[SQUEAKING] [RUSTLING] [CLICKING]

MICHAEL SIPSER: OK, why don't we begin. Hi, everybody. Let's see how many we got here, most of you. I'm sure the others will show up, hopefully, soon enough. So welcome back. We have today's lecture 4.

And let's just remember what we've been doing. In the last few lectures, we were exploring the regular languages, as described by finite automata and regular expressions. We showed how to convert them back and forth, those two models, to one another. And we also showed how to prove certain languages are not regular.

Now remember, finite automata are a very weak model of computation. They only have a limited memory, finite memory. And they still aren't able to do certain things with their finite memory, but they are-- if you can pair them with a general purpose computer, at least the way we think about it, is their capabilities are just extremely limited.

And so we're going, to over the next few lectures, explore some more powerful models. We started doing that last time, the context-free grammars. And as we'll see, there are certain things that you can do. Well, I think we saw that last time as well. There are some things you can do with context-free grammars that you cannot do with finite automata. But they still have their limitations, as we'll see.

So today what we're going to do, we're going to continue that discussion by looking at the definition of context-free grammars in a more formal way. One of the things that we do in this course is develop a practice with formalism, so that's going to be in the spirit of that. We also are going to look at their associated languages, called the context-free languages. So they are going to be the counterpart for context-free grammars of what the regular languages are for the finite automata regular expressions.

And then we're going to look at an automaton-based model, which is the counterpart to the grammars called the pushdown automata. And we'll see that those are equivalent in power. And finally-- well, and as part of that, we will show how to convert the context-free grammars to the pushdown automata. And that's what we're going to do today.

So we're going to move on then and return to our topic of context-free grammars that we began last time. And just to refresh your memory, so here was that example of a context-free grammar that we gave last time. And it has-- the way we're going to be writing context-free grammars is using a bit of a shorthand, which looks like this.

When you have multiple rules that have the same variable on the left-hand side, you can combine them into one line. So these two rules over here, S goes to 0S1 and S goes to R, can be written in one line as a little bit more compact way, this is standard, as S goes to 0S1 or R. That's the way you would read this. This is really two rules but written on one line.

So as you recall from last time, a context-free grammar has terminals, variables, and rules. Those are the parts that we speak of, as well as one of the variables being designated as a starting variable, which gets the whole thing going. So I'll remind you about how the computation goes.

So the variables are the symbols that appear in the left-hand side of the rules. The terminals are the other symbols that appear in the grammar. And the-- we take the grammar, and we use it to generate strings, according to a certain system.

And the system is that you start up by writing down the starting variable. And then once you've written down that variable, or whatever variables you have written down, you're allowed to substitute them according to the rules, the substitution rules, that are in the grammar. So you can keep on replacing the variables that you have with the corresponding right-hand sides. And then you do that over and over again until you don't have any variables left, only terminal symbols remain. And at that point, you have generated a string that's in the language of the grammar.

So the grammar's language is going to be a language over strings whose alphabet are the terminal symbols. So the terminal symbols in a certain sense play the same role as the input alphabet, say, for the finite automata. The variables are internal-working symbols for the grammar. The terminals are the symbols of which the language is written. We'll make that more precise in a minute when I give the formal definition.

So the result is the generated string. And the language of the grammar is the language of all generated strings that you can get using that grammar. And the important thing is that we call that language a context-free language. So that's like what we get from-- that's the analogous thing to the regular languages, but here we call them context-free languages, the things that you can get from a context-free grammar.

Again, just a quick recap of that example we did last time. So you start out by writing the start variable. And then I'm going to give you two views of that, either in terms of the tree of substitutions, which we call the parse tree, or in terms of the resulting string as you do the substitutions. So here is the parse tree, here is the resulting strings, here are the substitutions that you make.

And now we have R coming from S, and we have 00R11. And now we have R in turn becomes an empty string. And then the string that we generated is 0011. That's in the language of the grammar. And now if you play with this a little bit, you'll see that the language of the grammar is all strings that look like runs of zeros followed by runs of ones.

So is that clear? I think we're going to have a-- I think the next slide is going to have a check-in, and so hopefully that'll get us all together on the same page with this. Anyway.

So here's our formal definition anyway. We have a context-free grammar is a 4-tuple. There are four parts to a context-free grammar. These are the parts we've already been discussing, the variables, the terminal symbols, the rules.

The rules are always that they form a variable followed with an arrow to a string of variables and terminals. That's the way we just write that down. So this is the form of the rule, and then we have the special start variable. And we all wrap that up into a package, this 4-tuple. That's what the context-free grammar is.

Now we have here-- and now, maybe a little bit overkill, but let's talk about, formally speaking, the way the grammar actually processes and produces strings. So we're going to write-- the standard notation for this is that if you have two strings of variables and terminals-- so imagine you have an intermediate string that you've generated in the grammar so far, which might be like 00S11 from the previous line, so that's an intermediate string that is so far what you've generated, you're going to say maybe that's u, and v might be the next line down.

So that means we're going to write u arrow v-- and that arrow is-- the word we're going to use is "yields." We'll say u yields v if it can go from u to v just with one substitution step. And then we'll write u yields v in some number of steps-- or actually, we say u derives v if it can go to u to v with some number of substitutions instead of just one. And that's used with the yields arrow with the star above it. It means "some number of."

Another way of writing that is you can say, u goes to v if there are a bunch of one-step moves that you can make which take you from u to v. And that whole sequence is called a derivation of v from u. That's a sequence of steps that you go through doing these substitutions one by one to take you from u to v, according to the rules of the grammar.

And lastly, if u is a starting variable, then we call that sequence just the derivation of v. It could be the derivation from the start variable, but that's the assumed if you don't say it's a derivation from anything. The derivation of v in the grammar is the derivation of v from the start variable. It's just the sequence of substitutions that you make, kind of what I think you would expect.

Now, the language of the grammar is the set of all strings of terminal symbols that you can get from starting at the starting variable of the grammar. And that's called a context-free language, as I mentioned before. So it's a context-free language. It's the language of the grammar for some grammar.

So let's have a little check in here, again, nothing too hard, nothing to get worried about anyway. We're not counting correct here. So let's just see, I'm going to give you two things that look like grammars. Which of them are actually grammars?

And let me just pull that poll up here. So which of these are valid grammars here? Are they both? Neither? I mean, you could kind of make an argument either way for both of them. But both of them are a little-- have their own-- a bit of weirdness to them in a way if you study them for a second.

That's pretty much converged. Share the results. So, in fact, the correct answer is b. And why is only C2-- well, first of all-- well, what's wrong with C1? The problem with C1 is that the rules have things besides a single variable on the left-hand side. So having a B1 on the left-hand side is not legal in a context-free grammar.

In fact, there are other kinds of grammars. There's a kind of grammar called the context-sensitive grammar. The term "context-free" means you can replace the variable independent of its context in the intermediate string, so independent of what's around it.

But here, this substitution is going to-- you can replace B, but it depends on there being a 1 next to it. This is called a context-sensitive grammar, but it's not the kind of grammar we're going to be using, which are only context-free grammars. So C1 is out. That's not a legit context-free grammar.

C2, the thing that's a little weird about C2 is if you try to generate a string in C2, you'll see that there's no way to get rid of the variables, that you're always going to be stuck with a variable. Now, that doesn't violate the definition of a context-free grammar. So this is a context-free grammar, but it's not going to be able to generate any strings of only terminals.

So this is a context-free grammar whose language happens to be the empty language, but that's totally OK. So the correct answer here is B, that only C2 here is a valid context-free grammar.

Common-- let's just see. Common question, does a string u derive itself? Yes, a string u derives itself. That's a little bit of a-- little bit of an esoteric question there for us right now, but yes.

A string u-- in this definition here, u arrow star u is legit-- is legal. Maybe it isn't according to the way I've written it down here, but it is a legal thing. It's not going to really matter for you anyway, but it is legal.

OK, let's continue. Let's do another somewhat interesting example of a context-free grammar. This is a grammar that can generate arithmetical expressions involving pluses and times.

So here it is. It has how many rules? Well, there are six rules here. Each line represents two rules. So E goes to E plus T or T; T goes to T times F or F; and F goes to parentheses E parentheses or a.

And so the variables are going to be the symbols that appear on the left-hand side, E, T, and F. The terminal symbols, which are going to be the symbols of the language that you're going to be generating, is going to be the plus, the time symbols. The parentheses are just terminal symbols here. So they're not playing any special role besides that.

And then you have the a, which is representing the upper end on which those operators would be working if there was actually an expression you would use. But they're just symbols from the perspective of the grammar. And lastly, the start variable is going to be, as normally appears, on the upper left-hand side of the grammar, in terms of the way you write it down. So sometimes, you might specify a different start variable, but otherwise-- if it's not specified, it's the one in this corner here.

So let's just see some examples of using the grammar to generate a string. So here is a string in the language, a plus a times a. And this example will reveal some other interesting features of the grammar, but let's just see it in operation first. So, again, I'll try to write it to you in both ways in terms of the parse tree and the resulting string as you're doing the substitutions.

So the-- so first we start with the E, then we substitute E plus T. And we see the resulting string's E plus T. But now as we're doing additional substitutions, the resulting string that you get is going to evolve accordingly.

And so I hope it comes across that this tree here picture on the left shows you the structure of the various substitutions, whereas on the right, it just shows you the strings that you get as a result of those substitutions. So now you can generate this particular string, which is now in the language of this grammar. You can generate all sorts of other strings as well, such as parentheses a plus a parentheses times a and so on.

And in fact, this might be a piece of a programming language that you're trying to describe. And one application of context-free grammars is to describe the syntax of programming languages. What are the legal programs that you can write in that programming language?

And not only that, the grammar can be used to automatically generate the part of the compiler for that programming language, which will interpret the-- which will interpret the structure of the input, the so-called parser, which will figure out the meaning of the input to the compiler so that the compiler then can generate the code, or if it's an interpreter, it can interpret the resulting code that you've given it. But the very first step in both of those is to figure out the meaning. And the meaning is embedded within the structure of the parse tree.

Now in the case of this particular tree, just to give you some sense of what meaning I have in mind, this parse tree, due to the structure of this grammar, has the precedents for times over plus. So normally when we write down a plus a times a, you assume you're going to do the multiplication before you do the addition, even though it appears second. That's just the way we tend to write things.

And this grammar has grouped it that way for you. It groups the times lower down in the tree than the plus. So the times is going to be done before the plus, if you imagine doing this in terms of the way the tree is guiding you.

So the tree, as you can see, has a certain amount of meaning built into it. Now, we're not actually going to be using that in this course, but I just want to describe that as an illustration of how this material can get applied. So here is what I'm saying, that the tree contains additional information.

Now, that's also relevant if you happen to have a grammar which might allow multiple parse trees for the same string. That can happen. And this particular grammar does not allow that, but you might write other grammars, as we'll see in a minute, that could generate the same string in multiple ways with multiple different parse trees.

Now that might be undesirable when you have a programming language because typically you want it to be only a single meaning for your code. You don't want it to be ambiguous and have multiple meanings. But ambiguity does occur, and it's not necessarily something we're always going to see as a bad thing.

So I think as I mentioned last time, a lot of this subject originated with linguistics, and that's where the terminology comes from, grammar, and languages, and so on. The terminology for the subject really comes out of linguistics. In fact, one of the key players for that is an emeritus faculty member at MIT, Noam Chomsky. He was instrumental in setting a lot of this stuff up.

But the-- you can think of grammars as applying to natural human languages as well. So let me give you a little example as a pop-up. It's not directly a pop-up-- a check-in, not directly relevant to the material I just presented, but just a curiosity.

If you take the English sentence "The boy saw the girl with the mirror," does that-- is there only one natural interpretation for that sentence, or are there perhaps other natural interpretations for that sentence? So let me pose that to you as another poll here.

And so I ask you to think about how many different meanings you might find for a-- reasonable different meanings-- I mean, you can-- you're going to go wild, you can think of zillions of meanings. But I think in terms of reasonable meanings, how many meanings might you get for the sentence? People are seeing more meanings than I'm seeing, but that's OK.

So this is a quick-- why don't we just give this another 10 seconds here. And then most of you are in agreement with me. I can see here that you are seeing that there were two meanings.

The two meanings that I see here for this sentence are, when you say the boy saw the girl with the mirror is, who has the mirror? Is it the boy seeing the girl through the mirror, or is it the girl that has the mirror and the boy just happens to see her? So two very different meanings for the same sentence.

And that's in the nature of English is just the way-- it's an ambiguous structure there. And often we resolve that ambiguity in English with other information that we might have. But typically you don't want there to be ambiguity when you have a grammar, say, describing a programming language.

So let's continue on that. So talking a little bit more about ambiguity, I promised you an example where you might have an ambiguous grammar. So if you take these two grammars, G2 and G3, and G2 from the last slide and G3 is a similar grammar, in fact, it's the grammar that has the very same language-- that gives you the very same language.

So L of G2 equals L of G3. Both of them are describing these arithmetical expressions. But whereas G2 has a unique parse tree for every string that you generate, G3 can have multiple parse trees for the same string. So I'm just going to illustrate that here.

So here is the same string that we generated last time, a plus a times a. In G3, the parse tree is actually even simpler here. So here I'm showing you the-- there's just the two substituents that you need to make starting from E. And then to get the string a plus a times a, it's a simpler grammar in a sense. But there's another parse tree that'll give you the same result. And I've written that down below here upside down.

So the upper parse tree groups the times before the plus, more inside than the plus. But the lower parse tree doesn't have that precedence built into it and can alternatively interpret the plus as being of higher precedence than the times. And so in that sense, we have here a grammar which has two interpretations for this same string. And we call that-- whoops, we call that an ambiguous derivation, an ambiguously-derived string. And the grammar itself is called an ambiguous grammar.

So let us continue on from that. By the way, I think there's a question here that came in. Like, for example, a plus a, is that ambiguous in G2? No. If you try to apply it, you'll see the way that G2 can produce a plus a-- a plus a plus a is going to group the first two and then the second one-- then the last one. You can't derive things in multiple ways. Addition is associative, but the grammar doesn't-- it doesn't-- the grammar-- for the grammar, it's going to have a prescribed order for the way things get interpreted there. So that's ambiguity.

So let's introduce pushdown automata, which is going to be our automata counterpart for context-free languages. So the way I'm going to introduce pushdown automata, sort of shifting gears here now, is by first giving a new view of finite automata. Remember before when we presented a finite automaton, we gave it in terms of a state diagram, which I've shown here in miniature form on the picture.

We could do that for pushdown automata, but the picture tends to be very complicated. So I'm going to take a bit of a higher level description for pushdown automata, which is I'm calling a schematic view or a schematic diagram. And there I'm really not going to be showing you the individual states, but I'm going to be showing you the individual components of the machine, more of an abstraction-- from a more abstract perspective.

And so from that perspective, a finite automaton has here what I'm going to call the "finite control." So I'm going to be suppressing the details of the states in this picture. I'm going to represent those states as the control of the DFA or the NFA. They're really going to be the same from this pictorial point of view.

The input is going to appear as a string that's written down on what we're calling a "tape." Again, this is somewhat of an anachronistic terminology. Back in my days, people actually did feed their inputs into computers on a tape sometimes.

We don't do that so much anymore, but that terminology has stuck, and it's going to be a persisting later on in the course, too, so you might as well get used to it. So the input is going to appear on a tape, or sometimes callec an input tape.

And the way the machine actually will read that input-- whoops-- it's going to have a head, which is going to be starting at the left side and moving from left to right, reading the symbols that appear on the input tape one by one. So this is our picture of a finite automaton, just redone from last time, just a different way of picturing it.

Now, that's going to be setting the stage for the picture of a pushdown automaton, because a pushdown automaton is like a finite automaton, but it has an extra feature, it has an extra device attached to it. And that's called a "stack." So here's the schematic diagram for a pushdown automaton, and that's going to be a stack, which is going to be basically a form of auxiliary storage.

Now remember, part of the limitation for a finite automaton was that we had a limited amount of memory. So we were not able to do some very simple things, like counting, because we had a limited memory. So the pushdown automaton is going to be able to use its stack as a kind of unbounded memory, but a memory that's restricted in the way it can be used. So it's unlimited but still restricted, as we'll see.

So the way the pushdown automaton uses its extra memory on what we're calling the stack or a pushdown stack is that you can write symbols instead of only reading symbols. But those symbols can only be read at the very-- written or read at the very top of this list of symbols. And every time you add a new symbol, the other symbols that are already there get pushed down, so hence the name.

People also often refer to it as a stack of plates in a cafeteria, if you've ever seen those things, or you can remember back to the days when we went to cafeteria, which are getting further and further away. But even at the cafeteria, you had a stack of plates. And as you remove plates from them, they were on a spring, and they kept coming up. Or if you add more, they would go down.

And it's the same idea. Imagine these symbols here are on a spring, and the more symbols you add, the more they go down. Or if you move them, and read them, and remove them, then they move back up.

So a pushdown automaton operates like a finite-- like a non-deterministic finite automaton, as we'll see. Pushdown automata for us are always going to be allowed to be non-deterministic. So we're not going to be studying the pushdown automata that are restricted to be only deterministic. I'll say more about that in a second.

But they operate like an NFA, except they can write or read symbols from the top of the stack. And when they write, they're adding the symbol on, pushing down that stack. And when they're reading, they're removing symbols from the stack, and thereby lifting up the stack.

We give them special names. So those of you who have seen stacks already, this is, I'm sure, old hat for you. But I'm sure not everyone have seen stacks before.

So the special name for writing onto a stack is called a "push operation," so that you're pushing a new symbol down on the top of the stack, and it pushes everything down. Whereas when you're reading a symbol and removing it from the top of the stack, that's called a "pop." So that's reading and removing.

And we always think of those as going together. Writing and adding and reading and removing are combined. I mean, you might wonder, well, can't I just read it and leave it alone and not just remove it?

No. You can get that effect by reading it, which removes it, and then putting it back if you really want it to stay there. But the way we're setting it up is that reading comes with removing, writing comes with adding. Again, they're called pushing and popping.

So let's do an example. So we have here a pushdown automaton for a language we'll call D. We've seen that language before. It was-- actually, we used the same letter for it, strings of zeros followed by ones, where the numbers were the same of the two, 0 to the k 1 to the k.

We couldn't do that with a finite automaton. We will be able to do that with a pushdown automaton. And here I-- I thought I wrote down the input here, but OK.

So the basic idea is I'm going to give you an input now, and the pushdown automaton is supposed to test whether that input is in the language, whether it's of this form. Now it has the ability to use the stack because it's going to have to count how many zeros it has.

And so the way it's going to do it is it'll have a bunch of zeros, hopefully, and then a bunch of ones, and you want to see they are of the same number. It's going to take the zeros and store them on the stack until it sees a one. And then it's going to start to read the ones, and it's going to remove the zeros, matching them off one to one with the ones that it's seeing.

So you initially first read the zeros and push them onto the stack until you read a one, and then you read the ones while popping zeros from the stack. And you enter the accept state if the stack is empty. Just like with a finite automaton, entering the accept state only counts when you're at the end of the input.

So without even me needing to say anything, it's really saying you would enter the accept state if the stack is empty at the end of the input string. But that's implicit because it only takes effect at the end of the input string. If you enter an accept state alone in the middle somewhere, it doesn't matter. It doesn't affect anything.

And with that we're going to take a little break. And then we will be back shortly to look at pushdown automata again in a more-- with a more formal definition. Let me put that it's going to be five minutes. Let me see if I can figure out how to get my timer screen up here. Yes. And we will-- when the candle burns down to nothing, we will return and continue.

OK, our candle has burned down and has gone out. I never actually watched and see what would happened at the end. So we're good to go. Let's continue. Good. And let me put myself back in there.

All righty. So we were doing pushdown automata. And we just did that example of 0 to the k, 1 to the k. Now that you have a stack, we can do all sorts of fancy things that finite automata could not do just with their limited memory. So let's take a look at how we define pushdown automata.

So now pushdown automata is actually going to be a 6-tuple. So it's a little bit-- got some fancier stuff here to deal with, not too much, but a little bit. And so it has-- let's look at these a little bit more carefully since there's some novelty here. We have the input alphabet, just as we had before, sigma, but we also have gamma, which is the alphabet for using the stack.

Now, you might ask, why don't we just use the same alphabet? Well, it's really a matter of convenience that we would like to be able to have other symbols that could include the input alphabet, but could include other things as well. So it just gives you more flexibility in terms of what you're going to be writing on the stack.

The transition function, more complicated. So I think-- I don't know if I'm going to even say what the other things are, but these are the accepting states, this is the starting state. So that's the same as before. But the transition function is a much different animal here in a pushdown automaton. So let's just try to unpack that and understand what it's saying.

So the transition function tells us how the machine operates, how it goes from state to state, how it's going to read the input, how it reads it from the stack, and what am I to write on the stack, too, because that's going to all happen under program control.

So what this means here is that when the machine is in a particular state, reading a particular input symbol-- let's ignore the empty string subscript for the moment. So it's in a particular state reading a particular input symbol and with a certain stack symbol appearing at the top of the stack. So that's all information that's available to the controller of this pushdown automaton, the transition function, the current state, the next input symbol, and the symbol at the top of the stack.

And once we have that, we know what new state we can go into and what new symbol we can write on the top of the stack. So that's what the right-hand side of this function specification means. So this is where the input to the function, this is going to be the output of the function, state and new symbol to appear on the stack. So this is the popping symbol, this is the pushing symbol.

So now there are two things that bear explanation here first of all. Now this is a power set. So this is going to be representing, as we did before, a non-deterministic machine. We may have several possibilities, and we're going to represent that as a set of possibilities for the machine that it could go to at any point. I will give an example of how a pushdown automaton uses its non-determinism in a minute.

The other thing is these epsilons. So we have to understand why they are there. And we remember we had them for the NFAs, corresponding to when the NFA had an epsilon transition, the empty transition. So it could go along in that transition without reading any input. So this is going to play the same role here.

So if you have-- instead of an input symbol from sigma appearing in this part of the-- for the transition function, instead you have an epsilon appearing. That means that the transition-- that move of the machine can happen without reading any input symbol, just like for the NFAs. Or if you have an epsilon appearing for the stack symbol, that means you can make that transition without reading any stack symbol. So whatever is sitting on the top of the stack, it doesn't matter, the machine can make that move.

And it won't read anything either, or not going to pop anything. It's just going to be proceeding without looking at the stack at all. Or it might have both of them, in which case, it's going to go from one state to another state without looking at the input or at the top of the stack. So that's what the possibility of epsilon means for the transition function in those places.

The epsilon appearing over here means something a little different, but very similar. What that means is that we won't write anything on the top of the stack. That's going to be-- we will go to a new state, but without doing any writing. So we'll leave the stack alone.

So here it means we're not going to read anything if it's in this position, and this position means we're not going to write anything. So all of those things are valid and legal from the perspective of constructing a pushdown automaton.

And I've illustrated here, just with a little bit of an example, if you have delta that applies to some state q, reading an input symbol a and popping a c from the top of the stack, then you might have, let's say, in this case, two possibilities that you might end up going to. You might end up going to states r1 or to state r2. And in the former case, you'll end up writing a d, pushing a d onto the top of the stack. And in the latter case, you would be pushing an e onto the top of the stack.

So this is I'm trying to help you look at this notation. I hope that this is clear to you. I'm sure for some of you it's too slow, but others of you I'm trying to help along.

But if you're really struggling with this notation at this point, you're going to have to dig in and make sure you follow it. It's only going to get harder from now. I'm going to stop going over these kinds of points.

And if you're still struggling, you can't get it, this is not the right class for you, I'll be honest. So we're just going to be taking off like-- we're going to start to accelerate fairly quickly.

So it's a non-deterministic machine. We accept, like we did before. There might be several different threads of the computation. You're going to end up accepting if some of the threads, at least one of the threads ends up in an accept state at the end of the input string. That's when the machine overall accepts. It's just the way we normally think of non-determinism.

Again, you can use the models that we had before in terms of guessing or parallelism, whatever works for you. And sometimes different things work in different-- at different occasions. But that's how non-determinism works. We'll do an example of it now.

Here is a pushdown automaton for a different language we haven't seen before, I don't think, well, maybe we have, which is going to be using it's non-determinism in an essential way. This is a language where non-determinism is going to be critical. Without it, you can't-- a deterministic pushdown automaton, which is something, by the way, that people study.

And there's a section of my book about that, it's section 2.4, because it has relevance to applications. We're not going to address that in this course. So you can just skip section 2.4. It's pretty technical, I'll have to say, but still quite interesting and beautiful if you like that stuff. But it's technical. We won't do it.

So here is this input string ww reverse for all possible w's over our alphabet 0, 1. And what w reversed, by the way, means is writing w backwards. So this is all strings followed by a reversal of the same string, the string written backwards.

Really you can think of these as-- well, so these are strings that-- well, here's an example, 0, 1, 1, 1, 1, 0, the string written backwards. So this is a string in the language appearing on a tape as I described. So how is the machine going to recognize this line, which is going to be somewhat similar to before, but with one important difference. And if you imagine-- and I like to use this kind of anthropomorphizing these things, putting yourself in the place of the machine, and thinking how you would do it.

So if you imagine getting these symbols one by one, 0, 1, 1, you don't know what's coming next as you're getting the symbols. You have to figure out how to match off the second half with the first half. So you're going to put the first half on the stack, and then you're going to remove the first half and match it off with the second half.

Conveniently, the first half comes out backwards. The stack is a first in last out kind of thing. So it comes out in reverse order. So that's perfect for matching off with the second half.

But the tricky part with this language is, how do you know when you're at the middle because you don't get to see the rest. You only get to see what you've seen so far. You don't know what's coming.

So when you read that second one-- well, at this point, you're reading 0, 1, 1, now you're reading that second one, you don't know that perhaps it's just going to be a 0 following that and it's going to be all. So maybe you should be deciding so that this point here that I've marked is the midpoint. And you put 0, 1 on the tape and then start popping the second one and matching it off with the first one.

That would be a tempting thing to do, but you just don't know. And that's where the non-determinism is going to be essential. So let me write down more of this.

So what you're going to do is you're going to read and push input symbols, but non-deterministically guessing that you're at the middle. So you're going to non-deterministically either repeat that and continue to read and push symbols onto the stack, or you're going to go to 2 deciding that, or guessing that you're at the midpoint, and now it's time to start reading and popping instead of reading and pushing.

So you're going to read input symbols and popping the stack symbols, comparing the two, the symbols that you're reading at the top, the symbols you're removing from the stack. If they ever disagree, then this thread of the non-determinism rejects because either the input is not in the language or at least you have made a wrong choice as to where the midpoint is.

And then you're going to enter the accept state if the stack is empty. And ignore this part for the moment, this software reference. Let's just-- I'll speak to that in a second.

But I just want to make sure we understand that at an intuitive level how this machine is using it's non-determinism to recognize this language because the non-determinism is critical. And it's important that you understand it. So let me just make some side comments, and then we'll come back to this software remark.

So first of all, one question that comes up a lot-- well, I'm not paying attention to the chat here, sorry. So if you're not getting an answer from me, you try the TAs. But one of the-- one of the questions that comes up a lot when they're thinking about non-determinism for pushdown automata is, what happens to the stack?

The stack gets replicated in the non-determinism every time the machine forks, just like everything else gets replicated. So an entire-- every time there's a fork in the non-determinism and the machine branches into multiple possibilities, the entire machine replicates the current state, the current position of the head, the stack and its contents. All of that gets replicated.

And the two sides of the-- the two branches or the two sides of the fork each go on independently in their merry way, doing their own thing independently. And then if any one of them accepts, that's the only way they sort of-- a kind of a communication because the one that accepts raises the flag, and then the overall machine is set to accept.

So the non-determinism forks replicate the stack. [INAUDIBLE] saying it. I just want to make sure you got that. And then this language requires non-determinism, that I said earlier, so our PDA, pushdown automaton model is going to be non-deterministic. I mean, you might have examples which are deterministic, but the model is going to always allow non-determinism.

What's this bit about the software? So if you look at this formal definition here, it doesn't have anywhere in it the ability to test if the stack is empty. That's not part of the hardware specification, at least as we are describing it for a pushdown automaton. You can might imagine somebody else describes pushdown automata in some other way which gives that as a primitive, but we're not going to do that.

Why? Because we don't need a primitive for that. You can get the effect of testing if there's an empty stack, even if you don't have that as a primitive for the machine because what you can do is you can start the machine off when it-- and the very first thing it does is it writes a special symbol to mark the bottom-- what's going to eventually be the bottom of the stack.

There's going to be some special symbol, maybe a dollar sign symbol. That's the very first thing the machine does. And then it proceeds as before. And if it ever sees that dollar sign symbol again, it knows the stack is effectively empty.

So you can get the effect of testing for the static being empty, even if you don't have a primitive for that. And we're not going to actually fuss about details like that. So you can use-- when you're writing your homework sets, you can just use the assumption that you can test for an empty stack, which is what I'm going to do myself. So let's continue on. Yeah.

So now what we're going to do, we're going to prove our one-- so far we really haven't proved anything, we've just given some definitions and some examples. Today was going to-- now we're going to come to our big theorem, which actually is important and has some meat to it, and that is, how do we convert-- I claim that context-free grammars and pushdown automata are equivalent.

Well, we're going to prove that equivalence in one direction, converting the grammars to pushdown automata. So let me show you how that goes. In some ways-- and it's a nice proof, not super complicated, but it has some meat to it.

So if I give you a grammar here, what I'm going to tell you how to do is convert that grammar into pushdown automaton, which does the same language. So if you were checked out for a minute, please come back because we're starting this topic now, that you can think about this as a good re-entry point if you've been doing something else, which I can't tell, a good thing.

So converting a given grammar to a pushdown automaton, how is that going to work? So the idea is-- OK-- actually, before I tell you the idea, let's just think about it together. Again, I like to think about the pushdown automaton, building a pushdown automaton the way you would do it.

So a grammar is a generation device. It generates strings. A pushdown automaton-- or thinking about it as you, you are recognizing it. You are given an input, and you want to know, is it in the language?

So you want to know, is it possible for that grammar to generate that input you're given? So how are you going-- how are you going to do that? And how are you going to test if the input is in the language of the grammar?

Well, the thing that you would naturally do is you'd say, well, can I derive that string using the rules of the grammar? Let me start with the start string and try to do substitutions and see if I get the string I'm given. And if I can get it, then I know it's in the language. That's a natural thing to do. You're just going to try to do the substitution see if you can get to the string.

Now the thing is, there might be many different substitutions that you could make. And that seems like a really challenging, hard thing to figure out which substitutions to use among the many possibilities. That's where non-determinism is going to come in because you can think of yourself as guessing which substitutions to make. And you're always going to make the right guess.

So the choices of which substitutions to make, that's not going to be a problem for you. That's going to be managed by the non-determinism. So imagine you're always going to make the right substitution, but now the challenge is, how do you keep track of the intermediate results as you're doing those substitutions?

And that's where the stack is going to come in. The machine is going to write down those intermediate results on the stack. But even there, there's a subtlety. That's an important subtlety that you have to look at. So let's try pulling that together so far before I get to that subtlety.

So as I mentioned, the pushdown automaton is going to start out with a starting variable and is going to be guessing the substitutions to make. It's going to keep the intermediate results on the stack.

When it's done doing all the substitutions and it has only terminal strings on the stack, it can compare with the input and see if it got the right thing, so if it made all the right guesses. So you think of it as guessing, doing the right guesses, but in the end you have to check to make sure that you got the right-- that you did all the right things, and you will accept when things have matched up, and you made all the right guesses. So in the end, you have to check that you actually got that input from doing those substitutions.

So let's try to see this operating in action. And then you'll see the subtlety, the delicacy, a problem that's going to arise. Hopefully you're following, at least in part, what I'm saying so far.

So here is the input. We do know that that's an input in the language that-- we've been seeing this example multiple times. So here is the input appearing on the input tape, a plus a times a. And now the pushdown automaton is supposed to be accepting that input because it's in the language of the grammar.

So it's going to operate by first writing, to start off, the starting variable on the stack, and then doing the substitutions as we're going along. So we're going to substitute E goes to E plus T. So we do that first substitution.

And then we do the next substitution here, the E-- so if you're looking at this tree here, this is the right tree for that input. So we substitute E by T. So far so good. The automaton can make that substitution.

Then the next substitution is going to be a little-- so where E plus T, we substitute here. We got T plus T. And now we're going to substitute the T times F, which is this T over here. We want to substitute that. And that appears as T times as F now on the stack.

Now, if you're following me, you should be suddenly getting nervous because we just cheated. It's OK doing substitution-- doing these replacements at the very top of the stack because the pushdown automaton has access to the top. That's how stacks work.

But it does not have access deep down within this deck. That is not how-- that is not how stacks work. So that's cheating. But ignoring the cheating for the minute, if you could replace those-- do those substitutions deep down within the stack, this would all work. We would be good.

You would do the substitutions one after another until you ended up with no variables, and then you have the string here, and you're going to match it off and compare it with the input. It's all done in the right way so that the things are in the right order so after all the substitutions, you'd have a plus a times a sitting here in the stack, you compare that with the input, it's going to match up, and you'll end up accepting, all good.

So how do we deal with that problem here? The problem, access below the top of stack is cheating. What are we going to do instead? So the idea is actually pretty simple. Well, if you understood what I've said so far, fixing that is actually not too bad.

Sort of fading out here. I'll put some more light on my image. So how do we do that? How do we get the effect of the access below the top of the stack?

And the way we're going to do that is by making the-- what we're going to do, we're only going to do substitutions that we can make at the top of the stack. So whenever there's a variable at the top of the stack, we're going to do the substitution because the top we can access.

Now what happens if we have a terminal symbol sitting at the top blocking our way from accessing variables below? Well, actually, that's an easy case to handle because we have a terminal symbol sitting at the top, they're never going to change anyway, so you might as well match them with the input at that time.

So when you have a terminal sitting at the top, we'll just read another input symbol and match it off there. And we just keep reading the terminal symbols off until we have a variable sitting on the top, then we do a substitution. And we keep substituting variables until we have a terminal, then we read it, then we compare it with the input.

And in so doing, we will end up getting the same effect that I described before without ever needing to dig down into the interior of the stack and doing substitutions there. They're all going to rise up to the top. And we can always do them at the top. So anyway, I forgot to do that here. So instead, only substitute variables when they're at the top of the stack.

If a terminal's on the top, pop it and compare with the input, reject it if they're not equal. So if you ever have something which is not matching the way it's supposed to, I mean, that thread's just going to fail. Then there was not a-- a bad non-deterministic choice was made, or maybe the input was not in the language anyway, and there were no good choices. So my animation broke here, so let me just put the whole thing up in front of you.

So here is the actual construction. Push the start symbol on the stack. If the top of the stack is a variable, replace it with a corresponding right-hand side, doing a non-deterministic choice among the various possibilities. If it's a terminal, you pop it and match it with the next input symbol. And if the stack is empty, you accept.

So here is how the stack would actually look for this particular input. It would start off the same. You'd have E, and then substitute with E plus T. And then we're going to always do the substitutions at the top. So E gets substituted by F

Is that right? No. This slide I messed up. I apologize. So E gets substituted by T, which gets substituted by F.

And the point is that when you get to an A sitting at the top-- forgive the typos here, now we have a terminal symbol, and that's going to get matched off with the next input symbol and get removed. And now we have just the plus and the T left.

And then the plus is also a terminal symbol. That's going to get matched off with the next thing. We just have a T sitting on the top, and now we can do a substitution. So that's how it works.

That's all I wanted to say, I think. Oh, yeah, there's one just remark. So we're not going to prove this, but I do need to say this, that actually you can do the conversion in the other direction, too. You can convert a-- so a is a context-free language if and only if some pushdown automaton recognizes a.

And if you haven't seen if and only if, it's an expression I'm going to use, again, over and over. So you should get used to it. It stands for "if and only if," and which just means the implication goes both ways.

So "a is a context-free language" implies that some pushdown automaton recognizes a and vice versa. So there's really two things you need to prove whenever you have an if and only if. You have to prove both directions.

So thinking about it that way, splitting them in half, the forward direction we've already proved, converting a pushdown-- a context-free grammar to a pushdown automaton. The reverse direction we're not going to prove. It's in the book if you're curious, and you are responsible knowing that the fact is true, but you don't have to know the proof, which is a somewhat-- a little bit complicated. And I think it would take us too long to go through it, so you're not responsible for it.

So there's a last check in here that I have for you, which is just a question, which you can answer based on the material that we presented so far. Is every regular language also a context-free language? Just "yes," "no," or you're not sure.

So let me launch that as a poll here. OK, about to close. A new polling, and sharing results. This one, I think, you pretty much-- most of you got. Some of you are not sure.

Every language is in fact a context-free language. And the way to see that is that every regular language can be done by a DFA or an NFA, as we already showed. And a DFA or an NFA is really just a pushdown automaton that never uses its stack.

So you can always think of a DFA as a pushdown automaton. And we already argue that pushdown automata are equivalent to context-free grammars, and so they do the context-free languages. So anything that you can do with a DFA you can also do with a pushdown automaton, and so therefore all the regular languages are also context-free languages.

So with that, let's just pull things together, a little quick recap as to what we've been doing so far in the class. We have the regular languages and the context-free languages. We had the two forms of getting at them, the recognizer form, which is like the automata-based perspective, like either a DFA or an NFA in the case of the regular languages, the pushdown automaton for the context-free languages.

And for the generators, we had the regular expression for the regular languages and the context-free grammars-- for the context-free languages. And as we just pointed out in our last check-in, the regular languages form a subset and, in fact, a proper subset of the context-free languages, as shown in this Venn diagram because we have already exhibited languages that are context-free but not regular.

So a quick review. We've defined the context-free grammars and their associated languages, the context-free languages; we defined pushdown automata; and we showed how to convert context-free grammars to pushdown automata. And that's all I have for you today.

But here's a question I'll answer to everybody. Why do we restrict ourselves to a stack? Why don't we use random access memory? We will use random access memory for the next model, called a Turing machine. And we're going to introduce that I think the next lecture. So that's going to be the model that we're going to stick with throughout the term.

But we have not-- we were introducing weaker models as a kind of prelude to the more general-purpose computational model really to get ourselves warmed up and also because, for the weaker models, you can fully analyze them in a way that you cannot for Turing machines. As you will see, you can determine properties of languages for the weaker models that you cannot for the more general models.

And so I think that's helpful to have that perspective, that for some cases, you can get a full analysis, in some other cases you cannot. But anyway, that's the reason why we restrict it to the stack, besides the fact that these models have applications that I think are worth people seeing.

Why-- yeah, for some reason we chose a stack? Well, why did we choose a stack and not some other data structure for our temporary-- for our extra storage? And the reason for a stack, for one thing, the stack is exactly what you need to get you the correspondence with context-free grammars.

If we used some other storage, like a queue, for example, instead of a stack, in fact, you get a very different outcome. And it's an actually interesting exercise to see what happens. What do you get if you use a queue as an external storage instead of as a stack? It's a good homework problem. Maybe I'll assign it.

Let's see. And if a-- OK. We showed-- so we showed that non-determinism can be eliminated for finite automata. So NFAs and DFAs are equivalent. What about for pushdown automata?

The answer is no, they're not equivalent. I think I mentioned that earlier, but I don't mind repeating it. There are certain languages that can be done only with non-deterministic pushdown automata and cannot be done with deterministic pushdown automata, for example that language ww reverse that requires the non-determinism in order for the machine to be able to guess where the middle is.

So, OK, I'm going to head off. Thank you, guys, and see you on Tuesday.