[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Why don't we get started. So as I like to do, let's just review where we have been recently, which is to discuss context-free languages. We talked about the context-free grammars and the pushdown automata as a way of describing the context-free languages.

As you remember, the context-free languages are a larger class of languages than the regular languages, which is where we started, the languages of the finite automata. So when you add a stack, you get more power. You get more languages that you can do.

And we're very rapidly going to be moving on today to our main model for the semester, which is called the Turing machine. So let's just take a look at what we're going to be covering today. And that is, first, we're going to show that a technique analogous to the one we use for proving that languages are not regular, but this time for proving languages are not context-free.

So the pushdown automata and the grammar still have their limitations in terms of what we normally think a computer can do. And with that, we're going to use that as a kind of a lead-in to our general-purpose model, which is the Turing machine. And so we're going to talk about Turing machines and aspects of that.

I would want to comment-- so I have posted the solutions for the first problem set. I know you're starting to think about the second problem set now, which I have posted as well. If you want to get a sense of what I'm looking for in terms of the level of detail, you can look at the solutions to problem set one, because I consider those to be model solutions.

That's part of the reason why I post them, just to give you a sense of the level of detail that I'm looking for, which is not a whole lot. But I do want to make sure you're capturing the main ideas of what's involved in solving the problem. So have a look at those.

And for problem set two, which I'll talk about in a second-- so I'll just say a few words. If you want to pull that up, you can do that. But just to get you started on a few of the problems if you're finding some challenges there-- I don't want you to get stuck really before you even understand what the problem is saying. So for problem number one, if you looked at that, so that's a problem where you're asked to prove a certain language is not context free.

And by the way, all of the problems in this problem set except perhaps for the last, for number six, you'll be able to solve. We'll have enough material at the end of today's lecture to solve all of them. I believe that's right. Yeah, so number six, you should have enough as of Thursday's lecture to solve that.

So problem number one, it's proving a language is not context free. So we're going to introduce a method for doing that. That method is going to come in handy.

For parts B and C, if you look at the problem set, it has this strange-looking thing, sigma sigma sigma in parenthesis star. It's really just a regular expression that's very simple. You should just make sure you understand that that's a way of representing all strings whose length is a multiple of 3.

And if I stick a sigma in front of that, it's all strings whose length is 1 plus a multiple of 3. So once you understand that, and if you think about what kinds of strings are in the language C2, it'll help you to understand what happens when you take those unions. And parts B and C are not intended to be very hard, but you just have to understand what's going on.

Problem number two is about ambiguous grammars. I touched on that briefly in lecture. It's enough to solve the problem. The book has a little bit more detail about ambiguous languages, ambiguous grammars-- ambiguous grammars, I should say.

And so this is a grammar that's supposed to represent a fragment of a programming language with if thens and if then elses. I'm sure you're all familiar with those kinds of constructs in programming languages. And there is a natural ambiguity that comes up in a programming language.

If you have if some condition then statement one, else statement two, I presume you understand what the semantics of that is, what that means. And the tricky thing is that if you have-- those statements can themselves be if statements. And so if you have the situation where you have if then and if then else is what follows that, the question is, where does the else attach? Is it to the second if or to the first if?

So that's kind of a big hint on this problem, but that's OK. You need to take that and figure out how to get an actual member of the language which is ambiguously generated, and then show that it has-- show that it is by showing two parse trees or two leftmost derivations. If you read the book, you'll see that's an alternative way of representing a parse tree.

So and then what you're supposed to do is give a grammar for the same language which is unambiguous. You don't have to prove that it's unambiguous, because that's a bit of a chore. But as long as you understand what's going on, you should be able to come up with an unambiguous grammar which resolves that ambiguity.

And I don't have in mind changing the language by introducing new programming language constructs like a "begin end." That's not in the spirit of this problem, because that's a different-- it's grammar for a different language. So you need to be generating the same language without any other extraneous things going on that are going to resolve the ambiguity. The ambiguity needs to be resolved within the structure of the grammar itself So keep that in mind.

For problem number three about the queue automata, you know, that came up actually as a suggestion last lecture, I believe, or two lectures back. What happens if you take a pushdown automaton, but instead of a pushdown-- instead of a stack, you add a queue. What happens then?

Well actually, it turns out that the model you get is very powerful. And it turns out to be equivalent in power to a Turing machine. So you'll see arguments of that kind today, how you show that other models are equivalent-- no, not today.

So I apologize. This is going to be something that you'll-- I'm confusing myself here. Problem number three actually needs Thursday's lecture as well to really at least see examples of how you do that kind of thing. Yeah, so I'll try to send out a note clarifying this. By the end of Thursday, you'll be able to do everything, except for problem six. And for problem six, you'll need Tuesday's lecture, a week from today's lecture, to do.

So problem number four, that one you'll be able to do at the end of today. That's also going to-- the problem is I'm working on preparing Thursday's lecture too. So I'm getting a little-- I'm confusing myself.

Problem number four, you'll be able to do after Thursday's lecture. Maybe we should talk about that next lecture. Problem number five, you can do today, but maybe I'm not going to say anything about that. And problem number six, I won't say anything about either.

OK, so why don't we just jump in then and look at today's material. What about seven? Oh, seven is an optional problem. Oh, I should have mentioned that. Seven is always going to be an option. I indicate that with a star I should have made that clear on the actual description here, but seven is optional. It's just like we had for problem set one.

OK, let's move let's move on, then, to what we're going to talk about today. And just a little bit of review-- so we talked about the equivalence of context-free grammars and pushdown automata, as you remember. Oops, let me get myself out of the picture here.

As we mentioned last time, we actually proved one direction, but the other direction of that, you just have to know it's true, but you don't have to know the proof. The proof is a little bit lengthy, I would say. It's a nice proof, but it's pretty long.

And there are two important corollaries to that. If you know what a corollary is, it's just a simple consequence which doesn't need much of a proof, sort of a very straightforward consequence. First of all, I think we pointed out last time, one conclusion, one corollary you get is that every regular language is a context-free language, because a finite automaton is a pushdown automaton that just happens not to use its stack.

So immediately, you get that every language is context free. And second of all, you also immediately get that whenever you have a context-free language and a regular language and you take their intersection, you get back a context-free language. So context free intersect regular is context free.

That's actually mentioned in your homework as well as one of the 0.x problems which I give to try to get you-- you don't have to turn those in, but I suggest you look at them. I don't know how many of you are looking at them. But this is a useful fact.

And some of those other facts in 0.x problems are useful. So I encourage you to look at them. But anyway, intersection of context free and regular is context free.

You might ask, what about intersection of context free and context free? Do we have closure under intersection? The answer is, no, we do not have close to closure under intersection. We'll talk about that shortly.

So here is the proof sketch for-- I wanted to say that the intersection of context free and regular, why do we know that's still context free? Because the pushdown automaton for A can be simulating the finite automaton for B inside its finite control, inside its finite memory. The problem is, if you have two context-free languages, you have two pushdown automata, you can't simulate that with one pushdown automaton, because it has only a single stack. So if you're trying to take the intersection of two context-free languages with only a single stack, you're going to be in trouble, because it's hard to-- anyway, that's not a proof, but at least it shows you what goes wrong if you try to do the obvious thing.

OK, so if-- and just, here is an important point that was trying to make before. If A and B are both context free and you're taking the intersection, the result may not necessarily be a context-free language. So the class of context-free languages is not closed under its intersection. We'll comment on that in a bit.

The context-free languages are closed under the regular operations, however, union, intersection-- union, concatenation, and star. So you should feel comfortable that you know how to prove that. Again, it's one of the-- I think it's problem 0.2.

And I think the solution is even given in the book for it. So you just should know how to prove that. It's pretty straightforward.

OK, so let's move on then to basically conclude our work on context-free languages, to understand the limitations of context-free grammars, and what kinds of languages may not be context free. And how do you prove that? So how do you prove that, for some language, there is no grammar?

Again, you know, it's not enough just to, say, give an informal comment that, I couldn't think of a grammar, or some-- things of that kind. That's not going to be good enough. We need to have a proof.

So if we take the language here, 0 to the k, 1 to the k, 2 to the k, so those are strings which are runs of 0's followed by an equal number of 1's followed by an equal number of 2's, so just 0's, then 1's, then 2's, all the same length. That's a language which is not going to be a context-free language. And we'll give a method for proving that.

If you had a stack, you can match the 1's with the 0's, but then once you're done with that, the stack is empty. And how do you now make sure that the number of 2's corresponds to the number of 1's that you had? So again, that's an informal argument that's not good enough to be a proof, but it sort of gives an intuition.

So we're going to give a method for proving non-context-free-- languages are not context free using, again, a pumping lemma. But this is going to be a pumping lemma that applies to context-free language, not to regular languages. It looks very similar, but it has some extra wrinkles thrown in, because the other older pumping lemma was specific to the regular languages. And this is going to be something that applies to the context-free languages.

OK, so now let's just read it. And then we'll try to interpret it again. It's very similar in spirit. Basically, it says that, whenever you have a context-free language, all long strings in the language can be pumped in some kind of way. So it's going to be a little different kind of pumping than we had before. And you stay in the language.

OK, so before, we broke the string into three pieces where we could repeat that centerpiece as many times as you like. And you stay in the language. Here, we're going to end up breaking the string into five pieces. So s is going to be broken up into uvxyz.

And the way it's going to work here-- so here is a picture. So all long strings-- again, there is going to be a threshold. So whenever you have a language, there is going to be some cut-off length.

So all the longer strings in that language can be pumped. And you stay in the language. But the shorter strings, there is no guarantee.

So if you have a long string in the language of length at least this pumping length p, then you can break it up into five pieces. But now it's that second and fourth string that are going to play that special pumping role, which means that, what you can do is you can repeat those and you stay in the language. And it's important that you repeat them both, that v and that y, the same number of times.

So you're going to have a picture that looks something like this. And that is going to you repeat. If you repeat the v and you repeat the y, you get uvvxyyz.

Or if you look at over here, it would be uv squared xy squared z. And that's going to still be in the language. And then we have-- so that's one condition. We'll have to look at all of these conditions when we do the proof, but we just want to understand what the statement is right now.

So the second condition is that v and y together cannot be empty. And really, that's another way of saying, they can't both be the empty string, because if they were both the empty string, then repeating them wouldn't change s. And then of course it would stay in the language. So it would be kind of meaningless if they were allowed to be empty.

And the last thing is, again, going to be there as a matter of convenience for proving languages are not context free, because you have to make sure there is no possible way of cutting up the string. When you're trying to prove a language is not context free, you have to show the pumping fails. It's going to be helpful sometimes to limit the ways in which the string can be cut up, because then you have-- it's an easier job for you to work with it.

So here, it's a little different than before, but sort of similar, that vxy combine as a substring. So I show that over here. vxy together is not too long. So the vxy-- maybe it's better seen up here-- is going to be, at most, p. We'll do an example in a minute of using this.

OK, so again, here is our pumping lemma. I've just restated it. So we have it in front of us.

And we're going to do a proof. I'm just going to give you the idea of the proof first. And then we'll go through some of the details.

The idea is actually pretty simple. We give it-- call it a proof by picture. Again, remember what we're trying to do. We're trying to show that we have this context-free language A. And now all long strings in A have this pumping quality, that you can break them up into five pieces so that the second and the fourth piece can be repeated. And you stay in the language.

So how do we know that that's going to be true? Let's take a look at the proof here. And why is that going to be true? So first of all, I'd like to do it qualitatively rather than quantitatively.

So let's just imagine, instead of thinking-- we'll calculate what p is later. But just imagine that s is some really, really long string. That's the way I like to think about it. So s is just really long. What is that going to tell us? It's going to tell us something important about the way the grammar produces s, which is going to be useful in getting a way of pumping it.

So if s is really long, we're going to look at the parse tree for s. And we're going to conclude that the parse tree has to be really tall, because it's impossible for a very shallow parse tree to generate a very long string. And again, we'll quantify that in a second.

But intuitively, I think that's not too hard to see why that ought to be true. So if you have a long s, the parse tree has to be really tall, because the parse tree can't generate very many-- it can't expand by very much at each level. So we'll look at how much it can expand.

But it depends on the grammar, how much expansion have at each level. And it's going to be-- you can't have just in three levels some small grammar generating a string of length 1 million. You'll see that that's just impossible.

So once you know that the parse tree is really tall here, then you're actually almost done, because what does it mean to be really tall? It means that there is some path starting at the start variable E, I'm calling it in this parse tree, which goes down to some terminal symbol in s, which goes through many steps. That's what it means for the tree to be very tall. And each one of those steps is a variable until you get down to the very end.

OK, so that's the way parse trees look. You keep expanding variables until you get to a terminal. So here, you get some path that's really a long path. And once you have a long path that has many, many variables appearing on here, well, the grammar itself has only some fixed number of variables in it, so you're going to have to have a repetition coming among the variables that occur on that long path. Got that? So a long string forces a tall parse tree, forces a repetition on some path coming out of the start variable of some other variable that comes out.

Now that's going to tell us how to cut up s, because if you look at the subtrees of s that those two R variables are generating, shown like this, I'm going to use that-- so you have to follow what I'm saying here. So R here is generating this portion of s. And the lower R is generating a smaller portion of s, just looking at the subtree that you get here.

And that's going to tell us that we can cut up s accordingly. So u was that very first part out here generated by E, but not by the first R. R is generated-- v is generated by the first R, but not by the second R. The second R generates exactly x. And then we have y and z, similarly.

So that all follows from having a tall parse tree. And now we're finished. Now we know how to cut up s. How do we know we can repeat v and y and still be in the language?

Well, I'll actually show you that you're in the language by exhibiting a parse tree for the string uvvxyyz. Here it is. I'm going to get that parse tree by, when I expand this lower R, instead of expanding it to get x, I'm going to follow the same substitutions that I had when I expanded the upper R. So it's as if I took this larger subtree here and I substituted it in for the smaller subtree under the second R. And so I get a picture that looks like this.

So here I'm substituting under the second R the same subtree that I had originally coming out of the upper R, the first R. And so now this parse tree is generating the string uvvxyyz, which is what I'm looking for. And of course, you can do that again and again. And you're going to keep getting higher and higher exponents of v and y.

And in fact, you can even get the 0 exponent, which means that v and y both disappear altogether. And for that, you do something slightly different, which is that you replace the larger subtree by the smaller subtree. OK so here, which was originally that larger tree generating vxy, I stick instead the smaller subtree.

I do the substitutions from the smaller subtree. And I just get x there. And so now the string I generated is uxz, which is the same as uv to the 0 xy to the 0 z.

And that is the idea of the proof. Now, I think you could work out the quantities that you need in order to drive this proof. I'm going to do that for you. I actually hate writing down lots of inequalities, and equations, and so on, on the board, because I think they're just almost incomprehensible to follow. Or at least they would be for me. But I'm going to put them up there just for completeness sake.

So here we're going to give the details of this proof on the next slide here. Oh yeah, so I just want to give a name to this. I'm going to call this the cutting and pasting argument, because I'm cutting apart pieces of this parse tree and I'm pasting them in to other places within the parse tree to get new strings being generated. So this is a cutting and pasting argument.

So OK, let's take a look at the details here, just, well, we have to understand, well, how big does p actually need to be in order for this thing to kick in? Well, first of all, we have to understand how fast that parse tree can be growing as we go level to level. And that's going to be dependent on how big the right-hand sides of rules are.

I mean, that really tells you how many-- what's the fan out you know of each node? What's the maximum fan out? And that's going to be the maximum length of a right-hand side of any rule.

So for example, in that other grammar we had seen last time for arithmetic expressions, we had this E goes to E plus T, this rule here. And in terms of the parse tree, that would look like a little element like that. And that's actually the longest right-hand side that you can get. And so the parse tree can be growing by a factor of 3 each time.

Now, that's going to tell us how big the string needs to be that's being generated, what is the value of p in order to get a high enough parse tree so that you're going to get a repeated variable. Let's call the height of the parse tree for S h. So now if you-- this is just repeating what I just said. If you have a tree of height h and the maximum branching is b, then you get, at most, b to the h leaves, because each level, you get another factor of b coming up, because that's how much branching you have.

So each node at one level can become b nodes at the next level down. So you're multiplying by b each time. And if you have h levels, you're going to have b to the h leaves.

So the length of s, which are really the leaves here, is at most b to the h. The reason why it's at most and not exactly is you might be doing some substitutions which are shorter right-hand sides. OK, so to try to show this as a picture here, pulling that same picture we had before, we want h, the height, to be bigger than the number of variables to force a repetition.

So the number of variables is going to be written this way. V is the variables. V with bars around it is going to be the number of variables. And we want that height to be greater than the number of variables.

So once you know how high you want that tree to be in order to force a repetition, then it tells you how big s has to be. So V has to be bigger than b to the V, b to the size of V, because then the height that you're going to get is going to be greater than the size of V, which is-- so that's what you want. You want h to be greater than the size of V.

So you're going to set p to be one more than b to the V. And so if s is at least that length, this whole thing is going to kick in. And you're going to get that repeated variable.

So we'll let p to be that value where V is the number of variables in the grammar. And so if s is at least p, which is greater than b to the V, then the length of s is going to be greater than b to the V. So h is going to be what you want to make this thing work.

If you don't follow that, those inequalities, I sympathize with you. I would never follow that either in a lecture. So but I hope you get the idea. But we're not quite finished yet, because I want to now circle back, and look at these three conditions, and make sure that we've captured them all, because actually, it's not totally obvious in each of those cases that we've got them. So there is a few extra things we need to do.

OK, so this is concluding the argument. There are going to be at least V plus 1 variables in the longest path. So there is going to be a repetition. So now let's go back here and see, now that we have this picture with a repeated variable, how do we know we can get condition one? Well, that's just the cutting and pasting argument from the previous slide.

How do we know that v and y are not both empty? Well actually, that's not totally obvious, because it's possible that, when you generated v here and you generated y, maybe going from this R to that R, you got nothing new. You know, it could have been that R got replaced by T, another variable with nothing new coming out, and then T got replaced by R.

You substituted T for R and then R for T. And you've got nothing new coming out. And in that case, v and y would both be the empty string. And that would violate what we want.

The way you get around-- that and these are details here. If you're not totally following these points, don't worry. They're easy to describe. So I figure, let me present the whole thing in full detail.

So if going from this R to that R doesn't generate anything new, you're getting exactly the same things coming out-- v and y are just the empty string-- how do we avoid that from happening? There is a simple way to address that, which is to say, if you have this string s, when you take a parse tree, make sure you take a small-as-possible parse tree. You're not allowed to start off with an inefficient parse tree that can be shortened and still generate s.

I want the smallest possible parse tree. And that smallest possible parse tree can't have an R going to another R which is generating nothing new, because then you could always have eliminated that step. And you would still have a parse tree for s, but it would be a smaller parse tree.

So that would be-- I want you to start off with the smallest possible parse tree. And then you're going to be guaranteed that v or y is going to be something not empty. So that takes care of condition two.

Condition three-- you know, how do we know that vxy together is not very long? And basically, it's the same argument all over again. You just want to make sure that, when you're picking the repetition R, the two R's here, you pick the lowest possible repetitions that occur, if you have many choices. And those lowest two, those lowest repetitions, there is not going to be any lower repetition here. And then by the same argument, since once you have that very first R, there is no more repetitions occurring below, the vxy can't be very long, because that would, again, force another repetition to occur.

So anyway, those are the three conditions. And that's the proof of the pumping lemma for h free languages. Let's see how we use that.

OK, so let's do an example of proving a language not context free using the pumping lemma. How are you going to go about doing that? Because that's the kind of thing, at the very least, you need to know how to do this in order to do the homework. I'd like to motivate you that the stuff is so interesting and fun, but it doesn't work for everybody. So for you practical people out there, pay attention so you can do the homework.

OK, let's go back to that language we had a couple of slides back, 0 to the k, 1 to the k, 2 to the k. It's not a context-free language. We're going to show that now using the pumping lemma for context-free languages.

So it's going to do, similar to the proofs using for non-regular languages, proof by contradiction. So you, first you assume the language is context-free. And then we're going to apply the pumping lemma. And then we're going to get a contradiction.

So the pumping lemma gives that pumping length, as we described above. And now we just want to pick a longer string in the language and show that that longer string, which is supposed to be pumpable and stay in the language, in fact is not pumpable. So the pumping lemma says that you can divide it into five pieces satisfying the three conditions. Condition three implies that-- so now I'm going to I'm going to work through. I'm going to show you get a contradiction.

So condition three implies that you cannot contain both 0's and-- let's pull up a picture here. So here is s, 0's, 1's, and then 2's, all of the same length. Condition three-- so if you break it up, condition three says, vxy together cannot be too long.

Well, if vxy together is not too long, how could it be that, when you're repeating v and y, you stay in the language? For one thing, you can't have 0's, 1's, and 2's all occurring within v, x, and y. Some symbol is going to get left out. So then when you pump up, you're going to have unequal numbers of symbols. And so you're going to be out of the language.

OK, so no matter how you try to cut it up following condition three, which is one of the things that restricts the ways to cut it up, you're going to end up, when you pump up, going out of the language. And so therefore, it's not in-- D? D is wrong.

B, should say "B." I'm supposed to be able to write on this thing. I guess not. I didn't test that. Oh well, that's supposed to be a B.

So B is a context-free language, which includes-- so that's the assumption, that B is a context-free language. That's false. And we conclude that it's not a context-free language.

Let's do-- oh yeah, I have a check in here. So let's see what I'm going to ask you to think about. OK, my head is blocking part of the text? Oh, that was a while ago. Yes, so just one question by the way, in terms of applying the pumping lemma-- either v or y can be empty, but not both.

But anyway, let's get to this check in here. So let's look at these two languages, A1 and A2, which look very similar to B, but a little different. So it's A1 is 0 to the k, 1 to the k, 2 to the l, where k and l could be any numbers, any positive, non-negative numbers. So basically what this is saying is that the number of 0's and 1's are going to be equal, but the number of 2's can be anything, whereas A2, similar, but here, we're requiring the number of 1's and 2's to be equal. And the number of 0's can be anything.

Now, you can easily make, I hope-- you should make sure you can-- pushdown automata that can recognize A1 and A2, because let's just take A1. The pushdown automaton can push the 0's as it's reading them, pop them as it's reading the 1's to match them off and make sure that they're the same number of them. And then the 2's, it doesn't care how many there are. It just has to make sure that there are no strings, there are no letters coming out of order. But any number of 2's is fine. So you can easily make a pushdown automaton recognizing A1, similarly for A2.

So what can we conclude from that? Here are the three possibilities. Let me-- so look at that, the class of context-free languages is not closed under intersection. You can read it.

So I want to pull up the poll and launch that. Please fill that out. 10 seconds. Again, just, if you don't know the answer, just give any answers so that-- because we're not counting correctness.

There is still a few dribbling in. OK, five seconds. OK, end polling. Most of you got that right.

I don't know. Is it OK to share these things? I don't want to make people who didn't get the right feel bad. You know, but you should understand, I think if you're missing something, you should understand what you're missing.

The pumping lemma shows that A1 union A2 is not a context-free language? No. As I mentioned at the beginning, the context-free languages are closed under union. So the pumping lemma had better not show that these-- we already know that these two languages are context free, because we get them from pushdown automaton.

And we said at the beginning that context-free language is closed under union. So we know that these two are context free. So the pumping lemma better not show that they're not context free. Something would be terribly-- have gone terribly wrong if that were true.

And also we know also from a little bit of further reasoning that the context-free languages is not closed under complement by what we've already discussed, because they are closed under union. And as I pointed out, they're not closed under intersection. And so if they were closed under complement, De Morgan's Laws would say that closure under union and closure under complement would give you closure under intersection. But we don't have closure under intersection. So in fact, they're not closed under complement.

OK, so in fact, this does show us that the class of context-free language is not closed under intersection, because the intersection of A1 and A2, two context-free languages, is B. And B is not context free. So it shows that this is-- the closure under intersection does not hold.

So let us continue, then. We have one more example. Then we'll take a break. So the pumping lemma for context-free languages, again, here is the second example.

Here is the language F. We have actually seen this before. ww, two copies of a string, two copies of any string-- and we're going to show that's not a context-free language.

Assume that it is context free, the pumping lemma gives pumping length. Now, here you have to do a little bit more work. Often, the challenge in applying the pumping lemma in either case that we've seen involves choosing that string that you need to pump, that you're going to pump. So you have to choose s in F, which is longer than p, which s to go with.

So you might try this one, first glance. Here is a string that's in the language, because it's two copies of the string 0 to the p1 0 to-- and then 0 to the p1. So that's in the language, but it's a bad choice. Before I get ahead of myself, let's draw a picture of s, which I think is always helpful to see.

So here is runs of 0's and then a 1, runs of 0's and then a 1. Why is this a bad choice? Because you can pump that string and you remain in the language. There is a way to cut that string up and you'll stay in the language.

And the way to cut it up is to let the x be just that substring which is just the 1. And the v and y can be a couple of 0's or a single 0 on either side of that 1. And now that's going to be a small vxy.

But if you repeat v and y, you're going to stay in the language, because you'll just be adding 0's here. You'll be adding same number of 0's there. And then you're going to have a string which still looks like ww. And you'll still be in the language.

So that means that cutting it up doesn't get you out of the language under pumping. And the fact is that that's a bad choice for s, because there is that way of cutting it up. So you have to show there's no way-- you don't get to pick the way to cut it up.

You have to show that there is no way to cut it up in order to violate the pumping lemma. So if instead you use the string 0 to the p, 1 to the p, 0 to the p, 1 to the p-- so this is 0's followed by 1's followed by 0's followed by 1's all the same number of them-- that can't be pumped satisfying the three conditions. And just going through that-- now if you try to break it up, you're going to lose. Or the lemma is going to lose. You're going to be happy, but the lemma is not going to be happy, because it's not going-- it's going to violate the condition.

Condition three says vxy is not-- doesn't span too much, and in fact, can't span two runs of 0's or two runs of 1's. It's just not big enough, because they're more than p things-- they're p things apart. And this one string, this string vxy is only p long.

And so therefore, if you repeat v and y, you're going to have two runs of 0's or two 1's that have unequal length. And now that's not going to be the form ww. You're going to be out of the language. So I hope that's-- you've got a little practice with that.

I think we're at our break. And I will see you back here in five minutes, if I can get my timer launched here. OK, so see you soon. This is a good time, by the way, to message me or the TAs. And I'll try to be looking for if you have any questions.

In the pumping lemma, can x-- yeah, x can be epsilon in the pumping lemma. x can be epsilon. y can be epsilon, but x and y cannot both be epsilon, because then, when you pump, you'll get nothing new. Technically, v and y can include both 0's and 1's. Yeah, v and y can include both 0's and 1's.

So let me try to put that back, if that's will-- so v and y can have both 0's and 1's, but they can't have 0's from two different blocks. And you can't have 1's from two different blocks. So what's going to happen is either you're going to get things out of order when you repeat-- like, a v has both 0's and 1's in it.

When you repeat v, you're going to have 0's and 1's, and 0's and 1's, and 0's and 1's. That's clearly out of the language, so that's no good. Your only hope is to have v to be sticking only inside the 0's and y to be sticking only inside 0's or only inside 1's.

But now, if you repeat that and just look at what you're going to get, you're going to have a string which is going to be-- if you try to cut that string in half, it's not going to be of the right form. It's not going to be two copies of the same string, because it's going to have a run of 0's followed by a longer or shorter run of 0's, or a run of 1's followed by another run of 1's of unequal length. So there is no way this can be two strings, two copies of the same string, because that's what you required. F has to be two copies of the same string to be in the language.

OK, let me just see where-- we're running out of time here. Let me just put my timer here. We've only got 30 seconds.

And I'm sorry I'm not getting to answer all the questions here. OK, we are done with our break. It's going to come back.

And now we're shifting gears in a major way, because in a sense, everything we've done so far has been kind of a warm up. These limited computational models really are kind of helping us to set our understanding of automata and the definitions and the notation. And they're also going to be helpful in providing examples later on in the term.

But really, in terms of a model of computation, they don't cut it, because they cannot do very simple things that we normally think of a computer as being able to do. So here we're introducing another model of computation, called the Turing machine. And that's really going to be the model of what we're going to stick with for the rest of the semester, because that's going to be our model of a general-purpose computer, the way you normally think about it. So let's-- we'll spend a little time introducing it. And then we we'll continue that discussion next time.

So in terms of a schematic, actually, the Turing machine model is pretty simple. It's going to have states and all that stuff. So there is going to be a finite control here, which is going to include states and a transition function, as we'll describe in a minute. The point is that it's going to have the input appearing on a tape.

The key difference now is that the machine is going to be able to change the symbols on the tape. And so we think of the machine as being able to write as well as read the tape. So that's really the key feature of a Turing machine, is the ability to write on the tape. Everything else, in a sense, follows from that, and a few other differences. But so the fact that the head can read and write so that we can use the tape as storage much as we use the stack of storage, but it's not limited in the way we can access it the way a stack is-- so we kind of have very flexible access of the information on the tape.

Now, being able to write on the tape doesn't do any good if you can't go back and read what you've written later on. So we're going to make the head to be able to be two way. So the head can move left to right as before, but it can also move back left. And that's going to be under control of the transition function, so under program control, essentially.

The tape is going to be-- oops, sorry. The tape is infinite to the right. And so we're not going to limit how much storage the machine can have. So the tape is going to-- we'll think of as having, instead of just having the input on it, it's going to have the input.

But then the rest, it's going to have infinitely many blanks, blank symbols following the input. So the tape is infinite in the right-hand direction. And so there is infinitely many blanks. I'm going to use that symbol for the blank to follow the input.

You can accept or reject. Oh yeah, so that's another thing that's important. Normally, we think of-- in the previous machines, finite automata, pushdown automata, when you got to the end of the input, that's when the acceptance or rejection was decided. If you were going to accept it at the end of the input, then you accepted.

But you have to be in that location at the end of the input in order for that to take effect. That doesn't make any sense anymore, because the machine might go off beyond that, and still be computing, and come back and read the tape later on. So it only really makes sense to let the machine accept or reject upon entering the accept or reject state.

So we're going to have a special accept state and a special reject state, which is also a little different than before. And when the machine enters those states, then the machine-- then the action takes effect. The machine halts and then accepts or halts and then rejects. So we'll make that absolutely clear in the formal definition in a second, but just to get the spirit of it.

So I'm going to give you an example of the thing running. Sorry, me too-- again, my PowerPoint is having issues. OK, so here is a Turing machine recognizing that language b. Actually, I switched gears on you. Instead of 0's, 1's, and 2's, I made them a's, b's, and c's, but the same idea.

So I'm going to show you how the Turing machine operates. And then we'll give a formal definition. I hope that's on here. I think it is. In a second, but let's-- this is an informal discussion of how the machine is going to operate to do this language, a to the k, b to the k, c to the k, using its ability to write on the tape as well as read and move its head in both directions.

OK, so let me just first describe in English how this machine operates. And then we will see it in action on this little picture I have over here. So the way the machine is going to operate is the very first thing is the head is going to start here.

And the head is going to scan off to the right, making sure that the symbols appear in the correct order. So it's seeing that there are a's and b's and then c's, without checking the quantities, just that the order is correct. For that, you don't need to write. A finite automaton can check that the input is of the form a star, b star, c star.

So writing is not necessary. The machine, if it detects symbols out of order, it immediately rejects by going into a special reject state. Otherwise, it's going to return its head back to the left end.

And let me just show that here. So here is-- oh no. Before I illustrate it over here, let's go through the whole algorithm.

So the next thing that happens is you're going to scan right. And now you want to do the counting. So you're going to scan right again, but this time, you're going to make a bunch of passes over the input, a bunch of scans. And each time you make a scan, you're going to cross off one symbol of each type.

So you're going to cross off an a. You'll cross off a b. You'll cross off a c on a single scan. And then you repeat that, crossing off the next a, the next b, the next c. And you want to make sure that you've crossed off all of the symbols on the same run and not crossing off some symbols before other symbols, while other symbols will remain, because that would mean that the counts were not equal.

If you cross them off and they're all run out on the same scan, same pass, then we know that the numbers had to start off being equal. So I mean, this is a sort of baby stuff here, but I hope you get the idea. And we'll kind of illustrate it in a second.

If you have the last one of each symbol-- so what I mean by that is you just crossed off the last a, the last b, and the last c-- then that you originally had an equal number. And so you accept, because you're crossing off one of each on each scan. So if you cross off, on the last scan, each one of them gets crossed off, then you accept.

But if it was the last of some symbol but not of other symbols, so you crossed off the last a, but there were several b's remaining, then you started off with an unequal number of a's, b's, and c's. Then you can reject. Or if all symbols still remain after you have crossed them, one on each off, then you haven't done enough passes. And you're going to repeat from stage three and do that again another scan.

OK, so here is a little animation which shows this happening on this diagram. So here is the very first stage where you're scanning across, making sure things are in the right order. I didn't have to write on the tape. And now you're going to reset the head back to the beginning. This is, by the way, not the most efficient procedure for doing this.

Now we're going to do a scan crossing off a single a, a single b, and a single c. So here, I'm going to show that here, a single a, a single b, single c. And now as soon as you have crossed out that last c, we can return back to the beginning.

So scan right across-- so if all symbols remain, so there are still symbols remaining of each type, we're going to return to the left and repeat. Now we're getting another pass, single a, single b, single c get crossed off. Have we crossed them all off yet? No, there is-- of each type, there still are remaining ones.

So again, we return back to the beginning. Now we have a last pass, cross off the last a, the last b, the last c. The last one of each type was crossed off. So now we know we can accept, because the original string was in the language.

OK, so that's to give you at least some idea how the Turing machine can operate, more like the way you would think of a computer operating. Maybe it's very primitive. You could imagine counting also. And a Turing machine can count as well. But this is the simplest procedure that I can just describe for you without making it too complicated.

OK, so let's do a little checking on that. OK, so the way I'm describing this, how do you think? And in a sense, you don't quite know enough yet. But how do you think we're going to get this effect of crossing off with the Turing machine?

Are we going to get that by changing the model and adding that ability to cross off to the model? Are we going to use a tape alphabet that includes those crossed-off symbols among them? Or we'll just assume that all Turing machines come with an eraser and they can always cross off stuff. So what do you think is the nice way, sort of mathematically, to describe this ability to cross things off?

Yeah, again, most of you, again, I think are getting this. So there are, like, 10 laggards here. So please wrap it up so we can close the poll. Five seconds to go. OK, polling ending, get your last-- last call.

All right, share the results. So most of you got that right. All Turing machines come with the eraser-- I don't know. That was thrown in there as a joke, but it came in second. So don't feel bad if you got it, but that's not what I had in mind.

The way the Turing machine is going to be writing on the tape is to write a crossed-off symbol instead of the symbol that was originally there. So we're going to add these new crossed-off symbols. And that's going to be a common thing for us to do when we design Turing machines.

We're not going to get down to the implementation level for very long. We're going to very quickly shift to a higher level of discussion about the machines. But anyway, that's how you would do it if you were going to actually build a machine.

So let us then look at the formal definition. And personally, maybe I should have done that check in after the formal definition. That might have been clearer, but oh well.

OK, Turing-- here is the formal definition. This time, a Turing machine is a 7-tuple. And there is-- now here, we have sigma, which is the input alphabet. Gamma is the tape alphabet.

So now you're a little bit analogous to the stack from before where gamma was the stack alphabet. But these are the symbols that you're allowed to write on the tape-- that are allowed to be on the tape. So obviously, all of the input symbols are among the tape symbols, because they can appear on the tape. So you have sigma is a subset of gamma.

One thing I didn't mention here is that the input alphabet, we don't allow the blank symbol to be in the input alphabet, so that you can actually use the blank symbol as a delimiter for the end of the input, a marker for the end of the input. So in fact, and the blank symbol is always going to be in the tape alphabet. This is actually always going to be a proper subset because of the blank symbol. But we're just allowing-- it doesn't really matter. We're allowing the tape alphabet to have other symbols for convenience, so for example, these crossed-off symbols.

Now let's look at what the transition function, how that operates. So the transition function, remember, tells how the machine is actually doing its computation. And it says that, if you're in a certain state and the head is looking at a certain tape symbol, then you can go to a new state.

You write a new symbol at that location on the tape. And you can move the head either left or right. So that's how we get the effect of the head being able to be bi-directional.

And here is the writing on the tape. It comes up right here. So just an example here which says that, if we're in state two and the head is looking at an a currently on the tape, then we can move the state r. We change that a to a b. And we move the head right 1 square.

Now, this is important. When you give a certain input here to the Turing machine, it may compute around for a while, moving its head back and forth, as we were showing. And it may eventually halt by either entering the q accept state or the q reject state, which I didn't bring out here, but that's important. These are the accepting, rejecting, special states of the machine.

Or the machine may never enter one of those. It may just go on, and on, and on and never halt. We call that looping, a little bit of a misnomer, because looping implies some sort of a repetition. For us, looping just means not halting.

And so therefore, M has three possible outcomes for each input, this w. It might accept w by entering the accept state. It could reject w by entering the reject state, which means it's going to reject it by halting. Or we also say we can reject by looping. You can reject the string by running forever.

That's just the terminology that's common in the subject. So you either accept it by halting and accepting or rejecting it by either halting and rejecting or by just going forever. That's also considered to be rejecting, sort of rejecting in a sense by default. If you never actually have accepted it, then it's going to be rejected.

OK, check in three here-- all right, so now our last check in for the day, we say, this Turing machine model is deterministic. I'm just saying that. But if you look at the way we set it up, if you've been following the formal definition so far, you would understand why it's deterministic.

So let's just, as a way of checking that, how would we change this definition? Because we will look at the next lecture at non-deterministic Turing machines. So a little bit of a lead in to that, how would we change this definition to make it a non-deterministic Turing machine? Which of those three options would we use? So here, I'll launch that poll.

I've got about 10 people left. Let's give them another 10 seconds. OK, I think that's everybody who has answered it from before. So here, I think you pretty much almost all of you got the right idea.

It is B, in fact, because when we have the power set symbol here, that means there might be several-- there is a subset of possibilities. So that indicates several different ways to go. And that's the essence of non-determinism.

OK, so I think we're-- whoops. All right, so look, this is also kind of setting us up for next lecture and where we're going to be going with this. So these are basically two in a-- well, two or three important definitions here.

One is-- we talked about the regular languages from finite automata. We talked about the context-free languages from the grammars and the pushdown automata. What are the languages that the Turing machines can do? Those are called, in this course, anyway, Turing-recognizable languages, or T recognizable. Those are the languages that the Turing machine can recognize.

And so just to make sure we were on the same page on this, the language of the machine is the collection of strings that the machine accepts. So the things that are not in the language are the things that are rejected either by looping or by halting and rejecting. So only the ones that are accepted is the language.

Every machine has just a single language. It's the language of all strings that that machine accepts. And we'll say that and recognize that language, if that language is the collection of such strings that are accepted. And we will call that language a Turing-recognizable language, if there is some Turing machine that can recognize it.

Now, this feature of being able to reject by running forever is a little bit weird, perhaps. And from the standpoint of practicality, it's more convenient if the machine always makes a decision to accept or reject in finite time and doesn't just reject by going forever. And so we're going to bring out a special class of Turing machines that have that feature, that they always halt. The halting states, by the way-- maybe it didn't say this explicitly-- are the q accept and the q reject states. The accept and reject states are the halting states.

So if the machine halts, that means it ends up in one of those two. So it has made a decision of accepting or rejecting at the point at which it has halted. So we'll say a machine is a decider if it always halts on every input. So for every w fed in, the machine is eventually going to come to a q accept or a q reject. We call such a machine a decider.

And now we're going to say, a language is-- so we'll say that the machine decides a language if it's the language of the machine, so the collection of accepted strings, and the machine is the decider. We'll say that, instead of just recognizing the language, we'll say that it decides the language. And the Turing-decidable language is a language that the machine-- of all strings the machine accepts for some Turing machine which is a decider, which is a Turing machine that always halts.

So if a Turing machine may sometimes reject by looping, then it's only recognizing its language. If the Turing machine is always halting, so it's always rejecting by explicitly coming to a reject state and halting, then we'll say it's actually deciding the language. So then, in a sense, that's better. And we're going to distinguish between those two, because they're not the same.

There are some languages which can be recognized, but not decided. And so in fact, we're going to get the following picture here, that the Turing-recognizable languages are a proper subset. They include all of-- everything that's decidable, certainly is going to be recognizable, because being a decider is an additional restriction to impose, an additional requirement.

So everything that's decidable is going to be automatically recognizable. But there are things which are recognizable which are not decidable, as we'll see. I'll actually give an example of that, but not prove it next lecture.

And just for, just to complete out this picture, I'm going to also point out-- we haven't proven this yet, but we will prove it-- that the decidable languages also include all the context-free languages, which, in turn, include the regular languages, as was already seen. So we haven't shown this inclusion yet.

But actually, this is the picture that we get. So there is actually a hierarchy of containments here. Regular languages are a subset of the context-free languages, which are, in turn, a subset of the decidable languages, which in turn, are a subset of the Turing-recognizable languages.

And so with that, I think we're going to move to our little bit of a review of what we've done today. So we proved that pumping lemma as a tool for showing that languages are not context-free languages. We defined Turing machines, which is going to be our model that we're going to be focusing on for the rest of the term, not forgetting the other models, because they're going to be useful examples for us. And we defined Turing deciders, Turing machine deciders that halt on all inputs.

OK, so I think, with that, we have come to the end of today's lecture. I will stick around a little bit and answer questions in the chat. I will try to share them with everybody as I'm answering them so I'm not just talking to one person.

How is the concept applied in-- so I'm getting one question about the practicality of all this. Bunches of questions are coming in. So look, is this stuff all practical?

I would say, yes and no. I don't know which concept you have in mind. We're going to introduce lots of concepts in this course.

And the concept of the finite automata, and the pushdown automata, and context-free languages, definitely used in other subjects, in other fields in computer science and elsewhere-- these are very basic and fundamental notions. And so yes, and Turing machines-- well, I mean that's a model of a general computer. If you want to understand computation, you're going to need to understand some model.

And a Turing machine is a particularly simple model. And that's why we use it. As it turns out, it doesn't really matter what model you use, but we'll talk about that next time.

But yeah, I would say there is a lot of applied material in this course, as time has shown, whether it's led to things like public key cryptography, which is used on the internet, or understanding various algorithms. I mean, that's not the reason I study it. I study it because I'm more coming at it from more of a mathematical perspective. I just find the material very beautiful, and interesting, and challenging, but it does have applications.

Any other questions here? I think I'm going to sign off, then, to get myself set up for my office hours, which is on a different Zoom link. OK, so thank you, everybody. And see you on Thursday. Bye bye.