \rightarrow a, B \rightarrow Bb, B \rightarrow b\}$

Then G generates the string aaabb by the following derivation:

(1) $S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAaBb \Rightarrow aaaBb \Rightarrow aaabb$

Formally, given a grammar G, the two-place relation on strings called "derives in one step" and denoted by $\Rightarrow$ (or by $\Rightarrow_G$ if we want to remind ourselves that the relation is relative to G) is defined as follows:

$(u, v) \in \Rightarrow$ iff $\exists$ strings $w, x, y \in V^*$ and symbol $A \in (V - \Sigma)$ such that $u = xAy$, $v = xwy$, and $(A \rightarrow w) \in R$.

In words, two strings stand in the "derives in one step" relation for a given grammar just in case the second can be produced from the first by rewriting a single non-terminal symbol in a way allowed by the rules of the grammar.

$(u, v) \in \Rightarrow$ is commonly written in infix notation, thus: $u \Rightarrow v$.

This bears an obvious relation to the "yields in one step" relation defined on configurations of a finite automaton. Recall that there we defined the "yields in zero or more steps" relation by taking the reflexive transitive closure of the "yields in one step" relation. We'll do that again here, giving us "yields in zero or more steps" denoted by $\Rightarrow^*$ (or $\Rightarrow_G^*$, to be explicit), which holds of two strings iff the second can be derived from the first by finitely many successive applications of rules of the grammar. In the example grammar above:

- $S \Rightarrow AB$, and therefore also $S \Rightarrow^* AB$.
- $S \Rightarrow^* aAaB$, but not $S \Rightarrow aAaB$ (since aAaB cannot be derived from S in one step).
- $A \Rightarrow aAa$ and $A :\Rightarrow^* aAa$ (This is true even though A itself is not derivable from S. If this is not clear, read the definitions of $\Rightarrow$ and $\Rightarrow^*$ again carefully.)
- $S \Rightarrow^* S$ (taking zero rule applications), but not $S \Rightarrow S$ (although the second would be true if the grammar happened to contain the rule $S \rightarrow S$, a perfectly legitimate although rather useless rule). Note carefully the difference between $\rightarrow$, the connective used in grammar rules, versus $\Rightarrow$ and $\Rightarrow^*$, indicators that one string can be derived from another by means of the rules.

Formally, given a grammar G, we define a ***derivation*** to be any sequence of strings

$w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$

In other words, a derivation is a finite sequence of strings such that each string, except the first, is derivable in one step from the immediately preceding string by the rules of the grammar. We can also refer to it as a derivation of w_n from w_0. Such a derivation is said to be of length n, or to be a derivation of n ***steps***. (1) above is a 5-step derivation of aaabb from S according to the given grammar G.

Similarly, $A \Rightarrow aAa$ is a one-step derivation of aAa from A by the grammar G. (Note that derivations do not have to begin with S, nor indeed do they have to begin with a working string derivable from S. Thus, $AA \Rightarrow aAaA \Rightarrow aAaa$ is also a well-formed derivation according to G, and so we are entitled to write $AA \Rightarrow^* aAaa$).

The ***strings generated by*** a grammar G are then just those that are (i) derivable from the start symbol, and (ii) composed entirely of terminal symbols. That is, $G = (V, \Sigma, R, S)$ generates w iff $w \in \Sigma^*$ and $S \Rightarrow^* w$. Thus, derivation (l) above shows that the string aaabb is generated by G. The string aAa, however, is not generated by G, even though it is derivable from S, because it contains a non-terminal symbol. It may be a little harder to see that the string bba is not generated by G. One would have to convince oneself that there exists no derivation beginning with S and ending in bba according to the rules of G. (Question: Is this always determinable in general, given any arbitrary context-free grammar G and string w? In other words, can one always tell whether or not a given w is "grammatical" according to G? We'll find out the answer to this later.)

The ***language generated by*** a grammar G is exactly the set of all strings generated--no more and no less. The same remarks apply here as in the case of regular languages: a grammar generates a language iff every string in the language is generated by the grammar and no strings outside the language are generated.

And now our final definition (for this section). A language L is ***context free*** if and only if there exists a context-free grammar that generates it.

Our example grammar happens to generate the language a(aa)*bb*. To prove this formally would require a somewhat involved argument about the nature of derivations allowed by the rules of G, and such a proof would not necessarily be easily extended to other grammars. In other words, if you want to prove that a given grammar generates a particular language, you will in general have to make an argument which is rather specific to the rules of the grammar and show that it generates all the strings of the particular language and only those. To prove that a grammar generates a particular string, on the other hand, it suffices to exhibit a derivation from the start symbol terminating in that string. (Question: if such a derivation exists, are we guaranteed that we will be able to find it?) To prove that a grammar does not generate a particular string, we must show that there exists no derivation that begins with the start symbol and terminates in that string. The analogous question arises here: when can we be sure that our search for such a derivation is fruitless and be called off? (We will return to these questions later.)

2 Designing Context-Free Grammars

To design a CFG for a language, a helpful heuristic is to imagine generating the strings from the outside in to the middle. The nonterminal that is currently "at work" should be thought of as being at the middle of the string when you are building a string where two parts are interdependent. Eventually the "balancing" regions are done being generated, and the nonterminal that's been doing the work will give way to a different nonterminal (if there's more stuff to be done between the regions just produced) or to some terminal string (often ε) otherwise. If parts of a string have nothing to do with each other, do not try to produce them both with one rule. Try to identify the regions of the string that must be generated in parallel due to a correlation between them: they must be generated by the same nonterminal(s). Regions that have no relation between each other can be generated by different nonterminals (and usually should be.)

Here is a series of examples building in complexity. For each one you should generate a few sample strings and build parse trees to get an intuition about what is going on. One notational convention that we'll use to simplify writing language descriptions: If a description makes use of a variable (e.g., a^n), there's an implied statement that the description holds for all integer values ≥ 0.

Example 1: The canonical example of a context-free language is $L = a^n b^n$, which is generated by the grammar

$G = (\{S, a, b\}, \{a, b\}, R, S)$ where $R = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$.

Each time an a is generated, a corresponding b is generated. They are created in parallel. The first a, b pair created is the outermost one. The nonterminal S is always between the two regions of a's and b's. Clearly any string $a^nb^n \in L$ is produced by this grammar, since

$$\underbrace{S \Rightarrow aSb \Rightarrow \cdots \Rightarrow a^nSb^n}_{\text{n steps}} \Rightarrow a^nb^n,$$

Therefore $L \subseteq L(G)$.

We must also check that no other strings not in a^nb^n are produced by the grammar, i.e., we must confirm that $L(G) \subseteq L$. Usually this is easy to see intuitively, though you can prove it by induction, typically on the length of a derivation. For illustration, we'll prove $L(G) \subseteq L$ for this example, though in general you won't need to do this in this class.

Claim: $\forall$ x, $x \in L(G) \Rightarrow x \in L$. Proof by induction on the length of the derivation of G producing x.
Base case: Derivation has length 1. Then the derivation must be $S \Rightarrow \varepsilon$, and $\varepsilon \in L$.
Induction step: Assume all derivations of length k produce a string in L, and show the claim holds for derivations of length k + 1. A derivation of length k + 1 looks like:

$$S \Rightarrow \underbrace{aSb \Rightarrow \cdots \Rightarrow axb}_{\text{k steps}}$$

for some terminal string x such that $S \Rightarrow^* x$. By the induction hypothesis, we know that $x \in L$ (since x is produced by a derivation of length k), and so $x = a^nb^n$ for some n (by definition of L). Therefore, the string axb produced by the length k + 1 derivation is $axb = aa^nb^nb = a^{n+1}b^{n+1} \in L$. Therefore by induction, we have proved $L(G) \subseteq L$.

Example 2: $L = \{xy: |x| = |y| \text{ and } x \in \{a, b\}^* \text{ and } y \in \{c, d\}^*\}$. (E.g., ε, ac, ad, bc, bd, abaccc $\in L$.) Here again we will want to match a's and b's against c's and d's in parallel. We could use two strategies. In the first,

$G = (\{S, a, b, c, d\}, \{a, b, c, d\}, R, S)$ where $R = \{S \to aSc, S \to aSd, S \to bSc, S \to bSd, S \to \varepsilon\}$.

This explicitly enumerates all possible pairings of a, b symbols with c, d symbols. Clearly if the number of symbols allowed in the first and second halves of the strings is n, the number of rules with this method is $n^2 + 1$, which would be inefficient for larger alphabets. Another approach is:

$G = (\{S, L, R, a, b, c, d\}, \{a, b, c, d\}, R, S)$ where $R = \{S \to LSR, S \to \varepsilon, L \to a, L \to b, R \to c, R \to d\}$.

(Note that L and R are nonterminals here.) Now the number of rules is 2n+2.

Example 3: $L = \{ww^R : w \in \{a, b\}^*\}$. Any string in L will have matching pairs of symbols. So it is clear that the CFG G = ({S, a, b}, {a, b}, R, S}, where $R = \{S \to aSa, S \to bSb, S \to \varepsilon\}$ generates L, because it produces matching symbols in parallel. How can we prove L(G) = L? To do half of this and prove that $L \subseteq L(G)$ (i.e, every element of L is generated by G), we note that any string $x \in L$ must either be ε (which is generated by G (since $S \Rightarrow \varepsilon$)), or it must be of the form awa or bwb for some $w \in L$. This suggests an induction proof on strings:

Claim: $\forall x, x \in L \Rightarrow x \in L(G)$. Proof by induction on the length of x.
Base case: $\varepsilon \in L$ and $\varepsilon \in L(G)$.
Induction step: We must show that if the claim holds for all strings of length k, it holds for all strings of length $\geq$ k+2 (We use k+2 here rather than the more usual k+1 because, in this case, all strings in L have even length. Thus if a string in L has length k, there are no strings in L of length k +1.). If $|x| = k+2$ and $x \in L$, then x = awa or x = bwb for some $w \in L$. $|w| = k$, so, by the induction hypothesis, $w \in L(G)$. Therefore $S \Rightarrow^* w$. So either $S \Rightarrow aSa \Rightarrow^* awa$, and $x \in L(G)$, or $S \Rightarrow bSb \Rightarrow^* bwb$, and $x \in L(G)$.

Conversely, to prove that $L(G) \subseteq L$, i.e., that G doesn't generate any bad strings, we would use an induction on the length of a derivation.
Claim: $\forall x, x \in L(G) \Rightarrow x \in L$. Proof by induction on length of derivation of x.
Base case: length 1. $S \Rightarrow \varepsilon$ and $\varepsilon \in L$.
Induction step: Assume the claim is true for derivations of length k, and show the claim holds for derivations of length k+1. A derivation of length k + 1 looks like:

$$S \Rightarrow \underbrace{aSa \Rightarrow \cdots \Rightarrow awa}_{\text{k steps}}$$

or like

$$S \Rightarrow \underbrace{bSb \Rightarrow \cdots \Rightarrow bwb}_{\text{k steps}}$$

for some terminal string w such that $S \Rightarrow^* w$. By the induction hypothesis, we know that $w \in L$ (since w is produced by a derivation of length k), and so x = awa is also in L, by the definition of L. (Similarly for the second class of derivations that begin with the rule $S \rightarrow bSb$.)

As our example languages get more complex, it becomes harder and harder to write detailed proofs of the correctness of our grammars and we will typically not try to do so.

Example 4: $L = \{a^n b^{2n}\}$. You should recognize that $b^{2n} = (bb)^n$, and so this is just like the first example except that instead of matching a and b, we will match a and bb. So we want

$G = (\{S, a, b\}, \{a, b\}, R, S)$ where $R = \{S \rightarrow aSbb, S \rightarrow \varepsilon\}$.

If you wanted, you could use an auxiliary nonterminal, e.g.,

$G = (\{S, B, a, b\}, \{a, b\}, R, S)$ where $R = \{S \rightarrow aSB, S \rightarrow \varepsilon, B \rightarrow bb\}$, but that is just cluttering things up.

Example 5: $L = \{a^n b^n c^m\}$. Here, the c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together. A solution is

$G = (\{S, N, C, a, b, c\}, \{a, b, c\}, R, S)$ where $R = \{S \rightarrow NC, N \rightarrow aNb, N \rightarrow \varepsilon, C \rightarrow cC, C \rightarrow \varepsilon\}$.

This independence buys us freedom: producing the c's to the right is completely independent of making the matching $a^n b^n$, and so could be done in any manner, e.g., alternate rules like

$$C \rightarrow CC,\ C \rightarrow c,\ C \rightarrow \varepsilon$$

would also work fine. Thinking modularly and breaking the problem into more manageable subproblems is very helpful for designing CFG's.

Example 6: $L = \{a^n b^m c^n\}$. Here, the b^m is independent of the matching $a^n \ldots c^n$. But it cannot be generated "off to the side." It must be done in the middle, when we are done producing a and c pairs. Once we start producing the b's, there should be no more a, c pairs made, so a second nonterminal is needed. Thus we have

$G = (\{S, B, a, b, c\}, \{a, b, c\}, R, S)$ where $R = \{S \rightarrow \varepsilon, S \rightarrow aSc, S \rightarrow B, B \rightarrow bB, B \rightarrow \varepsilon\}$.

We need the rule $S \rightarrow \varepsilon$. We don't need it to end the recursion on S. We do that with $S \rightarrow B$. And we have $B \rightarrow \varepsilon$. But if n = 0, then we need $S \rightarrow \varepsilon$ so we don't generate any a…c pairs.

Example 7: $L = a^*b^*$. The numbers of a's and b's are independent, so there is no reason to use any rules like $S \rightarrow aSb$ which create an artificial correspondence. We can independently produce a's and b's, using

$G = (\{S, A, B, a, b\}, \{a, b\}, R, S)$ where $R = \{S \rightarrow AB, A \rightarrow aA, A \rightarrow \varepsilon, B \rightarrow bB, B \rightarrow \varepsilon\}$

But notice that this language is not just context free. It is also regular. So we expect to be able to write a regular grammar (recall the additional restrictions that apply to rules in a regular grammar) for it. Such a grammar will produce a's, and then produce b's. Thus we could write

$G = (\{S, B, a, b\}, \{a, b\}, R, S)$ where $R = \{S \rightarrow \varepsilon, S \rightarrow aS, S \rightarrow bB, B \rightarrow bB, B \rightarrow \varepsilon\}$.

Example 8: $L = \{a^m b^n : m \leq n\}$. There are several ways to approach this one. One thing we could do is to generate a's and b's in parallel, and also freely put in extra b's. This intuition yields

$G = (\{S, a, b\}, \{a, b\}, R, S)$ where $R = \{S \rightarrow aSb, S \rightarrow Sb, S \rightarrow \varepsilon\}$.

Intuitively, this CFG lets us put in any excess b's at any time in the derivation of a string in L. Notice that to keep the S between the two regions of a's and b's, we must use the rule $S \rightarrow Sb$; replacing that rule with $S \rightarrow bS$ would be incorrect, producing bad strings (allowing extra b's to be intermixed with the a's).

Another way to approach this problem is to realize that $\{a^m b^n : m \leq n\} = \{a^m b^{m+k} : k \geq 0\} = \{a^m b^k b^m : k \geq 0\}$. Therefore, we can produce a's and b's in parallel, then when we're done, produce some more b's. So a solution is

G = ({S, B, a, b}, {a, b}, R, S} where R = {S → ε, S → aSb, S → B, B → bB, B → ε}.

Intuitively, this CFG produces the matching a, b pairs, then any extra b's are generated in the middle. Note that this strategy requires two nonterminals since there are two phases in the derivation using this strategy.

Since $\{a^m b^n : m \le n\} = \{a^m b^k b^m : k \ge 0\} = \{a^m b^m b^k : k \ge 0\}$, there is a third strategy: generate the extra b's to the right of the balanced $a^m b^m$ string. Again the generation of the extra b's is now separated from the generation of the matching portion, so two distinct nonterminals will be needed. In addition, since the two parts are concatenated rather than imbedded, we'll need another nonterminal to produce that concatenation. So we've got

G = ({S, M, B, a, b}, {a, b}, R, S} where R = {S → MB, M → aMb, M → ε, B → bB, B → ε}.

Example 9: $L = \{a^{n_1} b^{n_1} \cdots a^{n_k} b^{n_k} : k \ge 0\}$. E.g., ε, abab, aabbaaabbbabab ∈ L. Note that $L = \{a^n b^n\}^*$ which gives a clue how to do this. We know how to produce matching strings $a^n b^n$, and we know how to do concatenation of strings. So a solution is

G = ({S, M, a, b}, {a, b}, R, S} where R = {S → MS, S → ε, M → aMb, M → ε}.

Any string $x = a^{n_1} b^{n_1} \cdots a^{n_k} b^{n_k} \in L$ can be generated by the canonical derivation

$$S$$

$\Rightarrow^*$ /* k applications of rule S → MS */

$$M^k S$$

$\Rightarrow$ /* one application of rule S → ε */

$$M^k$$

$\Rightarrow^*$ /* n_1 applications of rule M → aMb */

$$a^{n_1} M b^{n_1} M^{k-1}$$

$\Rightarrow$ /* one application of rule M → ε */

$$a^{n_1} b^{n_1} M^{k-1}$$

$\Rightarrow^*$ /* repeating on k-1 remaining M */

$$a^{n_1} b^{n} \cdots a^{n_k} b^{n_k}$$

Of course the rules could be applied in many different orders.

3 Derivations and Parse Trees

Let's again look at the very simple grammar G = (V, Σ, R, S), where

V = {S, A, B, a, b}, Σ = {a, b}, and R = {S → AB, A → aAa, A → a, B → Bb, B → b}

As we saw in an earlier section, G can generate the string aaabb by the following derivation:

(1) S ⇒ AB ⇒ aAaB ⇒ aAaBb ⇒ aaaBb ⇒ aaabb

Now let's consider the fact that there are other derivations of the string aaabb using our example grammar:

(2) S ⇒ AB ⇒ ABb ⇒ Abb ⇒ aAabb ⇒ aaabb

(3) S ⇒ AB ⇒ ABb ⇒ aAaBb ⇒ aAabb ⇒ aaabb

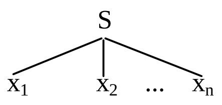
(4) S ⇒ AB ⇒ ABb ⇒ aAaBb ⇒ aaaBb ⇒ aaabb

(5) S ⇒ AB ⇒ aAaB ⇒ aaaB ⇒ aaaBb ⇒ aaabb

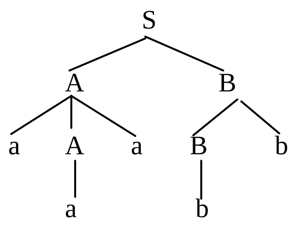
(6) S $\Rightarrow$ AB $\Rightarrow$ aAaB $\Rightarrow$ aAaBb $\Rightarrow$ aAabb $\Rightarrow$ aaabb

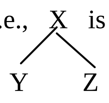
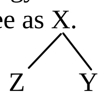
If you examine all these derivations carefully, you will see that in each case the same rules have been used to rewrite the same symbols; they differ only in the order in which those rules were applied. For example, in (2) we chose to rewrite the B in ABb as b (producing Abb) before rewriting the A as aAa, whereas in (3) the same.processes occur in the opposite order. Even though these derivations are technically different (they consist of distinct sequences of strings connected by $\Rightarrow$) it seems that in some sense they should all count as equivalent. This equivalence is expressed by the familiar representations known as ***derivation trees*** or ***parse trees***.

The basic idea is that the start symbol of the grammar becomes the root of the tree. When this symbol is rewritten by a grammar rule S $\rightarrow$ $x_1x_2...x_n$, we let the tree "grow" downward with branches to each of the new nodes x_1, x_2, …, x_n; thus:



When one of these x_i symbols is rewritten, it in turn becomes the "mother" node with branches extending to each of its "daughter" nodes in a similar fashion. Each of the derivations in (1) through (6) would then give rise to the following parse tree:



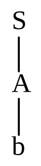
A note about tree terminology: for us, a tree always has a single root node, and the left-to-right order of nodes is significant; i.e.,  is not the same tree as .

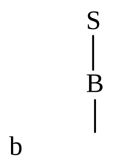
The lines connecting nodes are called ***branches***, and their top-to-bottom orientation is also significant. A ***mother node*** is connected by a single branch to each of the ***daughter nodes*** beneath it. Nodes with the same mother are called sisters, e.g., the topmost A and B in the tree above are sisters, having S as mother.

Nodes without daughters are called ***leaves***; e.g., each of the nodes labelled with a lower-case letter in the tree above. The string formed by the left-to-right sequence of leaves is called the ***yield*** (aaabb in the tree above).

It sometimes happens that a grammar allows the derivations of some string by nonequivalent derivations, i.e., derivations that do not reduce to the same parse tree. Suppose, for example, the grammar contained the rules S $\rightarrow$ A, S $\rightarrow$ B, A $\rightarrow$ b and B$\rightarrow$ b Then the two following derivations of the string b correspond to the two distinct parse trees shown below.

$S \Rightarrow A \Rightarrow b$ $S \Rightarrow B \Rightarrow b$

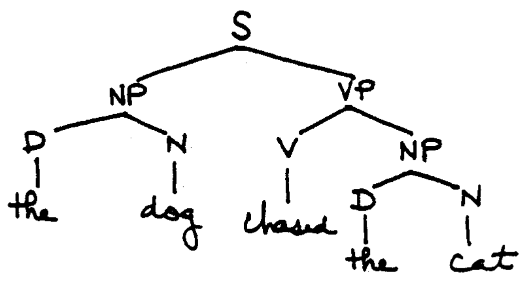




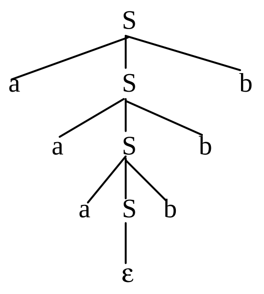
A grammar with this property is said to be ***ambiguous***. Such ambiguity is highly undesirable in grammars of programming languages such as C, LISP, and the like, since the parse tree (the syntactic structure) assigned to a string determines its translation into machine language and therefore the sequence of commands to be executed. Designers of programming languages, therefore, take great pains to assure that their grammars (the rules that specify the well-formed strings of the language) are unambiguous. Natural languages, on the other hand, are typically rife with ambiguities (cf. "They are flying planes," "Visiting relatives can be annoying," "We saw her duck," etc.), a fact that makes computer applications such as machine translation, question-answering systems, and so on, maddeningly difficult.

More Examples of Context-Free Grammars and Parse Trees:

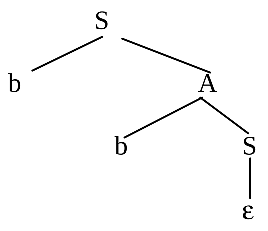
(1) $G = (V, \Sigma, R, S)$, where
$V = \{S, NP, VP, D, N, V, \text{chased}, \text{the}, \text{dog}, \text{cat}\}$,
$\Sigma = \{\text{chased}, \text{the}, \text{dog}, \text{cat}\}$
$R = \{S \rightarrow NP\ VP$
$NP \rightarrow D\ N$
$VP \rightarrow V\ NP$
$V \rightarrow \text{chased}$
$N \rightarrow \text{dog}$
$N \rightarrow \text{cat}$
$D \rightarrow \text{the}\ \}$

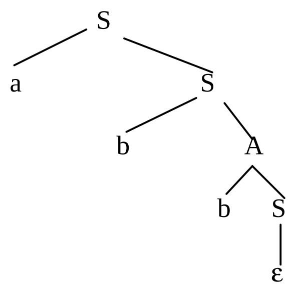


(2) $G = (V, \Sigma, R, S)$, where
$V = \{S, a, b\}$, $\Sigma = \{a, b\}$,
$R = \{S \rightarrow aSb$
$S \rightarrow \varepsilon$
$L(G) = \{a^n b^n : n \geq 0\}$



(3) $G = (V, \Sigma, R, S)$, where
$V = \{S, A, a, b\}$,
$\Sigma = \{a, b\}$,
$R = \{S \rightarrow aS$
$S \rightarrow bA$
$A \rightarrow aA$
$A \rightarrow bS$
$S \rightarrow \varepsilon$
$L(G) = \{w \in \{a, b\}^* :$ w contains an even number of b's$\}$





4 Designing Pushdown Automata

In a little bit, we will prove that for every grammar G, there is push down automaton that accepts L(G). That proof is constructive. In other words, it describes an algorithm that takes any context-free grammar and constructs the corresponding PDA. Thus, in some sense, we don't need any other techniques for building PDAs (assuming that we already know how to build grammars). But the PDAs that result from this algorithm are often highly nondeterministic. Furthermore, simply using the algorithm gives us no insight into the actual structure of the language we're dealing with. Thus, it is useful to consider how to design PDA's directly; the strategies and insights are different from those we use to design a CFG for a language, and any process that increases our understanding can't be all bad ....

In designing a PDA for a language L, one of the most useful strategies is to identify the different regions that occur in the strings in L. As a rule of thumb, each of these regions will correspond to at least one distinct state in the PDA. In this respect, PDAs are very similar to finite state machines. So, for example, just as a finite state machine that accepts a*b* needs two states (one for reading a's and one for reading b's after we're done reading a's), so will a PDA that acceps $\{a^nb^n\}$ need two states. Recognizing the distinct regions is part of the art, although it is usually obvious.

Example 1: In really simple cases, there may not be more than one region. For example, consider $\{a^na^n\}$. What we realize here is that we've really got is just the set of all even length strings of a's, i.e., (aa)*. In this case, the "border" between the first half of the a's and the second half is spurious.

Example 2: L = $\{a^nb^n\}$. A good thing to try first here is to make a finite state machine for a*b* to get the basic idea of the use of regions. (This principle applies to designing PDA's in general: use the design of a finite state machine to generate the states that are needed just to recognize the basic string structure. This creates the skeleton of your PDA. Then you can add the appropriate stack operations to do the necessary counting.) Clearly you'll need two states since you want to read a's, then read b's. To accept L, we also need to count the a's in state one in order to match them against the b's in state two. This gives the PDA M = ({s, f}, {a, b}, {I}, Δ, s, {f}), where Δ =

```
{ ((s, a, ε), (s, I)),          /* read a's and count them   */
  ((s, ε, ε), (f, ε)),          /* guess that we're done with a's and ready to start on b's   */
  ((f, b, I), (f, e))}.         /* read b's and compare them to the a's in the stack   */
```

(Notice that the stack alphabet need not be in any way similar to the input alphabet. We could equally well have pushed a's, but we don't need to.) This PDA nondeterministically decides when it is done reading a's. Thus one valid computation is

(a, aabb, ε) |- (s, abb, I) |- (f, abb, I),

which is then stuck and so M rejects along this path. Since a different accepting computation of aabb exists, this is no problem, but you might want to elimimate the nondeterminism if you are bothered by it. Note that the nondeterminism arises from the ε transition; we only want to take it if we are done reading a's. The only way to know that there are no more a's is to read the next symbol and see that it's a·b. (This is analogous to unfolding a loop in a program.) One other wrinkle: ε ∈ L, so now state s must be final in order to accept ε. The resulting deterministic PDA is M = ({s, f}, {a, b}, {I}, Δ, s, {s, f}), where Δ =

```
{ ((s, a, ε), (s, I)),          /* read a's and count them  */
  ((s, b, I), (f, ε)),          /* only go to second phase if there's a b   */
  ((f, b, I), (f, ε))}.         /* read b's and compare them to the a's in the stack   */
```

Notice that this DPDA can still get stuck and thus fail, e.g., on input b or aaba (i.e., strings that aren't in L). Determinism for PDA's simply means that there is at most one applicable transition, not necessarily exactly one.

Example 3: L = $\{a^mb^mc^nd^n\}$. Here we have two independent concerns, matching the a's and b's, and then matching the c's and d's. Again, start by designing a finite state machine for the language L' that is just like L in structure but where we don't care how many of each letter there are. In other words a*b*c*d*. It's obvious that this machine needs four states. So our PDA must also have four states. The twist is that we must be careful that there is no unexpected interaction between the two independent parts a^mb^m and c^nd^n. Consider the PDA M = ({1,2,3,4}, {a,b,c,d}, {I}, Δ, 1, {4}), where Δ =

```
{ ((1, a, ε), (1, I)),          /* read a's and count them  */
```

```
    ((1, ε, ε), (2, ε)),        /* guess that we're ready to quit reading a's and start reading b's */
    ((2, b, I), (2, ε)),        /* read b's and compare to a’s  */
    ((2, ε, ε), (3, ε)),        /* guess that we're ready to quit reading b’s and start reading c’s */
    ((3, c, ε), (3, I))         /* read c’s and count them  */
    ((3, ε, ε), (4, ε))}.       /* guess that we’re ready to quit reading c’s and start reading d’s  */
    ((4, d, I), (4, ε))}.       /* read d’s and compare them to c’s   */
```

It is clear that every string in L is accepted by this PDA. Unfortunately, some other strings are also, e.g., ad. Why is this? Because it's possible to go from state 2 to 3 without clearing off all the I marks we pushed for the a's That means that the leftover I’s are available to match d’s. So this PDA is accepting the language $\{a^mb^nc^pd^q : m \geq n \text{ and } m + p = n + q\}$, a superset of L. E.g., the string aabcdd is accepted.

One way to fix this problem is to ensure that the stack is really cleared before we leave phase 2 and go to phase 3; this must be done using a bottom of stack marker, say B. This gives M = ({s, l, 2, 3, 4}, {a, b, c, d}, {B, I}, Δ, s, {4}), where Δ =

```
  { ((s, ε, ε), (1, B)),        /* push the bottom marker onto the stack    */
    ((1, a, ε), (1, I)),        /* read a's and count them  */
    ((1, ε, ε), (2, ε)),        /* guess that we're ready to quit reading a's and start reading b's */
    ((2, b, I), (2, ε)),        /* read b's and compare to a’s  */
    ((2, ε, B), (3, ε)),        /* confirm stack is empty, then get readty to start reading c’s */
    ((3, c, ε), (3, I))         /* read c’s and count them  */
    ((3, ε, ε), (4, ε))}.       /* guess that we’re ready to quit reading c’s and start reading d’s  */
    ((4, d, I), (4, ε))}.       /* read d’s and compare them to c’s   */
```

A different, probably cleaner, fix is to simply use two different symbols for the counting of the a's and the c's. This gives us M = ({1,2, 3,4}, {a,b,c,d}, {A,C}, Δ, 1, {4}), where Δ =

```
  { ((1, a, ε), (1, A)),        /* read a's and count them  */
    ((1, ε, ε), (2, ε)),        /* guess that we're ready to quit reading a's and start reading b's */
    ((2, b, A), (2, ε)),        /* read b's and compare to a’s  */
    ((2, ε, ε), (3, ε)),        /* guess that we're ready to quit reading b's and start reading c's   */
    ((3, c, ε), (3, C)),        /* read c’s and count them  */
    ((3, ε, ε), (4, ε)),        /* guess that we’re ready to quit reading c’s and start reading d’s  */
    ((4, d, C), (4, ε))}.       /* read d’s and compare them to c’s   */
```

Now if an input has more a's than b's, there will be leftover A's on the stack and no way for them to be removed later, so that there is no way such a bad string would be accepted.

As an exercise, you want to try making a deterministic PDA for this one.

Example 4: L = $\{a^nb^n\} \cup \{b^na^n\}$). Just as with nondeterministic finite state automata, whenever the language we’re concerned with can be broken into cases, a reasonable thing to do is build separate PDAs for the each of the sublanguages. Then we build the overall machine so that it, each time it sees a string, it nondeterministically guesses which case the string falls into. (For example, compare the current problem to the simpler one of making a finite state machine for the regular language a*b* ∪ b*a*.) Taking this approach here, we get M = ({s, 1, 2, 3, 4), {a, b}, {I}, Δ, s, {2, 4}), where Δ =

```
  { ((s, ε, ε), (1, ε)),        /* guess that this is an instance of aⁿbⁿ   */
    ((s, ε, ε), (3, ε)),        /* guess that this is an instance of bⁿaⁿ   */
    ((1, a, ε), (1, I)),        /* a’s come first so read and count them   */
    ((1, ε, ε), (2, ε)),        /* begin the b region following the a’s  */
    ((2, b, I), (2, ε)),        /* read b’s and compare them to the a’s  */
    ((3, b, ε), (3, I)),        /* b’s come first so read and count them  */
    ((3, ε, ε), (4, ε)),        /* begin the a region following the b’s  */
    ((4, a, I), (4, ε))}.       /*  read a’s and compare them to the b’s  */
```

Notice that although $\varepsilon \in L$, the start state s is not a final state, but there is a path (in fact two) from s to a final state.

Now suppose that we want a deterministic machine. We can no longer use this strategy. The ε-moves must be eliminated by looking ahead. Once we do that, since $\varepsilon \in L$, the start state must be final. This gives us M = ({s, 1, 2, 3, 4}, {a, b}, {l}, Δ , s, {s, 2, 4}), where Δ =

```
{ ((s, a, ε), (1, ε)),       /* if the first character is a, then this is an instance of aⁿbⁿ  */
  ((s, b, ε), (3, ε)),       /* if the first character is b, then this is an instance of bⁿaⁿ  */
  ((1, a, ε), (1, I)),       /* a's come first so read and count them  */
  ((1, b, I), (2, ε)),       /* begin the b region following the a's  */
  ((2, b, I), (2, ε)),       /* read b's and compare them to the a's  */
  ((3, b, ε), (3, I)),       /* b's come first so read and count them  */
  ((3, a, I), (4, ε)),       /* begin the a region following the b's  */
  ((4, a, I), (4, ε))}.      /*  read a's and compare them to the b's  */
```

Example 5: L = {ww^R : w ∈ {a, b}*}. Here we have two phases, the first half and the second half of the string. Within each half, the symbols may be mixed in any particular order. So we expect that a two state PDA should do the trick. See the lecture notes for how it works.

Example 6: L = {ww^R : w ∈ a*b*}. Here the two halves of each element of L are themselves split into two phases, reading a's, and reading b's. So the straightforward approach would be to design a four-state machine to represent these four phases. This gives us M = ({1, 2, 3, 4), {a, b}, {a, b), Δ, 1, {4}), where Δ =

```
{ ((1, a, ε), (1, a))        /*push a's*/
  ((1, ε, ε), (2, ε)),       /* guess that we're ready to quit reading a's and start reading b's */
  ((2, b, ε), (2, b)),       /* push b's */
  ((2, ε, ε), (3, ε)),       /* guess that we're ready to quit reading the first w and start reading wᴿ  */
  ((3, b, b), (3, ε)),       /* compare 2nd b's to 1st b's */
  ((3, ε, ε), (4, ε)),       /* guess that we're ready to quit reading b's and move to the last region of a's */
  ((4, a, a), (4, ε))}       /* compare 2nd a's to 1st a's */
```

You might want to compare this to the straightforward nondeterministic finite state machine that you might design to accept a*b*b*a*.

There are various simplifications that could be made to this machine. First of all, notice that L = {$a^mb^nb^na^m$}. Next, observe that $b^nb^n = (bb)^n$, so that, in effect, the only requirement on the b's is that there be an even number of them. And of course a stack is not even needed to check that. So an alternate solution only needs three states, giving M = ({1, 2, 3}, {a, b}, {a}, {a}, Δ, 1, {3}), where Δ =

```
{ ((1, a, ε), (1, a))        /*push a's*/
  ((1, ε, ε), (2, ε)),       /* guess that we're ready to quit reading a's and start reading b's */
  ((2, bb, ε), (2, ε)),      /* read bb's */
  ((2, ε, ε), (3, ε)),       /* guess that we're ready to quit reading b's and move on the final group of a's  */
  ((3, a, a), (3, ε))}.      /* compare 2nd a's to 1st a's */
```

This change has the fringe benefit of making the PDA more deterministic since there is no need to guess where the middle of the b's occurs. However, it is still nondeterministic.

So let's consider another modification. This time, we go ahead and push the a's and the b's that make up w. But now we notice that we can match w^R against w in a single phase: the required ordering b*a* in w^R will automatically be enforced if we simply match the input with the stack! So now we have the PDA M= ({1, 2, 3}, {a, b}, {a, b}, Δ, 1, {3}), where Δ =

```
{ ((1, a, ε), (1, a))        /*push a's*/
  ((1, ε, ε), (2, ε)),       /* guess that we're ready to quit reading a's and start reading b's */
  ((2, b, ε), (2, b)),       /* push b's */
```

$((2, \varepsilon, \varepsilon), (3, \varepsilon))$,	/* guess that we're ready to quit reading the first w and start reading w^R */
$((3, a, a), (3, \varepsilon))$	/* compare w^R to w*/
$((3, b, b), (3, \varepsilon))\}$.	"

Notice that this machine is still nondeterministic. As an exercise, you might try to build a deterministic machine to accept this language. You'll find that it's impossible; you've got to be able to tell when the end of the strings is reached, since it's possible that there aren't any b's in between the a regions. This suggests that there might be a deterministic PDA that accepts L$, and in fact there is. Interestingly, even that is not possible for the less restrictive language $L = \{ww^R : w \in \{a, b\}^*\}$ (because there's no way to tell without guessing where w ends and w^R starts). Putting a strong restriction on string format often makes a language more tractable. Also note that $\{ww^R : w \in a^*b^+\}$ is accepted by a determinstic PDA; find such a·determinstic PDA as an exercise.

Example 7: Consider $L = \{w \in \{a, b\}^* : \#(a, w) = \#(b, w)\}$. In other words every string in L has the same number of a's as b's (although the a's and b's can occur in any order). Notice that this language imposes no particular structure on its strings, since the symbols may be mixed in any order. Thus the rule of thumb that we've been using doesn't really apply here. We don't need multiple states for multiple string regions. Instead, we'll find that, other than possible bookkeeping states, one "working" state will be enough.

Sometimes there may be a tradeoff between the degree of nondeterminism in a pda and its simplicity. We can see that in this example. One approach to designing a PDA to solve this problem is to keep a balance on the stack of the excess a's or b's. For example, if there is an a on the stack and we read b, then we cancel them. If, on the other hand, there is an a on the stack and we read another a, we push the new a on the stack. Whenever the stack is empty, we know that we've seen matching number of a's and b's so far. Let's try to design a machine that does this as deterministically as possible. One approach is M = $(\{s, q, f\}, \{a, b\}, \{a, b, c\}, \Delta, s, \{f\})$, where Δ =

1	$((s, \varepsilon, \varepsilon), (q, c))$	/* Before we do anything else, push a marker, c, on the stack so we'll be able to tell when the stack is empty. Then leave state s so we don't ever do this again.
2	$((q, a, c), (q ,ac))$	/* If the stack is empty (we find the bottom c) and we read an a, push c back and then the a (to start counting a's).
3	$((q, a, a), (q, aa))$	/* If the stack already has a's and we read an a, push the new one.
4	$((q, a, b), (q, \varepsilon))$	/* If the stack has b's and we read an a, then throw away the top b and the new a.
5	$((q, b, c), (q, bc))$	/* If the stack is empty (we find the bottom c) and we read a b, then start counting b's.
6	$((q, b, b), (q, bb))$	/* If the stack already has b's and we read b, push the new one.
7	$((q, b, a), (q, \varepsilon))$	/* If the stack has a's and we read a b, then throw away the top a and the new b.
8	$((q, \varepsilon, c), (f, \varepsilon))$	/* If the stack is empty then, without reading any input, move to f, the final state. Clearly we only want to take this transition when we're at the end of the input.

This PDA attempts to solve our problem deterministically, only pushing an a if there is not a b on the stack. In order to tell that there is *not* a b, this PDA has to pop whatever *is* on the stack and examine it. In order to make sure that there is always something to pop and look at, we start the process by pushing the special marker c onto the stack. (Recall that there is no way to check directly for an empty stack. If we write just ε for the value of the current top of stack, we'll get a match no what the stack looks like.) Notice, though, that despite our best efforts, we still have a nondeterministic PDA because, at any point in reading an input string, if the number of a's and b's read so far are equal, then the stack consists only of c, and so transition 8 $((q, \varepsilon, c), (f, \varepsilon))$ may be taken, even if there is remaining input. But if there is still input, then either transition 1 or 5 also applies. The solution to this problem is to add a terminator to L.

Another thing we could do is to consider a simpler PDA that doesn't even bother trying to be deterministic. Consider M $=(\{s\}, \{a, b\}, \{a, b\}, \Delta, s, \{s\})$, where Δ =

1	$((s, a, \varepsilon), (s, a))$	/* If we read an a, push a.
2	$((s, a, b), (s, \varepsilon))$	/* Cancel an input a and a stack b.
3	$((s, b, \varepsilon), (s, b))$	/* If we read b, push b.
4	$((s, b, a), (s, \varepsilon))$	/* Cancel and input b and a stack a.

Now, whenever we're reading a and b is on the stack, there are two applicable transitions: 1, which ignores the b and pushes the a on the stack, and 2, which pops the b and throws away the a (in other words, it cancels the a and b against each other).

Transitions 3 and 4 do the same two things if we're reading b. It is clear that if we always perform the cancelling transition when we can, we will accept every string in L. What you might worry about is whether, due to this larger degree of freedom, we might not also be able to wrongly accept some string not in L. In fact this will not happen because you can prove that M has the property that, if x is the string read in so far, and y is the current stack contents,

#(a, x) - #(b, x) = #(a, y) - #(b, y).

This formula is an invariant of M. We can prove it by induction on the length of the string read so far: It is clearly true initially, before M reads any input, since 0 - 0 - 0 - 0. And, if it holds before taking a transition, it continues to hold afterward. We can prove this as follows:

Let x' be the string read so far and let y' be the contents of the stack at some arbitrary point in the computation. Then let us see what effect each of the four possible transitions has. We first consider:

((s, a, ε), (s, a)): After taking this transition we have that x' = xa and y' = ay. Thus we have

#(a, x') - #(b, x')

= /* x' = xa */

#(a, xa) - #(b, xa)

= /* #(b, xa) = #(b, x)

#(a, xa) - #(b, x)

=

#(a, x) + 1 - #(b, x)

= /* induction hypothesis */

#(a, y) + 1 - #(b, y)

=

#(a, ay) - #(b, ay)

= /* y' = ay */

#(a, y') - #(b, y')

So the invariant continues to be true for x' and y' after the transition is taken. Intuitively, the argument is simply that when this transition is taken, it increments #(a, x) and #(a, y), preserving the invariant equation. The three other transitions also preserve the invariant as can be seen similarly:

((s, a, b), (s, ε)) increments #(a, x) and decrements #(b, y), preserving equality.

((s, b, ε), (s, b)) increments #(b, x) and #(b, y), preserving equality.

((s, b, a), (s, ε)) increments #(b, x) and decrements #(a, y), preserving equality.

Therefore, the invariant holds initially, and taking any transitions continues to preserve it, so it is always true, no matter what string is read and no matter what transitions are taken. Why is this a good thing to know? Because suppose a string x ∉ L is read by M. Since x ∉ L, we know that #(a, x) - #(b, x) ≠ 0, and therefore, by the invariant equation, when the whole string x has been read in, the stack contents y will satisfy #(a, y) - #(b, y) ≠ 0 Thus the stack cannot be empty, and x cannot be accepted, no matter what sequence of transitions is taken. Thus no bad strings are accepted by M.

5 Context-Free Languages and PDA's

Theorem: The context-free languages are exactly the languages accepted by nondeterministic PDA's.

In other words, if you can describe a language with a context-free grammar, you can build a nondeterministic PDA for it, and vice versa. Note here that the class of context-free languages is equivalent to the class of languages accepted by *nondeterministic* PDAs. This is different from what we observed when we were considering regular languages. There we showed that nondeterminism doesn't buy us any power and that we could build a *deterministic* finite state machine for every regular language. Now, as we consider context-free languages, we find that determinism does buy us power: there are languages that are accepted by nondeterministic PDAs for which no deterministic PDA exists. And those languages are context free (i.e., they can be described with context-free grammars). So this theorem differs from the similar theorem that we proved for regular languages and claims equivalence for nondeterministic PDAs rather than deterministic ones.

We'll prove this theorem by construction in two steps: first we'll show that, given a context-free grammar G, we can construct a PDA for L(G). Then we'll show (actually, we'll just sketch this second proof) that we can go the other way and construct, from a PDA that accepts some language L, a grammar for L.

Lemma: Every context-free language is accepted by some nondeterministic PDA.

To prove this lemma, we give the following construction. Given some CFG G = (V, Σ, R, S), we construct an equivalent PDA M in the following way. M = (K, Σ, Γ, Δ, s, F), where

K = {p, q }	(the PDA always has just 2 states)
s = p	(p is the initial state)
F = {q}	(q is the only final state)
Σ = Σ	(the input alphabet is the terminal alphabet of G)
Γ = V	(the stack alphabet is the total alphabet of G)
Δ contains	(1) the transition ((p, ε, ε), (q, S))
	(2) a transition ((q, ε, A), (q, α)) for each rule $A \rightarrow \alpha$ in G
	(3) a transition ((q, a, a), (q, ε)) for each $a \in \Sigma$

Notice how closely the machine M mirrors the structure of the original grammar G. M works by using its stack to simulate a derivation by G. Using the transition created in (1), M begins by pushing S onto its stack and moving to its second state, q, where it will stay for the rest of its operation. Think of the contents of the stack as M's expectation for what it must find in order to have seen a legal string in L(G). So if it finds S, it will have found such a string. But if S could be rewritten as some other sequence α, then if M found α it would also have found a string in L(G). All the transitions generated by (2) take care of these options by allowing M to replace a stack symbol A by a string α whenever G contains the rule $A \rightarrow \alpha$. Of course, at some point we actually have to look at the input. That's what M does in the transitions generated in (3). If the stack contains an expectation of some terminal symbol and if the input string actually contains that symbol, M consumes the input symbol and pops the expected symbol off the stack (effectively canceling out the expectation with the observed symbol). These steps continue, and if M succeeds in emptying its stack and reading the entire input string, then the input is accepted.

Let's consider an example. Let G = (V = {S, a, b, c}, Σ = {a, b, c}, R = {S → aSa, S → bSb, S → c}, S). This grammar generates $\{xcx^R : x \in \{a, b\}^*\}$. Carrying out the construction we just described for this example CFG gives the following PDA:

M = ({p, q}, {a, b, c}, {S, a, b, c}, Δ, p, {q}), where

Δ = { ((p, ε, ε), (q, S))
((q, ε, S), (q, aSa))
((q, ε, S), (q, bSb))
((q, ε, S), (q, c))
((q, a, a), (q, ε))
((q, b, b), (q, ε))
((q, c, c), (q, ε))}

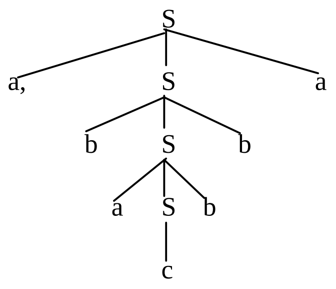
Here is a derivation of the string abacaba by G:

(1) S ⇒ aSa ⇒ abSba : ⇒ abaSaba ⇒ abacaba

And here is a computation by M accepting that same string:

(2) (p, abacaba, ε) |- (q, abacaba, S) |- (q, abacaba, aSa) |- (q, bacaba, Sa) |- (q, bacaba, bSba) |- (q, acaba, Sba) |- (q, acaba, aSaba) |- (q, caba, Saba) |- (q, caba, caba) |- (q, aba, aba) |- (q, ba, ba) |- (q, a, a) |- (q, ε, ε)

If you look at the successive stack contents in computation (2) above, you will see that they are, in effect, tracing out a derivation tree for the string abacaba:



M is alternately extending the tree and checking to see if leaves of the tree match the input string. M is thus acting as a ***top-down parser***. A parser is something that determines whether a presented string is generated by a given grammar (i.e., whether the string is ***grammatical*** or ***well-formed***), and, if it is, calculates a syntactic structure (in this case, a parse tree) assigned to that string by the grammar. Of course, the machine M that we have just described does not in fact produce a parse tree, although it could be made to do so by adding some suitable output devices. M is thus not a parser but a ***recognizer***. We'll have more to say about parsers later, but we can note here that parsers play an important role in many kinds of computer applications including compilers for programming languages (where we need to know the structure of each command), query interpreters for database systems (where we need to know the structure of each user query), and so forth.

Note that M is properly non-deterministic. From the second configuration in (2), we could have gone to (q, abacaba, bSb) or to (q, abacaba, c), for example, but if we'd done either of those things, M would have reached a dead end. M in effect has to guess which one of a group of applicable rules of G, if any, is the right one to derive the given string. Such guessing is highly undesirable in the case of most practical applications, such as compilers, because their operation can be slowed down to the point of uselessness. Therefore, programming languages and query languages (which are almost always context-free, or nearly so) are designed so that they can be parsed deterministically and therefore compiled or interpreted in the shortest possible time. A lot of attention has been given to this problem in Computer Science, as you will learn if you take a course in compilers. On the other hand, natural languages, such as English, Japanese, etc., were not "designed" for this kind of parsing efficiency. So, if we want to deal with them by computer, as for example, in machine translation or information retrieval systems, we have to abandon any hope of deterministic parsing and strive for maximum non-deterministic efficiency. A lot of effort has been devoted to these problems as well, as you will learn if you take a course in computational linguistics.

To complete the proof of our lemma, we need to prove that L(M) = L(G). The proof is by induction and is reasonably straightforward. We'll omit it here, and turn instead to the other half of the theorem:

Lemma: If M is a non-deterministic PDA, there is a context-free grammar G such that L(G) = L(M).

Again, the proof is by construction. Unfortunately, this time the construction is anything by natural. We'd never want actually to do it. We just care that the construction exists because it allows us to prove this crucial result. The basic idea behind the construction is to build a grammar that has the property that if we use it to create a leftmost derivation of some string s then we will have simulated the behavior of M while reading s. The nonterminals of the grammar are things like <s, Z, f'> (recall that we can use any names we want for our nonterminals). The reason we use such strange looking nonterminals is to make it clear what each one corresponds to. For example, <s, Z, f'> will generate all strings that M could consume in the process of moving from state s with Z on the stack to state f' having popped Z off the stack.

To construct G from M, we proceed in two steps: First we take our original machine M and construct a new "simple" machine M' (see below). We do this so that there will be fewer cases to consider when we actually do the construction of a grammar from a machine. Then we build a grammar from M'.

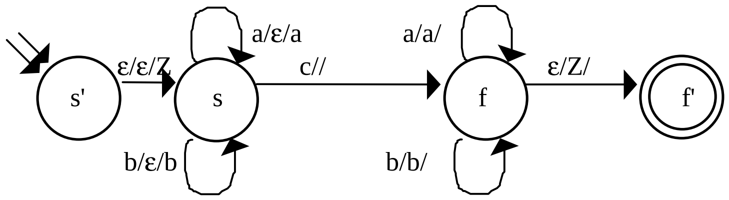
A pda M is ***simple*** iff:
(1) There are no transitions into the start state, and
(2) Whenever ((q, a, β), (p, γ)) is a transition in M and q is not the start state, then $\beta \in \Gamma$ and $|\gamma| \leq 2$.

In other words, M is simple if it always consults its topmost stack symbol (and no others) and replaces that symbol either with 0, 1, or 2 new symbols. We need to treat the start state separately since of course when M starts, its stack is empty and there is nothing to consult. But we do need to guarantee that the start state can't bypass the restriction of (2) if it also functions as something other than the start state i.e., it is part of a loop. Thus constraint (1).

Although not all machines are simple, there is an algorithm to construct an equivalent simple machine from any machine M. Thus the fact that our grammar construction algorithm will work only on simple machines in no way limits the applicability of the lemma that says that for any machine there is an *equivalent* grammar.

Given any PDA M, we construct an equivalent simple PDA M' as follows:
(1) Let M' = M.

(2) Add to M' a new start state s' and a new final state f'. Add a transition from s' to M's original start state that consumes no input and pushes a special "stack bottom" symbol Z onto the stack. Add transitions from of all of M's original final states to f'. These transitions should consume no input but they should pop the bottom of stack symbol Z from the stack. For example, if we start with a straightforward two-state PDA that accepts wcw^R, then this step produces:



(3) (a) Assure that $|\beta| \leq 1$. In other words, make sure that no transition looks at more than one symbol on the stack. It is easy to do this. If there are any transitions in M' that look at two or more symbols, break them down into multiple transitions that examine one symbol apiece.

(b) Assure that $|\gamma| \leq 1$. In other words, make sure that each transition pushes no more than one symbol onto the stack. (The rule for simple allows us to push 2, but you'll see why we restrict to 1 at this point in a minute.) Again, if M' has any transitions that push more than one symbol, break them apart into multiple steps.

(c) Assure that $|\beta| = 1$. We already know that $|\beta|$ isn't greater than 1. But it could be zero. If there are any transitions that don't examine the stack at all, then change them so that they pop off the top symbol, ignore it, and push it right back on. When we do this, we will increase by one the length of the string that gets pushed onto the stack. Now you can see why we did step (b) as we did. If, after completing (b) we never pushed more than one symbol, we can go ahead and do (c) and still be assured that we never push more than two symbols (which is what we require for M' to be simple).

We'll omit the proof that this procedure does in fact produce a new machine M' that is simple and equivalent to M.

Once we have a simple machine M' (K', Σ', Γ', Δ', s', f') derived from our original machine M (K, Σ, Γ, Δ, s, F), we are ready to construct a grammar G for L(M') (and thus, equivalently, for L(M). We let G = (V, Σ, R, S), where V contains a start symbol S, all the elements of Σ, and a new nonterminal symbol <q, A, p> for every q and p in K' and every A = ε or any symbol in the stack alphabet of M' (which is the stack alphabet of M plus the special bottom of stack marker). The tricky part is the construction of R, the rules of G. R contains all the following rules (although in fact most will be useless in the sense that the nonterminal symbol on the left hand side will never be generated in any derivation that starts with S):
(1) The special rule S → <s, Z, f'>, where s is the start state of the original machine M, Z is the special "bottom of stack" symbol that M' pushes when it moves from s' to s, and f' is the new final state of M'. This rule says that to be a string in L(M) you must be a string that M' can consume if it is started in state s with Z on the top of the stack and it makes it to state f' having popped Z off the stack. All the rest of the rules will correspond to the various paths by which M' might do that.

(2) Consider each transition ((q, a, B), (r, C)) of M' where a is either ε or a single input symbol and C is either a single symbol or ε. In other words, each transition of M' that pushes zero or one symbol onto the stack. For each such transition and each state p of M', we add the rule

<q, B, p> → a<r, C, p>.

Read these rule as saying that one way in which M' can go from q to p and pop B off the stack is by consuming an a, going to state r, pushing a C on the stack (all of which are specified by the transition we're dealing with), then getting eventually to p and popping off the stack the C that the transition specifies must be pushed. Think of these rules this way. The transition that motivates them tells us how to make a single move from q to r while consuming the input symbol a and popping the stack symbol B. So think about the strings that could drive M' from q to some arbitrary state p (via this transition) and pop B from the stack in the process. They include all the strings that start with a and are followed by the strings that can drive M' from r on to p provided that they also cause the C that got pushed to be dealt with and popped. Note that of course we must also pop anything else we push along the way, but we don't have to say that explicitly since if we haven't done that we can't get to C to pop it.

(3) Next consider each transition ((q, a, B), (r, CD)) of M', where C and D are stack symbols. In other words, consider every transition that pushes two symbols onto the stack. (Recall that since M' is simple, we only have to consider the cases of 0, 1, or 2 symbols being pushed.) Now consider all pairs of states v and w in K' (where v and w are not necessarily distinct). For all such transitions and pairs of states, construct the rule

$$<q, B, v> \rightarrow a<r, C, w><w, D, v>$$

These rules are a bit more complicated than the ones that were generated in (2) just because they describe computations that involve two intermediate states rather than one, but they work the same way.

(4) For every state q in M', we add the rule

$$<q, \varepsilon, q> \rightarrow \varepsilon$$

These rules let us get rid of spurious nonterminals so that we can actually produce strings composed solely of terminal symbols. They correspond to the fact that M' can (trivially) get from a state back to itself while popping nothing simply by doing nothing (i.e., reading the empty string).

See the lecture notes for an example of this process in action. As you'll notice, the grammars that this procedure generates are very complicated, even for very simple machines. From larger machines, one would get truly enormous grammars (most of whose rules turn out to be useless, as a matter of fact). So, if one is presented with a PDA, the best bet for finding an equivalent CFG is to figure out the language accepted by the PDA and then proceed intuitively to construct a CFG that generates that language.

We'll omit here the proof that this process does indeed produce a grammar G such that L(G) = L(M).

6 Parsing

Almost always, the reason we care about context-free languages is that we want to build programs that "interpret" or "understand" them. For example, programming languages are context free. So are most data base query languages. Command languages that need capabilities (such as matching delimiters) that can't exist in simpler, regular languages are also context free.

The interpretation process for context free languages generally involves three parts (although these logical parts may be interleaved in various ways in the interpretation program):

1. Lexical analysis, in which individual characters are combined, generally using finite state machine techniques, to form the building blocks of the language.
2. Parsing, in which a tree structure is assigned to the string.
3. Semantic interpretation, in which "meaning", often in the form of executable code, is attached to the nodes of the tree and thus to the entire string itself.

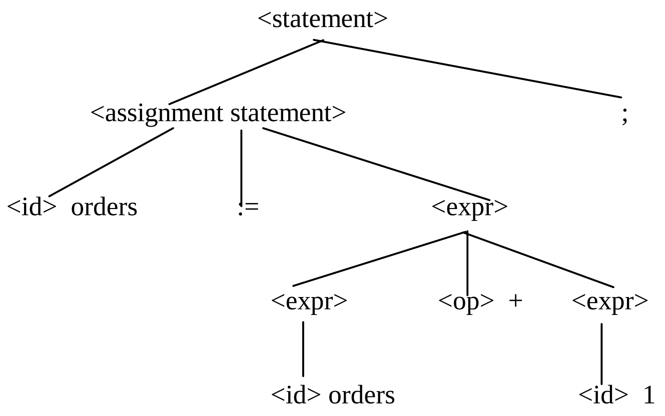
For example, consider the input string "orders := orders + 1;", which might be a legal string in any of a number of programming languages. Lexical analysis first divides this string of characters into a sequence of six ***tokens***, each of which corresponds to a basic unit of meaning in the language. The tokens generally contain two parts, an indication of what kind of thing they are and the actual value of the string that they matched. The six tokens are (with the kind of token shown, followed by its value in parentheses):

```
<id> (orders)      :=      <id> orders      <op> (+)      <id> (1)      ;
```

Assume that we have a grammar for our language that includes the following rules:

<statement> → <assignment statement> ;
<statement> → <loop statement> ;
<assignment statement> → <id> := <expr>
<expr> → <expr> <op> <expr>
<expr> → <id>

Using this grammar and the string of tokens produced above, parsing assigns to the string a tree structure like



Finally, we need to assign a meaning to this string. If we attach appropriate code to each node of this tree, then we can execute this statement by doing a postorder traversal of the tree. We start at the top node, <statement> and traverse its left branch, which takes us to <assignment statement>. We go down its left branch, and, in this case, we find the address of the variable orders. We come back up to <assignment statement>, and then go down its middle branch, which doesn't tell us anything that we didn't already know from the fact that we're in an assignment statement. But we still need to go down the right branch to compute the value that is to be stored. To do that, we start at <expr>. To get its value, we must examine its subtrees. So we traverse its left branch to get the current value for orders. We then traverse the middle branch to find out what operation to perform, and then the right branch and get 1. We hand those three things back up to <expr>, which applies the + operator and computes a new value, which we then pass back up to <assignment statement> and then to <statement>.

Lexical analysis is a straightforward process that is generally done using a finite state machine. Semantic interpretation can be arbitrarily complex, depending on the language, as well as other factors, such as the degree of optimization that is desired. Parsing, though, is in the middle. It's not completely straightforward, particularly if we are concerned with efficiency. But it doesn't need to be completely tailored to the individual application. There are some general techniques that can be applied to a wide variety of context-free languages. It is those techniques that we will discuss briefly here.

6.1 Parsing as Search

Recall that a parse tree for a string in a context-free language describes the set of grammar rules that were applied in the derivation of the string (and thus the syntactic structure of the string). So to parse a string we have to find that set of rules. How shall we do it? There are two main approaches:

1. Top down, in which we start with the start symbol of the grammar and work forward, applying grammar rules and keeping track of what we're doing, until we succeed in deriving the string we're interested in.
2. Bottom up, in which we start with the string we're interested in. In this approach, we apply grammar rules "backwards". So we look for a rule whose right hand side matches a piece of our string. We "apply" it and build a small subtree that will eventually be at the bottom of the parse tree. For example, given the assignment statement we looked at above, me might start by building the tree whose root is <expr> and whose (only) leaf is <id> orders. That gives us a new "string" to work with, which in this case would be orders := <expr> <op> <id>(1). Now we look for a grammar rule that matches part of this "string" and apply it. We continue until we apply a rule whose left hand side is the start symbol. At that point, we've got a complete tree.

Whichever of these approaches we choose, we'd like to be as efficient as possible. Unfortunately, in many cases, we're forced to conduct a search, since at any given point it may not be possible to decide which rule to apply. There are two reasons why this might happen:

- Our grammar may be ambiguous and there may actually be more than one legal parse tree for our string. We will generally try to design languages, and grammars for them, so that this doesn't happen. If a string has more than one parse tree, it is likely to have more than one meaning, and we rarely want to use languages where users can't predict the meaning of what they write.
- There may be only a single parse tree but it may not be possible to know, without trying various alternatives and seeing which ones work, what that tree should be. This is the problem we'll try to solve with the introduction of various specific parsing techniques.

6.2 Top Down Parsing

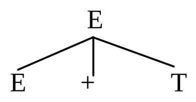
To get a better feeling for why a straightforward parsing algorithm may require search, let's consider again the following grammar for arithmetic expressions:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \text{id}$

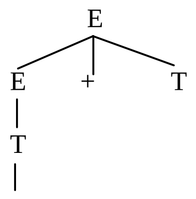
Let's try to do a top down parse, using this grammar, of the string id + id * id. We will begin with a tree whose only node is E, the start symbol of the grammar. At each step, we will attempt to expand the leftmost leaf nonterminal in the tree. Whenever we rewrite a nonterminal as a terminal (for example, when we rewrite F as id), we'll climb back up the tree and down another branch, each time expanding the leftmost leaf nonterminal). We could do it some other way. For example, we could always expand the rightmost nonterminal. But since we generally read the input string left to right, it makes sense to process the parse tree left to right also.

No sooner do we get started on our example parse but we're faced with a choice. Should we expand E by applying rule (1) or rule (2)? If we choose rule (1), what we're doing is choosing the interpretation in which + is done after * (since + will be at the top of the tree). If we choose rule (2), we're choosing the interpretation in which * is done after + (since * will be nearest the top of the tree, which we'll detect at the next step when we have to find a way to rewrite T). We know (because we've done this before and because we know that we carefully crafted this grammar to force * to have higher precedence than +) that if we choose rule (2), we'll hit a dead end and have to back up, since there will be no way to deal with + inside T.

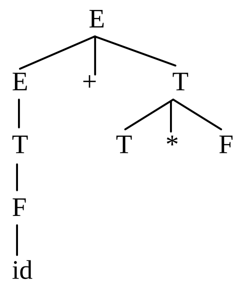
Let's just assume for the moment that our parser also knows the right thing to do. It then produces



Since E is again the leftmost leaf nonterminal, we must again choose how to expand it. This time, the right thing to do is to choose rule (2), which will rewrite E as T. After that, the next thing to do is to decide how to rewrite T. The right thing to do is to choose rule (4) and rewrite T as F. Then the next thing to do is to apply rule (6) and rewrite F as id. At this point, we've generated a terminal symbol. So we read an input symbol and compare it to the one we've generated. In this case, it matches, so we can continue. If it didn't match, we'd know we'd hit a dead end and we'd have to back up and try another way of expanding one of the nodes higher up in the tree. But since we found a match, we can continue. At this point, the tree looks like



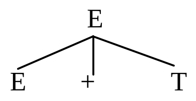
Since we matched a terminal symbol (id), the next thing to do is to back up until we find a branch that we haven't yet explored. We back all the way up to the top E, then down its center branch to +. Since this is a terminal symbol, we read the next input symbol and check for a match. We've got one, so we continue by backing up again to E and taking the third branch, down to T. Now we face another choice. Should we apply rule (3) or rule (4). Again, being smart, we'll choose to apply rule (3), producing



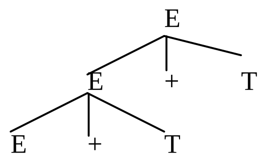
The rest of the parse is now easy. We'll expand T to F and then match the second id. Then we'll match F to the last id.

But how can we make our parser know what we knew?

In this case, one simple heuristic we might try is to consider the rules in the order in which they appear in the grammar. That will work for this example. But suppose the input had been id * id * id. Now we need to choose rule (2) initially. And we're now in big trouble if we always try rule (1) first. Why? Because we'll never realize we're on the wrong path and back up and try rule (2). If we choose rule (1), then we will produce the partial parse tree



But now we again have an E to deal with. If we choose rule (1) again, we have



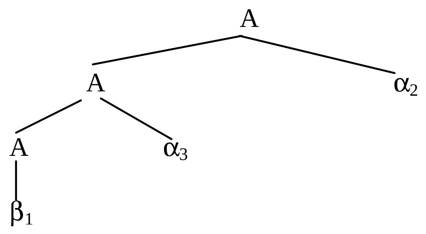
And then we have another E, and so forth. The problem is that rule (1) contains ***left recursion***. In other words, a symbol, in this case E, is rewritten as a sequence whose first symbol is identical to the symbol that is being rewritten.

We can solve this problem by rewriting our grammar to get rid of left recursion. There's an algorithm to do this that always works. We do the following for each nonterminal A that has any left recursive rules. We look at all the rules that have A on their left hand side, and we divide them into two groups, the left recursive ones and the other ones. Then we replace each rule with another related rule as shown in the following table:

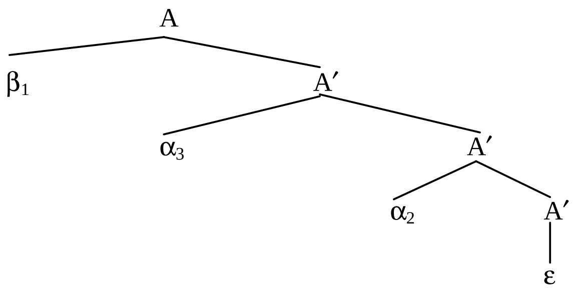
	Original rules	**New rules**
Left recursive rules:	$A \to A\alpha_1$	$A' \to \alpha_1 A'$
	$A \to A\alpha_2$	$A' \to \alpha_2 A'$
	$A \to A\alpha_3$ …	$A' \to \alpha_3 A'$ …
	$A \to A\alpha_n$	$A' \to \alpha_n A'$
		$A' \to \varepsilon$

Non-left recursive rules:	$A \rightarrow \beta_1$		$A \rightarrow \beta_1 A'$	
	$A \rightarrow \beta_2$	…	$A \rightarrow \beta_2 A'$	…
	$A \rightarrow \beta_n$		$A \rightarrow \beta_n A'$	

The basic idea is that, using the original grammar, in any successful parse, A may be expanded some arbitrary number of times using the left recursive rules, but if we're going to get rid of A (which we must do to derive a terminal string), then we must eventually apply one of the nonrecursive rules. So, using the original grammar, we might have something like



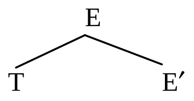
Notice that, whatever β_1, α_3, and α_2 are, β_1, which came from one of the nonrecursive rules, comes first. Now look at the new set of rules in the right hand column above. They say that A must be rewritten as a string that starts with the right hand side of one of the nonrecursive rules (i.e., some β_i). But, if any of the recursive rules had been applied first, then there would be further substrings, after the β_i, derived from those recursive rules. We introduce the new nonterminal A′ to describe what those things could look like, and we write rules, based on the original recursive rules, that tell us how to rewrite A′. Using this new grammar, we'd get a parse tree for $\beta_1\ \alpha_3\ \alpha_2$ that would look like



If we apply this transformation algorithm to our grammar for arithmetic expressions, we get

(1)	$E' \rightarrow + T E'$
(1′)	$E' \rightarrow \varepsilon$
(2)	$E \rightarrow T E'$
(3)	$T' \rightarrow * F T'$
(3′)	$T' \rightarrow \varepsilon$
(4)	$T \rightarrow F T'$
(5)	$F \rightarrow (E)$
(6)	$F \rightarrow \text{id}$

Now let's return to the problem of parsing id + id * id. This time, there is only a single way to expand the start symbol, E, so we produce, using rule (2),



Now we need to expand T, and again, there is only a single choice. If you continue with this example, you'll see that if you have the ability to peek one character ahead in the input (we'll call this character the ***lookahead character***), then it's possible to know, at each step in the parsing process, which rule to apply.

You'll notice that this parse tree assigns a quite different structure to the original string. This could be a serious problem when we get ready to assign meaning to the string. In particular, if we get rid of left recursion in our grammar for arithmetic expressions, we'll get parse trees that associate right instead of left. For example, we'll interpret a + b + c as

(a + b) + c using the original grammar, but

a + (b + c) using the new grammar.

For this and various other reasons, it often makes sense to change our approach and parse bottom up, rather than top down.

6.3 Bottom Up Parsing

Now let's go back to our original grammar for arithmetic expressions:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

Let's try again to parse the string id + id * id, this time working bottom up. We'll scan the input string from left to right, just as we've always done with all the automata we've built. A bottom up parser can perform two basic operations:

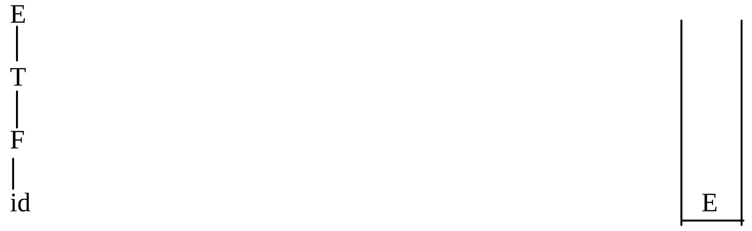
1. Shift an input symbol onto the parser's stack.
2. Reduce a string of symbols from the top of the stack to a nonterminal symbol, using one of the rules of the grammar. Each time we do this, we also build the corresponding piece of the parse tree.

When we start, the stack is empty, so our only choice is to get the first input symbol and shift it onto the stack. The first input symbol is id, so it goes onto the stack. Next, we have a choice. We can either use rule (6) to reduce id to F, or we can get the next input symbol and push it onto the stack. It's clear that we need to apply rule (6) now. Why? There are no other rules that can consume an id directly. So we have to do this reduction before we can do anything else with id. But could we wait and do it later? No, because reduction always applies to the symbols at the top of the stack. If we push anything on before we reduce id, we'll never again get id at the top of the stack. So it will just sit there, unable to participate in any rules. So the next thing we need to do is to reduce id to F, giving us a stack containing just F, and the parse tree (remember we're building up from the bottom):

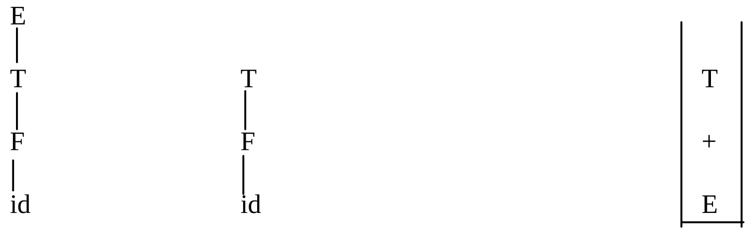
F
|
id

Before we continue, let's observe that the reasoning we just did is going to be the basis for the design of a "smart" deterministic bottom up parser. Without that reasoning, a dumb, brute force parser would have to consider both paths at this first choice point: the one we took, as well as the one that fails to reduce and instead pushes + onto the stack. That second path will dead end eventually, so even a brute force parser will eventually get the right answer. But for efficiency, we'd like to build a deterministic parser if we can. We'll return to the question of how we do that after we finish with this example so we can see all the places we're going to have to make our parser "smart".

At this point, the parser's stack contains F and the remaining input is + id * id. Again we must choose between reducing the top of the stack or pushing on the next input symbol. Again, by looking ahead and analyzing the grammar, we can see that eventually we will need to apply rule (1). To do so, the first id will have to have been promoted to a T and then to an E. So let's next reduce by rule (4) and then again by rule (2), giving the parse tree and stack:



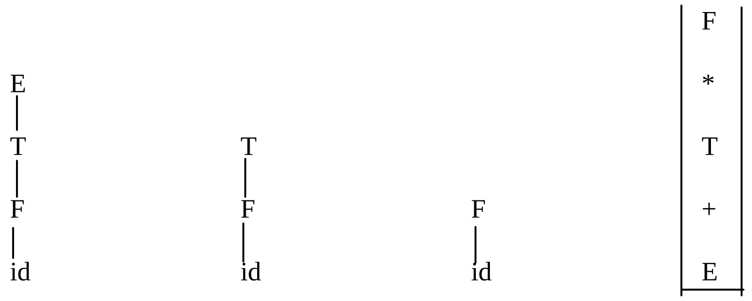
At this point, there are no further reductions to consider, since there are no rules whose right hand side is just E. So we must consume the next input symbol + and push it onto the stack. Now, again, there are no available reductions. So we read the next symbol, and the stack then contains id + E (we'll write the stack so that we push onto the left). Again, we need to promote id before we can do anything else, so we promote it to F and then to T. Now we've got:



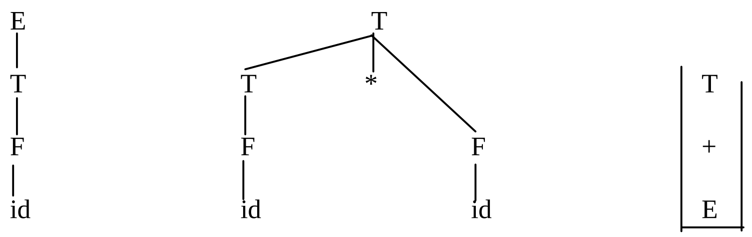
Notice that we've now got two parse tree fragments. Since we're working up from the bottom, we don't know yet how they'll get put together. The next thing we have to do is to choose between reducing the top three symbols on the stack (T + E) to E using rule (1) or shifting on the next input symbol. By the way, don't be confused about the order of the symbols here. We'll always be matching the right hand sides of the rules reversed because the last symbol we read (and thus the right most one we'll match) is at the top of the stack.

Okay, so what should we choose to do, reduce or shift? This is the first choice we've had to make where there isn't one correct answer for all input strings. When there was just one universally correct answer, we could compute it simply by examining the grammar. Now we can't do that. In the example we're working with, we don't want to do the reduction, since the next operator is *. We want the parse tree to correspond to the interpretation in which * is applied before +. That means that + must be at the top of the tree. If we reduce now, it will be at the bottom. So we need to shift * on and do a reduction that will build the multiplication piece of the parse tree before we do a reduction involving +. But if the input string had been id + id + id, we'd want to reduce now in order to cause the first + to be done first, thus producing left associativity. So we appear to have reached a point where we'll have to branch. Since our grammar won't let us create the interpretation in which we do the + first, if we choose that path first, we'll eventually hit a dead end and have to back up. We'd like not to waste time exploring dead end paths, however. We'll come back to how we can make a parser smart enough to do that later. For now, let's just forge ahead and do the right thing.

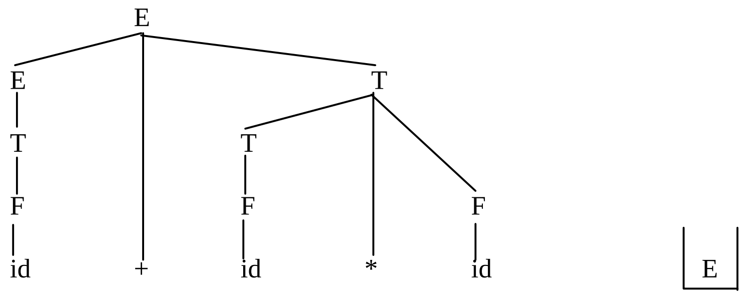
As we said, what we want to do here is not to reduce but instead to shift * onto the stack. So the stack now contains * T + E. At this point, there are no available reductions (since there are no rules whose right hand side contains * as the last symbol), so we shift the next symbol, resulting in the stack id * T + E. Clearly we have to promote id to F (following the same argument that we used above), so we've got



Next, we need to reduce (since there aren't any more input symbols to shift), but now we have another decision to make: should we reduce the top F to T, or should we reduce the top three symbols, using rule (3) to T? The right answer is to use rule (3), producing:



Finally, we need to apply rule (1), to produce the single symbol E on the top of the stack, and the parse tree:



In a bottom up parse, we're done when we consume all the input and produce a stack that contains a single symbol, the start symbol. So we're done (although see the class notes for an extension of this technique in which we add to the input and end-of-input symbol $ and consume it as well).

Now let's return to the question of how we can build a parser that makes the right choices at each step of the parsing process. As we did the example parse above, there were two kinds of decisions that we had to make:

- Whether to shift or reduce (we'll call these shift/reduce conflicts), and
- Which of several available reductions to perform (we'll call these reduce/reduce conflicts).

Let's focus first on shift/reduce conflicts. At least in this example, it was always possible to make the right decision on these conflicts if we had two kinds of information:

- A good understanding of what is going on in the grammar. For example, we noted that there's nothing to be done with a raw id that hasn't been promoted to an F.
- A peek at the next input symbol (the one that we're considering shifting), which we call the lookahead symbol. For example, when we were trying to decide whether to reduce T + E or shift on the next symbol, we looked ahead and saw that the next symbol was *. Since we know that * has higher precedence than +, we knew not to reduce +, but rather to wait and deal with * first.

So we as people can be smart and do the right thing. The important question is, "Can we build a parser that is smart and does the right thing?" The answer is yes. For simple grammars, like the one we're using, it's fairly straightforward to do so. For more complex grammars, the algorithms that are needed to produce a correct deterministic parser are way beyond the scope of this class. In fact, they're not something most people ever want to deal with. And that's okay because there are powerful tools for building parsers. The input to the tools is a grammar. The tool then applies a variety of algorithms to produce a parser that does the right thing. One of the most widely used such tools is yacc, which we'll discuss further in class. See the yacc documentation for some more information about how it works.

Although we don't have time to look at all the techniques that systems like yacc use to build deterministic bottom up parsers, we will look at one of the structures that they can build. A ***precedence table*** tells us whether to shift or reduce. It uses just two sources of information, the current top of stack symbol and the lookahead symbol. We won't describe how this table is constructed, but let's look at an example of one and see how it works. For our expression grammar, we can build the following precedence table (where \$ is a special symbol concatenated to the end of each input string that signals the end of the input):

V\Σ	(	)	id	+	*	\$
(						
)		●		●	●	●
id		●		●	●	●
+						
*						
E						
T		●		●		●
F		●		●	●	●

Here's how to read the table. Compare the left most column to the top of the stack and find the row that matches. Now compare the symbols along the top of the chart to the lookahead symbol and find the column that matches. If there's a dot in the correponding square of the table, then reduce. Otherwise, shift. So let's go back to our example input string id + id * id. Remember that we had a shift/reduce conflict when the stack's contents were T + E and the next input symbol was *. So we look at the next to the last row of the table, the one that has T as the top of stack symbol. Then we look at the column headed *. There's no dot, so we don't reduce. But notice that if the lookahead symbol had been +, we'd have found a dot, telling us to reduce, which is exactly what we'd want to do. Thus this table captures the precedence relationships between the operators * and +, plus the fact that we want to associate left when faced with operators of equal precedence.

Deterministic, bottom up parsers of the sort that yacc builds are driven by an even more powerful table called a parse table. Think of the parse table as an extension of the precedence table that contains additional information that has been extracted from an examination of the grammar.

7 Closure Properties of Context-Free Languages

Union: The CFL's are closed under union. Proof: If L_1 and L_2 are CFL's, then there are, by definition, CFG's $G_l = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ generating L_1 and L_2, respectively. (Assume that the non-terminal vocabularies of the two grammars are disjoint. We can always rename symbols to achieve this, so there are no accidental interactions between the two rule sets.) Now form CFG $G = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$. G generates $L_1 \cup L_2$ since every derivation from the start symbol of G must begin either $S \Rightarrow S_1$ or $S \Rightarrow S_2$ and thereafter to derive a string generated by G_1 or by G_2, respectively. Thus all strings in $L(G_1) \cup L(G_2)$ are generated, and no others.

Concatenation: The CFL's are closed under concatenation. The proof is similar. Given G_l and G_2 as above, form $G = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S\rightarrow S_1S_2\}, S)$. G generates L_1L_2 since every derivation from S must begin $S \Rightarrow S_1S_2$ and proceed thereafter to derive a string of L_1 concatenated to a string of L_2. No other strings are produced by G.

Kleene star: The CFL's are closed under Kleene star. Proof: If L is a CFL, it is generated by some CFG $G = (V, \Sigma, R, S)$. Using one new nonterminal symbol S', we can form a new CFG $G' = (V \cup S', \Sigma, R \cup (S' \rightarrow \varepsilon, S' \rightarrow S'S \}, S')$. G' generates L* since there is a derivation of ε ($S' \Rightarrow \varepsilon$), and there are other derivations of the form $S' \Rightarrow S'S \Rightarrow S'SS \Rightarrow ... \Rightarrow S'S...SS \Rightarrow S...SS$, which produce finite concatenations of strings of L. G generates no other strings.

Intersection: The CFL's are ***not*** closed under intersection. To prove this, it suffices to show one example of two CFL's whose intersection is not context free. Let $L_1 = \{a^ib^ic^j: i, j \geq 0\}$. L_1 is context-free since it is generated by a CFG with the rules $S \rightarrow AC$, $A \rightarrow aAb$, $A \rightarrow \varepsilon$, $C \rightarrow cC$, $C\rightarrow \varepsilon$. Similarly, let $L_2 = \{a^kb^mc^m : k, m \geq 0\}$. L_2 is context-free, since it is generated by a CFG similar to the one for L_1. Now consider $L_3 = L_1 \cap L_2 = \{a^nb^nc^n : n \geq 0\}$. L_3 is not context free. We'll prove that in the next section using the context-free pumping lemma. Intuitively, L_3 isn't context free because we can't count a's, b's, and c's all with a single stack.

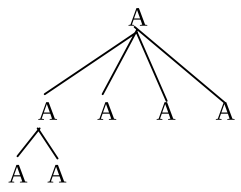
Intersection with regular languages: The CFL's are, however, closed under intersection with the regular languages. Given a CFL, L and a regular language R, the intersection $L \cap R$ is a CFL. Proof: Since L is context-free, there is some non-deterministic PDA accepting it, and since R is regular, there is some deterministic finite state automaton that accepts it. The two automata can now be merged into a single PDA by a straightforward technique described in class.

Complementation: The CFL's are ***not*** closed under complement. Proof: Since the CFL's are closed under union, if they were also closed under complement, this would imply that they are closed under intersection. This is so because of the set-theoretic equality $(\overline{\overline{L1} \cup \overline{L2}}) = (L1 \cap L2)$.

8 The Context-Free Pumping Lemma

There is a pumping lemma for context-free languages, just as there is one for regular languages. It's a bit more complicated, but we can use it in much the same way to show that some language L is not in the class of context-free languages. In order to see why the pumping lemma for context-free languages is true, we need to make some observations about parse trees:

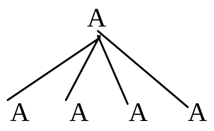
1. A ***path*** in a parse tree is a continuously descending sequence of connected nodes.
2. The ***length*** of a path in a parse tree is the number of connections (branches) in it.

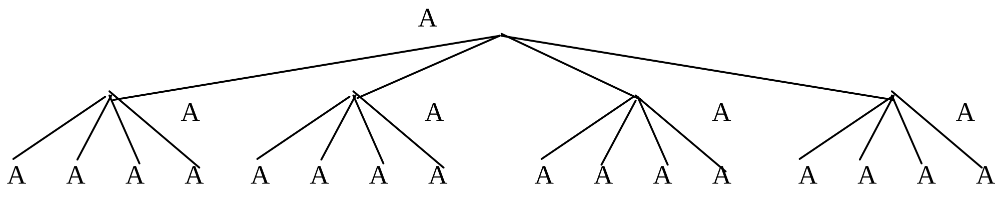


3. The ***height*** of a parse tree is the length of the longest path in it. For example, the parse tree above is of height 2.
4. The ***width*** of a parse tree is the length of its yield (the string consisting of its leaves). For example, the parse tree above is of width 5.

We observe that in order for a parse tree to achieve a certain width it must attain a certain minimum height. How are height and width connected? The relationship depends on the rules of the grammar generating the parse tree.

Suppose, for example, that a certain CFG contains the rule $A \rightarrow AAAA$. Focusing just on derivations involving this rule, we see that a tree of height 1 would have a width of 4. A tree of height 2 would have a *maximum* width of 16 (although there are narrower trees of height 2 of course).





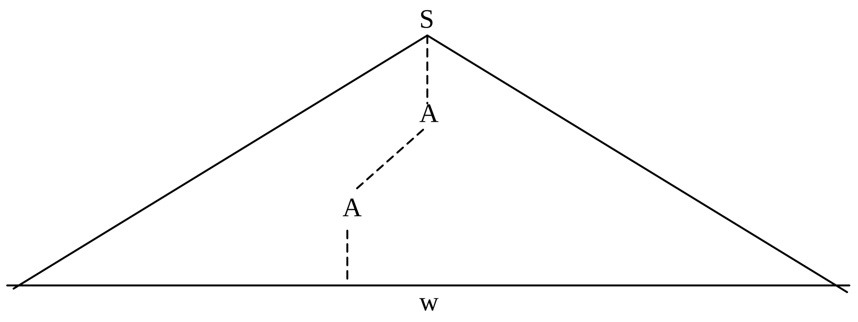
With height 3, the maximum width is 64 (i.e., 4^3), and in general a tree of height n has maximum width of 4^n. Or putting it another way, if a tree is wider than 4^n then it must be of height greater than n.

Where does the 4 come from? Obviously from the length of the right-hand side of the rule $A \rightarrow AAAA$. If we had started with the rule $A \rightarrow AAAAAA$, we would find that a tree of height n has maximum width 6^n.

What about other rules in the grammar? If it contained both the rules $A \rightarrow AAAA$ and $A \rightarrow AAAAAA$, for example, then the maximum width would be determined by the longer right-hand side. And if there were no other rules whose right-hand sides were longer than 6, then we could confidently say that any parse tree of height n could be no wider than 6^n.

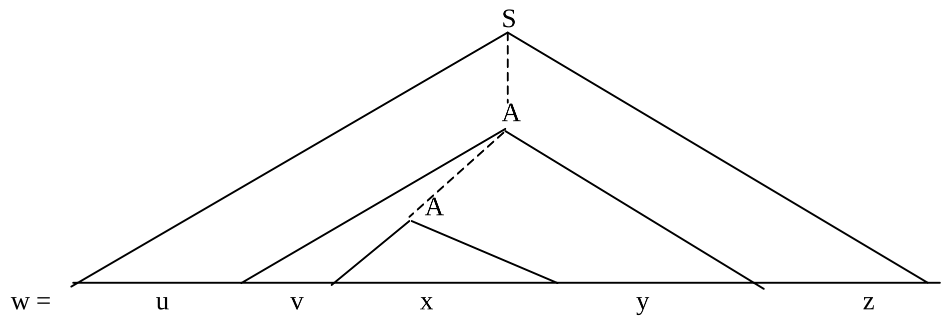
Let p = the maximum length of the right-hand side of all the rules in G. Then any parse tree generated by G of height m can be no wider than p^m. Equivalently, any parse tree that is generated by G and that is wider than p^m must have height greater than m.

Now suppose we have a CFG G = (V, Σ, R, S). Let n = |(V - Σ)|, the size of the non-terminal alphabet. If G generates a parse tree of width greater than p^n, then, by the above reasoning, the tree must be of height greater than n, i.e., it contains a path of length greater than n. Thus there are more than n + 1 nodes on this path (the number of nodes being one greater than the length of the path), and all of them are non-terminal symbols except possibly the last. Since there are only n distinct non-terminal symbols in G, some such symbol must occur more than once along this path (by the Pigeonhole Principle). What this says is that if a CFG generates a long enough string, its parse tree is going to be sufficiently high that it is guaranteed to have a path with some repeated non-terminal symbol along it. Let us represent this situation by the following diagram:

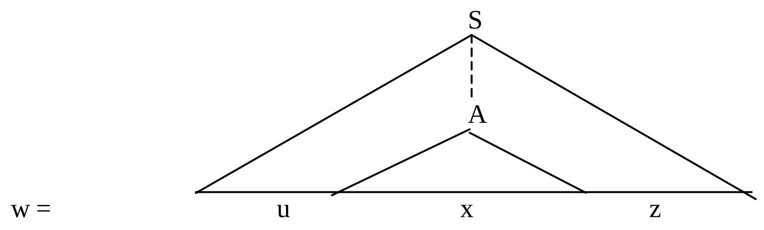


Call the generated string w. The parse tree has S as its root, and let A be a non-terminal symbol (it could be S itself, of course) that occurs at least twice on some path (indicated by the dotted lines).

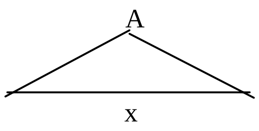
Another observation about parse trees: If the leaves are all terminal symbols, then every non-terminal symbol in the tree is the root of a subtree having terminal symbols as its leaves. Thus, the lower instance of A in the tree above must be the root of a tree with some substring of w as its leaves. Call this substring x. The upper instance of A likewise roots a tree with a string of terminal symbols as its leaves, and furthermore, from the geometry of the tree, we see that this string must include x as a substring. Call the larger string, therefore, vxy. This string vxy is also a substring of the generated string w, which is to say that for some strings u and z**,** w **=** uvxyz. Attaching these names to the appropriate substrings we have the following diagram:



Now, assuming that such a tree is generated by G (which will be true on the assumption that G generates some sufficiently long string), we can conclude that G must generate some other parse trees as well and therefore their associated terminal strings. For example, the following tree must also be generated:

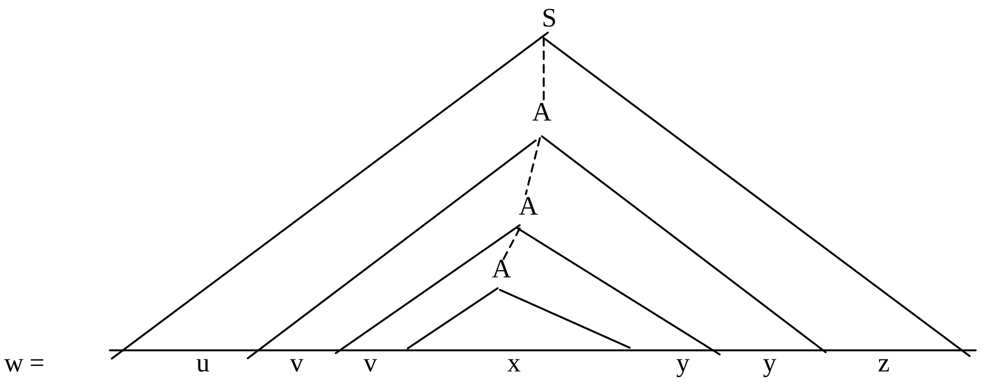


since whatever sequence of rules produced the lower subtree



could have been applied when the upper A was being rewritten.

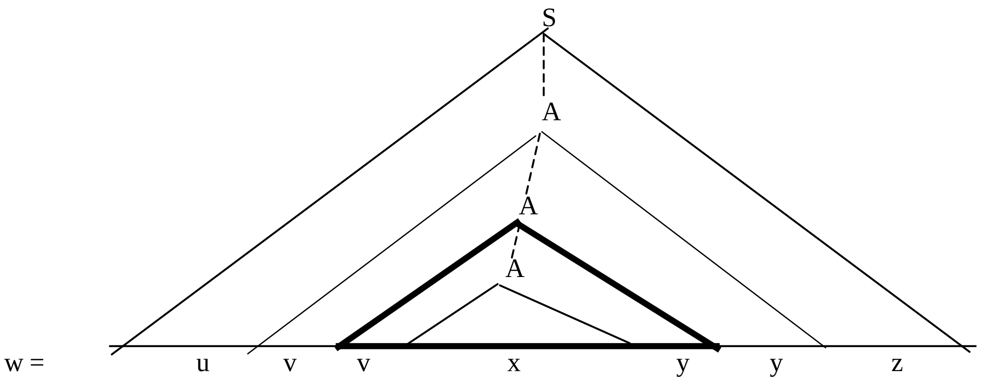
Similarly, the sequence of rules that expanded the upper A originally to yield the string vAy could have been applied to the lower A as well, and if the resulting third A were now rewritten to produce x, we would have:



Clearly this process could be repeated any number of times to give an infinite number of strings of the form

$u \quad v_1\ v_2\ v_3 \ldots v_n \quad x \quad y_1\ y_2\ y_3 \ldots y_n \quad z$, for all values of $n \geq 0$.

We need one further observation before we are ready to state the Pumping Lemma. Consider again any string w that is sufficiently long that its derivation contains at least one repeating nonterminal (A in our example above). Of course, there may be any number of occurrences of A, but let's consider the bottom two. Consider the subtree whose root is the second A up from the bottom (shown in bold):



Notice that the leaves of this subtree correspond to the sequence vxy. How long can this sequence be? The answer relies on the fact that this subtree contains exactly one repeated nonterminal (since we chose it that way). So the maximum height of this subtree is p^{n+1}. (Recall that p is the length of the longest rule in the grammar and n is the number of nonterminals in the grammar.) Why n+1? Because we have n+1 nonterminals available (all n of them plus the one repeated one). So we know that $|vxy|$ must be $\leq M$, where M is some constant that depends on the grammar and that is in fact p^{n+1}. We are now ready to state the Pumping Lemma for context-free languages.

Pumping Lemma for Context-Free Languages: Let G be a context-free grammar. Then there are some constants K and M

depending on G such that, for every string $w \in L(G)$ where $|w| > K$, there are strings u, v, x, y, z such that
(1) $w = uvxyz$,
(2) $|vy| > 0$,
(3) $|vxy| \leq M$, and
(4) for all $n \geq 0$, $uv^nxy^nz \in L(G)$.

Remarks: The constant K in the lemma is just p^n referred to above - the length of the longest right-hand side of a rule of G raised to the power of the number of non-terminal symbols. In applying the lemma we won't care what the value of K actually is, only that some such constant exists. If G generates an infinite language, then clearly there will be strings in L(G) longer than K, no matter what K is. If L(G) is finite, on the other hand, then the lemma still holds (trivially), because K will have a value greater than the longest strings in L(G). So all strings in L(G) longer than K are guaranteed to be "pumpable," but no such strings exist, so the lemma is trivially true because the antecedent of the conditional is false. Similarly for M, which is actually bigger than K; it is p^{n+1}. But, again, all we care about is that if L(G) is infinite then M exists. Without knowing what it is, we can describe strings in terms of it and know that we must have pumpable strings.

This lemma, like the pumping lemma for regular languages, addresses the question of how strings grow longer and longer without limit so as to produce infinite languages. In the case of regular languages, we saw that strings grow by repeating some substring any number of times: $xy^nz \in L$ for all $n \geq 0$. When does this happen? Any string in the language of sufficient length is guaranteed to contain such a "pumpable" substring. What length is sufficient? The number of states in the minimum-state deterministic finite state machine that accepts the language. This sets the lower bound for guaranteed pumpability.

For context-free languages, strings grow by repeating two substrings simultaneously: $uv^nxy^nz \in L$ for all $n \geq 0$. This, too, happens when a string in the language is of sufficient length. What is sufficient? Long enough to guarantee that its parse tree contains a repeated non-terminal along some path. Strings this long exceed the lower bound for guaranteed context-free pumpability.

What about the condition that $|vy| > 0$, i.e., v and y cannot both be the empty string? This could happen if by the rules of G we could get from some non-terminal A back to A again without producing any terminal symbols in the process, and that's possible with rules like $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, all perfectly good context-free rules. But given that we have a string w whose length is greater than or equal to K, its derivation must have included *some* rules that make the string grow longer; otherwise w couldn't have gotten as long as it did. Therefore, there must be some path in the derivation tree with a repeated non-terminal that involves branching rules, and along *this* path, at least one of v or y is non-empty.

Recall that the corresponding condition for regular languages was $y \neq \varepsilon$. We justified this by pointing out that if a sufficiently long string w was accepted by the finite state machine, then there had to be a loop in the machine and that loop must read something besides the empty string; otherwise w couldn't be as long as it is and still be accepted.

And, finally, what about condition (3), $|vxy| \leq M$? How does this compare to the finite state pumping lemma? The corresponding condition there was that $|xy| \leq K$. Since $|y| \leq |xy|$, this certainly tells us that the pumpable substring y is (relatively) short. $|xy| \leq K$ also tells us that y occurs close to the beginning of the string $w = xyz$. The context-free version, on the other hand, tells us that $|vxy| \leq M$, where v and y are the pumpable substrings. Since $|v| \leq |vxy| \leq M$ and $|y| \leq |vxy| \leq M$, we know that the pumpable substrings v and y are short. Furthermore, from $|vxy| \leq M$, we know that v and y must occur close to each other (or at least not arbitrarily far away from each other). Unlike in the regular pumping lemma, though, they do not necessarily occur close to the beginning of the string $w = uvxyz$. This is the reason that context-free pumping lemma proofs tend to have more cases: the v and y pumpable substrings can occur anywhere within the string w.

Note that this Pumping Lemma, like the one for regular languages, is an if-then statement not an iff statement. Therefore, it cannot be used to show that a language *is* context-free, only that it is *not*.

Example 1: Show that $L = \{a^nb^nc^n : n \geq 0\}$ is not context-free.

If L were context-free (i.e., if there were a context-free grammar generating L), then the Pumping Lemma would apply. Then there would be a constant K such that every string in L of length greater than K would be "pumpable." We show that this is not so by exhibiting a string w in L that is of length greater than K and that is not pumpable. Since we want to rely on clause 3 of the pumping lemma, and it relies on M > K, we'll actually choose w in terms of M.

Let w = $a^Mb^Mc^M$. (Note that this is a *particular string*, not a language or a variable expression for a string. M is some number whose exact value we don't happen to know; it might be 23, for example. If so, w would be the unique string $a^{23}b^{23}c^{23}$.) This string is of length greater than K (of length 3M, where M is greater than K, in fact), and it is a string in the language $\{a^nb^nc^n : n \geq 0\}$. Therefore, it satisfies the criteria for a pumpable string according to the Pumping Lemma--provided, of course, that L is context-free.

What does it mean to say that $a^Mb^Mc^M$ is pumpable? It means that there is *some* way to factor this string into five parts - u,v,x,y,z - meeting the following conditions:

1. v and y are not both the empty string (although any of u, x, or z could be empty),
2. $|vxy| \leq M$,
3. $uxz \in L$, $uvxyz \in L$, $uvvxyyz \in L$, $uvvvxyyyz \in L$, etc.; i.e., for all $n \geq 0$, $uv^nxy^nz \in L$.

We now show that there is *no way* to factor $a^Mb^Mc^M$ into 5 parts meeting these conditions; thus, $a^Mb^Mc^M$ is *not* a pumpable string, contrary to the stipulation of the Pumping Lemma, and thus L is *not* a context-free language.

How do we do this? We show that no matter how we try to divide $a^Mb^Mc^M$ in ways that meet the first two conditions, the third condition always falls. In other words, every "legal" division of $a^Mb^Mc^M$ falls to be pumpable-that is, there is some value of n for which $uv^nxy^nz \notin L$.

There are clearly a lot of ways to divide this string into 5 parts, but we can simplify the task by grouping the divisions into cases just as we did with the regular language Pumping Lemma:
Case 1: Either v or y consists of more than different letter (e.g., aab). No such division is pumpable, since for any $n \geq 2$, uv^nxy^nz will contain some letters not in the correct order to be in L. Now that we've eliminated this possibility, all the remaining cases can assume that both v and y contain only a's, only b's, or only c's (although one could also be ε).
Case 2: Both v and y are located within a^M. No such division is pumpable, since we will pump in only a's. So, for $n \geq 2$, uv^nxy^nz will contain more a's than b's or c's and therefore won't be in L. (Note that n = 0 also works.)
Cases 3, 4: Both v and y are located within b^M or c^M. No such division is pumpable, by the same logic as in Case 2.
Case 5: v is located within a^M and y is located within b^M. No such division is pumpable, since for $n \geq 2$, uv^nxy^nz will contain more a's than c's or more b's than c's (or both) and therefore won't be in L. (n = 0 also works here.)
Cases 6, 7: v is located within a^M and y is located within c^M, or v is located within b^M and y is located within c^M. No such division is pumpable, by the same logic as in Case 5.

Since every way of dividing $a^Mb^Mc^M$ into 5 parts (such that the 2nd and 4th are not both empty) is covered by (at least one of the above 7 cases, and in each case we find that the resulting division is not pumpable, we conclude that there is *no* division of $a^Mb^Mc^M$ that is pumpable. Since all this was predicated on the assumption that L was a context-free language, we conclude that L, is not context-free after all.

Notice that we didn't need to use condition the fact that |vxy| must be less than M in this proof, although we could have used it as an alternative way to handle case 6, since it prevents v and y from being separated by a region of size M, which is exactly the size of the region of b's that occurs between the a's and the c's.

Example 2: Show that $L = \{w \in \{a, b\ c\}^* \mid \#(a, w) = \#(b, w) = \#(c, w)\}$ is not context free. (We use the notation #(a, w) to mean the number of a's in the string w.)

Let's first try to use the pumping lemma. We could again choose w = $a^Mb^Mc^M$. But now we can't immediately brush off case 1 as we did in Example 1, since L allows for strings that have the a's, b's, and c's interleaved. In fact, this time there *are* ways to divide $a^Mb^Mc^M$ into 5 parts (v, y not both empty), such that the result is pumpable. For example, if v were ab and y were c, then uv^nxy^nz would be in L for all $n \geq 0$, since it would still contain equal numbers of a's, b's, and c's.

So we need some additional help, which we'll find in the closure theorems for context-free languages. Our problem is that the definition of L is too loose, so it's too easy for the strings that result from pumping to meet the requirements for being in L. We need to make L more restrictive. Intersection with another language would be one way we could do that. Of course, since the context-free languages are not closed under intersection, we can't simply consider some new language L' = L $\cap$ L2, where L2 is some arbitrary context-free language. Even if we could use pumping to show that L' isn't context free, we'd know nothing about L. But the context-free languages *are* closed under intersection with regular languages. So if we construct a new language L' = L $\cap$ L2, where L2 is some arbitrary *regular* language, and then show that L' is not context free, we know that L isn't either (since, if it were, its intersection with a regular language would also have to be context free). Generally in problems of this sort, the thing to do is to use intersection with a regular language to constrain L so that all the strings in it must have identifiable regions of symbols. So what we want to do here is to let L2 = a*b*c*. Then L' = L $\cap$ L2 = $a^nb^nc^n$. If we hadn't just proved that $a^nb^nc^n$ isn't context free, we could easily do so. In either case, we know immediately that L isn't context free.

Recursively Enumerable Languages, Turing Machines, and Decidability

1 Problem Reduction: Basic Concepts and Analogies

The concept of problem reduction is simple at a high level. You simply take an algorithm that solves one problem and use it as a subroutine to solve another problem. For example, suppose we have two TM's: C, which turns ❑w$\underline{❑}$ into ❑w❑w$\underline{❑}$ and S_L, which turns …❑w$\underline{❑}$ into …w$\underline{❑}$❑ (i.e., it shifts w one square to the left). Then we can build a TM M' that computes f(w) = ww by simply letting M' $=CS_L$. We have reduced the problem of computing f to the problem of copying a string and then shifting it.

Let's consider another example. Suppose we had a function sqr(m: integer): integer, which accepts an integer input m and returns m^2. Then we could write the following function to compute g(m) = $m^2 + 2m + 1$:

```
function g(m: integer): integer;
begin
    return sqr(m + 1);
end;
```

We have reduced the problem of computing $m^2 + 2m + 1$ to the problem of computing m + 1 and squaring it.

We are going to be using reduction specifically for decision problems. In other words, we're interested in a particular class of boolean functions whose job is to look at strings and tell us, for each string, whether or not it is in the language we are concerned with. For example, suppose we had a TM M that decides the language a^*b^+. Then we could make a new TM M' to decide a^*b^*: M' would find the first blank to the right of its input and write a single b there. It would then move its read head back to the beginning of the string and invoke M on the tape. Why does this work? Because $x \in a^*b^*$ iff $xb \in a^*b^+$. Looking at this in the more familiar procedural programming style, we are saying that if we have a function: f1(x: string): boolean, which tells us whether $x \in a^*b^+$, then we could write the following function that will correctly tell us whether $x \in a^*b^*$.

```
function f2(x: string): boolean;
begin
    return f1(x || 'b');
end;
```

If, for some reason, you believed that a^*b^* were an undecidable language, then this reduction would force you to believe that a^*b^+ is also undecidable. Why? Because we have shown that we can decide a^*b^* provided only that f1 does in fact decide a^*b^+. If we know that we can't decide a^*b^*, there must be something standing in the way. Unless we're prepared to believe that subroutine invocation is not computable or that concatenation of a single character to the end of a string is not computable, we must assign blame to f1 and conclude that we didn't actually have a correct f1 program to begin with.

These examples should all make sense. The underlying concept is simple. Things will become complicated in a bit because we will begin considering TM descriptions as the input stings about which we want to ask questions, rather than simple strings of a's and b's.

Sometimes, we'll use a slightly different but equivalent way of asking our question. Rather than asking whether a language is decidable, we may ask whether a problem is solvable. When we say that a problem is unsolvable, what we mean is that the corresponding language is undecidable. For example, consider the problem of determining, given a TM M and string w, whether or not M accepts w. We can ask whether this problem is solvable. But notice that this same problem can be phrased a language recognition problem because it is equivalent to being able to decide the language:

$$H = \{"M" "w" : w \in L(M)\}.$$

Read this as: H is the language that consists of all strings that can be formed from two parts: the encoding of a Turing

Machine M, followed by the encoding of a string w (which we can think of as the input to M), with the additional constraint that TM M halts on input w (which is equivalent to saying that w is in the language accepted by M).

“Solving a problem” is the higher level notion, which is commonly used in the programming/algorithm context. In our theoretical framework, we use the term “deciding a language” because we are talking about Turing Machines, which operate on strings, and we have a carefully constructed theory that lets us talk about Turing Machines as language recognizers.

In the following section, we’ll go through several examples in which we use the technique of problem reduction to show that some new problem is unsolvable (or, alternatively, that the corresponding language is undecidable). All of these proofs depend ultimately on our proof, using diagonalization, of the undecidability of the halting problem (H above).

2 Some Reductions Presented in Gory Detail

Example 1: Given a TM M, does M halt on input ε? (i.e., given M, is $\varepsilon \in L(M)$?) This problem is undecidable because we can reduce the Halting Problem H to it. What this means is that an algorithm to answer this question could be used as a subroutine in an algorithm (which is otherwise clearly effective) to solve the Halting problem. But we know the Halting problem is unsolvable; therefore this question is unsolvable. So how do we prove this?

First we'll prove this using the TM/language framework. In other words, we’re going to show that the following language LE is not decidable:

LE = {"M": $\varepsilon \in L(M)$}

We will show that if LE is decidable, so is H = {"M" "w" : $w \in L(M)$}.

Suppose LE is decidable; then some TM M_{LE} decides it. We can now show how to construct a new Turing Machine M_H, which will invoke M_{LE} as a subroutine, and which will decide H. In the process of doing so, we’ll use only clearly computable functions (other than M_{LE}, of course). So when we finish and realize that we have a contradiction (since we know that M_H can’t exist), we know that the blame must rest on M_{LE} and thus we know that M_{LE} cannot exist.

M_H is the Turing Machine that operates as follows on the inputs "M", "w":

1. Construct a new TM M*, which behaves as follows:
   1.1. Copy "w" onto its tape.
   1.2. Execute M on the resulting tape.
2. Invoke M_{LE}(M*).

If M_{LE} returns True, then we know that M (the original input to M_H) halts on w. If M_{LE} returns False, then we know that it doesn’t. Thus we have built a supposedly unbuildable M_H. How did we do it? We claimed when we asserted the existence of M_{LE} that we could answer what appears to be a more limited question, does M halt on the empty tape? But we can find out whether M halts on any other specific input (w) by constructing a machine (M*) that starts by writing w on top of whatever was originally on its tape (thus it ignores its actual input) and then proceeds to do whatever M would have done. Thus M* behaves the same on all inputs. Thus if we knew what it does on any one input, we’d know what it does for all inputs. So we ask M_{LE} what it does on the empty string. And that tells us what it does all the time, which must be, by the way we constructed it, whatever the original machine M does on w.

The only slightly tricky thing here is the procedure for constructing M*. Are we sure that it is in fact computable? Maybe we’ve reached the contradiction of claiming to have a machine to solve H not by erroneously claiming to have M_{LE} but rather by erroneously claiming to have a procedure for constructing M*. But that’s not the case. Why not? It’s easy to see how to write a procedure that takes a string w and builds M*. For example, if "w" is "ab", then M* must be:

$ERaRbL_{\square}M$, where E is a TM that erases its tape and then moves the read head back to the first square.

In other words, we erase the tape, move back to the left, then move right one square (leaving one blank ahead of the new tape contents), write a, move right, write b, move left until we get back to the blank that’s just to the left of the input, and then execute M.

The Turing Machine to construct M* is a bit too complicated to write here, but we can see how it works by describing it in a more standard procedural way: It first writes ER. Then, for each character in w, it writes that character and R. Finally it writes $L_{\square}M$.

To make this whole process even clearer, let's look at this problem not from the point of view of the decidability of the language H but rather by asking whether we can solve the Halting problem. To do this, let's describe in standard code how we could solve the Halting problem if we had a subroutine M_{LE}(M: TM) that could tell us whether a particular Turing Machine halts on the empty string. We'll assume a datatype TM. If you want to, you can think of objects of this type as essentially strings that correspond to valid Turing Machines. It's like thinking of a type Cprogram, which is all the strings that are valid C programs.

We can solve the Halting program with following function Halts:

```
Function Halts(M: TM, w: string): Boolean;
    M* := Construct(M, w);
    Return MLE(M*);
    end;

Function Construct(M: TM, w: string): TM;
    /* Construct builds a machine that first erases its tape.  Then it copies w onto its tape and moves its
    /* read head back to the left ready to begin reading w.  Finally, it executes M.
    Construct := Erase;         /* Erase is a string that corresponds to the TM that erases its input tape.
    For each character c in w do
        Construct := Construct || "R" || c;
        end;
    Construct := Construct || $L_{\square}$M;
    Return(Construct);

Function MLE(M: TM): Boolean;
    The function we claim tells us whether M halts on the empty string.
```

The most common mistake that people make when they're trying to use reduction to show that a problem isn't solvable (or that a language isn't decidable) is to do the reduction backwards. In this case, that would mean we would put forward the following argument: "Suppose we had a program Halts to solve the Halting problem (the general problem of determining whether a TM M halts on some arbitrary input w). Then we could use it as a subroutine to solve the specific problem of determining whether a TM M halts on the empty string. We'd simply invoke Halts and pass it M and ε. If Halts returns True, then we say yes. If Halts returns False, we say no. But since we know that Halts can't exist, no solution to our problem can exist either." The flaw in this argument is the last sentence. Clearly, since Halts can't exist, this particular approach to solving our problem won't work. But this says nothing about whether there might be some other way to solve our problem.

To see this flaw even more clearly, let's consider applying it in a clearly ridiculous way: "Suppose we had a program Halts to solve the Halting problem. Then we could use it as a subroutine to solve the problem of adding two numbers a and b. We'd simply invoke Halts and pass it the trivial TM that simply halts immediately and the input ε. If Halts returns True, then we return a+b. If Halts returns False, we also return a+b. But since we know that Halts can't exist, no solution to our problem can exist either." Just as before, we have certainly written a procedure for adding two numbers that won't work, since Halts can't exist. But there are clearly other ways to solve our problem. We don't need Halts. That's totally obvious here. It's less so in the case of attempting to build M_{LE}. But the logic is the same in both cases: flawed.

Example 2: Given a TM M, is $L(M) \neq \emptyset$? (In other words, does M halt on anything at all?). Let's do this one first using the solvability of the problem perspective. Then we'll do it from the decidability of the language point of view.

This time, we claim that there exists:

```
Function MLA(M: TM): Boolean;
    Returns T if M halts on any inputs at all and False otherwise.
```

We show that if this claim is true and MLA does in fact exist, then we can build a function MLE that solves the problem of determining whether a TM accepts the empty string. We already know, from our proof in Example 1, that this problem isn't solvable. So if we can do it using MLA (and some set of clearly computable functions), we know that MLA cannot in fact exist.

The reduction we need for this example is simple. We claim we have a machine MLA that tells us whether some machine M accepts anything at all. If we care about some particular input to M (for example, we care about ε), then we will build a new machine M* that erases whatever was originally on its tape. Then it copies onto its tape the input we care about (i.e., ε) and runs M. Clearly this new machine M* is oblivious to its actual input. It either always accepts (if M accepts ε) or always rejects (if M rejects ε). It accepts everything or nothing. So what happens if we pass M* to MLA? If M* always accepts, then its language is not the empty set and MLA will return True. This will happen precisely in case M halts on ε. If M* always rejects, then its language is the empty set and MLA will return False. This will happen precisely in case M doesn't halt on ε. Thus, if MLA really does exist, we have a way to find out whether any machine M halts on ε:

```
Function MLE(M: TM): Boolean;
    M* := Construct(M);
    Return MLA(M*);
    end;

Function Construct(M: TM): TM;  /* This time, we build an M* that simply erases its input and then runs M
                                  /*(thus running M on ε).
    Construct := Erase;         /* Erase is a string that corresponds to the TM that erases its input tape.
    Construct := Construct || M;
    Return(Construct);
```

But we know that MLE can't exist. Since everything in its code, with the exception of MLA, is trivially computable, the only way it can't exist is if MLA doesn't actually exist. Thus we've shown that the problem determining whether or not a TM M halts on any inputs at all isn't solvable.

Notice that this argument only works because everything else that is done, both in MLE and in Construct is clearly computable. We could write it all out in the Turing Machine notation, but we don't need to, since it's possible to prove that anything that can be done in any standard programming language can also be done with a Turing Machine. So the fact that we can write code for it is good enough.

Whenever we want to try to use this approach to decide whether or not some new problem is solvable, we can choose to reduce to the new problem any problem that we already know to be unsolvable. Initially, the only problem we knew wasn't solvable was the Halting problem, which we showed to be unsolvable using diagonalization. But once we have used reduction to show that other problems aren't solvable either, we can use any of them for our next problem. The choice is up to you. Whatever is easiest is the thing to use.

When we choose to use the problem solvability perspective, there is always a risk that we may make a mistake because we haven't been completely rigorous in our definition of the mechanisms that we can use to solve problems. One big reason for even defining the Turing Machine formalism is that it is both simple and rigorously defined. Thus, although the problem solvability perspective may seem more natural to us, the language decidability perspective gives us a better way to construct rigorous proofs.

So let's show that the language

$LA = \{"M": L(M) \neq \varnothing\}$ is undecidable.

We will show that if LA were decidable, then $LE = \{"M": \varepsilon \in L(M)\}$ would also be decidable. But of course, we know that it isn't.

Suppose LA is decidable; then some TM M_{LA} decides it. We can now show how to construct a new Turing Machine M_{LE}, which will invoke M_{LA} as a subroutine, and which will decide LE:

```
M_LE(M):    /* A decision procedure for LE = {"M": $\varepsilon \in$ L(M)}
1.  Construct a new TM M*, which behaves as follows:
    1.1.  Erase its tape.
    1.2.  Execute M on the resulting empty tape.
2.  Invoke M_LA(M*).
```

It's clear that M_{LE} effectively decides LE (if M_{LA} really exists). Why? M_{LE} returns True iff M_{LA} returns True. That happens, by its definition, if it is passed a TM that accepts at least one string. We pass it M*. M* accepts at least one string (in fact, it accepts all strings) precisely in case M accepts the empty string. If M does not accept the empty string, then M* accepts nothing and M_{LE} returns False.

Example 3: Given a TM M, is L(M) = Σ^*? (In other words, does M accept everything?). We can show that this problem is unsolvable by using almost exactly the same technique we just used. In example 2, we wanted to know whether a TM accepted anything at all. Now we want to know whether it accepts everything. We will answer this question by showing that the language

$L\Sigma = \{"M": L(M) = \Sigma^*\}$ is undecidable.

Recall the machine M* that we constructed for Example 2. It erases its tape and then runs M on the empty tape. Clearly M* either accepts nothing or it accepts everything, since its behavior is independent of its input. M* is exactly what we need for this proof too. Again we'll choose to reduce the language LE = {"M": $\varepsilon \in$ L(M)} to our new problem L:

If $L\Sigma$ is decidable, then there's a TM $M_{L\Sigma}$ that decides it. In other words, there's a TM that tells us whether or not some other machine M accepts everything. If $M_{L\Sigma}$ exists, then we can define the following TM to decide LE:

```
M_LE(M):    /* A decision procedure for LE = {"M": $\varepsilon \in$ L(M)}
1.  Construct a new TM M*, which behaves as follows:
    1.1.  Erase its tape.
    1.2.  Execute M on the resulting empty tape.
2.  Invoke M_LΣ(M*).
```

Step 2 will return True if M* halts on all strings in Σ^* and False otherwise. So it will return True if and only M halts on ε. This would seem to be a correct decision procedure for LE. But we know that such a procedure cannot exist and the only possible flaw in the procedure we've given is $M_{L\Sigma}$. So $M_{L\Sigma}$ doesn't exist either.

Example 4: Given a TM M, is L(M) infinite? Again, we can use M*. Remember that M* either halts on nothing or it halts on all elements of Σ^*. Assuming that $\Sigma \neq \varnothing$, that means that M* either halts on nothing or it halts on an infinite number of strings. It halts on everything if its input machine M halts on ε. Otherwise it halts on nothing. So we can show that the language

LI = {"M": L(M) is infinite}

is undecidable by reducing the language LE = {"M": $\varepsilon \in$ L(M)} to it.

If LI is decidable, then there is a Turing Machine M_{LI} that decides it. Given M_{LI}, we decide LE as follows:

```
M_LE(M):    /* A decision procedure for LE = {"M": $\varepsilon \in$ L(M)}
1.  Construct a new TM M*, which behaves as follows:
    1.1.  Erase its tape.
    1.2.  Execute M on the resulting empty tape.
2.  Invoke M_LI(M*).
```

Step 2 will return True if M* halts on an infinite number of strings and False otherwise. So it will return True if and only M halts on ε.

This idea that a single construction may be the basis for several reduction proofs is important. It derives from the fact that several quite different looking problems may in fact be distinguishing between the same two cases.

Example 5: Given two TMs, M_1 and M_2, is $L(M_1) = L(M_2)$? In other words, is the language

LEQ = {"M_1" "M_2": $L(M_1) = L(M_2)$} decidable?

Now, for the first time, we want to answer a question about the relationship of two Turing Machines to each other. How can we solve this problem by reducing to it any of the problems we already know to be undecidable? They all involve only a single machine. The trick is to use a constant, a machine whose behavior we are certain of. So we define M#, which halts on all inputs. M# is trivial. It ignores its input and goes immediately to a halt state.

If LEQ is decidable, then there is a TM M_{LEQ} that decides it. Using M_{LEQ} and M#, we can decide the language

LΣ = {"M": L(M) = Σ*} (which we showed in example 3 isn't decidable) as follows:

```
M_LΣ(M):    /* A decision procedure for LΣ
1.  Invoke M_LEQ(M, M#).
```

Clearly M accepts everything if it is equivalent to M#, which is exactly what M_{LEQ} tells us.

This reduction is an example of an easy one. To solve the unsolvable problem, we simply pass the input directly into the subroutine that we are assuming exists, along with some simple constant. We don't need to do any clever constructions. The reason this was so simple is that our current problem LEQ, is really just a generalization of a more specific problem we've already shown to be unsolvable. Clearly if we can't solve the special case (determining whether a machine is equivalent to M#), we can't solve the more general problem (determining whether two arbitrary machines are equivalent).

Example 6: Given a TM M, is L(M) regular? Alternatively, is

LR = {"M": L(M) is regular} decidable?

.

To answer this one, we'll again need to use an interesting construction. To do this, we'll make use of the language

H = {"M" "w" : $w \in L(M)$}

Recall that H is just the set of strings that correspond to a (TM, input) pair, where the TM halts on the input. H is not decidable (that's what we proved by diagonalization). But it is semidecidable. We can easily built a TM H_{semi} that halts whenever the TM M halts on input w and that fails to halt whenever M doesn't halt on w. All H_{semi} has to do is to simulate the execution of M on w. Note also that H isn't regular (which we can show using the pumping theorem).

Suppose that, from M and H_{semi}, we construct a machine M\$ that behaves as follows: given an input string w, it first runs H_{semi} on w. Clearly, if H_{semi} fails to halt on w, M\$ will also fail to halt. But if H_{semi} halts, then we move to the next step, which is to run M on ε. If we make it here, then M\$ will halt precisely in case M would halt on ε. So our new machine M\$ will either:

1. Accept H, which it will do if $\varepsilon \in L(M)$, or
2. Accept ∅, which it will do if $\varepsilon \notin L(M)$.

Thus we see that M\$ will accept either

1. A non regular language, H, which it will do if $\varepsilon \in L(M)$, or
2. A regular language ∅, which it will do if $\varepsilon \notin L(M)$.

So, if we could tell whether M\$ accepts a regular language or not, we'd know whether or not M accepts ε.

We're now ready to show that LR isn't decidable. If it were, then there would be some TM M_{LR} that decided it. But M_{LR} cannot exist, because, if it did, we could reduce LE = {"M": $\varepsilon \in L(M)$} to it as follows:

```
M_LE(M):    /* A decision procedure for LE = {"M": $\varepsilon \in L(M)$}
1.  Construct a new TM M$(w), which behaves as follows:
    1.1.  Execute $H_{semi}$ on w.
```

1.2. Execute M on ε.
2. Invoke M_{LR}(M$).
3. If the result of step 2 is True, return False; if the result of step 2 is False, return True.

$M_{LE,}$ as just defined, effectively decides LE. Why? If $\varepsilon \in L(M)$, then L(M$) is H, which isn't regular, so M_{LR} will say False and we will, correctly, say True. If $\varepsilon \notin L$, then L(M$) is ∅, which is regular, so M_{LR} will say True and we will, correctly, say False.

By the way, we can use exactly this same argument to show that LC = {"M": L(M) is context free} and LD = {"M": L(M) is recursive} are undecidable. All we have to do is to show that H is not context free (by pumping) and that it is not recursive (which we did with diagonalization).

Example 7: Given a TM M and state q, is there any configuration (p, $u\underline{a}v$), with $p \neq q$, that yields a configuration whose state is q? In other words, is there any state p that could possibly lead M to q? Unlike many (most) of the questions we ask about Turing Machines, this one is not about future behavior. (e.g., "Will the Turing Machine do such and such when started from here?") So we're probably not even tempted to try simulation (which rarely works anyway).

But there is a way to solve this problem. In essence, we don't need to consider the infinite number of possible configurations of M. All we need to do is to examine the (finite) transition table of M to see whether there is any transition from some state other than q (call it p) to q. If there is such a transition (i..e., if $\exists$ p, σ, τ such that $\delta(p, \sigma) = (q, \tau)$), then the answer is yes. Otherwise, the answer is no.

3 Rice's Theorem

Rice's Theorem makes a very general statement about an entire class of languages that are not recursive. Thus, for some languages, it is an alternative to problem reduction as a technique for proving that a language is not recursive.

There are several different forms in which Rice's Theorem can be stated. We'll present two of them here:

(Form 1) Any nontrivial property of the recursively enumerable languages is not decidable.

(Form 2) Suppose that C is a proper, nonempty subset of the class of all recursively enumerable languages. Then the following language is undecidable: LC = {<M>: L(M) is an element of C}.

These two statements look quite different but are in fact nearly equivalent. (Although Form 1 is stronger since it makes a claim about the language, no matter how you choose to define the language. So it applies given a grammar, for example. Form 2 only applies directly if the way you define the language is by a Turing Machine that semidecides it. But even this difference doesn't really matter since we have algorithms for constructing a Turing Machine from a grammar and vice versa. So it would just take one more step if we started with a grammar and wanted to use Form 2). But, if we want to prove that a language of Turing machine descriptions is not recursive using Rice's Theorem, you must do the same things, whichever description of it you prefer.

We'll consider Form 1 first. To use it, we first, we have to guarantee that the property we are concerned with is a property (predicate) whose domain is the set of recursively enumerable languages. Here are some properties P whose domains are the RE languages:
1) P is true of any RE language that contains an even number of strings and false of any RE language that contains an odd number of strings.
2) P is true of any RE language that contains all stings over its alphabet and false for all RE languages that are missing any strings over their alphabet.
3) P is true of any RE language that is empty (i.e., contains no strings) and false of any RE language that contains any strings.
4) P is true of any RE language

5) P is true of any RE language that can be semidecided by a TM with an even number of states and false for any RE language that cannot be semidecided by such a TM.
6) P is true of any RE language that contains at least one string of infinite length and false of any RE language that contains no infinite strings.

Here are some properties whose domains are not the RE languages:
1′) P is true of Turing machines whose first move is to write "a" and false of other Turing machines.
2′) P is true of two tape Turing machines and false of all other Turing machines.
3′) P is true of the negative numbers and false of zero and the positive numbers.
4′) P is true of even length strings and false of odd length strings.

We can attempt to use Rice’s Theorem to tell us that properties in the first list are undecidable. It won’t help us at all for properties in the second list.

But now we need to do one more thing: We must show that P is a *nontrivial* property. Any property P is nontrivial if it is not equivalent to True or False. In other words, it must be true of at least one language and false of at least one language.

Let’s look at properties 1-6 above:
1) P is true of {“a”, “b”} and false of {“a”}, so it is nontrivial.
2) Let’s just consider the case in which Σ is {a, b}. P is true of Σ* and false of {“a”}.
3) P is true of $\varnothing$ and P is false of every other RE language.
4) P is true of any RE language and false of nothing, so P is trivial.
5) P is true of any RE language and false of nothing, so P is trivial. Why? Because, for any RE language L there exists a semideciding TM M. If M has an even number of states, then P is clearly true. If M has an odd number of states, then create a new machine M′ identical to M except it has one more state. This state has no effect on M′s behavior because there are no transitions in to it. But it guarantees that M′ has an even number of states. Since M′ accepts L (because M does), P is true of L. So P is true of all RE languages.
6) P is false for all RE languages. Why? Because the definition of a language is a set of strings, each of finite length. So no RE language contains a string of infinite length.

So we can use Rice’s theorem to prove that the set of RE languages possessing any one of properties 1, 2, or 3 is not recursive. But it does not tell us anything about the set of RE languages possessing property 3, 4, or 5.

In summary, to apply this version of Rice’s theorem, it is necessary to do three things:
0) Specify a property P.
1) Show that the domain of P is the set of recursively enumerable languages.
2) Show that P is nontrivial by showing:
   a) That P is true of at least one language, and
   b) That P is false of at least on language.

Now let’s try to use Form 2. We must find a C that is a proper, nonempty subset of the class of all recursively enumerable language.

First we notice that this version is stated in terms of C, a subset of the RE languages, rather than P, a property (predicate) that is true of the RE languages. But this is an insignificant difference. Given a universe U, then one way to define a subset S of U is by a characteristic function that, for any candidate element x, returns true if $x \in S$ and false otherwise. So the P that corresponds to S is simply whatever property the characteristic function tests for. Alternatively, for any subset S there must exist a characteristic function for S (although that function need not be computable – that’s a different issue.) So given a set S, we can define the property P as “is a member of S.” or “possesses whatever property it takes to be determined to be n S.” So Form 2 is stated in terms of the set of languages that satisfy some property P instead of being stated in terms of P directly, but as there is only one such set for any property P and there is only one such property P (viewed simply as a truth table, ignoring how you say it in English) for any set, it doesn’t matter which specification we use.

Next we notice that this version requires that C be a proper, nonempty subset of the class of RE languages. But this is exactly the same as requiring that P be nontrivial. Why? For P to be nontrivial, then there must exist at least one language of which it is true and one of which it is false. Since there must exist one language of which it is true, the set of languages that satisfy it isn't empty. Since there must exist one language of which it is false, the set of languages that satisfy it is not exactly the set of RE languages and so we have a proper subset.

So, to use Form 2 of Rice's Theorem requires that we:
0) Specify some set C (by specifying a membership predicate P or some other way).
1) Show that C is a subset of the set of RE languages (which is equivalent to saying that the domain of its membership predicate is the set of RE languages)
2) Show that C is a proper nonempty subset of the recursive languages by showing that
   a) $C \neq \varnothing$ (i.e., its characteristic function P is not trivially false), and
   b) $C \neq RE$ (i.e., its characteristic function P is not trivially true).